# MAXAda for RedHawk Linux Reference Manual

concurrent

0898537-120
December 2004

Printed in U. S. A.

| Revision History: | Level: | Effective With: |
|---|---|---|
| Original Release -- March 1997 | 000 | MAXAda 1.0 |
| Previous Release -- November 2003 | 110 | MAXAda 3.4 |
| Current Release -- December 2004 | 120 | MAXAda 3.5-beta |

# Preface

## General Information

MAXAda™ is a tool set for the development of Ada programs on Concurrent computers under the PowerMAX OS environments. MAXAda processes the Ada language as specified by the Reference Manual for the Ada Programming Language, ANSI/ISO/IEC-8652:1995, referred to in this document as the Ada 95 Reference Manual or the RM. The Ada 95 Reference Manual may be obtained through the Superintendent of Documents, U.S. Government Printing Office, Washington, D.C. 20402.

The MAXAda documentation describes the operation of the Ada Programming Support Environment. It does not attempt to teach Ada or UNIX®.

The Ada 95 Reference Manual specifies all compiler-independent information about the Ada language. This document specifies all MAXAda-specific compiler-dependent information about the Ada language.

## Scope of Manual

This manual is a reference document and user guide for MAXAda.

## Structure of Manual

This manual consists of 12 chapters, four appendixes, a glossary, and an index. A brief description of the contents of each of the chapters of the manual is described as follows.

- Part 1 is Operations which contains Chapter 1 through Chapter 4. These chapters are the Introduction to MAXAda, Using MAXAda, MAXAda Concepts, and MAXAda Utilities.
- Part 2 is Run-Time which contains Chapter 5 through Chapter 7. These chapters are Run-Time Concepts, Run-Time Configuration, and Interrupt Handling.
- Part 3 is General Features which contains Chapter 8 through Chapter 9. These chapters are Shared Memory and Process Communication and Support Packages.
- Part 4 is Real-Time Features which contains Chapter 10 through Chapter 12. These chapters are Real-Time Extensions, Real-Time Event Tracing, and Real-Time Monitoring.
- Part 5 is Appendixes and Index which contains Appendix A (Troubleshooting), Appendix B (MAXAda Configuration), Appendix C (Ada NightView), Appendix M (Implementation-Defined Characteristics), a glossary, and an index.

## Syntax Notation

The following notation is used throughout this guide:

| | |
|---|---|
| *italic* | Books, reference cards, and items that the user must specify appear in *italic* type. Special terms and comments in code may also appear in *italic*. |
| **`list bold`** | User input appears in **`list bold`** type and must be entered exactly as shown. Names of directories, files, commands, options and man page references also appear in **`list bold`** type. |
| `list` | Operating system and program output such as prompts and messages and listings of files and programs appears in `list` type. Keywords also appear in `list` type. |
| *emphasis* | Words or phrases that require extra emphasis use *emphasis* type. |
| window | Keyboard sequences and window features such as push buttons, radio buttons, menu items, labels, and titles appear in window type. |
| `[ ]` | Brackets enclose command options and arguments that are optional. You do not type the brackets if you choose to specify such option or arguments. |
| `{ }` | Braces enclose mutually exclusive choices separated by the pipe ( \| ) character, where one choice must be selected. You do not type the braces or the pipe character with the choice. |
| `...` | An ellipsis follows an item that can be repeated. |
| `::=` | This symbol means *is defined as* in Backus-Naur Form (BNF). |

## Referenced Publications

The following publications are referenced in this document:

| | |
|---|---|
| 0890395 | *NightView™ User's Guide* |
| 0890398 | *NightTrace™ Manual* |
| 0890458 | *NightSim™ User's Guide* |
| 0890465 | *NightProbe™ User's Guide* |
| 0890493 | *Data Monitoring Reference Manual* |
| 0890514 | *NightBench™ User's Guide* |

# Contents

## Chapter 1 Introduction to MAXAda


MAXAda Utilities . . . . . 1-1
MAXAda Core Utilities . . . . . 1-3
Capabilities . . . . . 1-3
Run-Time Systems. . . . . 1-5
Supplied Environments . . . . . 1-6
Ada Bindings . . . . . 1-7
Complementary MAXAda Products . . . . . 1-7


## Chapter 2 Using MAXAda


Hello World - An Example . . . . . 2-1
Creating an environment . . . . . 2-1
Introducing units. . . . . 2-2
Defining a partition. . . . . 2-3
Building a partition . . . . . 2-4
Success!!! . . . . . 2-4
Let's look around.... . . . . 2-5
Listing the units in your environment . . . . . 2-5
Viewing the source for a particular unit . . . . . 2-5
Listing the partitions defined. . . . . 2-6
Looking at the Environment Search Path . . . . . 2-7
What are my options? . . . . . 2-7
Hello Galaxy - The Example Continues... . . . . 2-11
Setting up another environment . . . . . 2-11
Modifying an existing unit . . . . . 2-12
Building a unit with references outside the local environment . . . . . 2-13
Adding an environment to the Environment Search Path . . . . . 2-13
Making contact!!! . . . . . 2-14
Who resides here now?. . . . . 2-14
Hello Again... Ambiguous Units . . . . . 2-15
Resolving the ambiguity. . . . . 2-17
No more ambiguities!!! . . . . . 2-17


## Chapter 3 MAXAda Concepts


Environments. . . . . 3-1
Local Environments . . . . . 3-2
Foreign Environments . . . . . 3-2
Environment Search Path . . . . . 3-2
Naturalization . . . . . 3-3
Fetching. . . . . 3-3
Supplied Environments . . . . . 3-3
NFS Environments . . . . . 3-3
Freezing Environments. . . . . 3-4

Restoring Environments . . . 3-4
Relocating Environments . . . 3-4
Environment-wide Compile Options . . . 3-6
Units . . . 3-7
Unit Identification . . . 3-7
Configuration Pragmas . . . 3-9
Nationalities . . . 3-9
Local Units . . . 3-9
Foreign Units . . . 3-10
Ambiguous Units . . . 3-10
Artificial Units . . . 3-11
Unit Compile Options . . . 3-11
Partitions . . . 3-12
Types of Partitions . . . 3-12
Active Partitions . . . 3-12
Archives . . . 3-12
Shared Objects . . . 3-13
Lazy Versus Immediate Binding . . . 3-13
Position Independent Code . . . 3-14
Share Path . . . 3-14
Shared Objects and Special MAXAda Packages . . . 3-14
Issues to consider . . . 3-15
Elaboration and Finalization Methods . . . 3-16
Elaboration Methods . . . 3-17
Finalization Methods . . . 3-18
Main Subprogram Requirements . . . 3-19
Exit Status . . . 3-19
Compilation and Program Generation . . . 3-20
Compilation . . . 3-20
Automatic Compilation Utility . . . 3-20
Compile Options . . . 3-20
Environment-wide Options . . . 3-21
Permanent Unit Options . . . 3-21
Temporary Unit Options . . . 3-21
Effective Options . . . 3-22
Compilation States . . . 3-22
Consistency . . . 3-23
Interoptimization . . . 3-24
Programming Hints and Caveats . . . 3-25
Compiler Error Messages . . . 3-26
Lexical Errors . . . 3-27
Syntax Errors . . . 3-28
Semantic Errors . . . 3-29
General Errors . . . 3-30
Informational Messages . . . 3-31
Warnings . . . 3-32
Alerts . . . 3-32
Fatal Errors . . . 3-33
Internal Errors and Panics . . . 3-33
Link Options . . . 3-34
Linking Executable Programs . . . 3-36
Linking Ada Programs with Shared Objects . . . 3-36
Debugging . . . 3-38
Real-Time Debugging . . . 3-38

Selecting a Debug Level . . . 3-38
Degree of Interest . . . 3-40
Debug Information and cprs . . . 3-40

## Chapter 4 MAXAda Utilities

Common Options . . . 4-2
a.build . . . 4-3
Parallel Compilations and Dependency Analyses . . . 4-5
Inline Dependencies . . . 4-6
Forcing Attempts . . . 4-6
Why . . . 4-6
a.cat . . . 4-7
a.chmod . . . 4-8
a.compile . . . 4-9
a.demangle . . . 4-11
a.deps . . . 4-13
a.edit . . . 4-15
a.error . . . 4-16
a.expel . . . 4-21
a.fetch . . . 4-22
a.freeze . . . 4-25
a.help . . . 4-26
a.hide . . . 4-27
a.install . . . 4-28
a.intro . . . 4-30
a.invalid . . . 4-32
a.link . . . 4-33
a.ls . . . 4-35
Formatting the listing . . . 4-37
Dependent units . . . 4-39
Parts . . . 4-40
Sorting . . . 4-41
Filtering . . . 4-41
a.lssrc . . . 4-42
a.man . . . 4-44
References to the Ada 95 Reference Manual . . . 4-45
References to the MAXAda Reference Manual . . . 4-45
Access to Support Packages . . . 4-46
a.map . . . 4-47
a.mkenv . . . 4-53
a.monitor . . . 4-55
a.nfs . . . 4-56
a.options . . . 4-58
Option Sets . . . 4-59
Listing options . . . 4-59
Setting options . . . 4-60
Modifying options . . . 4-60
Clearing options . . . 4-60
Deleting options . . . 4-60
Keeping temporary options . . . 4-61
Setting options on foreign units . . . 4-61
a.partition . . . 4-62

Main Subprogram . . . . 4-64
Elaboration and Finalization . . . . 4-65
Case Sensitivity . . . . 4-65
Consistency . . . . 4-65
Link Options . . . . 4-65
Link Rule . . . . 4-67
Implicitly-Included Libraries . . . . 4-72
a.path . . . . 4-74
a.pclookup . . . . 4-76
a.pp . . . . 4-77
Commands . . . . 4-79
Expressions . . . . 4-80
Defaults . . . . 4-81
Examples . . . . 4-81
a.release . . . . 4-83
a.resolve . . . . 4-85
a.restore . . . . 4-86
a.rmenv . . . . 4-87
a.rmsrc . . . . 4-88
a.script . . . . 4-89
Generated Script - Options . . . . 4-91
a.syntax . . . . 4-92
a.tags . . . . 4-94
a.touch . . . . 4-97
a.trace . . . . 4-98
Compile Options . . . . 4-99
Negation (**!**) . . . . 4-99
Debug Level (**-g[***level***]**) . . . . 4-100
Opportunism (**-opp**) . . . . 4-100
Share Mode (**-sm**) . . . . 4-101
Not Shared (**-N**) . . . . 4-101
Optimization Level (**-O[***level***]**) . . . . 4-101
Qualifier Keyword (**-Q***keyword*[**=***value*]) . . . . 4-104
Suppress Checks (**-S**) . . . . 4-104
Qualifier Keywords (**-Q** options) . . . . 4-105
Link Options . . . . 4-109
Share Path . . . . 4-110
Incrementally Updateable Partition . . . . 4-110
Tracing . . . . 4-110
Task Weight . . . . 4-111
Shared Object Transitive Closure . . . . 4-111
Obscurity Checks . . . . 4-112


## Chapter 5 Run-Time Concepts


Tasking Model . . . . 5-1
Features . . . . 5-2
Performance . . . . 5-2
Task Weights . . . . 5-3
Bound Tasks . . . . 5-3
Multiplexed Tasks . . . . 5-3
Task Scheduling . . . . 5-3
Task Time Slices . . . . 5-3

Utilization of Multiple CPUs . . . . . 5-4
Ghost Tasks . . . . . 5-5
ADMIN Ghost Task . . . . . 5-5
TIMER Ghost Task . . . . . 5-5
Priorities . . . . . 5-5
OS Scheduling Policies . . . . . 5-7
Policy Selection by the Non-Tasking Run-Time . . . . . 5-8
Policy Selection by the Tasking Run-Time . . . . . 5-8
Restrictions for Priorities in the System.Interrupt_Priority Range . . . 5-9
Memory Management . . . . . 5-11
Text Memory . . . . . 5-11
Data Memory . . . . . 5-11
Collection Memory . . . . . 5-11
Stack Memory . . . . . 5-12
Other Memory . . . . . 5-12
Visibility of Memory . . . . . 5-13

## Chapter 6 Run-Time Configuration

General Pragmas . . . . . 6-1
Pragma RUNTIME_DIAGNOSTICS . . . . . 6-1
Pragma MAP_FILE . . . . . 6-2
Pragma QUEUING_POLICY . . . . . 6-2
Pragma TASK_DISPATCHING_POLICY . . . . . 6-2
Pragma LOCKING_POLICY . . . . . 6-3
Pragma SERVER_CACHE_SIZE . . . . . 6-4
Task and Group Configuration Concepts . . . . . 6-4
Task Names and Default Settings . . . . . 6-4
Task Specifiers in Task Pragmas . . . . . 6-5
Group Names and Default Settings . . . . . 6-7
Group Specifiers in Group Pragmas . . . . . 6-8
Task Attributes . . . . . 6-9
Pragma TASK_WEIGHT . . . . . 6-9
Pragma TASK_PRIORITY . . . . . 6-11
Pragma TASK_CPU_BIAS . . . . . 6-12
Pragma TASK_QUANTUM . . . . . 6-14
Pragma TASK_HANDLER . . . . . 6-15
Group Attributes . . . . . 6-18
Pragma GROUP_PRIORITY . . . . . 6-18
Pragma GROUP_CPU_BIAS . . . . . 6-19
Pragma GROUP_SERVERS . . . . . 6-19
Memory Attributes . . . . . 6-20
Pool Specifiers . . . . . 6-21
Pragma MEMORY_POOL . . . . . 6-23
Pragma POOL_CACHE_MODE . . . . . 6-25
Pragma POOL_LOCK_STATE . . . . . 6-25
Pragma POOL_SIZE . . . . . 6-26
Pragma POOL_PAD . . . . . 6-28
Protected Object Attributes . . . . . 6-28
Pragma PROTECTED_PRIORITY . . . . . 6-28

## Chapter 7 Interrupt Handling


Software Interrupts . . . . . 7-2
COURIER Ghost Tasks . . . . . 7-3
SHADOW Ghost Tasks . . . . . 7-4
Hardware Interrupts . . . . . 7-4
INTR_COURIER and COURIER Ghost Tasks . . . . . 7-5
SHADOW Ghost Tasks . . . . . 7-6
Privileges for Unrestricted Hardware Interrupts . . . . . 7-6
Interrupt Attachments . . . . . 7-6
Package Ada.Interrupts.Names . . . . . 7-6
Package Ada.Interrupts.Services . . . . . 7-7
Task Executives via Protected Handlers . . . . . 7-7
Example . . . . . 7-7
Description of Example . . . . . 7-10


## Chapter 8 Shared Memory and Process Communication


Shared Memory . . . . . 8-1
Shared Packages . . . . . 8-1
Pragma SHARED_PACKAGE . . . . . 8-1
Restrictions on Contents of Shared Packages . . . . . 8-4
Characteristics of Shared Packages . . . . . 8-4
Shared Package Semaphores . . . . . 8-5


## Chapter 9 Support Packages


Supplied Environments . . . . . 9-5
predefined . . . . . 9-6
vendorlib . . . . . 9-8
bit_ops . . . . . 9-9
ada.exceptions.addresses . . . . . 9-9
ada.numerics.constants . . . . . 9-10
ada.real_time.local . . . . . 9-10
runtime_configuration . . . . . 9-10
shared_memory_support . . . . . 9-10
system.addresses . . . . . 9-11
system.information . . . . . 9-11
system.storage_pools.standard . . . . . 9-11
system.storage_pools.standard.objects . . . . . 9-11
publiclib . . . . . 9-11
c_to_ada_types . . . . . 9-12
character_type . . . . . 9-12
curses . . . . . 9-12
qsort . . . . . 9-12
rtdm . . . . . 9-12
real_time_data_monitoring . . . . . 9-12
deprecated . . . . . 9-13
obsolescent . . . . . 9-13
posix_1003.1 . . . . . 9-14
posix_1003_1 . . . . . 9-14
posix_1003.5 . . . . . 9-15
sockets . . . . . 9-16

sockets . . . . . 9-16
general . . . . . 9-16
night_trace_bindings . . . . . 9-17
timers . . . . . 9-17

## Chapter 10 Real-Time Extensions

Mutual Exclusion Interfaces . . . . . 10-1
Spin Locks . . . . . 10-1
Binary Semaphores. . . . . 10-2
Tasking Semaphores. . . . . 10-4
Task Synchronization. . . . . 10-6
Cyclic Scheduling . . . . . 10-6
User Trace . . . . . 10-8
Low-Level Interfaces. . . . . 10-8
Indivisible Operations . . . . . 10-8
Rescheduling Control. . . . . 10-10
Client-Server Services . . . . . 10-11
Usermap Support . . . . . 10-11

## Chapter 11 Real-Time Event Tracing

Specifying Trace Events . . . . . 11-1
Predefined Trace Events. . . . . 11-2
Library Unit Elaboration. . . . . 11-2
User-Defined Trace Events. . . . . 11-2
user_trace package . . . . . 11-3
Specification. . . . . 11-4
Usage . . . . . 11-5
user_trace.raw package . . . . . 11-5
Specification. . . . . 11-7
NightTrace Binding. . . . . 11-8
Specification. . . . . 11-9
Usage . . . . . 11-12
NightView Debugger. . . . . 11-13
Tracing Options . . . . . 11-14
Tracing Options - Examples. . . . . 11-17
Logging Trace Events . . . . . 11-19
Logging Mechanisms . . . . . 11-19
Ada Executive . . . . . 11-19
Trace Buffer . . . . . 11-20
Timing Source . . . . . 11-20
NightTrace Daemon . . . . . 11-21
Log Files. . . . . 11-22
Viewing Trace Events . . . . . 11-23
User Table. . . . . 11-23
Viewing Trace Events with a.trace. . . . . 11-24
Viewing Trace Events with NightTrace . . . . . 11-25
Creating the NightTrace Configuration File . . . . . 11-25
Modifying the NightTrace Configuration File . . . . . 11-26

## Chapter 12 Real-Time Monitoring


Data Monitoring . . . 12-1
- Compiling . . . 12-1
- Eligible Data Objects . . . 12-1
- Eligible Data Types . . . 12-2
- real_time_data_monitoring Package . . . 12-2

Task Monitoring . . . 12-3
- a.monitor . . . 12-4
  - Menu Bar . . . 12-6
    - File . . . 12-6
    - View . . . 12-8
    - Options . . . 12-8
  - Task Bar . . . 12-11
  - Display Area . . . 12-13
    - Tasks . . . 12-13
    - Memory . . . 12-17
    - System . . . 12-20


## Appendix A Troubleshooting


Configuration Errors . . . A-1
- System Configuration . . . A-1
- Application Configuration . . . A-2
  - Using Tasks to Multithread Algorithms . . . A-2

User Errors . . . A-2
- Concurrent Access . . . A-2
- Hung Processes . . . A-3
- Client/Server Services . . . A-3

Run-Time Diagnostics . . . A-4
- Run-Time Diagnostic Messages . . . A-4

Compiler Errors . . . A-6


## Appendix B MAXAda Configuration


Capabilities . . . B-1


## Appendix C Ada NightView


Hints for Debugging Ada Programs with NightView . . . C-1
- Tasking Programs . . . C-1
  - Debugging Context . . . C-2
- Exception Handling and Interception . . . C-3
- Generics . . . C-4
- General NightView Operational Hints . . . C-4
  - Listing Source, Packages, and Subprograms . . . C-4
  - Disassembly . . . C-5
  - Interest Threshold . . . C-5
  - Expression Evaluation Syntax . . . C-5


## Appendix M Implementation-Defined Characteristics


RM Chapter 1: General . . . M-2

RM 1.1.2 Structure . . . . . M-2
RM 1.1.3 Conformity of an Implementation with the Standard. . . . . M-2
RM 1.1.4 Method of Description and Syntax Notation . . . . . M-4
RM Chapter 2: Lexical Elements . . . . . M-5
RM 2.1 Character Set . . . . . M-5
RM 2.2 Lexical Elements, Separators, and Delimiters . . . . . M-5
RM 2.8 Pragmas . . . . . M-6
RM Chapter 3: Declarations and Types. . . . . M-9
RM 3.5 Scalar Types . . . . . M-9
RM 3.5.2 Character Types . . . . . M-9
RM 3.5.4 Integer Types . . . . . M-9
RM 3.5.5 Operations of Discrete Types . . . . . M-10
RM 3.5.6 Real Types . . . . . M-10
RM 3.5.7 Floating Point Types. . . . . M-11
RM 3.5.9 Fixed Point Types. . . . . M-11
RM 3.6.2 Operations of Array Types . . . . . M-12
RM 3.9 Tagged Types and Type Extensions . . . . . M-12
RM Chapter 4: Names and Expressions . . . . . M-13
RM 4.1.4 Attributes . . . . . M-13
RM 4.3.1 Record Aggregates . . . . . M-16
RM Chapter 5: Statements. . . . . M-17
RM Chapter 6: Subprograms. . . . . M-18
RM Chapter 7: Packages . . . . . M-19
RM Chapter 8: Visibility Rules . . . . . M-20
RM Chapter 9: Tasks and Synchronizations . . . . . M-21
RM 9.6 Delay Statements, Duration, and Time . . . . . M-21
RM 9.10 Shared Variables . . . . . M-22
RM Chapter 10: Program Structure and Compilation Issues . . . . . M-23
RM 10.1 Separate Compilation . . . . . M-23
RM 10.1.4 The Compilation Process . . . . . M-23
RM 10.1.5 Pragmas and Program Units. . . . . M-24
RM 10.2 Program Execution . . . . . M-25
RM 10.2.1 Elaboration Control . . . . . M-29
RM Chapter 11: Exceptions. . . . . M-30
RM 11.4.1 The Package Exceptions. . . . . M-30
RM 11.5 Suppressing Checks. . . . . M-31
RM Chapter 12: Generic Units . . . . . M-32
RM Chapter 13: Representation Issues . . . . . M-33
RM 13.1 Representation Items. . . . . M-33
RM 13.2 Pragma Pack . . . . . M-34
RM 13.3 Representation Attributes . . . . . M-35
Address Attributes. . . . . M-35
Alignment Attributes. . . . . M-36
Size Attributes for Objects . . . . . M-39
Size Attributes for Subtypes . . . . . M-40
Component_Size Attributes . . . . . M-42
External_Tag Attributes . . . . . M-43
RM 13.4 Enumeration Representation Clauses . . . . . M-43
RM 13.5.1 Record Representation Clauses . . . . . M-44
RM 13.5.2 Storage Place Attributes. . . . . M-46
RM 13.5.3 Bit Ordering . . . . . M-46
RM 13.7 The Package System . . . . . M-47
RM 13.7.1 The Package System.Storage_Elements . . . . . M-47
RM 13.8 Machine Code Insertions. . . . . M-48

Pentium . . . . . M-48
Pentium Instruction Set . . . . . M-49
Pentium Register Set . . . . . M-54
Pentium Address Modes . . . . . M-54
Pentium Machine Code Example . . . . . M-55
RM 13.9 Unchecked Type Conversions . . . . . M-55
RM 13.11 Storage Management . . . . . M-59
RM 13.11.2 Unchecked Storage Deallocation . . . . . M-62
RM 13.12 Pragma Restrictions . . . . . M-62
RM 13.13.2 Stream-Oriented Attributes . . . . . M-63
RM Annex A: Predefined Language Environment . . . . . -64
RM A.1 The Package Standard . . . . . -64
RM A.3.2 The Package Characters.Handling . . . . . -65
RM A.4.4 Bounded-Length String Handling . . . . . -65
RM A.5.1 Elementary Functions . . . . . -65
RM A.5.2 Random Number Generation . . . . . -66
RM A.5.3 Attributes of Floating Point Types . . . . . -68
RM A.7 External Files and File Objects . . . . . -68
RM A.9 The Generic Package Storage_IO . . . . . -71
RM A.10 Text Input-Output . . . . . -71
RM A.10.7 Input-Output of Characters and Strings . . . . . -72
RM A.10.9 Input-Output for Real Types . . . . . -72
RM A.13 Exceptions in Input-Output . . . . . -72
RM A.15 The Package Command_Line . . . . . -73
RM Annex B: Interface to Other Languages . . . . . -74
RM B.1 Interfacing Pragmas . . . . . -74
RM B.2 The Package Interfaces . . . . . -78
RM B.3 Interfacing with C . . . . . -79
RM B.4 Interfacing with COBOL . . . . . -81
RM B.5 Interfacing with Fortran . . . . . -81
RM Annex C: Systems Programming . . . . . -83
RM C.1 Access to Machine Operations . . . . . -83
RM C.3 The Package Interrupts . . . . . -84
RM C.3.1 Protected Procedure Handlers . . . . . -87
RM C.3.2 The Package Interrupts . . . . . -87
RM C.4 Preelaboration Requirements . . . . . -87
RM C.5 Pragma Discard_Names . . . . . -88
RM C.7.1 The Package Task_Identification . . . . . -88
RM C.7.2 The Package Task_Attributes . . . . . -89
RM Annex D: Real-Time Systems . . . . . -91
RM D.1 Task Priorities . . . . . -91
RM D.2.1 The Task Dispatching Model . . . . . -91
RM D.2.2 The Standard Task Dispatching Policy . . . . . -91
RM D.3 Priority Ceiling Locking . . . . . -92
RM D.4 Entry Queuing Policies . . . . . -93
RM D.6 Preemptive Abort . . . . . -93
RM D.7 Tasking Restrictions . . . . . -94
RM D.8 Monotonic Time . . . . . -95
RM D.9 Delay Accuracy . . . . . -97
RM D.12 Other Optimizations and Determinism Rules . . . . . -97
RM Annex G: Numerics . . . . . -98
RM G.1 Complex Arithmetic . . . . . -98
RM G.1.1 Complex Types . . . . . -98
RM G.1.2 Complex Elementary Functions . . . . . -99

RM G.2 Numeric Performance Requirements . . . . . -99
RM G.2.1 Model of Floating Point Arithmetic. . . . . -100
RM G.2.3 Model of Fixed Point Arithmetic. . . . . -100
RM G.2.4 Accuracy Requirements for the Elementary Functions . . . . . -100
RM G.2.6 Accuracy Requirements for Complex Arithmetic . . . . . -101
RM Annex J: Obsolescent Features. . . . . -102
RM J.7.1 Interrupt Entries . . . . . -102
RM Annex K: Language-Defined Attributes . . . . . -103
RM Annex L: Pragmas . . . . . -104
Pragma ALL_CALLS_REMOTE - (not yet supported) . . . . . -106
Pragma ASSIGNMENT . . . . . -106
Pragma ASYNCHRONOUS - (not yet supported) . . . . . -106
Pragma ATOMIC . . . . . -106
Pragma ATOMIC_COMPONENTS . . . . . -107
Pragma ATTACH_HANDLER . . . . . -107
Pragma CONTROLLED . . . . . -107
Pragma CONVENTION. . . . . -108
Pragma DATA_RECORD - (obsolete). . . . . -109
Pragma DEBUG. . . . . -109
Pragma DEPRECATED_FEATURE . . . . . -110
Pragma DISCARD_NAMES . . . . . -110
Pragma DONT_ELABORATE . . . . . -110
Pragma ELABORATE . . . . . -111
Pragma ELABORATE_ALL . . . . . -111
Pragma ELABORATE_BODY . . . . . -111
Pragma EXPORT . . . . . -111
Pragma EXTERNAL_NAME - (obsolete). . . . . -112
Pragma FAST_INTERRUPT_TASK . . . . . -113
Pragma GROUP_CPU_BIAS . . . . . -113
Pragma GROUP_PRIORITY. . . . . -113
Pragma GROUP_SERVERS . . . . . -114
Pragma IMPLICIT_CODE. . . . . -114
Pragma IMPORT . . . . . -114
Pragma INLINE . . . . . -115
Pragma INSPECTION_POINT - (not yet supported) . . . . . -116
Pragma INTERESTING. . . . . -117
Pragma INTERFACE - (obsolete) . . . . . -117
Pragma INTERFACE_NAME - (obsolete) . . . . . -117
Pragma INTERFACE_OBJECT - (obsolete). . . . . -118
Pragma INTERFACE_SHARED - (obsolete) . . . . . -118
Pragma INTERRUPT_HANDLER . . . . . -118
Pragma INTERRUPT_PRIORITY. . . . . -118
Pragma LINK_OPTION - (obsolete) . . . . . -119
Pragma LINKER_OPTIONS . . . . . -119
Pragma LIST . . . . . -119
Pragma LOCKING_POLICY . . . . . -120
Pragma MAP_FILE . . . . . -120
Pragma MEMORY_POOL. . . . . -120
Pragma NORMALIZE_SCALARS - (not yet supported) . . . . . -121
Pragma OPT_FLAGS. . . . . -121
Pragma OPT_LEVEL. . . . . -122
Pragma OPTIMIZE . . . . . -122
Pragma PACK. . . . . -123
Pragma PAGE. . . . . -123

Pragma PASSIVE_TASK - (obsolete) . . . . . -123
Pragma POOL_CACHE_MODE . . . . . -124
Pragma POOL_LOCK_STATE . . . . . -124
Pragma POOL_PAD . . . . . -124
Pragma POOL_SIZE . . . . . -124
Pragma PREELABORATE. . . . . -125
Pragma PRIORITY. . . . . -125
Pragma PROTECTED_PRIORITY . . . . . -125
Pragma PURE. . . . . -127
Pragma QUEUING_POLICY. . . . . -127
Pragma REMOTE_CALL_INTERFACE - (not yet supported). . . . . -127
Pragma REMOTE_TYPES - (not yet supported). . . . . -127
Pragma RESTRICTIONS. . . . . -128
Pragma RETURN_CONVENTION. . . . . -128
Pragma REVIEWABLE - (not yet supported) . . . . . -129
Pragma RUNTIME_DIAGNOSTICS . . . . . -129
Pragma SERVER_CACHE_SIZE . . . . . -129
Pragma SHARE_BODY. . . . . -129
Pragma SHARE_MODE . . . . . -130
Pragma SHARED - (obsolete) . . . . . -131
Pragma SHARED_PACKAGE. . . . . -131
Pragma SHARED_PASSIVE - (not yet supported) . . . . . -131
Pragma SPECIAL_FEATURE . . . . . -131
Pragma STORAGE_SIZE . . . . . -132
Pragma SUPPRESS . . . . . -132
Pragma SUPPRESS_ALL . . . . . -133
Pragma TASK_CPU_BIAS . . . . . -133
Pragma TASK_DISPATCHING_POLICY. . . . . -133
Pragma TASK_HANDLER . . . . . -134
Pragma TASK_PRIORITY. . . . . -134
Pragma TASK_QUANTUM. . . . . -134
Pragma TASK_WEIGHT . . . . . -135
Pragma TDESC. . . . . -135
Pragma TRAMPOLINE . . . . . -135
Pragma VOLATILE . . . . . -135
Pragma VOLATILE_COMPONENTS. . . . . -136

## Illustrations

Figure 3-1. Package specification . . . . . 3-8
Figure 4-1. Profiling a Program . . . . . 4-7
Figure 4-2. Environment scenario containing obscurities . . . . . 4-29
Figure 4-3. Example of using **`a.fetch`** to resolve obscurities . . . . . 4-29
Figure 4-4. Link Rule Example . . . . . 4-81
Figure 5-1. Mapping of Various Priority Interpretations on PowerMAX OS. . . . . 5-7
Figure 5-2. Mapping of Various Priority Interpretations on RedHawk Linux . . . . . 5-9
Figure 6-1. Example Configuration for 6-Processor Series 6000 System . . . . . 6-28
Figure 6-2. Memory Usage on a 6-Processor Series 6000 System. . . . . 6-29
Figure 11-1. Viewing Trace Events . . . . . 11-23
Figure 12-1. Program Specification dialog . . . . . 12-21
Figure 12-2. Task display dialog . . . . . 12-24
Figure 12-3. Memory display dialog . . . . . 12-26
Figure 2-1. NightView Data Window . . . . . C-2

Figure M-1. An object of an elementary type . . . . . M-69
Figure M-2. An object of a composite type . . . . . M-69
Figure M-3. Convert small elementary object to large elementary object . . . . . M-70
Figure M-4. Convert large elementary object to small elementary object . . . . . M-70
Figure M-5. Convert small composite object to large composite object . . . . . M-70
Figure M-6. Convert large composite object to small composite object . . . . . M-71
Figure M-7. Convert small elementary object to large composite object . . . . . M-71
Figure M-8. Convert large elementary object to small composite object . . . . . M-71
Figure M-9. Convert small composite object to large elementary object . . . . . M-72
Figure M-10. Convert large composite object to small elementary object . . . . . M-72

## Tables

Table 1-1. MAXAda Utilities . . . . . 1-1
Table 1-2. MAXAda Core Utilities . . . . . 1-3
Table 2-1. Effective options for `hello` unit . . . . . 2-10
Table 2-2. Effective options for `hello` unit (after **`-keeptemp`**) . . . . . 2-11
Table 3-1. Effective options based on hierarchical relationship . . . . . 3-23
Table 3-2. Relevance of Options . . . . . 3-25
Table 3-3. MAXAda-supplied Shared Objects . . . . . 3-38
Table 4-1. Number of Parallel Dependency Analyses . . . . . 4-10
Table 4-2. **`a.ls -format`** — Descriptors . . . . . 4-45
Table 4-3. **`a.ls -format`** — Modifiers . . . . . 4-46
Table 4-4. Levels of Optimization . . . . . 4-120
Table 4-5. Linux PLDE Cross Development Libraries . . . . . 4-130
Table 4-6. Target Architectures . . . . . 4-130
Table 6-1. Stack Pool Sizes for Ghost Tasks . . . . . 6-33
Table 7-1. Erroneous Behavior Due to User-Defined Signal Handlers . . . . . 7-3
Table 9-1. Support environments . . . . . 9-1
Table 9-2. Support packages . . . . . 9-2
Table 9-3. predefined environment . . . . . 9-6
Table 9-4. vendorlib environment . . . . . 9-9
Table 9-5. publiclib environment . . . . . 9-13
Table 9-6. rtdm environment . . . . . 9-13
Table 9-7. obsolescent environment . . . . . 9-15
Table 9-8. posix_1003.1 environment . . . . . 9-15
Table 9-9. posix_1003.5 environment . . . . . 9-16
Table 9-10. sockets environment . . . . . 9-17
Table 9-11. general environment . . . . . 9-18
Table B-1. Required Privileges . . . . . B-2
Table B-3. Required Kernel Options . . . . . B-4
Table B-2. Required Capabilities . . . . . B-4
Table B-4. Significant Kernel Tunable Parameters . . . . . B-5
Table M-1. Alignment Restrictions . . . . . M-39

# 1
# Operations

**Replace with Part 1 tab**

# Part 1 - Operations

## Part 1 Operations

Chapter 1 Introduction to MAXAda ..... 1-1

Chapter 2 Using MAXAda ..... 2-1

Chapter 3 MAXAda Concepts ..... 3-1

Chapter 4 MAXAda Utilities ..... 4-1

# 1
# Introduction to MAXAda

MAXAda Utilities . . . . . 1-1
MAXAda Core Utilities . . . . . 1-3
Capabilities . . . . . 1-3
Run-Time Systems. . . . . 1-5
Supplied Environments . . . . . 1-6
Ada Bindings . . . . . 1-7
Complementary MAXAda Products . . . . . 1-7

# 1
# Introduction to MAXAda

MAXAda is a high-performance system intended for the large-scale development of Ada application, real-time, and systems software. MAXAda supports the Ada language specification as defined in the Ada 95 Reference Manual.

The run-time system provides a complete real-time implementation of all language-defined features. It can be configured to satisfy the demands of the most stringent real-time Ada applications as well as those of less critical, time-sharing applications.

## MAXAda Utilities

MAXAda consists of a number of utilities that provide support for library management, compilation and program generation, and debugging. Table 1-1 lists these tools and gives a brief description of each one.

**Table 1-1. MAXAda Utilities**

| **Environment Utilities** | |
|---|---|
| **a.mkenv** | Create an environment which is required for compilation, linking, etc. |
| **a.path** | Display or change the Environment Search Path for an environment |
| **a.options** | Set compilation options for the environment (or for units) |
| **a.rmenv** | Destroy an environment; compilation, linking, etc. no longer possible |
| **a.script** | Create script that will reproduce environment or part thereof |
| **a.nfs** | Display or change NFS aspects of an environment |
| **a.chmod** | Modify the UNIX file system permissions of an environment |
| **a.release** | Display release installation information |
| **a.restore** | Restore a damaged environment |
| **a.freeze** | Disallow changes to, and optimize uses of an environment |
| **Unit Utilities** | |
| **a.ls** | List units in the environment (state, source file, dependencies, etc.) |
| **a.options** | Set compilation options for units (or the environment) |
| **a.edit** | Edit the source of a unit, then update the environment |
| **a.cat** | Output the source of a unit |

**Table 1-1. MAXAda Utilities (Cont.)**

| | |
|---|---|
| **a.touch** | Make the environment consider a unit consistent with its source file's timestamp |
| **a.invalid** | Force a unit to be inconsistent thus requiring it to be recompiled |
| **a.resolve** | Resolve ambiguities created when a unit exists in multiple source files |
| **a.hide** | Mark units as being persistently hidden in the environment |
| **a.fetch** | Fetch the compiled form of a unit from another environment |
| **a.expel** | Expel fetched or naturalized units from the environment |
| **Source File Utilities** | |
| **a.intro** | Introduce source files (and units therein) to the environment |
| **a.rmsrc** | Remove knowledge of source files (and units therein) from the environment |
| **a.syntax** | Check the syntax of source files |
| **a.tags** | Generate a cross reference file |
| **Debug Utilities** | |
| **a.trace** | Format and display raw trace records |
| **a.map** | Display or edit the run-time configuration of an executable |
| **a.monitor** | Monitor tasking in real-time for debugging |
| **a.pclookup** | Filter standard input adding symbolic descriptions for pc values |
| **Compilation Utilities** | |
| **a.build** | Compile and link as necessary to build a unit, partition or environment |
| **a.partition** | Define or display a partition for the linker |
| **Internal Utilities** | |
| **a.install** | Install, remove, or modify a release installation |
| **a.pp** | Preprocess a source file |
| **a.deps** | Update environment with information about units within source files |
| **a.compile** | Compile the specification and/or body of one or more units |
| **a.error** | Process diagnostic messages generated by the compiler and other tools |
| **a.link** | Link a partition (an executable, archive or shared object file) |
| **Help Utilities** | |
| **a.help** | List usage and summary of each MAXAda utility |
| **a.man** | Invoke/position interactive help system (requires an X terminal) |

## MAXAda Core Utilities

Of the MAXAda Utilities listed in Table 1-1, there are four tools that form the "core" of the MAXAda system. These tools will most likely be used quite heavily and therefore are given special attention here.

**Table 1-2. MAXAda Core Utilities**

# Capabilities

RedHawk Linux provides a means to grant otherwise unprivileged users the authority to perform certain privileged operations. The **`pam_capability(8)`** (Pluggable Authentication Module) is used to manage sets of capabilities, called roles, required for various activities.

RedHawk systems should be configured with an `adauser` role which provides the capabilities required by MAXAda. In order to run MAXAda tasking programs on a RedHawk target, each MAXAda user must be configured to use (at a minimum) the capabilities specified below. In addition, the **`/etc/pam.d`** configuration files associated with the **`rsh`** and **`login`** services must be modified.

To configure user capabilities, edit the **`/etc/pam.d/rsh`** and **`/etc/pam.d/login`** files as `root`, adding the following line to each, if it is not already present:

```
session    required  /lib/security/pam_capability.so
```

Then edit **`/etc/security/capability.conf`** and define the `adauser` role (if it is not already defined) in the "ROLES" section:

```
role adauser cap_sys_admin cap_sys_nice cap_sys_rawio cap_ipc_lock
```

and, for each MAXAda user on the target system, add the following line at the end of the file:

```
user   username     adauser
```

where *username* is the login name of the user.

If the user requires capabilities not defined in the `adauser` role, add a new role which contains `adauser` and the additional capabilities needed, and substitute the new role name for `adauser` in the text above.

In order for the above changes to take effect, the user should log off and log back onto the target system.

**NOTE**

The **/etc/pam.d/rsh** and **/etc/pam.d/login** files, if edited as shown above, will allow capabilities to be granted to users who log into the system via **telnet**, **rlogin**, and **rsh**. Other methods of accessing the system may require that additional files in **/etc/pam.d** have similar modifications. For example, **/etc/pam.d/gdm**, **/etc/pam.d/kde**, and **/etc/pam.d/ssh**. To check to see if you have been granted capabilities, issue the following command:

**cat /proc/self/status**.

The last three lines labelled `CapInh`, `CapPrm`, and `CapEff` should have non-zero values if you have been granted capabilities.

See "Capabilities" on page B-1 for more detailed information.

# Run-Time Systems

The Ada Real-Time Multiprocessor System (ARMS) is broken into two libraries:

- The tasking run-time (**`libruntime.arms`**)
- The non-tasking run-time (**`libruntime.bart`**)

The **`a.link`** tool will link an application with the smaller, simpler non-tasking run-time whenever possible (see "a.link" on page 4-33). However, certain features require the use of the tasking run-time, including certain cases that do not require tasking.

The full set of features that require the tasking run-time is summarized here:

- Presence of any tasks or task types
- Presence of any protected units or protected types
- Delay until statements with a `delay_expression` of type `Ada.Calendar.Time`
- Use of any of the following pragmas:
  - `pragma TASK_PRIORITY`
  - `pragma TASK_CPU_BIAS`
  - `pragma TASK_QUANTUM`
  - `pragma GROUP_SERVERS`
  - `pragma GROUP_CPU_BIAS`
  - `pragma GROUP_PRIORITY`
  - `pragma MEMORY_POOL`
  - `pragma POOL_CACHE_MODE`
  - `pragma POOL_LOCK_STATE`
  - `pragma POOL_SIZE`
  - `pragma SHARED_PACKAGE`
- Semantic dependence on any of the following packages:
  - `Ada.Interrupts.Names.Services`
  - `Ada.Interrupts.Services`
  - `Ada.Synchronous_Task_Control`
  - `Ada.Task_Identification`
  - `Ada.Task_Attributes`
  - `Runtime_Configuration`
  - `Tasking_Semaphores`

- Use of the **-trace** link option (see "Link Options" on page 4-109)

The run-time system implements Ada tasks as states of execution that are served by one or more operating system clones. For critical real-time performance and predictability, the run-time system may be configured such that a single clone is dedicated to serve each Ada task. For applications with less stringent scheduling demands, it may be configured such that one or more clones serve all Ada tasks. (This is the default behavior). See "Task Weights" on page 5-3 for details.

See Chapter 5 - "Run-Time Concepts" for further information.

# Supplied Environments

MAXAda supplies a number of environments containing various packages that can be used for program development.

The Predefined Language Environment (**predefined**) contains packages as defined in Annex A of the Ada 95 Reference Manual. According to the Reference Manual, the library units listed in this Annex "shall be provided by every implementation".

The **vendorlib** environment contains a variety of extensions to MAXAda that can be utilized with Concurrent real-time services (see "Ada Bindings" below). It also provides shared memory support, run-time interfaces, and interfaces to system services.

The **publiclib** environment contains general-purpose, public-domain Ada packages. Concurrent neither owns nor supports any of the packages in **publiclib**; these packages are provided as a courtesy to users.

An interface is provided within the **rtdm** environment that allows for viewing and modifying data objects without prior knowledge of the objects themselves or their data types. More information about real-time data-monitoring is provided in Chapter 12, "Real-Time Monitoring".

The **obsolescent** environment corresponds to packages found within Annex J of the Ada 95 Reference Manual. These packages are designated by Annex J as having "functionality which is largely redundant with other features defined by this International Standard". Use of these features is not recommended in newly written programs.

And lastly, the **deprecated** environment is supplied for compatibility purposes with previous versions only. It will be removed in a future release of MAXAda.

Each of these environments and the packages contained within them are described in more detail in Chapter 9, "Support Packages".

# Ada Bindings

MAXAda provides several environments of Ada "bindings" to various libraries and services. Ada bindings furnish a pure Ada interface to libraries of routines and services which have been originally developed in another programming language.

MAXAda supplies Ada bindings to most Concurrent real-time features available with PowerMAX OS as well as interfaces to the run-time system. These bindings are available in the **vendorlib** environment of Ada packages. Within this environment, there is also an Ada binding to high-resolution timing devices which can be used to obtain high-resolution timings above and beyond the accuracy of the clock function provided in the **predefined** environment.

MAXAda also provides Ada bindings to other libraries and services in the MAXAda **bindings** directory. This directory holds several subdirectories, each containing an environment of Ada bindings to a specific library, service, or set of services.

Currently, the **bindings** directory contains the following environments:

| | |
|---|---|
| **general** | general-purpose Ada bindings |
| **sockets** | bindings to sockets |
| **posix_1003.1** | thin Ada bindings to IEEE-Std-1003.1 (POSIX 1003.1) and IEEE-Std-1003.1b (POSIX 1003.1b) |
| **posix_1003.5** | abstract Ada bindings to the IEEE-Std-1003.5-1992 standard (POSIX 1003.5). |

All of these are provided with MAXAda and are shipped with the MAXAda product.

Each of these environments and the packages contained within them are described in more detail in Chapter 9, "Support Packages".

# Complementary MAXAda Products

In addition to the Ada bindings supplied with MAXAda, several other Ada bindings and complementary utilities are available as stand-alone products. If they are desired, they must be purchased separately from MAXAda.

NightBench is a graphical user interface that organizes all of the information required for the development of MAXAda applications, ensures consistent, repeatable builds, and provides an efficient interface for editing, browsing, building, and debugging. For more information on NightBench, see the *NightBench User's Guide*.

NightTrace is a graphical debugging and performance-analysis tool that works with single and multi-process programs running on one or more CPUs. It may be used with Ada, C, and Fortran programs. For more information on NightTrace, see Chapter 11 and the *NightTrace User's Guide*.

NightSim is a graphical, non-intrusive tool for scheduling and monitoring real-time applications. It allows interactive control of the high-resolution Frequency-Based Scheduler (FBS) and interactive or deferred performance monitoring. It may be used with single and multi-process Ada, C, and Fortran programs running on one or more CPUs. For more information about NightSim, see the *NightSim User's Guide*.

NightView is a graphical source-level debugging and monitoring tool specifically designed for real-time applications. NightView can monitor, debug, and patch multiple real-time processes running on multiple processors with minimal intrusion. For more information on NightView, see the *NightView User's Guide*.

NightProbe is a real-time graphical tool for monitoring, recording, and altering program data within one or more executing programs without intrusion. It can be used in a development environment as a tool for debugging, or in a production environment to create a "control panel" for program input and output. For more information on NightProbe, see the *NightProbe User's Guide*.

Understand for Ada is an optional interactive development environment that provides for reverse engineering, automatic documentation, code navigation and comprehension, metrics, maintenance and cross-referencing. While it is designed to assist engineers who have inherited large amounts of Ada legacy code or whose Ada projects have grown to immense size or complexity, it is extremely useful for small projects as well.

# 2
# Using MAXAda


Hello World - An Example . . . . . 2-1
Creating an environment . . . . . 2-1
Introducing units. . . . . 2-2
Defining a partition. . . . . 2-3
Building a partition . . . . . 2-4
Success!!! . . . . . 2-4
Let's look around.... . . . . 2-5
Listing the units in your environment . . . . . 2-5
Viewing the source for a particular unit . . . . . 2-5
Listing the partitions defined. . . . . 2-6
Looking at the Environment Search Path . . . . . 2-7
What are my options? . . . . . 2-7
Hello Galaxy - The Example Continues... . . . . 2-11
Setting up another environment . . . . . 2-11
Modifying an existing unit . . . . . 2-12
Building a unit with references outside the local environment . . . . . 2-13
Adding an environment to the Environment Search Path . . . . . 2-13
Making contact!!! . . . . . 2-14
Who resides here now?. . . . . 2-14
Hello Again... Ambiguous Units . . . . . 2-15
Resolving the ambiguity. . . . . 2-17
No more ambiguities!!! . . . . . 2-17

# 2
# Using MAXAda

## Hello World - An Example

To demonstrate the ease of use of MAXAda, a simple example will be given. This example will traverse through the core functions needed to build an executable under the MAXAda system.

Building an executable under MAXAda can be broken down into as few as four steps:

- Creating an environment
- Introducing units
- Defining a partition
- Building the partition

This section will demonstrate each of these steps on a simple, but well-known example - Hello World.

**Before we begin...**

You must make sure that the path **`/usr/ada/bin`** is added to your `PATH` environment variable. This is the only path necessary to access the MAXAda utilities, regardless of the number of releases of MAXAda installed on the system.

Also, check to make sure you have the correct capabilities set (see "Capabilities" on page 1-3).

## Creating an environment

One of the first steps you must take in order to use MAXAda is to create an *environment*. MAXAda uses environments as its basic structure of organization. Environments contain all the information relevant to a particular project. All of the MAXAda utilities work within the context of a particular environment.

The MAXAda tool used to create an environment is **`a.mkenv`**. It requires a directory where this environment will reside.

For our example, we will create a new directory on our system and run **a.mkenv** from within that directory.

```
$ mkdir /pathname/earth
$ cd /pathname/earth
$ a.mkenv
```

**Screen 2-1. Creating an environment**

This creates the MAXAda internal directory structure that comprises the environment and that is essential before any other MAXAda tools can be utilized. This environment has the same name as the directory in which it was created. Our environment in this example, therefore, is **/pathname/earth**.

## Introducing units

*Compilation units* (henceforth referred to simply as *units*) are the basic building blocks of MAXAda environments. It is through units that MAXAda performs most all its library management and compilation activities. These units are, however, introduced into the system in the form of *source files*.

In our example, we have one unit, `hello`, that resides in a source file, **world.a**. This source file is just an ordinary text file.

```
with ada.text_io ;
procedure hello is
begin
     ada.text_io.put_line ("Hello World!!!") ;
end hello ;
```

**Screen 2-2. Source file `world.a` containing `hello` unit**

Create this source file within the directory in which you created your environment. (It is not necessary for the source file to reside in the same directory as the environment. You may specify a relative or absolute path name of the source file.)

We introduce this unit to the environment by using the **a.intro** utility. **a.intro** introduces each unit contained in the source file into the current environment by default.

```
$ a.intro world.a
```

**Screen 2-3. Introducing units from a source file**

The unit `hello` that was contained in the source file **`world.a`** is now a part of the environment **`earth`**.

From this point on, the unit `hello` is considered to be *owned* by the environment **`earth`**. Any functions performed on this unit must be managed by the environment through the MAXAda utilities.

## Defining a partition

If we want to create an executable program to use our unit, we must define a *partition*. We will be creating an *active partition* which is the type that corresponds to executable programs.

We must also name the partition. You can name your partition anything you want and then add units to it, but since this is a simple example, we are taking the most direct route.

Hence, our partition will be named **`hello`**, the same as the unit which will also function as our *main subprogram* (which is the only unit in our example). We will use the MAXAda utility **`a.partition`** to do this.

```
$  a.partition -create active hello
```

**Screen 2-4. Defining a partition**

Because it has the same name as the active partition being created, the unit `hello` is automatically added to this partition and designated the main subprogram .

**NOTE**

In the case where the partition has the same name as a library subprogram in the environment, that subprogram is assumed to be the main subprogram. Otherwise, no main subprogram is assumed.

The command in Screen 2-4 could have been explicitly specified as:

```
$  a.partition -create active -add hello! -main hello hello
```

This command creates an active partition named **hello**, containing the main unit, hello and all units on which it depends. (The **!** of the **hello!** argument to the **-add** parameter signifies that all units on which the hello unit depends should also be added to the partition definition - e.g. ada.text_io). In addition, this command designates the unit hello to be the main subprogram as specified by the **-main** option.

## Building a partition

The last step now is to build the executable. All the necessary steps have been done. Just issue **a.build**. This will build an executable file that you can run. (See "Compile Options" on page 4-99)

```
$  a.build
```

**Screen 2-5. Building a partition**

Because no arguments were specified, **a.build** tries to build everything it can within this environment. Since we've only defined one unit, hello, contained in one partition, **hello**, it will only build that.

## Success!!!

Now all that's left is to run the program as you would any other executable program. Enter the name of the executable, in this case **hello**.

```
$  hello
Hello World!!!
$
```

**Screen 2-6. Executing the program**

And there you have it! Your program has successfully been built and run.

# Let's look around...

Now that we have some substance to our environment, let's take a look around and see what things look like. We can use some of the MAXAda utilities to investigate the state of our environment and what's in it.

## Listing the units in your environment

Something you might want to do is to see what units are contained within this environment. **`a.ls`** provides this list for you. **`a.ls`** provides many different options, allowing you to sort the list by some attribute or filter the units based on certain criteria. We'll just take a look at a basic list of the units in the environment. This is done by issuing the **`a.ls`** command with no options from within your current environment.

```
$  a.ls
hello
$
```

**Screen 2-7. Listing the units in an environment**

You may want to see more information. You can do this by specifying the **`-l`** option to the **`a.ls`** command which will give you a long listing including the unit's date, type, compilation state, part, and name. (Even more information can be seen by specifying the **`-v`** option.)

```
$  a.ls -l
    Unit_Date        Item       State    Part  Name
03/05/97 10:04:26  subprogram  compiled  body  hello
$
```

**Screen 2-8. Listing the units in an environment (`-l` option)**

## Viewing the source for a particular unit

Once you know what units are in your environment, you may want to see the source for a particular unit. The MAXAda utility **`a.cat`** outputs the source of a given program unit. It outputs a filename header for the source file by default, but this can be suppressed by specifying the option **`-h`**.

The following figure shows how to view the source for the unit `hello` using **`a.cat`**.

```
$  a.cat hello
*********** /pathname/earth/world.a ****************
with ada.text_io ;
procedure hello is
begin
     ada.text_io.put_line ("Hello World!!!") ;
end hello ;
$
```

**Screen 2-9. Viewing the source for a particular unit**

## Listing the partitions defined

You may also want to see what partitions have been defined for an environment. You may do this by using the **a.partition** command with either the **-list** or **-List** option.

```
$  a.partition -list
hello
$
```

**Screen 2-10. Listing the partitions in an environment (-list option)**

The **-List** option gives you more detailed information for each partition, including what kind it is and which unit is designated as the main subprogram.

In the following figure, you can see that we have created **hello** to be an active partition with hello designated as its main subprogram.

```
$  a.partition -List
PARTITION: hello
   kind                 : active
   output file          : hello
   link options         :
   dependent partitions :
   link rule            : object,archive,shared_object
   main subprogram      : hello
   included units (+)   :
     hello!
   excluded units (-)   :

$
```

**Screen 2-11. Listing the partitions in an environment (-List option)**

## Looking at the Environment Search Path

Each MAXAda environment has an Environment Search Path associated with it. The Environment Search Path is your gateway to other environments. Upon creation of your environment, MAXAda defines the Environment Search Path so that you have access to the Predefined Language Environment, as specified in Annex A of the Ada 95 Reference Manual.

If you take a look at your Environment Search Path, you will see the path to the **`predefined`** environment. You can list your Environment Search Path by using the **`a.path`** utility.

```
$  a.path
Environment search path:
        /usr/ada/rel_name/predefined
$
```

**Screen 2-12. Viewing your Environment Search Path**

As you can see, the only environment in your Environment Search Path is that of the predefined functions.

**NOTE**

The Environment Search Path was the mechanism that made `ada.text_io` visible to the unit **`hello`**.

Using the Environment Search Path, you can use units that exist in foreign environments. All you need to do is add the environment's path to your Environment Search Path. It's as simple as that!

## What are my options?

MAXAda uses the concept of persistent compile options. These options are specified through **`a.options`** and are "remembered" at compilation time. They can apply to any of three areas: environment-wide compile options (which apply to all units within the environment), permanent unit options and temporary unit options (both of which apply and are unique to specific units).

Let's manipulate the options in our example to give an idea of how it all works.

First, we will consider the environment-wide compile options. These apply to all the units within the environment. Since we only have one unit right now, it will apply to that. However, if we add any others later, they will "inherit" these options automatically.

The *environment-wide compile options* are referenced by the **`-default`** flag to **`a.options`**. We'll use the **`-list`** flag to display what they're set to now:

```
$  a.options -list -default
default options: /pathname/earth

$
```

**Screen 2-13. Listing the environment-wide compile options**

You'll see that nothing is listed. That's because we haven't set anything yet. So let's set them to something and see what happens.

**a.options** provides the **-set** option to initialize or reset an option group. Let's set our environment-wide compile option set to contain the options **-g** and **-O2**. (These set the debug level to `full` and set the optimization level to `GLOBAL`, respectively. You can find out all about these options in "Compile Options" on page 4-99.)

```
$  a.options -set -default -g -O2
```

**Screen 2-14. Setting the environment-wide compile options**

Now let's list them again to see if they've taken effect:

```
$  a.options -list -default
default options: /pathname/earth
   -O -g
$
```

**Screen 2-15. Listing the environment-wide compile options (after `-set`)**

We can see that the environment-wide compile option set now consists of **-O** and **-g**. (Note that **-O** and **-O2** are equivalent.)

Remember, these options apply to all units in the environment and will be "inherited" by any units we add to this environment.

If we'd like to set particular options for a specific unit, we can use the *permanent unit compile options* for that unit. They're set in much the same way as environment-wide options, except that we need to specify the units to which they apply.

Let's set the permanent options for the unit `hello` so it is compiled at a `MAXIMAL` optimization level (**-O3**). This is done with the following command:

```
$  a.options -set -perm -O3 hello
```

**Screen 2-16. Setting the permanent unit options for `hello` unit**

We may decide that in addition to the specified options, we may want to "try out" some options or change particular options for a specific compilation but only "temporarily". The *temporary unit compile options* are for this purpose.

Say we want to produce no debug information for our `hello` unit for this particular compilation. We can set a temporary compile option for that.

```
$  a.options -set -temp -!g hello
```

**Screen 2-17. Setting the temporary unit options for `hello` unit**

In addition, we remember that we also want to open the source file in the **`vi`** editor if any errors occur. We can "add" this to the temporary option set by using the **`-mod`** flag to **`a.options`**.

```
$  a.options -mod -temp -ev hello
```

**Screen 2-18. Modifying the temporary unit options for `hello` unit**

If we list the temporary options for the unit `hello`, we will see that we now have **`-!g`** and **`-ev`** in the temporary option set:

```
$  a.options -list -temp hello
Unit                    Temporary

subprogram body hello   -ev -!g
$
```

**Screen 2-19. Listing the temporary options for `hello` unit**

These three option sets have a hierarchical relationship to one another which means that the environment-wide compile options are overridden by the permanent unit options which are, in turn, overridden by the temporary unit options. This relationship forms the

*effective compile options* for the unit, which the compiler will use during compilation. We can see these in Table 2-1.

**Table 2-1. Effective options for `hello` unit**

| | | | |
|---|---|---|---|
| Environment-wide options | **-g** | **-O2** | |
| Permanent unit options | | **-O3** | |
| Temporary unit options | **-!g** | | **-ev** |
| EFFECTIVE OPTIONS | **-!g** | **-O3** | **-ev** |

If we list the effective options for the `hello` unit, we will see similar results:

```
$  a.options -eff hello
Unit                    Effective

subprogram body hello   -O3 -ev -!g
$
```

**Screen 2-20. Listing the effective options for `hello` unit**

If, after we compile with these options, we find any particular option that we would like to delete, we can do so by using the **-del** flag. For example, let's delete the error emmision option from the temporary options.

```
$  a.options -del -temp -ev hello
```

**Screen 2-21. Deleting from the temporary options set for `hello` unit**

And if we like the other temporary options so much that we'd like to make them permanent, MAXAda provides the **-keeptemp** flag to propagate all the temporary options for a particular unit to the permanent option set for that same unit. If we do this,

```
$  a.options -keeptemp hello
```

**Screen 2-22. Propagating the temporary options to the permanent set**

the temporary option **-!g** will become a permanent unit option for the unit `hello`.

The effective options will now resemble that of Table 2-2:

**Table 2-2. Effective options for `hello` unit (after `-keeptemp`)**

| | | | |
|---|---|---|---|
| Environment-wide options | **`-g`** | **`-O2`** | |
| Permanent unit options | **`-!g`** | **`-O3`** | |
| Temporary unit options | | | |
| EFFECTIVE OPTIONS | **`-!g`** | **`-O3`** | |

If we list the effective options for the `hello` unit, we will see similar results:

```
$  a.options -eff hello
Unit                    Effective

subprogram body hello   -O3 -!g
$
```

**Screen 2-23. Listing the effective options for `hello` unit (after `-keeptemp`)**

See “a.options” on page 4-58 for a complete description of the functionality of this MAX-Ada utility.

# Hello Galaxy - The Example Continues...

## Setting up another environment

Let’s set up another environment with a function that our `hello` unit can contact.

Let’s set up a new environment, **`galaxy`**, and introduce a source file very similar to **`world.a`**. We’ll call this file **`planet.a`** and it will contain the following unit, `alien`. The file is shown in Screen 2-24.

```
with ada.text_io;
procedure alien is
begin
  ada.text_io.put_line("Greetings from Outer Space!!!");
end alien;
```

**Screen 2-24. Source file `planet.a` containing `alien` unit**

Create a different directory `/pathname/galaxy` to contain our new environment and place the source file, **`planet.a`** in it. From within that directory, the following commands will create our environment and introduce the source file into it.

```
$  a.mkenv
$  a.intro planet.a
```

**Screen 2-25. Setting up another environment**

**NOTE**

We have not compiled this unit nor have we created a partition and included the unit in the partition to be built. This was intentional to demonstrate a point later in the example.

## Modifying an existing unit

Now we must go back to our original environment **`earth`** that contains our original unit `hello`.

We will update the unit `hello` so that it references the new `alien` unit. We do this by using the **`a.edit`** utility. **`a.edit`** edits the source file that contains the unit specified. It does this by using the editor referenced in the `EDITOR` environment variable. It then updates the environment so that the automatic compilation utility, **`a.build`**, knows that this unit needs to be rebuilt.

**NOTE**

**`a.edit`** is the supported method for modifying units that have been introduced into the environment. Any modifications to the units other than through the tools provided is discouraged, although the tools support it as well as possible.

Specify the unit name to the **`a.edit`** command.

```
$  a.edit hello
```

**Screen 2-26. Editing a unit**

Add the following lines to the `hello` unit.

```
with ada.text_io ;
with alien;
procedure hello is
begin
     ada.text_io.put_line ("Hello World!!!") ;
     alien;
end hello ;
```

**Screen 2-27. Reference the `alien` unit within the `hello` unit**

Save the changes to the file.

## Building a unit with references outside the local environment

Now let's try to build it.

Issue the **`a.build`** command as before.

```
$  a.build
a.build: error: required spec of alien does not exist in
   the environment
a.build: warning: subprogram body hello will not be
   built because required spec of alien does not exist
   in the environment
a.build: info: partition hello will not be built
   because required spec of alien does not exist in the
   environment
a.build: error: errors encountered during build

$
```

**Screen 2-28. Building the partition with reference to `alien` unit**

Because the `alien` unit does not exist in the current environment AND because we have not manually added it to our Environment Search Path, **`a.build`** cannot find it and therefore complains.

## Adding an environment to the Environment Search Path

This is easily remedied by adding the new environment's path to the Environment Search Path for the **`earth`** environment using the **`a.path`** utility.

```
$  a.path -a /pathname/galaxy
```

**Screen 2-29. Adding an environment to the Environment Search Path**

You can see that it has been added to your Environment Search Path by issuing the **a.path** command with no parameters again.

```
$  a.path
Environment search path:
       /usr/ada/rel_name/predefined
       /pathname/galaxy
$
```

**Screen 2-30. Viewing the updated Environment Search Path**

## Making contact!!!

Now try to issue **a.build** again. This time it will be successful.

After it is successfully built, run the **hello** executable again.

```
$  a.build
$  hello
Hello World!!!
Greetings from Outer Space!!!
$
```

**Screen 2-31. Executing the new `hello` - contact is made!**

## Who resides here now?

Let's take a look at who inhabits our environment **earth** now. Remember before when we issued the **a.ls** command, we saw that our environment contained the lone unit `hello`. Let's issue the command again and see what has happened since we made contact with the `alien`.

```
$  a.ls
alien hello
$
```

**Screen 2-32. Listing the units**

You can now see that the unit `alien` has been added to the list of units in this environment.

Although they are both listed *local* to this environment, they each have a different means of citizenship.

- The unit `hello` was introduced directly into this environment. Therefore, it is regarded as a *native* unit.
- The `alien` unit, however, was never formally introduced into the local environment. It was found on the Environment Search Path.

  Now, remember that the `alien` unit was not compiled in its original foreign environment. The **`a.build`** command, when run in this local environment, could not find a compiled form of the `alien` unit on the Environment Search Path and had to do something in order to build the partition. It therefore compiled the `alien` unit in the local environment.

  This compiled form of a foreign unit within the local environment is considered *naturalized* by the system.

**NOTE**

If the `alien` unit had been compiled in its own foreign environment, **`a.build`** would have found that compiled form on the Environment Search Path and would have used that when linking the **`hello`** executable together. In that case, an **`a.ls`** would have only shown the local unit `hello` as before.

**FURTHER NOTE**

The **`-noimport`** option will inhibit the automatic naturalization behavior of **`a.build`**. If it had been used in this example, **`a.build`** would have reported an error.

# Hello Again... Ambiguous Units

Let's see what happens when we introduce a unit having the same name as one already introduced into our environment.

We'll create a source file, **newunit.a**, in our **earth** environment containing a unit named hello:

```
with ada.text_io ;
procedure hello is
begin
     ada.text_io.put_line ("I am a new unit - Hello!!!") ;
end hello ;
```

**Screen 2-33. Source file newunit.a containing different hello unit**

Now, when we try to introduce the source file containing this unit into our environment, we will see an error message:

```
$  a.intro newunit.a
a.intro: error: body of unit "hello" already exists in
     another source file
$
```

**Screen 2-34. Introducing a unit that already exists in the environment**

This is because MAXAda provides a mechanism that detects the case where two versions of the same unit appear among all the source files owned by the environment.

Upon introducing a unit having the same name as a previously introduced unit, MAXAda labels both units as *ambiguous*. It will then refuse to perform any operations on either of the two versions, or on any units depending on the ambiguous unit.

For example, you will not be able to build the partition that contains this unit. If you try, you will get the following warning:

```
$  a.build
a.build: warning: subprogram body hello will not be
     built because it is ambiguous
$
```

**Screen 2-35. Building a unit that already exists in the environment**

The user will be forced to choose which of the two units should actually exist in the environment by "removing" the other.

## Resolving the ambiguity

The only option at this point is to remove the unit which doesn't belong. MAXAda provides the **`a.resolve`** tool specifically for this case.

**`a.resolve`** provides an option that allows you to list out the multiple sources of the ambiguous unit. Screen 2-36 shows this feature:

```
$  a.resolve -l hello
subprogram body hello is in:
        newunit.a
        world.a
$
```

**Screen 2-36. Listing the multiple source files for an ambiguous unit**

**`a.resolve`** allows you to "choose" which of the units you would like to remain in the environment. Let's choose the newer unit, `hello`, from the source file **`newunit.a`**.

```
$  a.resolve -r newunit.a hello
```

**Screen 2-37. Resolving the ambiguity**

## No more ambiguities!!!

Let's build again now that the ambiguities are resolved... and execute the file to see our results:

```
$  a.build
$  hello
I am a new unit - Hello!!!
$
```

**Screen 2-38. No more ambiguities!!!**

**NOTE**

While MAXAda is refusing to perform any operations on the ambiguous units, the compilation state of the original unit remains intact in the environment. This is useful in case the original unit is selected instead of the newly added one. If this is the case, the original unit (and all units dependent on it) would not have to be recompiled.

In our example, however, we have chosen the newly added unit, so the unit must be compiled in order for the partition to be built.

# 3
# MAXAda Concepts


Environments . . . 3-1
Local Environments . . . 3-2
Foreign Environments . . . 3-2
Environment Search Path . . . 3-2
Naturalization . . . 3-3
Fetching . . . 3-3
Supplied Environments . . . 3-3
NFS Environments . . . 3-3
Freezing Environments . . . 3-4
Restoring Environments . . . 3-4
Relocating Environments . . . 3-4
Environment-wide Compile Options . . . 3-6
Units . . . 3-7
Unit Identification . . . 3-7
Configuration Pragmas . . . 3-9
Nationalities . . . 3-9
Local Units . . . 3-9
Foreign Units . . . 3-10
Ambiguous Units . . . 3-10
Artificial Units . . . 3-11
Unit Compile Options . . . 3-11
Partitions . . . 3-12
Types of Partitions . . . 3-12
Active Partitions . . . 3-12
Archives . . . 3-12
Shared Objects . . . 3-13
Lazy Versus Immediate Binding . . . 3-13
Position Independent Code . . . 3-14
Share Path . . . 3-14
Shared Objects and Special MAXAda Packages . . . 3-14
Issues to consider . . . 3-15
Elaboration and Finalization Methods . . . 3-16
Elaboration Methods . . . 3-17
Finalization Methods . . . 3-18
Main Subprogram Requirements . . . 3-19
Exit Status . . . 3-19
Compilation and Program Generation . . . 3-20
Compilation . . . 3-20
Automatic Compilation Utility . . . 3-20
Compile Options . . . 3-20
Environment-wide Options . . . 3-21
Permanent Unit Options . . . 3-21
Temporary Unit Options . . . 3-21
Effective Options . . . 3-22
Compilation States . . . 3-22
Consistency . . . 3-23
Interoptimization . . . 3-24

Programming Hints and Caveats . . . . . 3-25
Compiler Error Messages . . . . . 3-26
Lexical Errors . . . . . 3-27
Syntax Errors . . . . . 3-28
Semantic Errors . . . . . 3-29
General Errors . . . . . 3-30
Informational Messages . . . . . 3-31
Warnings . . . . . 3-32
Alerts . . . . . 3-32
Fatal Errors . . . . . 3-33
Internal Errors and Panics . . . . . 3-33
Link Options . . . . . 3-34
Linking Executable Programs . . . . . 3-36
Linking Ada Programs with Shared Objects . . . . . 3-36
Debugging . . . . . 3-38
Real-Time Debugging . . . . . 3-38
Selecting a Debug Level . . . . . 3-38
Degree of Interest . . . . . 3-40
Debug Information and cprs . . . . . 3-40

# 3
# MAXAda Concepts

MAXAda uses the concept of *environments* as its basic structure of organization. These environments take advantage of various utilities provided by MAXAda to manipulate *compilation units* (referred to simply as *units*) that may form *partitions*.

Utilities for library management, compilation and program generation, and debugging are provided by MAXAda.

This chapter will discuss in further detail the concepts of environments, units and partitions and their relationship to library management, program generation, and debugging.

## Environments

MAXAda uses the concept of environments as its basic structure of organization. These environments are very closely related to *environments* as defined in the *Ada 95 Reference Manual*.

Environments may include:

- units that have been introduced
- partitions that have been defined
- Environment Search Paths
- references to source files (which generally contain units)
- other information used internally by MAXAda

Environments maintain *separate compilation information* collected from previous compilations.

There are different types of environments:

- *local environments* - see ("Local Environments" on page 3-2)
- *foreign environments* - see ("Foreign Environments" on page 3-2)
- *NFS environments* - see ("NFS Environments" on page 3-3)

MAXAda permits local environments to reference *foreign* environments thus providing visibility to the units and partitions therein. This feature allows programmers to work on local versions of individual program units while retrieving the remainder of the program from previously-developed environments.

A MAXAda environment may be initialized or created in any desired location in a filesystem using the **`a.mkenv`** utility which is discussed in "a.mkenv" on page 4-53.

MAXAda provides several other utilities to maintain, modify and report on the contents of environments. See “MAXAda Utilities” on page 1-1 to see a list of these tools.

**NOTE**

Any modifications to the environment other than through the tools provided by MAXAda is discouraged, although the tools support it as well as possible.

An environment may be frozen, making it unalterable (see “Freezing Environments” on page 3-4).

If an environment becomes damaged, MAXAda provides tools to help to correct the problem (see “Restoring Environments” on page 3-4).

MAXAda supports the relocation of environments to other locations in the filesystem hierarchy or even to other systems. Some advance planning may be required, however. See “Relocating Environments” on page 3-4 for some considerations to keep in mind.

## Local Environments

By default, MAXAda uses the current working directory as its *local environment*. All MAXAda utilities perform their actions within this local environment unless the **`-env`** option is explicitly specified.

For example, if no environment is specified with the **`a.mkenv`** tool, MAXAda will set up its internal directory structure for that environment within the current working directory.

When used with any of the MAXAda utilities, however, the **`-env`** option allows the user to specify a target environment other than the current working directory. The actions of the MAXAda utility using this option will be performed in the environment specified and not in the local environment. (See Chapter 4, “MAXAda Utilities” for more details on using this parameter with each of the tools.)

## Foreign Environments

MAXAda uses the Environment Search Path to reference units within foreign environments. These units can be used as foreign units or can be brought into the local environment through naturalization or fetching. MAXAda also provides a number of supplied environments.

### Environment Search Path

MAXAda uses the concept of an *Environment Search Path* to allow users to specify that units from environments other than the current environment should be made available in the current environment. This Environment Search Path relates only to each particular environment and each environment has its own Environment Search Path.

By placing the location of another environment on the *Environment Search Path* for a given environment, all the units from the other environment are conceptually added to the given environment, unless that would involve replacing a unit which was either introduced manually into the environment by a user, or would replace a unit which was introduced from yet a third environment which precedes the other environment in the Environment Search Path. In order to add or delete environments on your Environment Search Path, you may use the **`a.path`** tool. See "a.path" on page 4-74.

### Naturalization

At times, it is necessary for the compilation system to make local copies of units that exist in foreign environments. For example, if a foreign unit is referenced within a local unit and no compilation has been done on that foreign unit in that foreign environment, a local copy of the foreign unit will be compiled within the current environment, using any options that would apply to the foreign unit. These *naturalized* units take precedence over units that are in the Environment Search Path.

### Fetching

It may be desirable for users to force copies of specified units from other environments into the current environment. This eliminates any requirement that the unit be compiled in the foreign environment, so long as it is compiled locally. The **`a.fetch`** tool is provided for that purpose. Units that are fetched also take precedence over units that are in the Environment Search Path. See "a.fetch" on page 4-22.

### Supplied Environments

The *Ada 95 Reference Manual* states that certain units *must* exist in an environment. These units are shipped with MAXAda and the environment in which they exist (**`predefined`**) is automatically added to the *Environment Search Path* for the local environment when it is first created.

A number of other environments are supplied with MAXAda. See Chapter 9, "Support Packages" for a complete discussion of these environments.

## NFS Environments

MAXAda supports the creation and use of environments on NFS-mounted filesystems only to a limited extent. This is because NFS caches make it difficult to guarantee file consistency when an environment is being modified by two or more systems nearly simultaneously. The limitations are designed to avoid problems caused by those deficiencies in the NFS model. They are:

- Modification operations (e.g. **`a.compile`**) can only be performed on an environment from the system that is that environment's "owner".
    - If an environment is created on a local (non-NFS) filesystem, then the environment's owner is its local system. If the environment is

moved to another filesystem on a different system, the environment's owner is its new local system.

- If an environment is created on an NFS-mounted filesystem, then the environment's owner is the system which created the environment. Note that this means that the environment cannot be modified even on the system on which it is local. If the environment is moved to another filesystem on a different system, then its owner is still the system which created the environment.

- Read-only operations (e.g. **`a.ls`**) can always be performed from any system.

In addition, MAXAda provides a new utility to display or change the NFS aspects of an environment. See "a.nfs" on page 4-56 for more information.

## Freezing Environments

An environment may be frozen using the **`a.freeze`** utility. This changes an environment so that it is unalterable.

A frozen environment is able to provide more information about its contents than one that is not frozen. Therefore, accesses to frozen environments from other environments function much faster than accesses to unfrozen environments.

Any environment which will not be changed for a significant period of time and which will be used by other environments is a good candidate to be frozen to improve compilation performance.

See "a.freeze" on page 4-25 for information on this utility.

## Restoring Environments

In rare instances, an environment may become damaged (e.g. through a system crash or power failure). In these cases, **`a.restore`** may be used to correct problems with the environment as much as possible.

MAXAda stores "backup" information about the environment internally which can be used to restore a damaged environment.

See "a.restore" on page 4-86 for information on this utility.

## Relocating Environments

Although there is no MAXAda-defined mechanism for physically moving an environment, MAXAda supports the relocation of environments to other locations in the filesystem hierarchy or even to other systems. This may be done using commands similar to the following:

```
cd old-location
find . -depth -print | cpio -pdmu new-location
```

where ***old-location*** is the directory of the original environment and ***new-location*** is the directory where the environment is to be relocated.

Some advance planning may be required, however.

MAXAda preserves pathnames as specified by the user. If a relative pathname is specified, MAXAda stores it as that relative pathname. Likewise, if an absolute pathname is specified, MAXAda stores it as that absolute pathname.

**NOTE**

If the **-env** option is used to specify an environment other than the current directory, MAXAda must alter any relative pathnames to be relative to the environment so that future MAXAda tool invocations from different current working directories function properly. Still, MAXAda will attempt to keep the relative pathname relative.

The basic rule for environment relocation is that all pathnames specified to the environment must make sense in both the original and relocated locations. If not, then the tools will most likely issue fatal errors because they will be unable to find source files or environments. (See "Fatal Errors" on page 3-33 for more information.)

The MAXAda utilities which specify pathnames to the environment are:

**a.path**
**a.fetch -from ...**

Both tool invocations provide the locations of foreign environments.

If planning to move a group of environments en masse, it would be appropriate to specify those foreign environments with relative pathnames, assuming that the relative pathnames would remain meaningful in the relocated locations. Any other required environments that would not be moved probably should be specified with absolute pathnames (or keywords in the case of the MAXAda-supplied environments - see Chapter 9 for a list of these keywords).

(See "a.path" on page 4-74 and "a.fetch" on page 4-22 for more information.)

**a.intro**

If planning to move the source files along with the environment, then they should be specified with relative pathnames (or simple file names if they are in the same directory as the environment), assuming that those relative pathnames would remain meaningful in the relocated locations. If the source files are located in a fixed position without regard to the location, though, they probably should be specified with absolute pathnames.

(See "a.intro" on page 4-30 for more information.)

**a.partition -o ...**

This pathname is less likely to cause a tool to fail, because it designates the location at which the output file will be created, as opposed to a location where some pre-existing file must be found. So, the only failure that could occur in a relocated environment would be if the output file specified a directory which did not exist, or was otherwise unwritable.

Regardless, the same care should be taken so that the output file will be created where it is expected. Use relative pathnames if the partition's output file location should be relocated along with the environment. Use absolute pathnames if the output file location should be at a fixed location regardless of the location of the environment.

(See "a.partition" on page 4-62 for more information.)

MAXAda does provide the **`a.script`** tool to achieve something like an environment relocation. The major difference is that in the new version, nothing is built. The same considerations with regard to absolute/relative pathnames that straightforward **`cpio(1)`** copies have must be taken into account when using **`a.script`**: all the foreign environment, source file, and partition output file paths have to be meaningful both in the original environment (the one on which **`a.script`** was run) and in the new one (the one created by the script generated by **`a.script`**). (See "a.script" on page 4-89 for more information.)

## Environment-wide Compile Options

Environment-wide compile options apply to all units within an environment. They are described in detail on page 3-21.

# Units

Compilation units (or simply units) are the basic building blocks of MAXAda environments. Instead of dealing with source files for library management and compilation activities, MAXAda focuses on the concept of units from the Ada 95 Reference Manual (10.1). According to the Reference Manual, a *compilation unit* can be a

- Subprogram declaration
- Package declaration
- Generic declaration
- Generic instantiation
- Library unit body
  - subprogram body or package body
- Subunit
  - subprogram body, package body, task body, or protected unit body
- Configuration pragma

## Unit Identification

For many of the MAXAda utilities in Chapter 4, the following definition is given:

*unit-id* is defined by the following syntax:

*unit***[/***part***] | all[/***part***]**

where *part* is the **specification**, **body**, or **all**; abbreviations are accepted.

Units are identified by their name and by their *part*. The part can be specified as:

- **specification**
- **body**
- all

**NOTE**

Abbreviations are most commonly used when specifying the part. For instance, instead of writing out the full unit-id:

**foo/specification**

it is much simpler to use the abbreviated form:

**foo/s**

*Specifications* can be

- subprogram declarations (including renaming declarations)
- package declarations (including renaming declarations)
- generic declarations (including renaming declarations)
- generic instantiations

*Bodies* can be

- subprogram bodies
- package bodies
- subunits

When **all** is specified as the *part*, it refers to the specification and the body.

For most MAXAda utilities, part, if unspecified, defaults to **body**.

Consider the following specification:

```
package tax_options is
   itemize : boolean := FALSE;
end tax_options;
```

**Figure 3-1. Package specification**

After this unit has been introduced, the **/specification** (or **/s**) suffix must be specified in order to edit it:

```
$  a.edit tax_options/s
```

If nothing is specified, *part* defaults to **body** and the following error message is issued:

```
$ a.edit tax_options
a.edit: fatal: Body of unit "tax_options" could not be
located
$
```

A keyword that can be used in place of a *unit* name is **all**. When used alone as the unit-id for most MAXAda utilities, **all** implies all units within the environment. **all** takes the same part options as any other unit-id. For example,

```
$  a.ls all/s
```

lists the specifications of all of the units within the current environment.

However, in the absence of a part, **all** indicates all units in the environment, not just bodies. That is, **all** is equivalent to **all/all**.

## Configuration Pragmas

Configuration pragmas are syntactical entities that are not part of a unit. Configuration pragmas can appear either at the beginning of a source file containing library units or independently in a source file with no units.

If the configuration pragmas appear independently in a source file with no units, they are considered to be *independent configuration pragmas*. When independent configuration pragmas are first compiled, they must be remembered and are applied to any future compilations in the environment. These are handled automatically by the **`a.build`** tool, but there are restrictions. Independent configuration pragmas may only be compiled when all units local to the environment are either `uncompiled` or inconsistent. (See "Consistency" on page 3-23) If independent configuration pragmas are added to an environment with `compiled` units, **`a.build`** will generate error messages. The user may ignore them or may invalidate all the units in the environment to force the independent configuration pragmas to take effect. See "a.invalid" on page 4-32 for more information on invalidating units.

If the configuration pragma is in a source file with library units, the configuration pragmas must precede those units in the file. They then apply only to those units sharing the same source file. This is handled automatically by **`a.build`**, and there are no particular restrictions.

See "RM Annex L: Pragmas" on page M-104 for a complete listing of pragmas supported by MAXAda.

## Nationalities

Compilation units in MAXAda have a nationality associated with them. Units can be either *local* or *foreign*.

### Local Units

Compilation units that are *local* to a system can be one of three types:

*native*

Native compilation units are introduced into an environment by using the **`a.intro`** function.

Once a unit is introduced into an environment, it is considered to be owned by that environment and any functions performed on that unit should be managed by the environment through the MAXAda utilities.

*naturalized*

Sometimes, the compiled form of a foreign unit is not available when it is needed locally for a build. In this case, the system automatically makes a local compilation. This local compiled form is considered to be naturalized.

A naturalized unit retains the compile options from its original environment. These options can only be altered by changing them in the original environment.

*fetched*

In some cases, it may be desirable for users to manually fetch the compiled form of a unit from another environment into the local environment. This may be necessary to avoid *obscurities*, but this is rarely required. (See "a.fetch" on page 4-22 for an example of fetching a unit to avoid obscurities.)

A fetched unit retains the unit-specific options from the original unit but these options may be changed in the local environment. However, it does not retain the environment-wide options of its original environment. It uses those of the current environment instead.

Naturalized or fetched units must be *expelled* from the environment by using **`a.expel`** if they are no longer desired.

### Foreign Units

*Foreign units* are those units that exist in other environments which are on the Environment Search Path. The user is not required to do anything special in order to use these units. They become automatically available once their environment is added to the Environment Search Path.

## Ambiguous Units

MAXAda provides a mechanism that detects the case where two versions of the same unit appear among all the source files introduced to the environment.

Upon introducing a unit having the same name as a previously introduced unit, MAXAda labels both units as *ambiguous*. It will then refuse to perform any operations on either of the two versions, or on any units depending on the ambiguous unit. The user will be forced to choose which of the two units should actually exist in the environment by "removing" the other.

MAXAda provides the **`a.resolve`** tool to select the desired unit. See "Hello Again... Ambiguous Units" on page 2-15 for an example of this situation.

**NOTE**

The **`a.hide`** utility (see page 4-27) may also be used to remove the ambiguous unit but it is usually simpler to use the **`a.resolve`** tool.

Another way of possibly removing an ambiguity is to use **`a.rmsrc`** (see page 4-88). However, this will also remove other units contained in that source file from the environment, which may not be what the user intended.

While MAXAda is refusing to perform any operations on the ambiguous units, the compilation state of the original unit remains intact in the environment. This is useful in case the original unit is selected instead of the newly added one. If this is the case, the original unit (and all units dependent on it) would not have to be recompiled.

## Artificial Units

At times, the implementation may create units to fill internal roles such as bodies of instances. These units are created, utilized, and sometimes discarded during the compilation phase. The user may use the **`-art`** option to **`a.ls`** to display the artifical units in the environment. See "a.ls" on page 4-35 for more information.

## Unit Compile Options

Each unit has a set of permanent and temporary compile options associated with it. These compile options are described in detail on page 3-21.

# Partitions

A *partition* is an executable, archive, or shared object that can be invoked outside of MAXAda. Partitions consist of one or more units that have been introduced into the environment. The units included in a partition are those that the user explictly assigns and units which they require. MAXAda manages these units and their dependencies, as well as link options and configuration information for each partition within the context of an *environment*. A partition definition must include one or more units in order to be built.

A more complete definition of *partition* can be found in 10.2(2) of the *Ada 95 Reference Manual*.

A partition within MAXAda is created and maintained by using the **`a.partition`** function. This function provides tools to create a partition, add or delete units from a partition, designate a main unit for the partition, and various other utilities.

In much the same way that options and configuration information concerning compilation are associated with units, linker options and configuration information for linking are associated with partitions. Partitions are basically recipes to the linker which indicate how to build a target file from units.

## Types of Partitions

MAXAda defines three types of partitions:

- Active Partitions
- Archives
- Shared Objects

### Active Partitions

The simplest form of partition is the active partition which describes how to build an executable program. This corresponds to the *active partition* defined in Section 10.2 of the *Ada 95 Reference Manual*.

### Archives

An *archive* is a collection of routines and data that is associated with an application during the link phase. Archives are useful for linking into other, potentially non-Ada, applications. Archives are usually designated with a **`.a`** suffix.

Archives differ from shared objects by the form of the object contained within it. Archives contain statically-built (i.e. non-shared) objects within them. (See “Position Independent Code” on page 3-14 for more details)

Because archives are non-active partitions, they may set elaboration and finalization methods using the **`-elab`** and **`-final`** options to **`a.partition`**. (See “Elaboration and

Finalization Methods" on page 3-16 for more details.) For the same reasons, they may not set a main subprogram using the **-main** option to **a.partition**.

## Shared Objects

**NOTE**

Currently MAXAda for RedHawk Linux only supports statically linked Ada code; however, you can link with system shared libraries. Support for Ada shared objects is anticipated in a future release.

A *shared object* is a collection of routines and data that is associated with an application during the link and execution phases. Shared objects are useful for linking into other Ada or non-Ada applications. Shared objects are usually designated with a **.so** suffix.

Shared objects differ from archives by the form of the object contained within it. Shared objects are dynamically built (i.e. shared) objects that contain position independent code. (See "Position Independent Code" on page 3-14 for more details)

At link time, routines and data objects from a shared object may satisfy unresolved references from an application, but they are not copied into the resultant application's executable image. The actual associations and memory allocations occur during the initial phase of the application's execution; this is termed the *dynamic linking* phase. Because of this, it is possible for shared objects to be changed and these changes to affect the application that has linked with them. However, due to this dynamic linking property of shared objects, it is often not necessary to rebuild the calling application after the shared object has changed.

During dynamic linking, all shared objects that the application requires are allocated and linked into the application's address space, sharing as many physical memory pages with other concurrently executing applications as possible. Therefore, totally dissimilar applications may share the same physical pages for the same shared object. This applies to the memory for the actual code or machine instructions in the shared object. The memory for the data segments in a shared object is usually replicated for each application using that shared object.

Because shared objects are non-active partitions, they may set elaboration and finalization methods using the **-elab** and **-final** options to **a.partition**. (See "Elaboration and Finalization Methods" on page 3-16 for more details.) For the same reasons, they may not set a main subprogram using the **-main** option to **a.partition**.

### Lazy Versus Immediate Binding

After the dynamic linker successfully locates all of the shared objects required for the application program, it maps their memory segments into the application program's address space.

The dynamic linker uses internal symbol tables to satisfy symbol references in the application program. Entries in these tables describe the final location of symbols found in the shared objects; this is termed *relocation*. All data references are immediately relocated.

By default, the dynamic linker does not fully relocate all subprogram references in the application program (or the shared objects themselves, because they can reference other shared objects or routines in the application program). If an as-yet unrelocated reference occurs, control passes once again to the dynamic linker which then relocates the reference. This is termed *lazy binding*.

To force immediate binding of all references, the user may invoke the program with the `LD_BIND_NOW` environment variable set. See *Compilation Systems Volume 1 (Tools)* for more information.

### Position Independent Code

In order to create a shared object, the compiler must generate code in a position-independent manner. *Position independence* refers to the fact that the generated code cannot rely on labels, data, or routines being in known locations; these locations are not known until dynamic linking occurs. *Position independent code* (PIC) requires additional indirections at run-time; therefore, routines within shared objects are inherently slightly slower than non-shared versions of those routines.

You control whether a unit is compiled as position independent code via a compilation setting called *share mode*. When the share mode is set to `shared` or `both`, compilations are performed generating position independent code. Units with this share mode must be included in a shared object to be used . They cannot be statically linked.

See “Share Mode (`-sm`)” on page 4-101 for more details on share modes.

### Share Path

Because the actual association of a shared object with a user application does not occur until execution time, the shared object must exist on the target system in a specific location, configurable by the user. By default, the path name of the shared object is that defined by the target of the partition.

When creating a partition, you may specify an alternative path name (or *share path*) for the shared object. The shared object will still be built at the pathname specified for the target, but it must be placed at the share path before any executables using it can be run. Alternatively, a soft link can be created by using the **`-sl`** option to the **`a.partition`** command when defining the shared object.

See “Share Path” on page 4-110 for more details.

### Shared Objects and Special MAXAda Packages

When linking with MAXAda shared objects, it is possible that certain packages specially recognized by the MAXAda run-time library may be quietly linked with user programs, even if not specified in a `with` clause in the user’s source code.

MAXAda associates the specification and bodies of a package using externally visible symbol names, instead of strictly using the dependency information as calculated by the `with` clauses starting at the main unit of the user program.

This package is:

- Package `default_handler` in **`vendorlib`**

If the user were to supply his own copy of this package and compile it in shared mode, all programs that use that shared object would use the new version, even if the main unit (or any of the main unit's dependents) do not specify the package in a `with` clause.

For example, if the user supplied a body to `default_handler` that printed the associated program counter register value with an otherwise unhandled exception, all programs using the shared object that contained the user's copy of the `default_handler` package body would exhibit the same behavior.

### Issues to consider

While the use of shared objects almost always reduces disk space utilization on the target architecture and often improves development productivity by minimizing application link time, it may or may not actually improve run-time memory utilization. The following issues should be considered.

1. Are the shared objects configured with an appropriate *granularity* (i.e. the number of Ada units located in each shared object) with respect to the particular client application programs that will be concurrently executing?

   For example, it is possible that if only two application programs concurrently execute and use large granular shared objects, more memory may potentially be used than in a non-shared object scenario. There is a trade-off between small granularity and manageability.

2. Will the application make use of local memory, and if so, how many applications will be executing out of the same local memory pools using the same shared object?

3. What disk storage capacity does the system have? The difference in size between ordinary objects and PIC objects is negligible. However, note that when choosing share mode `both`, the disk storage requirement for the object files in the environment is approximately doubled.

4. What time constraints are there? The share mode `both` effectively doubles the amount of time required for the code generation phase of compilation because it is executed twice: one time to generate the code for the ordinary object, and one time for the PIC object.

# Elaboration and Finalization Methods

Elaboration and finalization are taken care of in active Ada partitions for all archive and shared object partitions included via the link rule (see “Link Rule” on page 4-67) or dependent partitions list (see the **`-add`** option to **`a.partition`** on page 4-62). In all other cases, (for example, calling an Ada subprogram from within C++ code, or using a routine that exists in an archive that hasn't been included in the active partition), this must be done explicitly.

The elaboration and finalization routines do have an effect the second and subsequent times they are called. This is contrary to the advice in RM B.1(39) (see page M-76), but permits the Ada code to be elaborated and finalized multiple times. This is useful if used in a foreign language subsystem where the designers of that subsystem do not know how many times the subsystem will be initialized and finalized. In such a case, Ada elaboration and finalization can be performed multiple times without worry.

The elaboration routine should never be called multiple times without an intervening call to the finalization routine. The results from such actions would be unpredictable.

Also, multiple Ada partitions can be elaborated, used, and finalized within a foreign language program so long as the user is careful to ensure that no two partitions ever contain the same unit. In such a case, link errors of the following form could occur:

```
ld: .../partition1(ELAB_partition1): fatal error: symbol
`symbol_name` multiply-defined, also in file .../
partition2(ELAB_partition2)
```

Even though link errors of this form may not always occur, the use of two partitions which contain the same unit could result in the unit being elaborated or finalized twice, which could produce unpredictable results.

Elaboration and finalization methods are specified by using the **`-elab`** and **`-final`** options, respectively, to **`a.partition`**. (See “a.partition” on page 4-62.)

### NOTE to Fortran Users

For active Ada partitions that make interface calls to Fortran, calls to the `f_init` and `f_exit` routines in the Fortran library are made automatically to ensure that Fortran I/O works correctly. For archive or shared object Ada partitions that make interface calls to Fortran, neither `f_init` nor `f_exit` is called. This is so that they will not interfere with the calls automatically made to those routines when the main program of an executable is Fortran. However, this means that when Fortran code is called from an archive or shared object Ada partition which is, in turn, called from a non-Ada, non-Fortran main program, the user must arrange to call `f_init` before using the Ada partition and `f_exit` afterward.

## Elaboration Methods

Elaboration methods are specified by using the **`-elab`** option to **`a.partition`**. (See "a.partition" on page 4-62.)

MAXAda provides three methods for *elaboration*:

- `none`

  This is the default. Nothing will be done for elaboration. This is generally not recommended for partitions used outside the Ada development environment, but may be useful for partitions containing only pure and preelaborated units.

- `auto`

  An elaboration routine is generated at link time and is called before the main subprogram even runs. The user does not need to be concerned about the routine itself or calling it. Elaboration is handled automatically when this option is specified.

  This option is not available for archives.

  **NOTE**

  This option should not be used for partitions that will be included via the link rule (see "Link Rule" on page 4-67) or dependent partitions list (see the **`-add`** option to **`a.partition`** on page 4-62) in active Ada partitions because the automatic elaboration will interfere with the elaboration for the active Ada partition.

- `user,`*routine_name*

  An elaboration routine named *routine_name* is generated at link time. The user specifies the actual name for *routine_name* and makes a call to this routine at some point in the foreign language source. The actual call to this elaboration routine should be made before any Ada code is called.

  This option may be used for partitions that will be included both via the link rule (see "Link Rule" on page 4-67) or dependent partitions list (see the **`-add`** option to **`a.partition`** on page 4-62) in active Ada partitions and in foreign language partitions.

  **NOTE**

  If this option is used, *routine_name* should not be called for partitions that will be included via the link rule (see "Link Rule" on page 4-67) or dependent partitions list (see the **`-add`** option to **`a.partition`** on page 4-62) in active Ada partitions because the elaboration performed by *routine_name* will interfere with the elaboration for the active Ada partition.

See "Elaboration and Finalization Methods" on page 3-16 for more information.

## Finalization Methods

Finalization methods are specified by using the **-final** option to **a.partition**. (See "a.partition" on page 4-62.)

MAXAda provides the same three methods for *finalization*:

- `none`

  This is the default. Nothing will be done for finalization. This is generally not recommended for partitions used outside the Ada development environment, but may be useful for partitions containing only pure and preelaborated units.

- `auto`

  A finalization routine is generated at link time and is called after the main subprogram runs. The user does not need to be concerned about the routine itself or calling it. Finalization is handled automatically when this option is specified.

  This option is not available for archives.

**NOTE**

> This option should not be used for partitions that will be included via the link rule (see "Link Rule" on page 4-67) or dependent partitions list (see the **-add** option to **a.partition** on page 4-62) in active Ada partitions because the automatic finalization will interfere with the finalization for the active Ada partition.

- `user,`*routine_name*

  A finalization routine named *routine_name* is generated at link time. The user specifies the actual name for *routine_name* and makes a call to this routine at some point in the foreign language source. The actual call to this finalization routine should be made after all Ada code is called.

  This option may be used for partitions that will be included both via the link rule (see "Link Rule" on page 4-67) or dependent partitions list (see the **-add** option to **a.partition** on page 4-62) in active Ada partitions and in foreign language partitions.

**NOTE**

> If this option is used, *routine_name* should not be called for partitions that will be included via the link rule (see "Link Rule" on page 4-67) or dependent partitions list (see the **-add** option to **a.partition** on page 4-62) in active Ada partitions because the finalization performed by *routine_name* will interfere with the finalization for the active Ada partition.

See "Elaboration and Finalization Methods" on page 3-16 for more information.

# Main Subprogram Requirements

A main subprogram must be a non-generic library subprogram without parameters that is either a procedure or a function returning `STANDARD.INTEGER` (predefined type).

## Exit Status

Upon program termination, the exit status is determined by the first applicable following rule:

- If the `Ada.Command_Line.Set_Exit_Status` procedure was called, the program's exit status is the last value used in a call to this procedure.
- If the main subprogram propagated an (unhandled) exception to the environment task, the exit status is the value 42, as required by the POSIX 1003.5 standard.
- If the main subprogram was a procedure which returned normally, the exit status is `Ada.Command_Line.Success`, which is the value 0.
- If the main subprogram was a function which returned normally, the exit status is the result of the call to that main subprogram.

# Compilation and Program Generation

The compiler operates in several distinct phases, designed to satisfy the needs of the entire software development process. These phases include:

- Determination of compilation unit dependencies
- Syntax checking
- Semantic checking
- Code generation and optimization
- Instruction scheduling
- Machine-code assembly

Various options can be specified with the **`a.options`** command in order to control compilation phases. For example, during preliminary software development, it is often useful to limit the compilation phases to syntax and semantic checking. Errors from these phases can be brought up into a text editor automatically for fast, iterative editing and compiling.

## Compilation

MAXAda uses an Ada compiler that partially supports the Ada language specification as defined in the Ada 95 Reference Manual.

### Automatic Compilation Utility

MAXAda provides **`a.build`** for automatic compilation and program generation. **`a.build`** calls various internal tools to create an executable image of the program. See “a.build” on page 4-3 for more information.

### Compile Options

Unlike most compilation systems, MAXAda uses the concept of *persistent options*. These options do not need to be specified on the command line for each compilation. Rather, they are stored as part of the environment or as part of an individual unit’s information. These options are “remembered” when the MAXAda compilation tools are used.

There are three “levels” of compilation options:

- Environment-wide options
- Permanent unit options
- Temporary unit options

These levels have a hierarchical relationship to one another. Environment-wide options can be overridden by permanent unit options which can be overridden by temporary unit options. The set of *effective options* for a unit are that unit’s sum total of these three option

sets, with respect to this hierarchical relationship. See "Effective Options" on page 3-22 for more information.

See "Compile Options" on page 4-99 for a list of options that may be specified.

### Environment-wide Options

*Environment-wide options* apply to all units within that environment. All compilations within this environment then observe these environment-wide options unless overridden.

Environment-wide options can be overridden by

- individual unit compile options (permanent or temporary - see below)
- command-line options (which change temporary options on a unit)
- pragmas in the source of the units themselves

See "Compile Options" on page 3-20 for more information.

### Permanent Unit Options

Each unit has its own set of options permanently associated with it that override those specified for the environment. They may be specified and later modified via the **`a.options`** utility.

See "Compile Options" on page 3-20 for more information.

See the description of "a.options" on page 4-58 for more details.

### Temporary Unit Options

Each unit also has a set of options that may be temporarily associated with it that override those that are permanently associated with it.

- If a unit is manually compiled (using **`a.compile`** - see page 4-9) with any specified options, these are added to its set of temporary options.
- The temporary options may also be set using the **`a.options`** tool.

Temporary options allow users to "try out" options under consideration. By designating these options as "temporary", the user can first see the effect these options have and then decide if this is what is desired. If so, MAXAda provides a way to add these temporary options to the set of permanent options for that unit using **`a.options`**. If these options are not what the user desires, **`a.options`** also provides a way to eliminate all temporary options from a unit (or from all units in the environment).

Another case in which temporary options might also prove useful is one in which a unit needs to be compiled with debug information. If this is not the manner in which the unit is normally compiled, a temporary option can be set for that unit to be compiled with debug information. When the debug information is no longer needed, the temporary option can be removed and the unit can be recompiled in its usual manner.

See "Compile Options" on page 3-20 for more information.

See the description of "a.options" on page 4-58 for more details.

### Effective Options

These levels have a hierarchical relationship to one another. Environment-wide options can be overridden by permanent unit options which can be overridden by temporary unit options. The set of *effective options* for a unit are that unit's sum total of these three option sets, with respect to this hierarchical relationship. Table 3-1 shows an example of a unit's effective options based on the relationship between its environment-wide options, permanent unit options, and temporary unit options.

**Table 3-1. Effective options based on hierarchical relationship**

| | | | | |
|---|---|---|---|---|
| Environment-wide options | | **-g** | **-O2** | **-ee** |
| Permanent unit options | **-!S** | | **-O3** | |
| Temporary unit options | **-S** | **-!g** | | |
| EFFECTIVE OPTIONS | **-S** | **-!g** | **-O3** | **-ee** |

As shown in this example, compilation options can be negated by preceding the option with the "**!**" symbol. Therefore, the option "**-!g**" means no debug information should be generated for this unit. Because it is a temporary option for only this particular unit, all other units in the environment will be compiled with debug information (due to the "**-g**" environment-wide option listed in the example).

See "Compile Options" on page 3-20 for more information.

In addition, see "Compile Options" on page 4-99 and "Qualifier Keywords (`-Q` options)" on page 4-105 for a list of available compilation options.

## Compilation States

Units in the environment can be in any of several different compilation states:

- `uncompiled`

  The state of a newly-introduced unit, or one that has been invalidated. The environment is aware of the unit and some basic dependency information but very little else.

- `parsed`

  In this state, some semantic information about the unit has been generated. There is a complete picture of the meaning of the unit, but none of the actual implementation.

- `drafted`

  All semantic information has been produced, but no actual object files have been created.

- `compiled`

  Object files have been generated for the unit

The benefit of having this information generated at each of these states for each unit in the environment is that it allows the compilation utility to use this information to produce bet-

ter code in the unit currently being compiled. (See "Interoptimization" on page 3-24 for more information.)

**a.build** allows the user to compile units to a specified state using the **-state** option, however, `compiled` is the only valid state allowed for this option in the current release. See "a.build" on page 4-3 for more information.

**NOTE**

Only the `uncompiled` and `compiled` states are available at this time. These states are documented because they are visible in such utilities as **a.build**, **a.compile**, and **a.ls**.

## Consistency

Along with compilation states comes the idea of *consistency.* Each unit is considered consistent up to a particular state. This means that it is valid *up to that state of compilation*. Any recompilation of the unit can start from that state. It does not need to go through the earlier stages of recompilation.

Modification of a unit may possibly change its consistency. Modifications include:

- changes to the source file itself
- changes to any of the options
- changes to any required units upon which this unit depends

For example, if the source of a unit has been modified since it was last compiled, the semantics of the unit are potentially changed. New semantic information about the unit must be generated. Therefore, it is considered "consistent up to the `uncompiled` state". This means that when it is recompiled, it must start at the inconsistent state, `uncompiled`.

Not all changes to a unit make it "consistent up to the `uncompiled` state". Changing the options on a unit may not affect the syntax or semantics of a unit and therefore do not require a total recompilation.

Each option, in fact, has *relevance*, that is, how "inconsistent" a unit becomes if this option is changed. Table 3-2 lists the relevance for each option.

**Table 3-2. Relevance of Options**

| Option | Relevance |
|---|---|
| **-e** | `compiled` |
| **-g** | `drafted` |
| **-N** | `uncompiled` |
| **-opp** | `parsed` |
| **-O** | `parsed` |

**Table 3-2. Relevance of Options**

| Option | Relevance |
|---|---|
| **-i** | compiled |
| **-w** | compiled |
| **-sm** | drafted |
| **-S** | drafted |
| **-Qinline_line_count** | parsed |
| **-Qinline_nesting_depth** | parsed |
| **-Qinlines_per_compilation** | parsed |
| **-Qinline_statement_limit** | parsed |
| **-Qinteresting** | drafted |
| **-Qopt_class** | drafted |
| **-Qoptimize_for_space** | drafted |
| **-Qoptimization_size_limit** | drafted |
| **-Qobjects** | drafted |
| **-Qloops** | drafted |
| **-Qunroll_limit** | drafted |
| **-Qgrowth_limit** | drafted |
| **-Qwiden_trees** | drafted |
| **-Qtarget** | drafted |
| **-Qdb_basic_block** | compiled |
| **-Qdb_region** | compiled |
| **-Qdb_routine** | compiled |
| **-Q** | drafted |

For example, if only the **debug_level** option on a unit is changed, the syntax and semantics of the unit will not be affected. Therefore, it is not necessary to go through the parsed or drafted states again since nothing will change. However, the object files that will be generated for this unit will change so the unit is considered "consistent up to the drafted state".

For example, if only the **-e** option on a unit is changed, the syntax, semantics, and resultant object file of the unit will not be affected. In this case, it is not necessary to recompile the unit at all. Therefore, the unit is considered "consistent up to the compiled state".

## Interoptimization

MAXAda provides a method of optimization that controls the compilation order such that all language-dependence rules are obeyed.

There are currently two levels of interoptimization available:

| | |
|---|---|
| `0` (none) | no effort to attain interoptimization |
| `1` (inlining) | better ordering of compilation of units such that inlined subprogram calls will be performed whenever possible |

See the **`-IO`** option of "a.build" on page 4-3 for using this option with the compilation utility. Further information can be found by referring to "Inline Dependencies" on page 4-6.

## Programming Hints and Caveats

In general, programs that are to be debugged with NightView should not be optimized, although they may be interoptimized. Optimization levels `GLOBAL` and `MAXIMAL` should be reserved for thoroughly tested code.

Further optimizations for speed can often be accomplished by combining the use of `OPT_LEVEL (MAXIMAL)` with other pragmas. In some applications, judicious use of pragma `SUPPRESS` and pragma `INLINE` will contribute to even faster execution speeds; however, excessive use of pragma `INLINE` in large applications is not recommended. (See "Pragma SUPPRESS" on page M-132 and "Pragma INLINE" on page M-115.)

The higher levels of optimization are also subject to compiler configuration parameters. Refer to "Compile Options" on page 4-99 for more information about these parameters.

Optimization parameters can also be manipulated by using the implementation-defined pragma `OPT_FLAGS`. Refer to "Pragma OPT_FLAGS" on page M-121.

All optimizations performed at the various levels of optimization are done in compliance with the Ada 95 Reference Manual. At some levels, some operations may not be invoked if their only possible effect is to propagate a predefined exception. These optimizations are permitted under RM 11.6.

Components in records may be misaligned because of the following practices:

- Using representation clauses
- Using the predefined pragma `PACK`. (See "Pragma PACK" on page M-123.)
- Doing unchecked conversions to access types

There is no misaligned handler. The hardware allows misaligned integer (fixed-point) data accesses, but floats and long floats must be word-aligned. There is a performance penalty for misaligned accesses.

# Compiler Error Messages

This section describes the different types of compilation errors that can occur and illustrates the procedures MAXAda uses to handle error messages. It also shows the ways in which the **a.error** utility can be used to examine error messages produced by the compiler. (See “a.error” on page 4-16 for details.) The compiler writes all error messages to the standard error stream, **stderr**.

A list of the several categories of error messages appears next, followed by descriptions with examples of each category.

- Lexical Errors
- Syntax Errors
- Semantic Errors
- General Errors
- Informational Messages
- Warnings
- Alerts
- Fatal Errors
- Internal Errors

**NOTE**

Many diagnostics contain references to the *MAXAda Reference Manual* which can be used by the **a.man** utility to provide further assistance in determining the cause and/or solution for the error. See “References to the MAXAda Reference Manual” on page 4-45 for more information about how to use these references with the **a.man** tool.

## Lexical Errors

*Lexical errors* are errors in the formation of literals, identifiers, and delimiters. The compiler performs no semantic analysis on a unit containing lexical errors, but attempts to correct the error to minimize its impact on the discovery of further lexical and syntax errors. Screen 3-1 illustrates:

```
   1:   procedure MY_ _PROGRAM is
A ------------------^
B --------------------^
C ----------------------^
A:lexical error: trailing '_' not allowed
B:lexical error: token starts badly:  "_"
C:syntax error: "program" deleted
   2:   end MY_PROGRAM;
A ------^
B --------------------^
A:syntax error: "end" replaced by "begin"
B:syntax error: "null ; end ;" appended
```

**Screen 3-1. Lexical Errors with `-e` Option**

Each line that contains an error is listed, prefixed with a line number, and followed with a description of the errors that were found. This description includes one or more lines 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. Screen 3-2 illustrates:

```
   1:   procedure MY_PROGRAM? iz
A --------------------------^
B ----------------------------^
A:lexical error: illegal character "?"
B:syntax error: "iz" deleted
   2:   end MY_PROGRAM?;
A ------^
B --------------------^
A:syntax error: "end" replaced by "is new"
B:lexical error: illegal character "?"
```

**Screen 3-2. Lexical Errors with `-e` Option**

## Syntax Errors

*Syntax errors* are errors in the form of grammatical constructs. The compiler performs no semantic analysis on a unit containing syntax errors, but attempts to correct an error to minimize its impact on the discovery of further lexical and syntax errors. Screen 3-3 illustrates:

```
   4:   end OLD_PROGRAM;
A ----------^
A:syntax error: RM 6.3(4): subprogram was given a different name
   7:      for X = 1..10 do
A ---------------^
B -----------------------^
A:syntax error: "=" replaced by "in"
B:syntax error: "do" replaced by "loop"
  11:   end A_PROGRAM;
A ----------^
B --------------------^
A:syntax error: "a_program" replaced by "loop"
B:syntax error: "null ; end ;" appended
```

**Screen 3-3. Example of Syntax Errors with `-e` Option**

## Semantic Errors

*Semantic error*s are those made in the semantic usage of language constructs. The compiler generates no code for units with semantic errors. It generates code for a unit that is error-free, even if other units in the file have semantic errors. All semantic error messages refer to the specific section, subsection, or paragraph within the Ada 95 Reference Manual. Screen 3-4, Screen 3-5, and Screen 3-6 illustrate:

```
   3:         subtype WORK_DAY is WEEK_DAY range 1..5;
A ------------------------------------------------^
A:error: RM 3.5(4): range constraint has wrong type
```

**Screen 3-4. Semantic Errors with `-e` Option**

```
 1:package NEW_PACKAGE is
 2:   type SUN_GLASSES is (grey, green, blue);
 3:   type MY_GLASSES is access SUN_GLASSES;
 4:   type SCREEN is (green, black);
 5:   type MY_SCREEN is access SCREEN;
 6:end NEW_PACKAGE;
 7:
 8:package body NEW_PACKAGE is
 9:   function MY_FUNCTION return MY_GLASSES is
10:      I: MY_GLASSES := null;
11:   begin
12:      return I;
13:   end MY_FUNCTION;
14:
15:   function MY_FUNCTION return MY_SCREEN is
16:      I: MY_SCREEN := null;
17:   begin
18:      return I;
19:   end MY_FUNCTION;
20:
21:   procedure ASSIGN is
22:   begin
23:      MY_FUNCTION.all := green;
A --------------------^
A:error: RM 5.2(9): assignment statement is ambiguous. Could be:
A:error:        my_function, line 9 of new_package
A:error:        my_function, line 15 of new_package
24:     end ASSIGN;
25:
26:end NEW_PACKAGE;
```

**Screen 3-5. Semantic Errors with `-el` Option**

```
   2:         type INT1 is range 1..UPPER;
A --------------------------------^
A:error: RM 3.5.4(3): bounds must be static simple expressions
```

**Screen 3-6. Semantic Errors with `-e` Option**

## General Errors

Errors that are semantic in nature but do not fall within a specific Ada 95 Reference Manual reference are called *general errors*. The compiler does not generate code for units with general errors. However, it generates code for a unit that is error-free, even if other units in the file have errors. Screen 3-7 illustrates:

```
a.build: error: required spec of FACTORIAL does not exist
                in the environment
```

**Screen 3-7. Example of General Errors**

## Informational Messages

The MAXAda compiler may generate an informational message to a user if an internal compiler limit has been exceeded. Most of these internal limits deal with optimization in the compiler's back end. (See "Compile Options" on page 4-99.)

Because most optimization parameters can be manipulated by users via the **`a.options`** tool, informational messages are helpful because they may indicate that certain optimizations are not being performed due to the values of these constraints. This information is helpful to users because it may point out areas where optimizations are being missed, and that in order to perform the maximum amount of optimization possible, the limits should be raised. Limits can be raised:

- By changing the default values with **`a.options`** for all compilations (See "a.options" on page 4-58.)
- By inserting the appropriate optimization values through the use of the `OPT_FLAGS` pragma for compilation units where optimizations are being missed (See "Pragma OPT_FLAGS" on page M-121.)
- By using an appropriate qualifier flag (**`-Q`***parameter*) for the optimizer parameter that is being exceeded. (See "Qualifier Keywords (`-Q` options)" on page 4-105.)

For example, if the environment's configuration has a value of 128 for the parameter `OBJECTS`, then only 128 variables in a given subprogram will be considered as candidates for optimization in the back end of the MAXAda compiler. If a subprogram contains more than 128 objects, the compiler will inform the user that opportunities for optimization may be missed. The informational message will identify which parameter(s) have been exceeded, and will also suggest what an appropriate value for the offending parameter(s) should be in order to take advantage of the maximum amount of optimization possible. For instance, if more than 128 objects exist within a compilation and the default parameter is set to 128, then the following message will appear:

```
info: Only first 128 most frequently occurring variables
      out of 200 total variables were optimized.  Check
      configuration parameters.
```

**Screen 3-8. Example of Warnings**

This informs the user that there were actually 200 objects in the given compilation, and that in order to achieve maximum optimization, the configuration value for `OBJECTS` should be raised to 200 for this compilation. If the suggested value(s) for a compilation are not reasonable values to set as configuration parameters, then the implementation-defined pragma `OPT_FLAGS` can be used to modify the values of optimization parameters for individual compilation units.

Informational messages may be suppressed by specifying the **`-i`** compile option. The **`-w`** compile option also suppresses informational messages.

## Warnings

An error that is not sufficiently serious to prevent code generation or that indicates questionable use of a construct generates a *warning* message. Warning messages may be suppressed by specifying the **-w** compile option. Screen 3-9 illustrates:

```
   6:      for i in 1..10 loop
A ------------^
A:warning: id hides outer definition
```

**Screen 3-9. Example of Informational Messages**

## Alerts

An *alert* is a diagnostic message that conveys information to the user about packages, pragmas, or options that are obsolete in this release. Support for such features will normally be removed in the next production release of MAXAda. Alerts typically refer to the correct method for achieving the desired effect (if such behavior is still meaningful). Alerts do not prevent code generation. Alerts cannot be suppressed through command line options; the only method of preventing alerts is to refrain from using features which are obsolete.

In many cases, the alert indicates that the compilation system is automatically taking the appropriate action for the user. Screen 3-10 illustrates:

```
   2:   pragma memory_pool (lock_pages) ;
A ------------------------^
A:alert: RM Appendix F: This form of pragma memory_pool is obsolete
A:alert: RM Appendix F: it is supported in this release only for
backward compatibility
A:alert: RM Appendix F: use pragma pool_lock_state instead (it is
being activated now)
```

**Screen 3-10. Example of Alerts**

## Fatal Errors

*Fatal errors* are those of such severity that meaningful recovery is impossible and compilation of the file stops. A fatal error can occur if a MAXAda environment is not created with **`a.mkenv`** before compilation. Screen 3-11 illustrates:

```
a.compile: fatal: invalid environment: /pathname/noenv
```

**Screen 3-11. Example of Fatal Errors**

## Internal Errors and Panics

*Internal errors* and *panics* are those due to faults within the compiler. All internal errors and panics should be reported to the Concurrent Customer Support Center.

Internal errors and panics may indicate that a program is erroneous, and they occur because the compiler is unable to process the erroneous program. The following example does not generate an internal error in the current release; it is provided to show an example of the error message. Screen 3-12 illustrates:

```
   --
   begin
        declare
             X : ADDRESS;
             Z : BOOLEAN := X = test'ADDRESS;
A ------------^
A:internal: assertion error at file type_util.c, line 184
        begin
            null;
        end;
   end test;
```

**Screen 3-12. Example of Internal Errors**

## Link Options

MAXAda supports a set of link options for each partition. These link options are persistent and may be specified using any of the following methods:

- **`a.link`** command line

  Options specified directly to **`a.link`** (see "a.link" on page 4-33 for details) may be useful for experimental links, but should not be used during the normal course of development, because specifications made in this manner are *not* persistent.

- partition definition

  Link options are specified for a particular partition by using the following options to **`a.partition`**:

  | | |
  |---|---|
  | **`-oset`** *opts* | Set the link option list to *opts* |
  | **`-oappend`** *opts* | Append *opts* to the link option list |
  | **`-oprepend`** *opts* | Prepend *opts* to the link option list |
  | **`-oclear`** | Clear the link options list |

  where:

  *opts* is a single parameter containing one or more link options; note that *opts* may need to be quoted.

  For example:

  **`a.partition -oset -c`** *partition_name*

  A list of available link options (*opts*) may be found under "Link Options" on page 4-109.

  For more information about setting link options with **`a.partition`**, see "Link Options" on page 4-65.

- environment-wide link options

  Link options that affect all the partitions in the entire environment may be specified using the **`-default`** option to **`a.partition`** in combination with the **`-o`***commands* listed above.

  For example:

  **`a.partition -default -oset -c`**

  sets the environment-wide link options to **`-c`**.

  To list the environment-wide link options, issue:

  **`a.partition -default`**

  by itself.

In addition, the environment may be created with a set of environment-wide link options using the **-oset** *opts* option to **a.mkenv** (see "a.mkenv" on page 4-53 for details).

For example:

```
a.mkenv -oset -c
```

sets the environment-wide link options to **-c** when the environment is created.

- source code

  Link options may also be specified within the source code itself using pragma LINKER_OPTIONS (see "Pragma LINKER_OPTIONS" on page M-119).

  For example:

```
pragma Linker_Options("-c");
```

Link options are interpreted in the order specified above and in the order specified by the user when using **a.link** or **a.partition**. The order of link options specified in the source code is arbitrary among various units, but is in the order specified by the user within any single unit.

In the event of a conflict between two link options, an earlier one will override a later one, generally. The exceptions are benign. For instance, if two contradictory **-trace:buffersize** options are specified, the larger of the two values is selected regardless of the order.

# Linking Executable Programs

MAXAda provides a linker that verifies and creates an ELF executable image of all component units required for a given main unit. The linker can be invoked directly but should be called from the compilation utility **a.build**.

## Linking Ada Programs with Shared Objects

The following table lists the MAXAda-supplied shared object partitions and their corresponding environments.

**Table 3-3. MAXAda-supplied Shared Objects**

| SHARED OBJECT | ENVIRONMENT |
|---|---|
| **libdeprecated.so** | **deprecated** |
| **libgeneral.so** | **bindings/general** |
| **libobsolete.so** | **obsolescent** |
| **libposix1.so** | **bindings/posix_1003.1** |
| **libposix5.so** | **bindings/posix_1003.5** |
| **libpredefined.so** | **predefined** |
| **libpublic.so** | **publiclib** |
| **librtdm.so** | **rtdm** |
| **libsockets.so** | **bindings/sockets** |
| **libvendor.so** | **vendorlib** |

These partitions are expected to be installed in **/usr/ada/***release***/lib** (where *release* is the name of the MAXAda release).

If a user application requires the MAXAda shared libraries but the application will run on a target system without the MAXAda product, then those libraries must be installed independently on the target system.

Users are granted limited rights to copy the required shared libraries from a development system to a licensed run-time system. Contact Concurent Customer Support for details on these rights.

**IMPORTANT**

Users cannot copy shared libraries from the AXI for MAXAda product. If these libraries are required for the run-time system, a copy of the AXI for MAXAda product must be purchased and installed on the target machine. These libraries include:

**/usr/ada/***release***/lib/libX.so**
**/usr/ada/***release***/lib/libmotif.so**
**/usr/ada/***release***/lib/libstars.so**

# Debugging

## Real-Time Debugging

In addition to the symbolic debugging capabilities provided by **nview**, and the post-analysis debugging capabilities provided by the tracing mechanism, MAXAda also provides several ways to debug programs in real-time. The **a.monitor** utility may be used to monitor an Ada program while it is running; the utility displays Ada task state information, CPU, stack, and memory usage. See "a.monitor" on page 4-55 as well as Chapter 12 - Real-Time Monitoring for more information.

**NOTE**

NightView (**nview**) requires a non-zero debug level and level `full` (`2`) for full support. See "Debug Level (`-g[level]`)" on page 4-100.

## Selecting a Debug Level

There are trade-offs to be considered when selecting the debug level with which to compile a single unit or application. While the full level of debug information provides more information for such programs as the **nview** debugger, it does so at the cost of additional disk space in object files and the final executable. Note, however, that there is no additional space required in memory as an application is executing. Note also that **nview** debugging requires only the program image.

If users intend to use other programs, such as **nview**, then careful debug-level selection must be made. Good candidates for compiling at full debug level are units that:

- Are few in number
- Are reasonably self-contained
- Contain frequently used type information
- Need to be debugged

If users expect to debug only certain portions of an application, it is possible to compile only those certain units with the full level of debug information and to compile the remainder of the application with the none or lines level. Thus, only a portion of the application requires additional disk space.

This technique is very useful. However, the user must be careful because it can actually be counterproductive and produce object files requiring more disk space than would be required otherwise. This is because the compiler attempts not to duplicate debug information whenever possible.

### Example Scenario:

Assume the following code fragment:

```
package types is
   type rec is record
      ...
   end record;
end types;

with types;
procedure user1 is
   var : types.rec;
   ...
begin
   ...
end user1;

with types;
procedure user2 is
   var : types.rec;
   ...
begin
   ...
end user2;
```

### Example 1:

Assume that all of these units are compiled with the full level of debug information. The debug information for unit `types` includes a description of the type `rec`.The debug information for each of the units `user1` and `user2` includes descriptions of their respective variables, `var`; however, the descriptions of those variables need not fully describe the type `rec`. Their debug information just refers to the debug information already described in unit `types`.

### Example 2:

Assume that the unit `types` was compiled with the none level of debug information. Further assume that the units `user1` and `user2` are compiled with the full level of debug information. The debug information for the unit `types` does not include a description of the type `rec`. The debug information for each of the units `user1` and `user2` includes descriptions of their respective variables, `var`. Unlike the previous example, though, these descriptions cannot simply reference the debug information for `rec` in the unit `types` because it does not exist there. So, they must each include the debug information for `rec` locally. Unfortunately, because neither references the other on any `with` clause and because language rules prohibit any dependency from one to the other in the absence of such a `with` clause, they cannot share the debug information, and it is duplicated in each of them. This was not the case in the first example.

## Degree of Interest

Pragma INTERESTING indicates in the debug information the degree of interest of a named unit, object, component or exception (see “Pragma INTERESTING” on page M-117). This information is only useful if full debug information is enabled (see “Pragma DEBUG” on page M-109 or “Debug Level (`-g[level]`)” on page 4-100).

This information is useful in conjunction with the `Real_Time_Data_Monitoring` package. A minimum interest "threshold" may be specified to restrict the set of objects or components to be monitored using the `interest_threshold` parameter (see “rtdm” on page 9-12).

This information is also useful in conjunction with the NightView debugger. A minimum interest threshold may be specified via the **`interest`** command to restrict the set of routines to be displayed in various circumstances.

In addition, the `-Qinteresting` compile option may be used to indicate the default degree of interest for every entity in the compilation. See “Qualifier Keywords (`-Q` options)” on page 4-105 for more information.

## Debug Information and cprs

The **`cprs`** utility (see **`cprs(1)`**), supplied with PowerMAX OS, reduces the size of an application by removing duplicate type information. The Ada compiler reduces the value of this tool by already referencing the debug information for types defined in other units from those other units. However, the **`cprs`** utility can still reduce the size of Ada applications. Also, if debug code from other languages is included in an application, then **`cprs`** can significantly reduce the size of those portions as well.

If users compile only certain units with full debug information, it is possible to produce duplicate debug information for types in several units. Also, even if an entire application is compiled with full debug information, anonymous types are frequently duplicated in several units, as are types for certain compiler-generated constructs.

# 4
# MAXAda Utilities


Common Options . . . . . 4-2
a.build . . . . . 4-3
Parallel Compilations and Dependency Analyses . . . . . 4-5
Inline Dependencies . . . . . 4-6
Forcing Attempts . . . . . 4-6
Why . . . . . 4-6
a.cat . . . . . 4-7
a.chmod . . . . . 4-8
a.compile . . . . . 4-9
a.demangle . . . . . 4-11
a.deps . . . . . 4-13
a.edit . . . . . 4-15
a.error . . . . . 4-16
a.expel . . . . . 4-21
a.fetch . . . . . 4-22
a.freeze . . . . . 4-25
a.help . . . . . 4-26
a.hide . . . . . 4-27
a.install . . . . . 4-28
a.intro . . . . . 4-30
a.invalid . . . . . 4-32
a.link . . . . . 4-33
a.ls . . . . . 4-35
Formatting the listing . . . . . 4-37
Dependent units . . . . . 4-39
Parts . . . . . 4-40
Sorting . . . . . 4-41
Filtering . . . . . 4-41
a.lssrc . . . . . 4-42
a.man . . . . . 4-44
References to the Ada 95 Reference Manual . . . . . 4-45
References to the MAXAda Reference Manual . . . . . 4-45
Access to Support Packages . . . . . 4-46
a.map . . . . . 4-47
a.mkenv . . . . . 4-53
a.monitor . . . . . 4-55
a.nfs . . . . . 4-56
a.options . . . . . 4-58
Option Sets . . . . . 4-59
Listing options . . . . . 4-59
Setting options . . . . . 4-60
Modifying options . . . . . 4-60
Clearing options . . . . . 4-60
Deleting options . . . . . 4-60
Keeping temporary options . . . . . 4-61
Setting options on foreign units . . . . . 4-61
a.partition . . . . . 4-62

Main Subprogram . . . . 4-64
Elaboration and Finalization . . . . 4-65
Case Sensitivity . . . . 4-65
Consistency . . . . 4-65
Link Options . . . . 4-65
Link Rule . . . . 4-67
Implicitly-Included Libraries . . . . 4-72
a.path . . . . 4-74
a.pclookup . . . . 4-76
a.pp . . . . 4-77
Commands . . . . 4-79
Expressions . . . . 4-80
Defaults . . . . 4-81
Examples . . . . 4-81
a.release . . . . 4-83
a.resolve . . . . 4-85
a.restore . . . . 4-86
a.rmenv . . . . 4-87
a.rmsrc . . . . 4-88
a.script . . . . 4-89
Generated Script - Options . . . . 4-91
a.syntax . . . . 4-92
a.tags . . . . 4-94
a.touch . . . . 4-97
a.trace . . . . 4-98
Compile Options . . . . 4-99
Negation (**!**) . . . . 4-99
Debug Level (**-g[***level***]**) . . . . 4-100
Opportunism (**-opp**) . . . . 4-100
Share Mode (**-sm**) . . . . 4-101
Not Shared (**-N**) . . . . 4-101
Optimization Level (**-O[***level***]**) . . . . 4-101
Qualifier Keyword (**-Q***keyword*[**=***value*]) . . . . 4-104
Suppress Checks (**-S**) . . . . 4-104
Qualifier Keywords (**-Q** options) . . . . 4-105
Link Options . . . . 4-109
Share Path . . . . 4-110
Incrementally Updateable Partition . . . . 4-110
Tracing . . . . 4-110
Task Weight . . . . 4-111
Shared Object Transitive Closure . . . . 4-111
Obscurity Checks . . . . 4-112

# 4
# MAXAda Utilities

MAXAda consists of a number of utilities that provide support for library management, compilation and program generation, and debugging. This section will go through these tools and give an overview of their uses. The utilities appear in alphabetical order. For easy reference, the command syntax and options available for each utility are provided in tabular format. Available options for each tool are also provided by specifying the **`-H`** (Help) command-line option when invoking the utility.

Each section describes a command, shows the command's syntax and discusses the options that can be specified. For each option flag listed in the "Option" column, a mnemonic and a short description are provided in the columns labeled "Meaning" and "Function," respectively.

See "MAXAda Utilities" on page 1-1 for a complete listing of these utilities. In addition, refer to "Common Options" on page 4-2 for those options relative to all utilities.

# Common Options

There are a number of options that are the same for each utility. They are listed for each tool but are also listed below.

| Option | Meaning | Function |
|---|---|---|
| **-env** *env* | environment | Specify an environment pathname |
| **-rel** *release* | release | Specify a MAXAda release (other than the default release) |
| **-H** | help | Display syntax and options for that particular function |

*unit-id* is defined by the following syntax:

*unit***[/***part***] | all[/***part***]**

where *part* is the **specification**, **body**, or **all**; abbreviations are accepted.

See “Unit Identification” on page 3-7 for more information about the *unit-id*.

# a.build

## Compile and link as necessary to build a unit, partition or environment

The syntax of the **a.build** command is:

**a.build [***options***] [***partition* **...]**

The following represents the **a.build** options:

| Option | Meaning | Function |
|---|---|---|
| **-allparts** | all partitions | Build all partitions in the environment. This option is not allowed if the **-o** option is specified. |
| **-attempt** | force attempts | Attempt those compilations and links that will fail, but skip subsequent dependent compilations and links |
| **-Attempt** | force attempts! | Attempt those compilations and links that will fail, including subsequent dependent compilations and links |
| **-bypass** | bypass optional | Bypasses optional dependencies if they cannot be satisfied by the build |
| **-C** "*compiler"* | compiler | Use *compiler* to compile units (may be used to pass options to the compiler, e.g. **a.build -C "a.compile -b"**) |
| **-env** *env* | environment | Specify an environment pathname |
| **-H** | help | Display syntax and options for this function |
| **-i** | infos | Suppress **a.build** information messages<br><br>See "Informational Messages" on page 3-31 for more details. |
| **-IO**[level] | inter optimization | Set level of interoptimization (0-1)<br><br>See "Interoptimization" on page 3-24 and "Inline Dependencies" on page 4-6 for more details. |
| **-L** "*linker*" | linker | Use "*linker*" to link partitions (may be used to pass options to the linker) |
| **-noimport** | no import | Forestall automatic recompilation of out-of-date units from other environments in the current environment |
| **-nosource** | no source | Skip checks of the source timestamps for out-of-date units (should only be used if no source files have changed) |
| **-o** *file* | output | Override the output file for the partition being built. Only a single partition file name is allowed with this option. |
| **-part** *partition* | partition | Build the given *partition*, all included units and all units upon which they directly or indirectly depend |
| **-p**[*n*] | parallel | Perform up to *n* parallel compilations (n defaults to number of CPUs) |

| Option | Meaning | Function |
|---|---|---|
| **-pd**[*n*] | parallel dependencies | Perform up to *n* parallel dependency analyses; (*n* defaults to number of CPUs * 2) |
| **-r** *unit* | require | Build the given *unit*, all units upon which it directly or indirectly depends, and all units which directly or indirectly depend upon it. This option is not allowed if the **-o** option is specified. |
| **-rel** *release* | release | Specify a MAXAda release (other than the default release) |
| **-rfile** *file* | require from file | Build the units in *file*, all units upon which they directly or indirectly depend, and all units which directly or indirectly depend upon them |
| **-source** *file* | source file | Build all units defined in the given source *file* and all units upon which they directly or indirectly depend |
| **-state** *s* | state | Build all specified units to compilation state *s* (`compiled` is the only valid state allowed for this option in the current release) |
| **-stop** | stop on errors | If an error is encountered, stop building (normally, any units not dependent upon the erroneous units would be built) |
| **-u** *unit* | unit | Compile the specified *part*(s) of the specified unit; if no part is specified, both `specification` and `body` are built. The *unit* parameter can be "`all`". This option is not allowed if the **-o** option is specified. |
| **-ufile** *file* | units from file | Build the units in *file* and all units upon which they directly or indirectly depend |
| **-V** | verify | List compilations that would occur, but do not actually perform them |
| **-v** | verbose | Display compilations as they are done |
| **-vv** | very verbose | Display commands as they are done |
| **-w** | warnings | Suppress **a.build** warnings<br><br>See "Warnings" on page 3-32 for more details. |
| **-Why** | why | List reasons for compilations that would occur, and the compilations themselves, but do not actually perform them |

**NOTE**

Specified partitions are equivalent to partitions passed as arguments to the **-part** option. If no options are specified, then all units and partitions in the environment are built.

MAXAda provides the **a.build** utility to build partitions and units in an environment. **a.build** determines which units must be compiled to build the given target, preprocessing those units marked for preprocessing, and calls the linker to produce the desired partition. **a.build** examines the current environment (and the environments on the Environment Search Path), determines and automatically executes the proper sequence of compilations and links necessary to build the given partition.

Targets to **a.build** can be:

| | |
|---|---|
| partitions | which can be specified directly, with the **-part** option, or with the **-allparts** option |
| units | which can be specified with the **-r** or **-u** option, depending upon the desired result |

If the **-u** option is specified, **a.build** ensures the named *unit* is up-to-date, recompiling any dependencies if necessary.

## Parallel Compilations and Dependency Analyses

If the **-p** option is used, then **a.build** attempts to build as much as it can in parallel, making use of the available resources. If an integer parameter, *n*, is supplied, then *n* parallel compilations are distributed across the CPUs on the system. If no integer parameter is given, then **a.build** attempts to distribute a number of parallel compilations that is consistent with the number of CPUs on the system. Using the **-p** option can greatly enhance compilation speed if used to compile a large MAXAda library and system resources are available.

The **a.build** tool not only does its compilations in parallel when the **-p** option is active, but it also does its dependency analysis in parallel. By default, twice the number of parallel dependency analyses are used as are specified with **-p**. However, the **-pd** option can be used to control the number of parallel dependency analyses independently. If an integer parameter, *n*, is supplied with **-pd**, then *n* parallel dependency analyses are used. If no integer parameter is given, then **a.build** attempts to use twice the number of CPUs on the system. Finally, the **-pd** option can be used without the **-p** option, if that is desired. In that case, compilations will be single-stream, while dependency analyses will be in parallel.

See the following matrix for a complete description of the interaction of the **-p** and **-pd** options.

**Table 4-1. Number of Parallel Dependency Analyses**

| | **-p** | **-p** *n'* | No **-p** |
|---|---|---|---|
| **-pd** | Twice number of CPUs on system | Twice number of CPUs on system | Twice number of CPUs on system |
| **-pd** *n* | *n* | *n* | *n* |
| No **-pd** | Twice number of CPUs on system | Twice *n'* | 1 |

Normally, **a.build** attempts to build all units in the current MAXAda environment and all units on the Environment Search Path that are required. The **-noimport** option can be used to prevent automatic recompilation of out-of-date units from other environments.

See “Compile Options” on page 3-20 and “Link Options” on page 3-34 for more information.

## Inline Dependencies

With the interoptimization level set to "inlining" (e.g. **-IO1**), the **a.build** utility detects inline dependencies and attempts to honor them. To honor them, **a.build** must determine a valid compilation order that permits all requested inline calls to actually be performed inline.

Sometimes an inline dependency creates a dependency loop. In such instances, particular inline dependencies may have to be broken in order to break dependency loops. The **a.build** utility notifies users of dependency loops and issues a message when inline dependency loops must be broken in order to proceed with dependency analysis. If such loops exist, then it is possible that some requested inline calls may not actually be performed inline.

## Forcing Attempts

In situations where **a.build** has already tried to compile a unit but has encountered errors, it will not attempt to compile the unit again if it has not been modified. On subsequent compilations, **a.build** will report to the user a message similar to:

```
a.build: error: MAX(060) 3-23: subprogram body
sem_errors will not be built because it contains
semantic errors
```

However, the user may wish to see the specific errors that were reported on the first attempt. The **-attempt** and **-Attempt** options are for this purpose. When **a.build** is run with these either of these options, it will try to recompile units that have encountered errors in previous compilation attempts.

**NOTE**

Similar functionality exists in the NightBench Program Development Environment using the Attempt compiles and links that will fail checkbox under the Settings page of the Builder window. See the *NightBench User's Guide* (0890514) for more details.

See "Compiler Error Messages" on page 3-26 for more information about the types of errors you may encounter in this situation, especially "Syntax Errors" on page 3-28 and "Semantic Errors" on page 3-29.

## Why

The **-Why** option lists reasons why all the entities that would be built are inconsistent, and then shows the commands that would be executed to make things consistent. (This latter part is like **a.build -V**).

# a.cat

## Output the source of a unit

The syntax of the **a.cat** command is:

**a.cat [***options***]** *unit-id*

The following represents the **a.cat** options:

| Option | Meaning | Function |
|---|---|---|
| **-env** *env* | environment | Specify an environment pathname |
| **-H** | help | Display syntax and options for this function |
| **-h** | no header | Does not output filename header |
| **-l** | line numbers | Prepend each line of source with its line number |
| **-rel** *release* | release | Specify a MAXAda release (other than the default release) |

*unit-id* is defined by the following syntax:

*unit***[/***part***]**

where *part* is the **specification** or **body**; abbreviations are accepted.

The **a.cat** command is similar to the UNIX **cat(1)** command in functionality. It accepts as its argument a *unit_id* and prints to **stdout** the source file in which this unit is found.

By default, it outputs a header containing the full path name of the source file. This can be suppressed by specifying the **-h** option.

Also, line numbers can be prepended to each line of source by using the **-l** option.

# a.chmod

### Modify the UNIX file system permissions of an environment

The syntax of the **a.chmod** command is:

**a.chmod [***options***]** *access_mode*

The following represents the **a.chmod** options:

| Option | Meaning | Function |
|---|---|---|
| **-a** | all | In addition to internal environment files, change the permissions of all files associated with the environment, including source files and partition targets |
| **-env** *env* | environment | Specify an environment pathname |
| **-f** | force | Force, if some environment components are missing |
| **-H** | help | Display syntax and options for this function |
| **-i** | ignore | Quietly ignore all non-fatal errors |
| **-q** | query | Display the permissions on the current environment |
| **-rel** *release* | release | Specify a MAXAda release (other than the default release) |
| **-s** | source only | Only change the permissions of the source files associated with the environment; no other files are affected |

*access_mode* is a symbolic or octal digit parameter indicating the desired file system permission. For example,

**777** all permissions (read, write, execute) for all users

**u-x** removes execute permission from the file's owner

**+x** gives execute privileges to user, group, and others

For details, see the **chmod(1)** manual page.

# a.compile

### Compile the specification and/or body of one or more units

> **INTERNAL UTILITY**
>
> This tool is used internally by **a.build** which is the recommended utility for compilation and program generation.
>
> **a.compile** is not intended for general usage.

The syntax of the **a.compile** command is:

**a.compile [***options***] [***compile_options***] [***unit-id*** ...]**

The following represents the **a.compile** options:

| Option | Meaning | Function |
|---|---|---|
| **-b** | object | Send symbol object listing to **stdout** |
| **-env** *env* | environment | Specify an environment pathname |
| **-fetch** | fetch | For specified units from other environments, fetch them first |
| **-H** | help | Display syntax and options for this function |
| **-HC** | help compile | Display list of compile options |
| **-HQ** | help qualifier | Display list of qualifier keywords (**-Q** options)<br><br>See "Qualifier Keywords (-Q options)" on page 4-105 for more details. |
| **-inter** *fd1 fd2* | interactive | Execute in interactive mode control file descriptor 1 (*fd1*) and response file descriptor 2 (*fd2*) |
| **-noimport** | no imports | Prevent the automatic local recompilation of out-of-date foreign instantiations |
| **-pipeline** | pipeline | Perform optimization and code generation in parallel with subsequent compilations for limited parallelism; requires **-inter** option; primarily for use with **a.build -p** |
| **-pragma** *file* | config pragmas | Compile *independent configuration pragmas* from the given *file*<br><br>See "Configuration Pragmas" on page 3-9 for more information. |
| **-quiet** | quiet options | Suppress display of effective options |
| **-R** | recompile instantiations | Recompile out-of-date instantiations |
| **-rel** *release* | release | Specify a MAXAda release (other than the default release) |

| Option | Meaning | Function |
|---|---|---|
| **-state** *s* | state | Compile the specified unit to compilation state *s* (`compiled` is the only valid state allowed for this option in the current release) |
| **-V** | very verbose | Print subordinate tool command lines |
| **-v** | verbose | Print header for each compilation |
| **-vv** | very verbose | Print results of each compilation |

*unit-id* is defined by the following syntax:

*unit***[/***part***] | all[/***part***]**

where *part* is the **specification**, **body**, or **all**; abbreviations are accepted.

If *compile_options* are specified to this command, they are added to the set of temporary unit options. For instance, if the temporary compile options for the unit `hello` consist of **-S** and the following command is issued

```
$ a.compile -g hello
```

the temporary unit options will now consist of **-S** and **-g**.

The *file* specified by the **-pragma** option may only contain independent configuration pragmas. See "Configuration Pragmas" on page 3-9 for more information.

See "Compile Options" on page 4-99 for list of compile options.

# a.demangle

## Output the source of a unit

The syntax of the **a.demangle** command is:

**a.demangle [***options***]**

The following represents the **a.demangle** options:

| Option | Meaning | Function |
|---|---|---|
| **-env** *env* | environment | Specify an environment pathname |
| **-f***c* | file name | File name quote character *c* |
| **-H** | help | Display syntax and options for this function |
| **-k** | keep | Keep quote characters for unknown names |
| **-r***c* | routine | Routine name quote character *c* |
| **-rel** *release* | release | Specify a MAXAda release (other than the default release) |

The **a.demangle** utility is a filter that accepts MAXAda symbol names, such as those found in a MAXAda object file, and returns Ada unit names in expanded form.

To recognize symbol names, **a.demangle** requires the location of the MAXAda environment in which those symbols exist. For instance, if the symbol names are taken from an executable, then **a.demangle** requires the location of the environment in which that executable was linked. If a **-env** option is specified, **a.demangle** uses the given environment. Otherwise, **a.demangle** assumes that the current working directory is the location of the environment.

By default, **a.demangle** expects symbol names to be the first word on each line of **stdin**, optionally followed by whitespace and any additional text. The **a.demangle** utility returns the corresponding Ada unit name on **stdout**, followed by the unaltered optional whitespace and text.

For example, the command:

```
$ a.demangle
```

if given a line such as:

**A_foo.5S13.bar..BODY** *some additional text*

would return, assuming the symbol was recognized:

```
bar.foo (body) some additional text
```

If either the **-f** or **-r** option is present, **a.demangle** no longer expects symbol names at the beginning of each line of **stdin**. Instead, if the **-r** option is present and followed immediately with no intervening whitespace by a character, that character will serve as a

quote character for MAXAda symbols (routines). Similarly, if the **-f** option is present and followed by a character, that character will serve as a quote character for source file names.

The quote characters specified by **-f** and **-r** must not be identical.

If **a.demangle** locates two (or more) matching quote characters on a line, it interprets the text between them to be a MAXAda symbol name or source file name, depending on the quote character. These lines are returned on **stdout** with all recognized symbols replaced by their corresponding Ada unit names and all source file names left unchanged. All other lines and all unrecognized names are returned unchanged. The command:

```
$ a.demangle -f@ -r#
```

if given a line such as:

```
Routine #A_foo.5S13.bar..BODY# is located in file
    @bar_b.a@
```

might return, assuming everything was recognized:

```
Routine bar.foo (body) is located in file bar_b.a
```

Normally, when the **-f** or **-r** option is present, quote characters are removed regardless of whether or not the symbol or source file name is recognized. If the **-k** option is specified, however, quote characters remain if they enclose text which is unrecognized.

**NOTE**

The **-f** option is supported only for backward-compatibility.

# a.deps

### Update environment with information about units within source files

**INTERNAL UTILITY**

This tool is used internally by MAXAda.

**a.deps** is not intended for general usage.

The syntax of the **a.deps** command is:

**a.deps [***options***] [***source_file* **...]**

The following represents the **a.deps** options:

| Option | Meaning | Function |
|---|---|---|
| **-e**[**e** \| **l** \| **L** \| **v**] | errors | Control error emission style:<br>**-e** list syntax errors for files **a.deps** is unable to parse to **stdout** with related source lines<br>**-ee** embed syntax errors in files that **a.deps** is unable to parse and invoke $EDITOR<br>**-el** list source files to **stdout**, interspersed with any syntax errors — only source files that **a.deps** is unable to parse<br>**-eL** list source files to **stdout**, interspersed with any syntax errors — even source files that **a.deps** is able to parse<br>**-ev** embed syntax errors in files that **a.deps** is unable to parse and invoke **vi**<br>The default behavior is to list syntax errors to **stderr** with file name, and line and column number. |
| **-env** *env* | environment | Specify an environment pathname |
| **-p**[*n*] | parallel | Use *n* parallel introductions (*n* defaults to the number of CPUs) |
| **-rel** *release* | release | Specify a MAXAda release (other than the default release) |
| **-s** *file_list* | file list | Read *file_list* for a list of files to process<br>If - is specified for *file_list*, read file list from **stdin** |
| **-v** | verbose | Echo files as they are processed |
| **-H** | help | Display syntax and options for this function |

| Option | Meaning | Function |
|---|---|---|
| **-P** | preprocess | Inform the environment that preprocessing is always required for all source files included in this invocation of **a.deps** |
| **-!P** | no preprocess | Inform the environment that preprocessing should never be performed for any source file included in this invocation of **a.deps** |
| **-V** | very verbose | Echo units encountered for each file |

This tool behaves exactly as the **a.intro** utility. See page 4-30 for more information.

# a.edit

### Edit the source of a unit, then update the environment

The syntax of the **a.edit** command is:

**a.edit [***options***]** *unit-id*

The following represents the **a.edit** options:

| Option | Meaning | Function |
|---|---|---|
| **-e** *editor* | editor | Use *editor* instead of `$EDITOR` |
| **-env** *env* | environment | Specify an environment pathname |
| **-H** | help | Display syntax and options for this function |
| **-i** | inhibit | Do not immediately notify the environment that the unit has changed |
| **-rel** *release* | release | Specify a MAXAda release (other than the default release) |
| **-s** | syntax | Check the syntax of the unit's source file after editing |
| **-v** | verbose | Display invocations of the editor, **a.syntax**, and **a.deps** as they occur |

*unit-id* is defined by the following syntax:

*unit***[/***part***]**

where *part* is the **specification** or **body**; abbreviations are accepted.

# a.error

### Process diagnostic messages generated by the compiler and other tools

**INTERNAL UTILITY**

This tool is used internally by **a.build** which is the recommended utility for compilation and program generation.

**a.error** is not intended for general usage.

The syntax of the **a.error** command is:

**a.error [***options***]**

The following represents the **a.error** options:

| Option | Meaning | Function |
|---|---|---|
| **-e** *editor* | editor | Embed error messages in the source file and invoke the specified *editor* |
| **-env** *env* | environment | Specify an environment pathname |
| **-H** | help | Display syntax and options for this function |
| **-l** | listing | Produce listing to **stdout** |
| **-N** | no line #'s | Do not display line numbers |
| **-o** | order | Do not sort the order of the diagnostics by file and line number; process each diagnostic in the order given |
| **-rel** *release* | release | Specify a MAXAda release (other than the default release) |
| **-s** | suppress | Suppress non-error lines |
| **-t***number* | tabs | Change the default tab settings; the default is 8. (there is no space between **t** and *number*) |
| **-v** | **vi** | Embed error messages in the source file and invoke the **vi** editor |
| **-W** | warnings | Ignore warnings |

Compiler output may be redirected into a file and examined with the aid of the **a.error** command or can be piped directly into **a.error** via the **-e** compile option.

**a.error** reads the specified file or the standard input, determining the source file(s) containing errors and processing the errors according to the options given.

**NOTE**

Perhaps more generally useful are the **-e** compile options (**-e**, **-ee**, **-el**, **-eL**, **-ev**), which automatically call **a.error** to process any compiler error messages resulting from the current compilation. See "Compile Options" on page 4-99 for a complete list of compile options.

Screen 4-1 shows the file **badtry.a**. This file containing errors is used to illustrate various ways MAXAda tools can use **a.error** to process error messages.

```
-- file is badtry.a --

with ADA.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;
    ADA.TEXT_IO.PUT(INTEGER'IMAGE(SUM));
    ADA.TEXT_IO.PUT(REAL'IMAGE(AVG));
  end MAIN;
```

**Screen 4-1. File badtry.a**

Before it can be compiled, the file must be introduced into a MAXAda environment, and a partition must be created for it:

```
$ a.mkenv
$ a.intro badtry.a
$ a.partition -create active badtry
```

The file can be compiled and the output directed as follows (**stdout** is redirected to the file **badtry.errors**):

```
$ a.build 2> badtry.errors
```

Screen 4-2 shows the contents of file **badtry.errors**.

```
/badenv/badtry.a, line 5, char 29: lexical error: deleted "f"
/badenv/badtry.a, line 5, char 19: syntax error: " identifier"
inserted
/badenv/badtry.a, line 10, char 4: syntax error: ";" inserted
/badenv/badtry.a, line 17, char 8: syntax error: RM95 6.3(3):
subprogram was given a different name:
a.build: error: errors encountered during build
```

**Screen 4-2. File badtry.errors**

This file can simply be listed, if desired, but it is more useful to use **a.error** as follows.

```
$ a.error  -l  badtry.errors
```

outputs the listing that appears in Screen 4-3.

```
Non-specific diagnostics:

a.build: error: errors encountered during build

**********************  /badenv/badtry.a  ************************

   1:-- file is badtry.a --
   2:
   3:with ADA.TEXT_IO;
   4:procedure BADTRY is
   5:     subtype T is range 1..1f;
A ---------------------^
B -------------------------------^
A:syntax error: " identifier" inserted
B:lexical error: deleted "f"
   6:     COUNT : T;
   7:     SUM : INTEGER;
   8:     type REAL is digits 6;
   9:     AVG : REAL
  10:   begin
A ------^
A:syntax error: ";" inserted
  11:     for COUNT in T loop
  12:     SUM := SUM + I;
  13:     end loop;
  14:     AVG := SUM / COUNT;
  15:     ADA.TEXT_IO.PUT(INTEGER'IMAGE(SUM));
  16:     ADA.TEXT_IO.PUT(REAL'IMAGE(AVG));
  17:   end MAIN;
A ----------^
A:syntax error: RM95 6.3(3): subprogram was given a different name:
  18:
```

**Screen 4-3. a.error -l Output Listing**

The preceding 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 `1f` being a malformed integer literal. 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.

With the **-v** option, **a.error** writes the error messages directly into the original source file and calls the **vi** text editor. Line numbers are suppressed, error messages marked with the pattern `###`, and the editor positioned in the file with the cursor at the point of the first error.

After the compilation,

```
$ a.error  -v  <  badtry.errors
```

calls **vi**. Screen 4-4 shows the screen output.

```
-- file is badtry.a --

with ADA.TEXT_IO;
procedure BADTRY is
     subtype T is range 1..1f;
------------------^A                                                  ###
---------------------------^B                                         ###
--### A:syntax error: " identifier" inserted
--### B:lexical error: deleted "f"
     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;
     ADA.TEXT_IO.PUT(INTEGER'IMAGE(SUM));
     ADA.TEXT_IO.PUT(REAL'IMAGE(AVG));
   end MAIN;
-------^A                                                             ###
--### A:syntax error: RM95 6.3(3): subprogram was given
                                    a different name:
~
~
~
~
"/badenv/badtry.a" 26 lines, 877 characters
```

**Screen 4-4. a.error -v Output Listing**

The ### is provided so that error messages can be easily found and subsequently deleted. For example, if invoked with the **-v** (**vi**) option, **a.error** embeds error text in the source file and then invokes the **vi** editor. All error text can easily be found and removed with simple editor commands by searching for the ### pattern and deleting. In **vi**, for instance, the sequence "**:g/###/d**" deletes all lines matching the ### pattern.

**NOTE**

> The **-o** option to **a.error** displays each diagnostic in the order in which it was encountered without sorting the diagnostics by file and line number. This option has no effect when used in conjunction with the **-e**, **-v**, or **-l** options to **a.error** (or the associated **-e**, **-ee**, **-el**, **-eL**, **-ev** compile options. See "Compile Options" on page 4-99 for a complete list of compile options.)

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 and the error corrected, a second copy of the same messages appears.

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.

The preferred method for achieving the same results is to modify the default options for the environment so that the **vi** editor is invoked whenever errors are encountered during

compilation. The following command sets this as a default option for the entire environment:

```
$ a.options -default -mod -ev
```

To compile the unit, simply issue **a.build**:

```
$ a.build
```

Now, when errors are encountered during compilation, the **vi** editor will be automatically opened to the source file with the error messages embedded in it. Also, upon leaving the editor, the compiler offers to recompile the file.

This method is generally faster for rapid interactive program development because it does not require any intermediate files. Also, because the environment-wide options are persistent, whenever **a.build** is called, these options are "remembered" and do not need to be specified again.

For more information about compiler error messages, see "Compiler Error Messages" on page 3-26.

# a.expel

### Expel fetched or naturalized units from the environment

The syntax of the **a.expel** command is:

**a.expel [***options***]** *unit-id* **...**

The following represents the **a.expel** options:

| Option | Meaning | Function |
|---|---|---|
| **-env** *env* | environment | Specify an environment pathname |
| **-rel** *release* | release | Specify a MAXAda release (other than the default release) |
| **-v** | verbose | Print message for each expelled unit |
| **-H** | help | Display syntax and options for this function |

*unit-id* is defined by the following syntax:

*unit***[/***part***] | all[/***part***]**

where *part* is the **specification**, **body**, or **all**; abbreviations are accepted.

Local versions of foreign units may be created via the **a.fetch** tool (see "a.fetch" on page 4-22) and the **a.build** tool (see "a.build" on page 4-3 for details and "Hello Again... Ambiguous Units" on page 2-15 for an example). These versions are called *fetched* and *naturalized*, respectively. (See "Nationalities" on page 3-9 for a more detailed discussion.)

It may be desirable to later remove these local versions, thus making the foreign versions once again visible. The **a.expel** tool is provided for this purpose.

**NOTE**

Other methods exist for removing native units. See "a.rmsrc" on page 4-88 and "a.hide" on page 4-27 for more information.

# a.fetch

### Fetch the compiled form of a unit from another environment

The syntax of the **a.fetch** command is:

**a.fetch [***options***]** *unit-id* **...**

The following represents the **a.fetch** options:

| Option | Meaning | Function |
|---|---|---|
| **-d** | default | Use default supplied libraries with **-from** |
| **-env** *env* | environment | Specify an environment pathname |
| **-from** *env* | from env | Specify an environment pathname from which to fetch the unit(s) |
| **-H** | help | Display syntax and options for this function |
| **-rel** *release* | release | Specify a MAXAda release (other than the default release) |
| **-v** | verbose | Display a message for each fetched unit |

*unit-id* is defined by the following syntax:

*unit***[/***part***]** **| all[/***part***]**

where *part* is the **specification**, **body**, or **all**; abbreviations are accepted.

At times, it may be desirable for users to be able to force copies of specified units from other environments into the current environment. This may be necessary to avoid *obscurities*.

Obscurities occur when the natural behavior of MAXAda and the Environment Search Path mechanism prevent an intended file from being used for a particular compilation.

For example, consider the following environment dependency scenario:

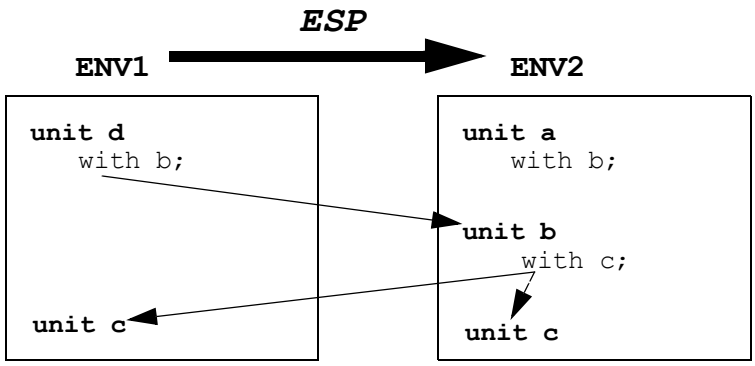



**Figure 4-1. Environment scenario containing obscurities**

In Figure 4-1:

- **`unit c`** exists in both **`ENV1`** and **`ENV2`** but may have completely different functionalities
- **`unit d`** has a `with b` statement inside it in **`ENV1`**
- **`unit b`** does not exist in **`ENV1`**
- **`unit b`** exists in **`ENV2`** on the Environment Search Path (*ESP*) for **`ENV1`**
- **`unit b`** has a `with c` statement inside it in **`ENV2`**

When **`unit d`** is compiled, the following obscurity arises: Because **`unit d`** requires **`unit b`** and **`unit b`** does not exist in **`ENV1`**, the Environment Search Path will be searched. **`unit b`** is found in **`ENV2`** but has a "`with c`" statement inside it. Since **`unit c`** exists in **`ENV1`**, the compilation utility will use the local **`unit c`** contained in **`ENV1`**, instead of the foreign **`unit c`** in **`ENV2`** which is required by **`unit b`**.

In order for **`unit d`** to use the foreign **`unit b`** and the local copy of **`unit c`**, and for everything to be consistent, you may "fetch" a copy of **`unit b`** to your local environment.

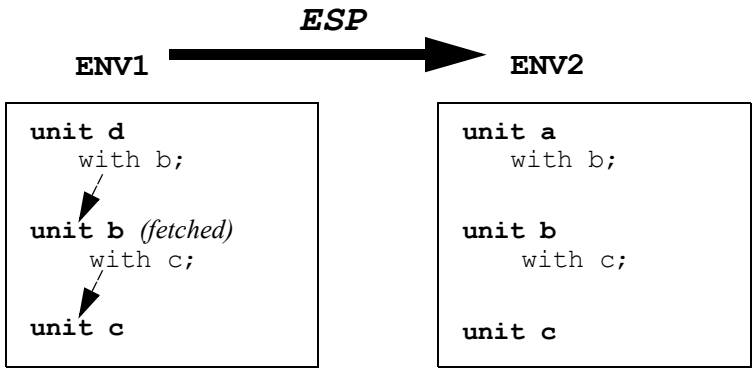



**Figure 4-2. Example of using `a.fetch` to resolve obscurities**

When using the **`a.build`** compilation utility, however, this obscurity is automatically taken care of by creating a *naturalized* copy of **`unit b`** in **`ENV1`**. The options that existed in the original copy are persistent in a naturalized copy. They can only be altered in the original environment. If you wish to change the options on a foreign unit in the local environment, you must fetch it.

The **`-from`** option allows the user to specify an environment pathname from which to fetch the unit(s). In addition, you may specify certain environments using their "keywords". See Chapter 9 for a list of these keywords.

**NOTE**

If the **`-from`** option is not specified, **`a.fetch`** will try to "find" the specified unit by searching the Environment Search Path.

The **-d** option can be used for ambiguity resolution for those environments specified with the **-from** option. If no **-from** option is specified, the **-d** option has no effect.

For example, if the user says:

```
a.fetch -from publiclib curses
```

the package **curses** would be fetched from the **/usr/ada/***release_name***/publiclib** environment due to the use of the **publiclib** keyword. However, if there exists a directory named **publiclib** in the current working directory, that directory takes precedence. The **-d** option may be used to override this behavior if, in fact, the user desires to use **/usr/ada/***release_name***/publiclib**.

For example:

```
a.fetch -d -from publiclib curses
```

always uses the **/usr/ada/...** version, whereas:

```
a.fetch    -from publiclib curses
```

fetches from the local directory if it exists or from **/usr/ada/...** otherwise.

The **a.expel** tool is provided to allow a fetched unit to be removed from the local environment, thus restoring visibility to the foreign version. See “a.expel” on page 4-21 for details.

# a.freeze

### Freeze an environment, preventing changes

The syntax of the **a.freeze** command is:

**a.freeze [***options***]**

The following represents the **a.freeze** options:

| Option | Meaning | Function |
|---|---|---|
| **-env** *env* | environment | Specify an environment pathname |
| **-q** | query | Displays an environment's frozen status and its environmental consistency |
| **-t** | transitive | Freeze specified environment and required environments |
| **-rel** *release* | release | Specify a MAXAda release (other than the default release) |
| **-u** | unfreeze | Thaw the environment, allowing changes |
| **-v** | verbose | Displays the environment(s) being frozen (or thawed) |
| **-H** | help | Display syntax and options for this function |

An environment may be frozen using the **a.freeze** utility. This changes an environment so that it is unalterable.

A frozen environment is able to provide more information about its contents than one that is not frozen. Therefore, accesses to frozen environments from other environments function much faster than accesses to unfrozen environments.

Any environment which will not be changed for a significant period of time and which will be used by other environments is a good candidate to be frozen to improve compilation performance.

# a.help

### List usage and summary of each MAXAda utility

The syntax of the **`a.help`** command is:

```
a.help
```

# a.hide

## Mark units as being persistently hidden in the environment

The syntax of the **a.hide** command is:

**a.hide [***options***]** *unit-id* **...**

The following represents the **a.hide** options:

| Option | Meaning | Function |
|---|---|---|
| **-env** *env* | environment | Specify an environment pathname |
| **-rel** *release* | release | Specify a MAXAda release (other than the default release) |
| **-l** | list | List hidden units and their corresponding source files |
| **-s** *file* | source file | Restrict operations to the source *file* |
| **-u** | undo | Make the specified hidden units visible |
| **-v** | verbose | Print message for each removed unit |
| **-H** | help | Display syntax and options for this function |

*unit-id* is defined by the following syntax:

*unit***[/***part***] | all[/***part***]**

where *part* is the **specification**, **body**, or **all**; abbreviations are accepted.

There are times when a source file may contain units other than those the user would like introduced into the environment. **a.intro** introduces all units contained within a particular source file into the environment (unless they have previously been hidden). In order to "remove" any unwanted units from the environment, the **a.hide** tool is provided. Using **a.hide**, the units specified are no longer visible to the environment.

This is also a way to resolve ambiguities. Upon introducing a unit having the same name as a previously introduced unit, MAXAda labels both units as *ambiguous*. It will then refuse to perform any operations on either of the two versions, or on any units depending on the ambiguous unit. The user will be forced to choose which of the two units should actually exist in the environment by "removing" the other. Normally, this is done with the **a.resolve** tool. However, the **a.hide** utility, in combination with the **-s** option to specify which *source_file* the unit belongs, can be applied to one of the units to resolve the ambiguity. See "Ambiguous Units" on page 3-10 for more information.

In order to reveal the unit so that it is no longer hidden, the **-u** option is provided. Also, the **-l** option is provided to list the hidden units and their corresponding source files.

These operations can also be modified with the **-s** option to operate on only those hidden units from a particular source *file*.

# a.install

### Install, remove, or modify a release installation

The syntax of the **a.install** command is:

**a.install -rel** *release* **[***options***]**

The following options are available with the **a.install** command:

| Option | Meaning | Function |
|---|---|---|
| **-a** *attr* | attribute | Set the attribute list for the selected release installation to *attr* |
| **-d** | default | Mark the selected release installation as the system-wide default |
| **-env** *env* | environment | Specify an environment pathname |
| **-f** | force | Permit the removal of the last release on the system without confirmation |
| **-H** | help | Display syntax and options for this function |
| **-i** *path* | install | Install the release located at *path* in the release database (the name is determined from the **-rel** option) |
| **-m** *path* | move | Move the selected release installation to *path* |
| **-r** | remove | Remove the specified release installation from the release database |
| **-rel** *release* | release | Specify a MAXAda release (REQUIRED) |
| **-v** | verbose | Report changes as they are made |

**NOTE**

Only the System Administrator (or a super user) can invoke **a.install** with the **-a**, **-d**, **-i**,**-m**, or **-r** options.

The **-i**, **-m**, and **-r** options may never be used together.

The **a.install** utility is the tool that allows users to register installations with the system's MAXAda database. It may be used to install, move, remove, and set attributes to installations.

When the **-i** option is given, then the MAXAda structure located at the specified path name is registered with the database as a valid installation. The name of the installation is registered as the release given by the **-rel** option. Therefore, the **-rel** option is required when using the **-i** option to install a MAXAda installation.

For example, the following command:

```
$ a.install -rel newada -d -i /somedir/ada_dir
```

assumes that **`/somedir/ada_dir`** contains a valid MAXAda directory structure and "installs" this version of MAXAda in the database as **`newada`**.

When the **`-d`** option is used, then **`a.install`** registers the installation with the database, and also marks the installation as the system-wide default installation (as in the above example).

After MAXAda is installed, it may need to be configured. See Appendix B for more information on "MAXAda Configuration".

# a.intro

### Introduce source files (and units therein) to the environment

The syntax of the **a.intro** command is:

**a.intro [***options***] [***source_file* **...]**

The following represents the **a.intro** options:

| Option | Meaning | Function |
|---|---|---|
| **-e**[**e** \| **l** \| **L** \| **v**] | errors | Control error emission style:<br>**-e** list syntax errors for files **a.intro** is unable to parse to **stdout** with related source lines<br>**-ee** embed syntax errors in files that **a.intro** is unable to parse and invoke $EDITOR<br>**-el** list source files to **stdout**, interspersed with any syntax errors — only source files that **a.intro** is unable to parse<br>**-eL** list source files to **stdout**, interspersed with any syntax errors — even source files that **a.intro** is able to parse<br>**-ev** embed syntax errors in files that **a.intro** is unable to parse and invoke **vi**<br>The default behavior is to list syntax errors to **stderr** with file name, and line and column number |
| **-env** *env* | environment | Specify an environment pathname |
| **-H** | help | Display syntax and options for this function |
| **-P** | preprocess | Inform the environment that preprocessing is always required for all source files introduced with this option |
| **-!P** | no preprocess | Inform the environment that preprocessing should never be performed for any source file introduced with this option |
| **-p**[*n*] | parallel | Use *n* parallel introductions (*n* defaults to the number of CPUs) |
| **-rel** *release* | release | Specify a MAXAda release (other than the default release) |
| **-s** *file_list* | file list | Read *file_list* for a list of files to process<br>If - is specified for *file_list*, read file list from **stdin** |
| **-V** | very verbose | Echo units encountered for each file |
| **-v** | verbose | Echo files as they are processed |

The **-P** option allows the user to specify that every *source_file* listed in this invocation of **a.intro** should be preprocessed. By default, only files with a **.pp** extension are preprocessed. This option allows files with other extensions to be preprocessed.

The **-!P** option allows the user to specify that preprocessing should not be performed for any *source_file* listed in this invocation of **a.intro**. Files with a **.pp** extension listed in

combination with this option will never be preprocessed, thereby overriding the default functionality.

The error emission options allow you to process error messages in a number of ways. Syntactic errors in the file that **a.intro** is parsing are listed to **stdout** when the **-e** option is specified. This lists only the erroneous lines with an explanation for each error.

More useful perhaps is the **-el** option which lists entire source files with errors to stdout with error messages interspersed at the positions where they occur. This option also lists the line number for each line in the source file and displays a banner with the source file's name at the top of the listing. The **-eL** option provides the same functionality but will list the source file even if no errors have occurred.

The **-ev** option embeds the errors directly into the source file, and then opens the source file with the **vi** editor. Error messages are marked with the pattern ###, and the editor is positioned in the file with the cursor at the point of the first error. Each error is marked where it is found in the file and an explanation is given. Each error line is prefixed with --, which denotes an Ada comment so that the compiler can still process that file if the error messages have not been deleted. MAXAda prompts to recheck syntax when editing is completed. The **-ee** option provides the same functionality but opens the source file with the editor designated by the EDITOR environment variable.

The **-s** option takes as its argument a *file_list* containing the names of all the files to be processed by **a.intro**. This is useful in order to introduce many files at once. Each file must be on a separate line in the *file_list*.

> If **-** is specified for *file_list*, **a.intro** uses input from **stdin**. This is provided mainly so that users can pipe output from another UNIX command to **a.intro**.

**a.rmsrc** can be used to eliminate the association of source files with the environment. **a.rmsrc** removes all knowledge of source files (and units therein) from the environment. See "a.rmsrc" on page 4-88 for more information.

# a.invalid

### Force a unit to be inconsistent thus requiring it to be recompiled

The syntax of the **a.invalid** command is:

**a.invalid [***options***] [***unit-id* **...]**

The following represents the **a.invalid** options:

| Option | Meaning | Function |
|---|---|---|
| **-env** *env* | environment | Specify an environment pathname |
| **-H** | help | Display syntax and options for this function |
| **-pragma** *file* | config pragmas | Invalidate independent configuration pragmas from the given *file* |
| **-rel** *release* | release | Specify a MAXAda release (other than the default release) |
| **-source** *file* | source file | Invalidate all units in the specified file |
| **-v** | verbose | Display a message for each invalidated unit |

*unit-id* is defined by the following syntax:

*unit***[/***part***] | all[/***part***]**

where *part* is the **specification**, **body**, or **all**; abbreviations are accepted.

The **a.invalid** tool is used to force a unit and any units that depend on it to be considered inconsistent, usually to force them to be rebuilt by **a.build**.

The **a.touch** tool is provided to allow the opposite functionality. See “a.touch” on page 4-97 for more information.

**NOTE**

The *file* specified by the **-pragma** option may only contain independent configuration pragmas.

# a.link

### Link a partition (an executable, archive or shared object file)

**INTERNAL UTILITY**

This tool is used internally by **a.build** which is the recommended utility for compilation and program generation.

**a.link** is not intended for general usage.

The syntax of the **a.link** command is:

**a.link [***options***] [***link-options***]** *partitions* **...**

The following represents the **a.link** options:

| Option | Meaning | Function |
|---|---|---|
| **-E** | elaboration | List dependent units in elaboration order, suppressing execution |
| **-env** *env* | environment | Specify an environment pathname |
| **-F** | files | List dependent files, suppressing execution |
| **-H** | help | Display syntax and options for this function |
| **-HA** | help arch | Display architectures and descriptions |
| **-HL** | help link | Display link options |
| **-i** | information | Suppress informational messages |
| **-map** | map | Display a map created by **a.map** |
| **-meth** | methods | Display the link method for each unit |
| **-o** *file* | output | Override the default output for the partition and place the output in *file* |
| **-rel** *release* | release | Specify a MAXAda release (other than the default release) |
| **-U** | units | List dependent units, suppressing execution |
| **-V** | verify | Display the link commands, suppressing execution |
| **-v** | verbose | Display links as they are done |
| **-vv** | very verbose | Display the link commands before execution |
| **-w** | warnings | Suppress warning messages |

See “Link Options” on page 4-109 for list of link options.

**NOTE**

Intermediate files are created during the linking process. If temporary file space (**/tmp**) is limited, **a.link** recognizes the TMPDIR environment variable and utilizes that location, if it is defined. This may be useful for large programs or programs with many units when **/tmp** is small or limited.

# a.ls

## List units in the environment (state, source file, dependencies, etc.)

The syntax of the **a.ls** command is:

**a.ls [***options***] [***unit-id* **...]**

The following represents the **a.ls** options:

| Option | Meaning | Function |
|---|---|---|
| **-all** | all | Include units from all environments on the Environment Search Path<br><br>**-all** is automatically assumed if *unit-id* is specified without the **-local** option (see below) |
| **-art** | artificial | Include artificial units (those created by the implementation to support generic instantiations) |
| **-b** | body | Filter candidate units to include only bodies |
| **-C***state* | compiled | Filter candidates by compilation state. *state* may be one of the following: `uncompiled`, `compiled`, `!uncompiled`, or `!compiled`<br><br>If *state* is omitted, `compiled` is used<br><br>If multiple **-C** options are specified, they are ORed together. If a not option (e.g., `!uncompiled`) is used, only one **-C** option is allowed<br><br>**NOTE:** There is no space between the **-C** option and *state*. |
| **-D** | depend! | Include all units on which the specified unit(s) directly or indirectly depend (the transitive closure)<br><br>Filtration has no effect on such inclusions |
| **-d** | depend | Include all units upon which the specified *unit-id*(s) directly depends<br><br>Filtration has no effect on such inclusions (for example, those units mentioned in a "`with`" statement for the specified *unit-id*(s)) |
| **-e** | everything | Provide an all-encompassing listing; add the following information to the verbose listing: temporary, permanent, and effective option sets, nationality (visa), home and originating environments, consistency |
| **-env** *env* | environment | Specify an environment pathname |
| **-F** | flag | Append annotations to units as follows:<br>bodies are appended with "`/b`"<br>specifications are appended with "`/s`" |
| **-f** *file* | file | Filter candidate units to include those found in the Ada source *file* |
| **-format** *fmt* | format | The information supplied for each unit is selected and formatted based on the format descriptor *fmt* |

| Option | Meaning | Function |
|---|---|---|
| **-format help** | format help | Display list of format descriptors |
| **-H** | help | Display syntax and options for this function |
| **-h** | headers | Suppress headers on long and verbose listings |
| **-i** | inconsistent | Mark units that are inconsistent with their source files or are inconsistent with units on which they depend with a trailing asterisk (*) character |
| **-l** | long | Provide a long listing including: unit's date, type, compilation state, part, and name |
| **-local** | local | Filter candidate units to include only those found in the local environment (default) |
| **-m** | main | Filter the candidate units to those which may be main subprograms |
| **-N** | name | Sort units by name in ascending order |
| **-n** | number | Include a total count of the number of units, categorized by compilation state |
| **-R** | require | Include all units that depend on the specified unit(s). Filtration has no effect on such inclusions |
| **-r** | reverse | Reverse the sorting order |
| **-rel** *release* | release | Specify a MAXAda release (other than the default release) |
| **-S** | source | List the source file instead of each unit |
| **-s** | specification | Filter the candidate units to include only specifications |
| **-t** | time | Sort the units by compilation time, i.e., most recently compiled units to least recently compiled units |
| **-u** | unit | Include all parts of the specified *unit-id*(s); include specification, body, and subunits<br><br>Filtration has no effect on such inclusions |
| **-ufile** *file* | units file | Obtain the list of *unit-id*(s) from the specified *file*; the *unit-id*(s) in *file* may include regular expressions |
| **-v** | verbose | Provide a verbose listing; add the following information to the long listing: source file, date, scope, source file name, options, etc. |
| **-1** | one, single | Display output in a single column |

*unit-id* is defined by the following syntax:

*unit***[/***part***] | all[/***part***]**

where *part* is the **specification**, **body**, or **all**; abbreviations are accepted.

Units may be specified as regular expressions as accepted by **compile(3G)**.

The behavior of **a.ls** with no options or *unit-id* specified is to list the names of all the units within the local environment (if no options are specified, **-local** is assumed). The

information is displayed in multiple columns. This can be overridden with the **-1** (single) option.

If *unit-id* is specified, the **-all** option is assumed. That is, **a.ls** will search the local environment and those on the Environment Search Path to find the given *unit-id*. However, if the **-local** option is used with a specified *unit-id*, **a.ls** will search for the *unit-id* in the local environment only.

The **-S** option lists the source file names instead of the unit names.

The **-n** option lists the units in the environment, providing a total count of the number of units and giving subtotals for `uncompiled`, `parsed`, `drafted`, and `compiled` units.

To see more information than is provided in a default listing, **a.ls** provides a number of options:

| | |
|---|---|
| **-l** | Provides a long listing consisting of the unit's name, date, type, compilation state, and part |
| **-v** | Provides a verbose listing consisting of the source file's name, source file's date, class, and any generic information in addition to the information provided by the **-l** option |
| **-e** | Provides an all-encompassing listing consisting of the temporary, permanent and effective compile option sets, the unit's nationality (visa), it's home environment, originating environment, and consistency state in addition to the information provided by the **-v** option |
| **-format** | Provides a method to display only the fields that are desired |

The options **-l**, **-v**, **-e**, **-format**, and **-1** options are mutually exclusive.

## Formatting the listing

The **-format** option to **a.ls** allows you to format the information listed for each unit based on a format descriptor, *fmt*, which takes the form:

```
"%[Modifier]Descriptor random_text %[Modifier]Descriptor..."
...
```

Characters encountered in the quoted format string which are not part of a descriptor are echoed in the output. Any character other than 'a'..'z' and '_' serve to terminate the current descriptor; any such characters are echoed.

The descriptors and their potential modifiers are listed in Table 4-2:

**Table 4-2. `a.ls -format` — Descriptors**

| Descriptor | Modifier | Meaning |
|---|---|---|
| **ambiguous** | **C** | Is the unit ambiguous: `ambiguous` or `unambiguous` |
| **artificial** | **C** | Is the unit `real` or `artificial`? (`artificial` units are created by the compiler for some generic instantiations) |
| **class** | **C** | Description of unit's library class: `library`, `subunit`, or `nested` |
| **consistent** | **CL** | Description of the unit's consistency: `consistent` or `inconsistent` (the reason is included with the **L** modifier) |
| **date** | **CL** | The date and time the unit last changed compilation state |
| **dependency_kind** | **C** | The kind of dependency another unit has on this unit: `semantic`, `opt_subp_spec`, `inline`, `optimization`, or `compiler_decision` |
| **environment** | **CLHOFN** | The pathname to the environment associated with the unit; as modified by **H**(ome), **O**(rigin), **F**(rom), or **N**(ative) |
| **generic_info** | **CQ** | Description of the unit's genericity: `generic`, `instance-of...`, or `null` |
| **item** | **C** | `package`, `subprogram`, `task` or `protected` |
| **main** | **C** | Indicates whether the unit can be a main subprogram: `yes`, `no`, or `maybe` (`maybe` indicates determination incomplete until unit is compiled) |
| **missing** | **C** | `missing` or `present` |
| **name** | **C** | The name of the unit |
| **options** | **CEPT** | The unit's effective, permanent, or temporary option set as selected by the **E**, **P**, or **T** modifier |
| **part** | **C** | The unit's part: `spec` or `body` |
| **srcdate** | **CL** | The date and time of the source file associated with the unit |

**Table 4-2. `a.ls -format` — Descriptors (Cont.)**

| Descriptor | Modifier | Meaning |
|---|---|---|
| **`srcfile`** | **`CL`** | The name of the source file associated with the unit |
| **`state`** | **`C`** | Unit's compilation state: `uncompiled`, parsed drafted, or `compiled` `missing` if the unit cannot be found |
| **`visa`** | **`CL`** | Description of the unit's passport: `native`, `fetched`, `naturalized`, or `foreign`<br><br>The **`L`** modifier appends information about the visa of a `foreign` unit (i.e. was it `naturalized` or `fetched` in the foreign environment) |

Descriptors may be abbreviated to any unique shortened form.

The modifiers have the following meanings:

**Table 4-3. `a.ls -format` — Modifiers**

| Modifier | Meaning | Description |
|---|---|---|
| **`C`** | column | Causes the current item to be padded with sufficient trailing blanks to form a column; this modifier is allowed for any descriptor |
| **`L`** | long | Causes the long-form of the item to be output: date descriptors will include microseconds; path descriptors will be forced into fully-rooted filename notation |
| **`Q`** | quiet | Curtails output of the current item if it is not applicable or has null text; otherwise `[]` would be output |
| **`E, P, T`** | options | Selects between the effective, permanent, or temporary option sets; only legal for the option descriptor |

For example, in an environment that contains the unit `hello`, the following **`-format`** option to **`a.ls`** produces the following output:

```
$ a.ls -format "%name was introduced on %srcdate"
hello was introduced on 05/01/97'15:11:25
```

## Dependent units

**`a.ls`** allows you to list those units upon which specified units depend

Consider an environment which solely contains the following source file:

```
with bar ;
package foo is
end foo ;
```

Note that the unit `foo` depends upon `bar` but unit `bar` cannot be located.

Issuing the command

```
$ a.ls -d foo
```

would result in the following output:

```
bar~   foo
```

**NOTE**

The "~" is appended to the unit name when the unit itself cannot be located and a short listing has been specified.

To see a long listing of the same:

```
$ a.ls -l -d foo
```

results in the following output:

```
Unit_Date          Item        State       Part  Name
n/a                n/a         missing     spec  bar
04/21/97'15:59:24  package     uncompiled  spec  foo
```

The **-R** option includes all units that *depend on* the specified units.

## Parts

The **-F** option to **a.ls** designates the parts of a unit by appending "/s" to unit specification names and "/b" to unit body names.

Using the above example, the following command

```
$ a.ls -F -d foo
```

would result in the following output:

```
bar/s~ foo/s
```

(Note the "~" which appears because this is a short listing and `bar` cannot be located.)

The -u option includes all parts of the specified unit-id(s). This includes the specifications, bodies, and subunits.

There are also a number of options available to filter the listing for with respect to parts:

| | |
|---|---|
| **-s** | Only list unit specifications |
| **-b** | Only list unit bodies |

## Sorting

There are a few options to **a.ls** with which to sort the output. They are:

| | |
|---|---|
| **-N** | Sort units by name in ascending order |
| **-t** | Sort units by compilation time - most recently compiled units to least recently compiled units |
| **-r** | Reverse the sorting order |

## Filtering

There are a few options to **a.ls** with which to filter the output. They are:

| | |
|---|---|
| **-C***state* | Filter units by compilation *state* - `compiled`, `uncompiled`, `!compiled`, and `!uncompiled` |
| **-f** *file* | Filter units to include only those found in the *file* |
| **-m** | Filter units to include only main subprograms |

The **-i** flag also helps to determine which units are inconsistent with their source files or are inconsistent with units on which they depend by appending a trailing "`*`" after the unit name.

# a.lssrc

### List source files associated with the environment

The syntax of the **a.lssrc** command is:

**a.lssrc [***options***] [***source-file***]**

The following represents the **a.lssrc** options:

| Option | Meaning | Function |
|---|---|---|
| **-env** *env* | environment | Specify an environment pathname |
| **-H** | help | Display syntax and options for this function |
| **-l** | long | Display pre-processing/configuration pragma information |
| **-rel** *release* | release | Specify a MAXAda release (other than the default release) |
| **-s** *file_list* | file_list | Read *file_list* for a list of files |

**a.lssrc** provides information about source files introduced to the environment. The information available via this tool is specific only to the source file. For information about units contained within the source file, the **a.ls** tool should be used. See "a.ls" on page 4-35 for more information.

With no options, **a.lssrc** provides a list of the names of all source files introduced to the environment. This includes source files that contain no units, and source files that contain only independent configuration pragmas (see "Configuration Pragmas" on page 3-9). In this respect, it differs from **a.ls -S**.

If a *source-file* name is specified on the command line or the **-s** option is used with a file containing a list of source file names, only the mentioned source files will be listed.

If the **-l** option is specified, **a.lssrc** provides additional information directly associated with the source file. This information appears enclosed in square brackets on the same line following each listed source file. The two pieces of information that can be provided are:

- `pre-processed`
- `configuration pragmas`

If the `pre-processed` indication appears, it means that the file will always be filtered by the **a.pp** tool before being compiled (see "a.pp" on page 4-77). Files introduced with the `.pp` suffix will be marked as `pre-processed` by default. Other files will not. This indication can be set or changed by the **-P** and **-!P** options to the **a.intro** tool (see "a.intro" on page 4-30).

If the `configuration pragmas` indication appears, it means that the file contains only independent configuration pragmas (see "Configuration Pragmas" on page 3-9).

This command may be useful if you wanted to completely remove your environment and later reproduce it. You might want to do:

```
a.lssrc > .source_files
```

before you remove the environment and subsequently,

```
a.intro -s .source_files
```

intro a newly-created environment.

# a.man

### Invoke/position interactive help system (requires an X terminal)

The syntax of the **a.man** command is:

**a.man [***options***] [***manual* **[***topic***]]**

The following represents the **a.man** options:

| Option | Meaning | Function |
|---|---|---|
| **-display** *disp* | X display | Select an X terminal |
| **-env** *env* | environment | Specify an environment pathname |
| **-l** | list | Lists available online manuals |
| **-man** *manpage* | man page | Display **man** page for specified *manpage* |
| **-x** | exact | Requires an exact match on specified manual or topic arguments; without this option, the help system is activated at the appropriate "bookshelf" or "find" section |
| **-rel** *release* | release | Specify a MAXAda release (other than the default release) |
| **-H** | help | Display syntax and options for this function |

**a.man** invokes the interactive HyperHelp system as directed by options and arguments. If a HyperHelp session for the user is already active, **a.man** will position the existing session to the specified topic or manual.

To see a list of the names of each online *manual* available for viewing with HyperHelp, issue:

```
$ a.man -l
```

To open a specific *manual*, issue **a.man** with the name of that *manual*:

```
$ a.man maxada
```

If the manual is not recognized (and is not interpreted as a *topic*), then HyperHelp is opened to the Bookshelf.

To view a particular *topic* within a specific *manual*, issue either that *topic* along with the *manual* in which it is contained, or the *topic* alone.

```
$ a.man maxada a.build
```

or

```
$ a.man a.build
```

will position the HyperHelp system to the description of the **a.build** command.

Topics for the MAXAda Reference Manual include the names of all MAXAda utilities, all pragmas recognized by MAXAda, all MAXAda-supplied environments, and various Ada bindings.

**NOTE**

The *topic* argument is meant as a shortcut for positioning the HyperHelp session. The list of topics recognized by **a.man** is short and obviously not meant to be comprehensive. Direct use of HyperHelp is intended for general manual browsing and selection.

If a *topic* is not recognized, but the *manual* is, HyperHelp will be positioned at the "Find" window for that *manual*.

## References to the Ada 95 Reference Manual

In addition to the MAXAda topics mentioned above, **a.man** can also position the user within a specified section of the Ada 95 Reference Manual. For instance:

```
$ a.man 1.1.5
```

will position the user at that section in the RM.

This is short for:

```
$ a.man rm 1.1.5
```

Furthermore, **a.man** can position to the exact paragraph within the RM:

```
$ a.man "1.1.5(10)"
```

**NOTE**

Due to the shell's parsing of the command line, the double-quotes may be necessary so that the topic passed includes the paragraph number between the parentheses.

This is helpful when MAXAda issues error messages with references to the Ada 95 Reference Manual. The user may enter the reference as an argument to **a.man** and view the related section.

## References to the MAXAda Reference Manual

MAXAda also issues error messages that reference the *MAXAda Reference Manual (0890516)*. These references can also be used with **a.man**. For instance, if a user encounters the following message:

```
MAX(060) 6-27: too few pragma arguments
```

this message can be issued to **a.man**:

```
$ a.man "MAX(060) 6-27: too few pragma arguments"
```

to bring up the related online help topic.

The text of the error message is not necessary, however. The user may also issue the following command:

```
$ a.man "MAX(060) 6-27"
```

or

```
$ a.man 6-27
```

to bring up the same online topic.

## Access to Support Packages

**a.man** can also be used to view the source of the packages contained in the various environments shipped with MAXAda (see Chapter 9 - "Support Packages" for more information).

The user may enter the fully-expanded name of a package and MAXAda will bring up a hyperlink to the actual source of the package. For example:

```
$ a.man ada.task_identification
```

opens HyperHelp to the position of `ada.task_identification` in the list of MAXAda-supplied packages. The entry in this list is a hyperlink to the actual `ada.task_identification` package in the `default` release installed on the system (see "a.release" on page 4-83 to find out more information about the releases installed on your system). The user may then follow this link to bring up the source in the HyperHelp viewer.

In addition, **a.man** provides shortcuts to many of these packages. In many cases, the leading "`ada.`" may be omitted for the same functionality. For example:

```
$ a.man task_identification
```

brings up the `ada.task_identification` package in the same manner as the previous command.

# a.map

### Display or edit the run-time configuration of an executable

The syntax of the **a.map** command is:

**a.map [***options***]** *executable_file*

The following represents the **a.map** options:

| Option | Meaning | Function |
|---|---|---|
| **-assoc** | associative | Use alternate associative list for output |
| **-bound** | bound | Change the default weight to bound |
| **-c** | check | Check executable for possible inconsistencies |
| **-E** | edit | Edit configuration with the editor designated by the shell environment variable `$EDITOR` |
| **-e**[**e** \| **l** \| **L** \| **v**] | errors | Control error emission style:<br>**-e** list errors in configuration file to **stdout** with related source lines<br>**-ee** embed errors in configuration file and invoke $`EDITOR`<br>**-el** list configuration file to **stdout**, interspersed with any errors — only if there are errors<br>**-eL** list configuration file to **stdout**, interspersed with any errors — even if there are no errors<br>**-ev** embed errors in configuration file and invoke **vi**<br>The default behavior is to list errors to **stderr** with file name, and line and column number |
| **-env** *env* | environment | Specify an environment pathname |
| **-g** | ghosts | Include ghost task information |
| **-H** | help | Display syntax and options for this function |
| **-i** | information | Suppress information messages |
| **-l** *file* | listing | List configuration to the specified *file*<br>If **-** is specified for *file*, list configuration to **stdout** |
| **-lock** | lock | Change the default `lock_state` to locked |
| **-rel** *release* | release | Specify a MAXAda release (other than the default release) |
| **-m** *file* | modify | Modify configuration from the specified *file*<br>If **-** is specified for *file*, read configuration from **stdin** |
| **-multiplexed** | multiplexed | Change the default weight to multiplexed (multiplexed tasks are not supported in this release) |
| **-p** | pragmas | Write example pragmas in configuration output |

| Option | Meaning | Function |
|---|---|---|
| **-r** | resolve | Resolve DEFAULT values in configuration output (such output cannot be used to modify) |
| **-s** | stacks | Associate stacks with their tasks rather than with other memory specifications |
| **-V** *file* | verify | Verify configuration from the specified *file* (this does not modify the program)<br><br>If **-** is specified for *file*, read configuration from **stdin** |
| **-v** | verbose | Emit verbose information about changes |
| **-w** | warnings | Suppress warning messages |

There are five basic areas of run-time configuration: General, Memory, Tasks, Groups, and Protected.

General area

contains configuration parameters that affect the entire run-time system, including:

- RUNTIME_DIAGNOSTICS

  see “Pragma RUNTIME_DIAGNOSTICS” on page 6-1

- QUEUING_POLICY

  see “Pragma QUEUING_POLICY” on page 6-2

- DISPATCHING_POLICY

  see “Pragma TASK_DISPATCHING_POLICY” on page 6-2

- LOCKING_POLICY

  see “Pragma LOCKING_POLICY” on page 6-3

- SERVER_CACHE_SIZE

  see “Pragma SERVER_CACHE_SIZE” on page 6-4

- TRACING_ENABLED

  see “Tracing Options” on page 11-14

- TRACING_MECHANISM

  see “Tracing Options” on page 11-14

- TRACING_BUFFERSIZE

  see “Tracing Options” on page 11-14

Memory area

contains configuration parameters for regions of memory, including:

- `pool`

  see “Pragma MEMORY_POOL” on page 6-23

- `cache_mode`

  see “Pragma POOL_CACHE_MODE” on page 6-25

- `lock_state`

  see “Pragma POOL_LOCK_STATE” on page 6-25

- `size`

  see “Pragma POOL_SIZE” on page 6-26, “Pragma STORAGE_SIZE” on page M-132, and “RM 13.11 Storage Management” on page M-59

- `pad`

  see “Pragma POOL_PAD” on page 6-28

Task area

contains configuration parameters for tasks, task types, and named task objects, including:

- `weight`

  see “Pragma TASK_WEIGHT” on page 6-9

- `priority`

  see “Pragma TASK_PRIORITY” on page 6-11

- `quantum`

  see “Pragma TASK_QUANTUM” on page 6-14

- `cpu_bias`

  see “Pragma TASK_CPU_BIAS” on page 6-12

Group area

contains configuration parameters for task groups, including:

- `servers`

  see “Pragma GROUP_SERVERS” on page 6-19

- `priority`

  see “Pragma GROUP_PRIORITY” on page 6-18

- `cpu_bias`

see “Pragma GROUP_CPU_BIAS” on page 6-19

Protected area

contains configuration parameters for protected objects, including: priorities, interrupt handlers, and attached interrupts.

- `priority`

  see “Pragma PROTECTED_PRIORITY” on page 6-28

Options are provided to:

- Produce a listing of a program's current run-time configuration, in either of two formats
- Modify a program’s run-time configuration based on a configuration file, in either of the same two formats
- Modify some aspects of the run-time configuration, without need of a configuration file

**NOTE**

One of the following options is required by **`a.map`**:
**`-c`**, **`-l`**, **`-m`**, **`-E`**, **`-V`**, **`-bound`**, **`-multiplexed`**, **`-lock`**

The **`-l`** option causes a configuration listing based on the specified program to be output to the specified file name. If − is specified for the file name, output is directed to standard output.

By default, the format of the output is in a tabular format. The tabular format lists, for each area, the appropriate program entities, one per line. Each configuration parameter associated with the program entities is listed in a particular column on that line.

If desired, an alternate format can be specified using the **`-assoc`** option. This associative format lists, for each area, all the configuration parameters for the appropriate kind of program entity together, one configuration parameter per line. This format can be slightly amended by the **`-s`** option. With the **`-s`** option, stack memory pools are listed in the Tasks area near their corresponding tasks, instead of in the Memory area.

The **`-g`** option allows ghost tasks, overhead tasks defined internally by the run-time system in certain circumstances, to be output along with user-defined tasks. See “Ghost Tasks” on page 5-5 for a description of ghost tasks.

In addition to the aforementioned formats, the **`-p`** option causes example pragmas to be emitted as comments so they do not interfere with the normal format. These pragmas correspond to the run-time configuration as detected in the program.

The **`-r`** option performs a subtle change on configuration listings. It resolves any configuration values which would be listed as DEFAULT, by using the values of the appropriate pseudo-entities. For instance, assume that in some hypothetical program, the lock_state for the DEFAULT memory region was specified as LOCKED, with the **`-r`** option. The lock_state value for every memory region which had not specified any lock_state value

would be listed as LOCKED instead of DEFAULT. Because an application of such output back into a program would cause drastic changes in the program, and because these drastic changes would most likely not be desired, configuration output produced with the **-r** option is marked in such a way that it will not be accepted by the **-m** or **-V** options. In addition, the **-r** option cannot be specified with the **-E** option.

The **-m** option causes a configuration file to be read and applied to the specified program. Either format emitted by the **-l** option is accepted as input. If **-** is specified as the file name for the **-m** option, input is read from standard input. (Neither the **-ee** nor the **-ev** options are allowed in this case because both of these options modify the configuration file and the **-** indicates that the configuration file be read from standard input.) The input need not be complete; only the particular parameters to be changed are required. In fact, entire areas can be omitted if no changes in those areas are required. Furthermore, within each area, the order of program entities is irrelevant.

The **-V** option performs the same actions as the **-m** option, except that it does not apply the actual specified changes to the program. Its purpose is to verify the contents of an input file before actually applying that input file to a program. It performs all syntactic and semantic analysis and emits the same diagnostic messages as would the **-m** option.

The **-E** option allows users to edit the run-time configuration of a program with the editor specified by the environment variable, `$EDITOR`. It performs a listing based on a particular program to a temporary file, invokes `$EDITOR` on that file, and then applies the edited temporary file to the same program. The format of the output is controlled in the same way as it is via the **-l** option, except that the **-l** is replaced by the **-E** option. Note also that no file name is required with the **-E** option.

The **-e** option invokes the **a.error** tool on any diagnostic messages emitted. This causes diagnostic messages to be emitted along with the offending line from the configuration file. This is useful for easily relating line and column information in a diagnostic message directly to the corresponding text in the configuration file.

The **-el** option is the same as the **-e** option, except that upon any error, it produces a full listing of the configuration file, instead of just the offending lines. When the **-eL** option is used, the source file is listed even if no errors have occurred.

The **-ee** and **-ev** options invoke the **a.error** tool in such a way as to cause it to insert any error messages back into the configuration file, and then invoke `$EDITOR` on that configuration file, allowing the error to be corrected. The user then has the option of applying the corrected file to the program. This can be done with the **-m**, **-V**, and **-E** options, and can be done iteratively.

The **-bound** and **-multiplexed** options set the task weight for the partitions to which they are applied, however multiplexed tasks are not supported in this release. For more information, see "Task Weights" on page 5-3 and "Pragma TASK_WEIGHT" on page 6-9.

The **-lock** option causes the lock_state for the DEFAULT memory region to become LOCKED. The result is that any memory region which has not specified its own lock_state becomes LOCKED.

The **-c** option neither performs a configuration listing nor modifies the configuration in any way. It merely performs a few sanity checks on the specified program and produces diagnostics if there are any dubious configuration values. These same checks are performed with the **-m**, **-V**, and **-E** options, but the **-c** option provides a way to perform the checks without changing the program configuration.

The **-v** option produces verbose output which details every configuration parameter that is changed via the **-m**, **-V**, or **-E** option.

The **-w** option suppresses all warning diagnostics produced by **a.map**. The **-i** option suppresses all information diagnostics produced by **a.map**.

# a.mkenv

### Create an environment which is required for compilation, linking, etc.

The syntax of the **a.mkenv** command is:

**a.mkenv [***options***] [***compile_options***] [***environment_pathname***]**

The following represents the **a.mkenv** options:

| Option | Meaning | Function |
|---|---|---|
| **-env** *env* | environment | Specify an environment pathname |
| **-f** | force | Force environment creation, even if it or some portion of it already exists |
| **-H** | help | Display syntax and options for this function |
| **-HA** | help arch | Display list of supported target architectures and descriptions |
| **-HC** | help compile | Display list of compile options |
| **-HL** | help link | Display list of link options |
| **-HQ** | help qualifier | Display list of qualifier keywords (**-Q** options)<br><br>See "Qualifier Keywords (-Q options)" on page 4-105 for more details. |
| **-oset** *opts* | link options | Set the default link options list for the environment to *opts*<br><br>Note that *opts* may need to be quoted |
| **-rel** *release* | release | Specify a MAXAda release (other than the default release) |

**a.mkenv** takes an optional *environment_pathname*. If issued with no parameters:

```
a.mkenv
```

then **a.mkenv** will attempt to create an environment in the current directory based on the default release. (See "a.release" on page 4-83 for more information regarding MAXAda releases.)

If an *environment_pathname* is given:

```
a.mkenv dir
```

then **a.mkenv** will attempt to make the directory specified by *environment_pathname* (`dir`) and, if successful, will create an environment in that directory based on the default release or the release specified by the **-rel** option.

The **-env** option is used only when an *environment_pathname* **IS NOT** specified:

```
a.mkenv -env dir
```

In this case, **a.mkenv** will attempt to create an environment in the directory specified by the *env* parameter (`dir`) based on the default release or the release specified by the **-rel** option. If an *environment_pathname* is specified, the **-env** option is ignored.

**NOTE**

If the directory specified by the *env* parameter does not exist, **a.mkenv** will fail.

The **-rel** option specifies which release of **a.mkenv** to use in creating this environment. (See "a.release" on page 4-83 for more information regarding MAXAda releases.)

The **-f** option forces creation of an environment even if one has already been created or if only a portion of it already exists. (If the **a.mkenv** tool is interrupted or fails for some reason such as not enough disk space, power failure, etc., the creation of the environment may not have completed.) Trying to recover from this failure by running the **a.mkenv** tool again may result in a message similar to the following:

```
a.mkenv: fatal: environment already exists
```

The **-f** option will force this environment to be created, thereby overriding such error messages.

The *compile_options* specified with this command become the environment-wide compile options and apply to all units introduced into this environment. (See "Environment-wide Options" on page 3-21 for more information). They may be changed by using **a.options**. They may also be overridden for particular units by permanent or temporary unit options or pragmas. See "Compile Options" on page 3-20 for a more detailed explanation of this relationship.

Use **a.mkenv -HC** for a list of *compile_options*. Also, "Compile Options" on page 4-99 provides a similar list.

Default link options for the environment are specified using the **-oset** *opts* option. Use **a.mkenv -HL** for a list of *opts*. Also, "Link Options" on page 4-109 provides a similar list.

An environment can be removed with **a.rmenv**. See "a.rmenv" on page 4-87 for details.

# a.monitor

### Monitor tasking in real-time for debugging

The syntax of the **a.monitor** command is:

**a.monitor [***options***] [***executable_file* **[***pid***]]**

The following represents the **a.monitor** options:

| Option | Meaning | Function |
|---|---|---|
| **-a**<br>**--ascii** | ascii | Print to **stdout** instead of invoking X client |
| **-env** *env* | environment | Specify an environment pathname |
| **-g**<br>**--ghosts** | ghosts | Include implementation (ghost) tasks |
| **-rel** *release* | release | Specify a MAXAda release (other than the default release) |
| **-r** *s*<br>**--rate=***s* | refresh | Set the refresh rate to *s* seconds |
| **-s**<br>**--short** | short | Provide short description of task status |
| **-h**<br>**--help** | help | Display syntax and options for this function |
| **-x**<br>**--snapshot** | snapshot | Implies **--ascii**, print snapshot and exit. |

The MAXAda **a.monitor** utility provides users with a full-screen real-time program monitor. It provides an interactive menu interface that allows users to cyclically monitor task and memory information  Currently, a **.monitor** can only monitor Ada tasking programs.

See “a.monitor” on page 12-4 for more details on the use of this utility.

# a.nfs

### Display or change NFS aspects of an environment

The syntax of the **a.nfs** command is:

**a.nfs [***options***]**

The following represents the **a.nfs** options:

| Option | Meaning | Function |
|---|---|---|
| **-env** *env* | environment | Specify an environment pathname |
| **-f** | force | Use with **-take** to take an environment that was not previously given; beware that this may expose problems caused by NFS caches if the environment was used recently from its original owner system |
| **-give** | give away | Prepare environment currently modifiable from the local system to be modifiable from another system |
| **-rel** *release* | release | Specify a MAXAda release (other than the default release) |
| **-q** | query | Query environment NFS status and owner |
| **-take** | take | Make environment modifiable from local system |
| **-write** | write | Query ability to modify environment from local system |
| **-H** | help | Display syntax and options for this function |

MAXAda supports the creation and use of environments on NFS-mounted filesystems only to a limited extent. (See "NFS Environments" on page 3-3.)

The **a.nfs** tool provides a means for determining from what system an environment can be modified, and a means for changing that system.

**a.nfs -q** displays whether or not the environment is on an NFS filesystem and what system is capable of modifying it. It responds with the output:

`status:` either `NFS` or `local`

`owner:` `local`, indicating the local system, or

`its local system`, indicating the system on which the environment's filesystem is local, or

a system name

**a.nfs -write** is a simple query to determine if the environment is writable from the current system. It will respond either with the string:

```
writable
```

or

```
not writable because ...
```

and the reason.

**a.nfs -give** and **a.nfs -take** are designed to work together to provide a means of changing the owner of an environment safely with respect to NFS caches. On the system that is the current owner, the command **a.nfs -give** should be executed. After that is done, the environment effectively is owned by no system at all. Any system can then execute **a.nfs -take** and become the new owner of that environment. By doing this in two steps it is possible to ensure that any caches with pending modifications are synchronized with the real environment file, and that any caches which might be stale are invalidated and reloaded from the real environment file.

**a.nfs -take -f** is designed to work with an environment that is owned by another system which no longer exists, is down, or is otherwise unable to modify the environment and therefore cannot execute the **a.nfs -give** command. In this case, because the current owner is unable to modify the environment, there is no possibility for the NFS caches to create problems. In that case, **a.nfs -take -f** will forcibly take an environment on which **a.nfs -give** was never run.

**CAUTION**

Never execute **a.nfs -take -f** on an environment where the current owner has modified the environment recently. Instead execute **a.nfs -give** on the current owner and then **a.nfs -take** on the new owner.

# a.options

### Set compilation options for units or the environment

The syntax of the **a.options** command is:

**a.options [***options***] [***compile_options***] [***unit-id*** ...]**

The following represents the **a.options** options:

| Option | Meaning | Function |
|---|---|---|
| **-clear** | clear | Clear all designated options for the specified entities |
| **-default** | default | Operate on the default options for the entire environment |
| **-del** | delete | Delete the designated options from the specified entities |
| **-eff** | effective | Display the effective options (based on temporary, permanent, environment defaults) |
| **-env** *env* | environment | Specify an environment pathname |
| **-fetch** | fetch | Apply the options to fetched copies (for specified units from other environments) |
| **-H** | help | Display syntax and options for this function |
| **-h** | header | Remove the header from the option list output |
| **-HC** | help compile | Display list of compile option*s* |
| **-HQ** | help qualifier | Display list of qualifier keywords (**-Q** options)<br><br>See "Qualifier Keywords (-Q options)" on page 4-105 for more details. |
| **-keeptemp** | keep temporaries | Propagate the temporary options for the units into the set of permanent options |
| **-list** | list | List the option sets for the specified entities |
| **-mod** | modify | Modify the designated options for the specified entities |
| **-perm** | permanent | Operate on the permanent options (this is the default) |
| **-rel** *release* | release | Specify a MAXAda release (other than the default release) |
| **-set** | set | Set the designated options for the specified entities |
| **-temp** | temporary | Operate on the temporary options |
| **-v** | verbose | Display a message for each change |

*unit-id* is defined by the following syntax:

*unit***[/***part***] | all[/***part***]**

where *part* is the **specification**, **body**, or **all**; abbreviations are accepted.

## Option Sets

As discussed in "Compile Options" on page 3-20, there are three different levels of options in MAXAda. These three option sets are designated by the following flags to **a.options**:

| Flag | Designation | Operates on |
|---|---|---|
| **-default** | environment-wide compile options | all units |
| **-perm** | permanent unit compile options | specified units |
| **-temp** | temporary unit compile options | specified units |

In addition, the *effective options* are derived from these three and their hierarchical relationship to one another. This set is discussed in greater detail in "Effective Options" on page 3-22.

## Listing options

The option sets may be viewed using the **-list** option to **a.options**. When issued alone, **-list** shows the permanent, temporary, and effective option sets for the units specified. For example, the following command lists those option sets for the unit `hello`,

```
$ a.options -list hello
```

By combining the **-list** option and the desired option set's flag, only that option set is displayed for the specified units. For instance, to view the permanent options for the unit `hello`,

```
$ a.options -list -perm hello
```

This only lists the permanent options for the units specified. You may specify multiple unit names, or you may use the keyword **all** to specify all units in the environment.

To list the effective options for all units in the environment,

```
$ a.options -list -eff all
```

However, this particular option does the same thing when issued alone,

```
$ a.options -eff all
```

Note that since the **-default** flag operates on all the units in the environment by definition, there is no need to specify any unit names. To list the default options,

```
$ a.options -list -default
```

## Setting options

The option sets may be initialized or reset by using the **-set** flag to **a.options**. This sets the specified options for the units designated. Any previous options for the set designated are replaced. For example,

```
$ a.options -set -perm -g hello
```

sets the debug level to `full` in the permanent option set for the unit `hello`.

If the following command is issued,

```
$ a.options -set -perm -ee hello
```

the permanent option set will only contain the **-ee** option (the previous **-g** option will have been replaced).

## Modifying options

In order to modify an option set, the **-mod** flag to **a.options** is used. This flag adds the specified options to the designated set, while retaining any other options that existed in this grouping. For instance, after the following command,

```
$ a.options -set -temp -g hello
```

the temporary option set for the unit `hello` consists of **-g**.

To add an error emission compile option to this set,

```
$ a.options -mod -temp -ev hello
```

The temporary option set for `hello` now consists of **-g** and **-ev**.

## Clearing options

All of the options may be cleared from a designated option set by using the **-clear** option to **a.options**. To clear all of the temporary options from all units in the environment,

```
$ a.options -clear -temp all
```

## Deleting options

The **-del** flag to **a.options**is more specific than the **-clear** option and allows specified options to be deleted from a particular option set.

For example, if the environment-wide compile option set (**-default**) contains **-ee**, **-!g** and **-S**, the following command,

```
$ a.options -del -!g -default
```

will remove the **-!g** option from the set and leave **-ee** and **-S** to remain as the environment-wide compile options.

## Keeping temporary options

Temporary options may be propagated into the permanent set by using the **-keeptemp** option to **a.options**. This moves the temporary options into the permanent option set and clears the temporary set. The following command does this for all units in the environment,

```
$ a.options -keeptemp all
```

See "Compile Options" on page 4-99 for more information.

Also, see the example of this in "What are my options?" on page 2-7.

## Setting options on foreign units

Options for units in foreign environments cannot be changed using **a.options** in the local environment. In order to change the options on a foreign unit, it must first be fetched. This can be done automatically by specifying the **-fetch** option in addition to the options to be applied to the foreign unit.

A fetched copy of the unit will be created in the local environment and those options specified will be applied.

# a.partition

### Define or display a partition for the linker

The syntax of the **a.partition** command is:

**a.partition [***options***] [***partitions ...***]**

The following represents the **a.partition** options:

| Option | Meaning | Function |
|---|---|---|
| **-a** | all | Display all partitions in the environment<br><br>(Normally, only those originating in the environment are displayed) |
| **-add** "*units*" | add | Add *units* to the partitions while retaining previously added units<br><br>*units* is a single parameter; the names of individual units should be comma-separated and enclosed in double quotes |
| **-addfile** *file* | add from file | As **-add**, but reads units from *file* |
| **-case** | case-sensitive | If specified, unit names will be interpreted in a case-sensitive manner |
| **-cons** | show consistency | Display consistency of each partition, with a reason if inconsistent |
| **-create** *kind* | create | Create the new named partitions as *kind* where *kind* could be `active`, `shared_object` (`so`), or `archive` (`ar`) |
| **-default** | default | Operate on the default link options list for the entire environment |
| **-del** "*units*" | delete | Delete *units* from the partitions<br><br>*units* is a single parameter; the names of individual units should be comma-separated and enclosed in double quotes |
| **-delfile** *file* | delete from file | As **-del**, but reads units from *file* |
| **-elab** *method* | elaboration method | Set the elaboration method for non-active partitions used from programs other than the active partitions<br><br>where *method* can be:<br>`none`<br>`auto`<br>`user`,*routine_name*<br>(*routine_name* is a name of the user's choice) |
| **-env** *env* | environment | Specify an environment pathname |
| **-f** | force | Force creation of existing partitions and removal of nonexistent partitions |

| Option | Meaning | Function |
|---|---|---|
| **-final** *method* | finalization method | Set the finalization method for non-active partitions used from programs other than the active partitions<br><br>where *method* can be:<br>none<br>auto<br>user,*routine_name*<br>(*routine_name* is a name of the user's choice) |
| **-file** *name* | file | Create partitions corresponding to the descriptions provided in file *name*<br><br>The syntax of the file *name* is identical to the output of the **-List** option |
| **-H** | help | Display syntax and options for this function |
| **-HA** | help arch | Display architectures and descriptions |
| **-HL** | help link | Display link options |
| **-List** | list all | Display all partitions and information about them |
| **-list** | list | List all partition names |
| **-main** *name* | main | Set the main unit for the specified active partition to *name* |
| **-o** *file* | output | Set the name of the corresponding partition output file to be created |
| **-oappend** *opts* | append options | Append *opts* to the link option list<br><br>Note that *opts* may need to be quoted |
| **-oclear** | clear options | Clear the link options list |
| **-oprepend** *opts* | prepend options | Prepend *opts* to the link option list<br><br>Note that *opts* may need to be quoted |
| **-oset** *opts* | set link options | Set the link option list to *opts*<br><br>Note that *opts* may need to be quoted |
| **-parts** *list* | partition list | Set the dependent (comma-separated) partition list for each partition |
| **-rel** *release* | release | Specify a MAXAda release (other than the default release) |
| **-remove** | remove | Remove the specified partitions |
| **-rule** *rule* | link rule | Set the link rule for the partitions (see below for syntax) |
| **-set** "*units*" | set | Add *units* to the partitions, and remove all others<br><br>*units* is a single parameter; the names of individual units should be comma-separated and enclosed in double quotes (see below) |
| **-setfile** *file* | set from file | As **-set**, but reads units from *file* |

*units* is defined by the following syntax:

**[[***unit_name***[!][,[+-]***unit_name***[!]]...** (comma-separated list)
**+** indicates an included unit (the default)

**-** indicates an excluded unit
**!** indicates all units directly or indirectly required by the given unit

**NOTE**

You may specify multiple *partitions* to **a.partition** and all *options* specified will apply to every one of those *partitions*. Each *option*, however, may only be specified once. If a particular *option* is repeated on the command line, the last occurrence of that *option* overrides all others.

Issuing **a.partition** with only a partition name and no options provides detailed information about that partition. This same information is provided for *all* partitions in the environment by specifying the **-List** option. This information includes:

- the kind of partition (**object**, **archive**, or **shared_object**)
- its resultant output file
- the link options associated with this partition
- partitions upon which this partition depends
- the link rule for this partition
- the unit designated as the main subprogram
- all included and excluded units

**NOTE**

The link options listed in this manner are those link options associated directly with the listed partitions, not their effective set. To see the environment-wide link options, use **a.partition -default**. See "Link Options" on page 3-34 for more information.

## Main Subprogram

The **-main** option to **a.partition** specifies a unit that will act as the main subprogram for an active partition. In the case where the partition has the same name as a library subprogram in the environment, that subprogram is assumed to be the main subprogram. Otherwise, no main subprogram is assumed and one must be explicitly specified using this option, if desired.

## Elaboration and Finalization

**a.partition** uses the **-elab** option to set the elaboration method for non-active partitions and the **-final** option to set the finalization method for non-active partitions.

See "Elaboration and Finalization Methods" on page 3-16 for more information.

## Case Sensitivity

The **-case** option ensures that unit names specified to **a.partition** (with the **-add**, **-addfile**, **-del**, **-delfile**, **-set**, **-setfile**, and **-main** options) will be interpreted in a case-sensitive manner. Usually, unit names are interpreted in a case-insensitive manner because Ada identifiers are case-insensitive. But some artificial units contain upper-case letters (precisely because they cannot conflict with user-specified names), so it is occasionally useful to be able to indicate those units. (See "Artificial Units" on page 3-11 for more information.)

## Consistency

The **-cons** option displays the *consistencies* of any partitions mentioned. If no partitions are mentioned, it displays the consistencies of all local partitions. In addition, you can display the consistencies of foreign partitions using the **-cons** option in combination with the **-a** option.

## Link Options

Link options are specified for a particular partition using the following options to **a.partition**:

| | |
|---|---|
| **-oset** *opts* | Set the link option list to *opts* |
| **-oappend** *opts* | Append *opts* to the link option list |
| **-oprepend** *opts* | Prepend *opts* to the link option list |
| **-oclear** | Clear the link options list |

where:

*opts* is a single parameter containing one or more link options; note that *opts* may need to be quoted.

**NOTE**

Be sure to specify the link options within the double quotes and ensure that they are specified as listed on page 4-109. For example, if the link option **-bound** is desired, the leading "**-**" must be specified as well.

For example, to set the link options for the partition **hello** to include the link options **-skipobscurity** and **-forgive**:

```
$  a.partition -oset "-skipobscurity -forgive" hello
```

Issuing **a.partition** with the partition name will show the link options for this partition:

```
$  a.partition hello
PARTITION: hello
   kind                 : active
   output file          : hello
   link options         : -skipobscurity -forgive
   dependent partitions :
   link rule            : object,archive,shared_object
   main subprogram      : hello
   included units (+)   :
      hello!
   excluded units (-)   :
```

To append a link option to this set, use the **-oappend** option:

```
$  a.partition -oappend "-trace" hello
```

The link options now will be:

```
$  a.partition hello
PARTITION: hello
   kind                 : active
   output file          : hello
   link options         : -skipobscurity -forgive -trace
   dependent partitions :
   link rule            : object,archive,shared_object
   main subprogram      : hello
   included units (+)   :
      hello!
   excluded units (-)   :
```

To clear all link options for this partition, use the **-oclear** option:

```
$  a.partition -oclear hello
```

The user may also specify link options that affect all partitions within the environment using the **-default** option in combination with those listed above.

For instance, to set the environment-wide set of link options to include the link option **-skipobscurity**:

```
$  a.partition -default -oset -skipobscurity
```

**NOTE**

The environment-wide set of link options may be set when creating the environment by using the **-oset** *opts* option to **a.mkenv** (see "a.mkenv" on page 4-53).

You may list the environment-wide set of link options by specifying:

```
$  a.partition -default
default link options: .
   -skipobscurity
```

Use **a.partition -HL** for a list of *opts*. Also, "Link Options" on page 4-109 provides a similar list.

In addition, "Link Options" on page 3-34 provides further discussion of this topic.

## Link Rule

The **-rule** option to **a.partition** sets the *link rule* for a given partition. The link rule is an ordering of the link methods which instructs the linker how to acquire each unit or system library during the linking process.

A *link method* specifies the manner in which a unit is included in the linking process. It can instruct the linker to

- use the object of a unit directly (**object** method)
- utilize the unit contained in an archive (**archive** method)
- include the unit found within a shared object (**shared_object** method)

**NOTE**

Using the object directly (the **object** method) is the most common method of utilizing units.

The link *rule* is defined by the following syntax:

*method* **[-***part***] . . . [,***method* **[-***part***] . . . ] [,***method* **[-***part***] . . . ]**

where *method* is one of the following: **object**, **archive**, or **shared_object** (or their respective abbreviations: **obj**, **ar**, **so**)

and *part* is the name of any partition, system library, or class of partitions/libraries that is

to be excluded by the linker for that particular method. Note that for each method, multiple partitions can be specified, separated by dashes (with no spaces between).

A list of *part* items to be excluded can be specified for the **archive** or **shared_object** methods. No such list can be specified for the **object** method.

To indicate that a partition name is to be excluded for a particular method, its name should be specified.

To indicate that a system library is to be excluded for a particular method, it must be specified in the form:

**-l***name*

which is the standard shorthand notation for **lib***name***.a** or **lib***name***.so**.

**NOTE**

The libraries listed as exceptions here will only affect libraries that would be included in the link implicitly. See "Implicitly-Included Libraries" on page 4-72 for more information.

To indicate that a class of partitions or libraries is to be excluded for a particular method, one of three keywords should be specified:

- **ada**
- **system**
- **user**

The **ada** keyword indicates all partitions and libraries that are part of MAXAda (those located within **/usr/ada/***release_name***/lib**). The **system** keyword indicates all libraries that are part of the PowerMAX OS operating system (those located within **/lib**, **/usr/lib**, or **/usr/ccs/lib**). The **user** keyword indicates all other libraries.

The default link rule differs for each type of partition:

| Partition | Default Link Rule |
|---|---|
| active | **object, archive, shared_object** |
| archive | **object** |
| shared_object | **object** |

The ordering of these methods within the link rule tell the linker which link method is preferred for each unit. For example, the following link rule:

```
-rule shared_object, archive, object
```

directs the linker to search first for each unit within shared object partitions visible in the current environment. It will continue to search for the unit in shared objects on the Environment Search Path until one is found. If the unit is not found within any shared objects along the Environment Search Path, the linker will search in any archive partitions in the current environment or on the Environment Search Path. Finally, if still not found, the linker will attempt to use the actual object for the unit.

In the case of system libraries, the linker will attempt to use either the shared object or archive of a system library based on the ordering of the link methods in the link rule.

The link rule is specified by the user and can combine any number of methods in any order.

In addition, the link rule can also specify certain partitions or system libraries to be passed over by the linker when searching for each unit. This allows the user greater control as to how units are included in the linking process. Specifying the **-***part* modifier after the appropriate method in the link rule instructs the linker to exclude a particular partition or system library.

To exclude the archive partition `notme` and the system library **`libux.so`** from the partition `rulexamp`, you would issue the following command:

```
a.partition -rule object,archive-notme,shared_object--lux ruleexamp
```

Note that the notation to exclude a system library is slightly different (the **-l***name* follows the **-**, appearing in the link rule as two dashes in a row).

Consider a more complicated example:

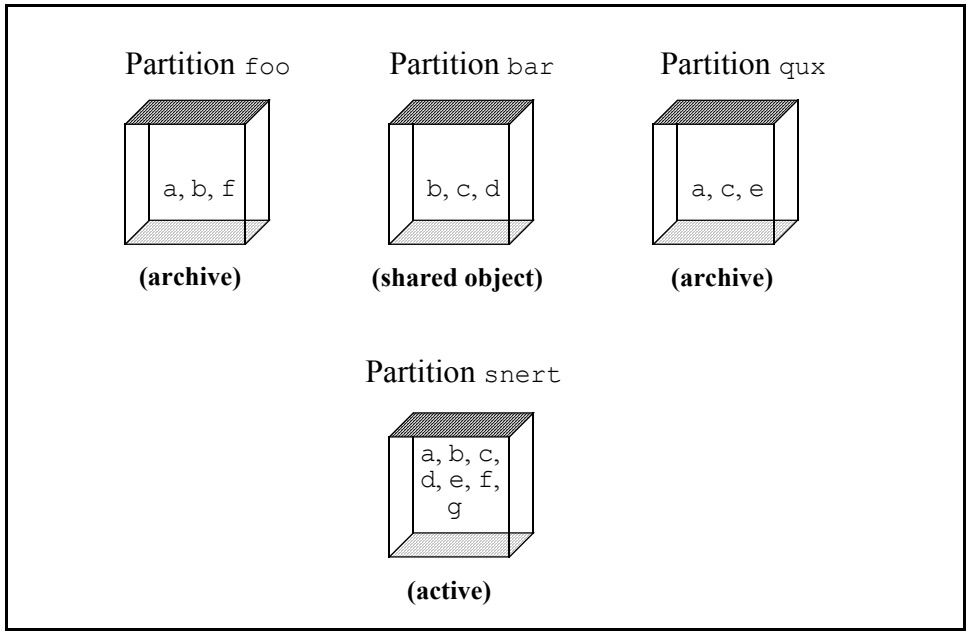



**Figure 4-3. Link Rule Example**

In Figure 4-3, the following is given:

- partition `foo` is an archive partition and contains units `a`, `b`, and `f`
- partition `bar` is a shared object partition and contains units `b`, `c`, and `d`
- partition `qux` is an archive partition and contains units `a`, `c`, and `e`
- partition `snert` is an active partition and contains units `a`, `b`, `c`, `d`, `e`, `f`, and `g`

If the following link rule is specified for partition `snert`

```
-rule archive, shared_object, object
```

the units will be used from the following partitions:

| Unit | Partition |
|---|---|
| a | foo (**ar**), qux (**ar**) |
| b | foo (**ar**) |
| c | qux (**ar**) |
| d | bar (**so**) |
| e | qux (**ar**) |
| f | foo (**ar**) |
| g | (**obj**) |

The linker tries to use the appropriate method for each unit. For example, when searching for unit `d`, the linker first looks in all archive partitions in the current environment and on the Environment Search Path. Since none of the archive partitions on the Environment Search Path contain unit `d`, the linker then searches all shared objects on the Environment Search Path. It finds unit `d` in shared object partition `bar` and uses it.

Note that the linker will decide arbitrarily which of the two partitions will be used for unit `a`.

Also note that since no archives or shared objects on the Environment Search Path contained unit `g`, the linker will use the object file for this unit.

By using the *-part* option with **`-rule`**, the determination of which methods to use for each unit can be more precise.

If the following link rule is specified for partition `snert`

```
-rule archive-foo, shared_object, object
```

the units will be used from the following partitions:

| Unit | Partition |
|---|---|
| a | qux (**ar**) |
| b | bar (**so**) |
| c | qux (**ar**) |
| d | bar (**so**) |
| e | qux (**ar**) |
| f | (**obj**) |
| g | (**obj**) |

Since `foo` was excluded as a potential archive partition, the ambiguity of which partition is to be used for unit `a` no longer exists. Also, shared object partition `bar` is used for unit `b` because there were no archive partitions that contained that unit. And lastly, since `foo` was the only partition that contained unit `f`, the linker will not be able to find this unit in any of the partitions on the Environment Search Path and therefore will use the object for unit `f`.

See also “Partitions” on page 3-12 for more information.

## Implicitly-Included Libraries

The following are libraries which may be included implicitly during the linking phase:

| | |
|---|---|
| **-lc** | required for all programs |
| **-lccur_rt** | required for tasking program or programs that use other real-time features |
| **-lgen** | required for AXI |
| **-lfrtbegin** | required for interfacing to GNU Fortran 77 |
| **-lg2c** | required for interfacing to GNU Fortran 77 |
| **-lgcc** | required for all programs |
| **-lgcc_eh** | required for all programs |
| **-lICE** | required for Xt via AXI |
| **-lm** | required for interfacing to Fortran |
| **-lMrm** | required for Motif 2.1 via AXI |
| **-lntrace** | required for programs that use NightTrace bindings or **-trace:mechanism=ntraceud** |
| **-lPW** | required for Motif 2.1 via AXI |
| **-lruntime*** | required for all programs |
| **-lsemaf** | required for tasking programs |
| **-lsemat** | required for tasking programs |
| **-lSM** | required for Xt via AXI |
| **-lX11** | required for X11R6 via AXI |
| **-lXAda** | required for AXI |
| **-lXext** | required for X11R6 via AXI |
| **-lXm** | required for Motif 2.1 via AXI |
| **-lXmAda** | required for AXI |
| **-lXmu** | required for X11R6 via AXI |
| **-lXp** | required for Motif 2.1 via AXI |
| **-lXt** | required for Xt via AXI |

In addition, on Linux, the following object files are included in the link phase:

```
/usr/lib/crt1.o
/usr/lib/crti.o
/usr/lib/crtbeginT.o
/usr/lib/crtend.o
/usr/lib/crtn.o
```

# a.path

### Display or change the Environment Search Path for an environment

The syntax of the **a.path** command is:

**a.path [***options***]**

The following represents the **a.path** options:

| Option | Meaning | Function |
|---|---|---|
| **-A** *path* | append | Append *path* to the end of the Environment Search Path |
| **-a** *path1* **[***path2***]** | append | Append *path1* after *path2*. If *path2* is not specified, this option is identical to the **-A** option |
| **-d** | default | Use the default supplied libraries |
| **-env** *env* | environment | Specify an environment pathname |
| **-f** | full path | Display full environment pathnames |
| **-H** | help | Display syntax and options for this function |
| **-I** *path* | insert | Insert *path* at the beginning of the Environment Search Path |
| **-i** *path1* **[***path2***]** | insert | Insert *path1* before *path2*. If *path2* is not specified, this option is identical to the **-I** option |
| **-P** | purge | Remove all paths in the Environment Search Path |
| **-R** *path1 path2* | replace | Replace *path1* with *path2* |
| **-r** *path* | remove | Remove *path* from the Environment Search Path |
| **-rel** *release* | release | Specify a MAXAda release (other than the default release) |
| **-t** | transitive | Display transitive closure of environments in the Environment Search Path |
| **-v** | verbose | If combined with any other **a.path** option, display the Environment Search Path after the operation is complete |
| **-w** | warnings | Suppress warning messages |
| **-x** *path* | exclude | Remove all but *path* from the Environment Search Path |

MAXAda uses the concept of an Environment Search Path to allow users to specify that units from environments other than the current environment should be made available in the current environment. See “Environment Search Path” on page 3-2 for a more complete discussion.

MAXAda supplies a number of environments with the product. These environments are listed in Chapter 9, “Support Packages”.

The **predefined** environment is automatically added to the path when **a.mkenv** is used to create an environment. Any of the other environments may be added to the path, if

desired. They can be specified by their full pathnames or by their “keywords”. See Chapter 9 for a list of these keywords.

# a.pclookup

### Filter standard input adding symbolic descriptions for pc values

The syntax of the **a.pclookup** command is:

**a.pclookup [***options***]** *executable-file*

The following represents the **a.pclookup** options:

| Option | Meaning | Function |
|---|---|---|
| **-c** *code* | code | Do not read **stdin** or process the executable file, rather immediately print symbolic description of the specified exception code |
| **-e** [*options*] | **a.error** | Pass appropriately filtered lines through **a.error**<br><br>If *options* is supplied, it is passed to **a.error** |
| **-env** *env* | environment | Specify an environment pathname |
| **-H** | help | Display syntax and options for this function |
| **-i** | insert | Insert symbolic information immediately after the program counter; normally, symbolic information is appended to the line |
| **-l** *loadmap* | load map | The file *loadmap* is expected to contain the output of **ldd(1)** and is used for executable files that use shared libraries<br><br>If not specified but needed, **a.pclookup** automatically invokes **ldd(1)** |
| **-p** *pc_value* | program counter | Print the symbolic description of the specified program counter (*pc_value*) immediately<br><br>Do not read **stdin** |
| **-r** | rooted | Show fully rooted pathnames to source<br><br>Implied by **-e** option |
| **-rel** *release* | release | Specify a MAXAda release (other than the default release) |
| **-s** | symbol | Always list the raw symbol name associated with the function containing it |
| **-t** *tag* | tag | Change the tag; used to identify all program counters to translate from the default value of `pc=` to *tag*<br><br>*tag*s are case-insensitive |

# a.pp

### Preprocess a source file

**INTERNAL UTILITY**

This tool is used internally by **a.build** which is the recommended utility for compilation and program generation.

**a.pp** is not intended for general usage.

The syntax of the **a.pp** command is:

**a.pp [***options***] [***in_source_file* **[***out_source_file***]]**

The following represents the **a.pp** options:

| Option | Meaning | Function |
|---|---|---|
| **-a** | allow | Allow blanks between prefix and directive<br><br>By default, blanks are not allowed between the prefix and the command, except those specifically placed in the prefix (e.g. "`pragma `") |
| **-b** | blank | Blank out uncompiled lines |
| **-ca** *string* | comment after | Comment uncompiled lines; end uncompiled lines with *string*<br><br>null is the default "comment after" string |
| **-cb** *string* | comment begin | Comment uncompiled lines; start uncompiled lines with *string*<br><br>"`--`" is the default "comment begin" string |
| **-D** *name* | define | Define *name* before line 1; that is, set *name*=`TRUE` |
| **-E** *name val* | equivalence | Define *name* as *val* before line 1; that is, set *name*=*val* |
| **-e** | eliminate | Eliminate uncompiled lines |
| **-env** *env* | environment | Specify an environment pathname |
| **-H** | help | Display syntax and options for this function |
| **-I** *file* | include | Include directives *file* before line 1<br><br>*file* is assumed to contain **a.pp** commands and source lines. If *file* does not specify a complete path name, it is searched for under the directory containing the file in which the **include** command is found |

| Option | Meaning | Function |
|---|---|---|
| **-l***lang* | language | Use the default options for language *lang* (`ada`, `c`) |
| | | **a.pp** defaults to Ada-mode preprocessing; however, on option it can be made to perform preprocessing using a **cpp(1)**-like syntax |
| **-m** | minimize | Eliminate blank lines |
| **-p** *prefix* | prefix | Use *prefix* as the directive prefix |
| | | "`pragma` " is the default prefix (upper or lower case - with a space after) |
| **-rel** *release* | release | Specify a MAXAda release (other than the default release) |
| **-s** | spaces | Replace uncompiled characters with spaces |
| **-U** *name* | undefine | Undefine *name* |
| **-u** | unknown | Comment unknown directives |
| | | By default, unknown directives are left without comments |

**NOTE**

If **a.pp** is invoked automatically by MAXAda, it is treated essentially as a pass of the compiler for the given source. Output from **a.pp** goes to a temporary file which is destroyed after the compilation like any other intermediate file in the compilation.

**a.pp** automatically includes the command file **.pprc** from the source file directory before any options are processed, if it exists. If file names are not given on the command line it reads from **stdin** and writes to **stdout**.

**a.pp** is not case sensitive. Any directive or argument (except file names) may be given in upper case or lower case.

Users may explicitly invoke **a.pp** to produce a preprocessed, pure Ada, **.a** source file. However, several of the MAXAda tools will automatically invoke **a.pp** on a unit if its corresponding source file name has a **.pp** suffix.

Also, if a unit is introduced with the **-P** option to **a.intro**, it will be preprocessed by **a.pp** automatically, regardless of the extension of its corresponding source file. Conversely, if a unit is introduced with the **-!P** option, it will not be preprocessed, regardless of its source file extension. (See "a.intro" on page 4-30 for more information.)

The input to the **a.pp** preprocessor consists of command lines and source lines. Command lines begin with a specific prefix, which can be set by the **-p** option. Source lines are written to the output file unchanged.

It's handy when you need the character positions within the resultant file to be the same as the character positions in the original.

The default behavior of **a.pp** (or when using the **-b** option) results in a line-to-line correspondence with the original file. The **-s** option is provided when a full character-to-character correspondence is needed.

See “Defaults” on page 4-81 for information on the default behavior of **a.pp** and the default values of its options.

See “Examples” on page 4-81 for some examples of using **a.pp**.

## Commands

Command lines begin with a specific prefix, which can be set by the **-p** option.

Commands must appear on the same line as the prefix, and there may be intervening whitespace only between the command and the prefix if the **-a** option is specified.

The following commands are available:

| Command | Description |
|---|---|
| **include** *filename* | The file *filename* is assumed to contain **a.pp** commands and source lines. If *filename* does not specify a complete path name it is searched for under the directory containing the file in which the include command is found. |
| **define** *name* [*value*] | Define the symbol *name* and optionally give it value. |
| **undefine** *name* | Remove the definition of the symbol *name*. |
| **undef** *name* | Remove the definition of the symbol *name.* |
| **ifdef** *name* | If the symbol *name* is presently defined, the following lines up to the next **a.pp else, elsifdef** or **endif** command line are written to the output file. |
| **ifndef** *name* | If the symbol *name* is not presently defined, the following lines up to the next **a.pp else, elsifndef** or **endif** command line are written to the output file. |
| **if** *expression* | If the *expression* evaluates as TRUE, the following lines are written to the output file. See the following section on expression syntax for the format of **a.pp** expressions. |
| **else**<br>**elsif** *expression*<br>**elsifdef** *name*.<br>**elsifndef** *name* | These various forms of an **else** clause specify conditional compilation boundaries. The form of the **else** clause used must match the form of its associated **if** clause. See the following section on expression syntax for the format of **a.pp** expressions. |
| **endif** | This command marks the end of a conditional compilation clause. |

| Command | Description |
|---|---|
| **substitute** *name* | This command will result in the value of the symbol *name* being written to the output file. |
| **blanks_after_prefix** | Allow blanks between the prefix and the command. |
| **no_blanks_after_prefix** | Do not allow blanks between the prefix and the command. |
| **prefix** *string* | The *string* will become the command prefix for subsequent commands. *String* must be enclosed in quotes if it contains spaces. |
| **blank_lines** | Lines that are conditionally removed from the input file will be blanked out in the output file. |
| **comment_before** *string* | Lines that are conditionally removed from the input file will be written to the output file preceded by *string*, (usually the characters for the beginning of a comment in the source language). |
| **comment_after** *string* | Lines that are conditionally removed from the input file will be written to the output file followed by *string*, (usually the comment terminator for the source language). |
| **eliminate_lines** | Lines that are conditionally removed from the input file will not be written to the output file. |
| **comment_unknown** | Lines from the input file which contain the prefix string, but are not recognized as valid **a.pp** commands, will be written to the output file as comments. |
| **no_comment_unknown** | Unrecognized command lines will be written as is to the output file. |

Commands to **a.pp** can be embedded in the actual source file, however, this is not recommended. An appropriate place for commands is the **.pprc** file, or in separate command files that may be included using the **-I** option of **a.pp**.

See the following section on expression syntax for the format of **a.pp** expressions.

## Expressions

The expression syntax of **a.pp** is similar to that of the C preprocessor. The arguments to the equality operators may be performed as character string arguments (either with or without quotes) and with *names* defined through the use of other commands. Expressions and sub-expressions can be placed within parentheses as desired.

The following operators are available in **a.pp**:

```
not
and, &&
or, ||
=, ==
/=, !=
```

**`defined(`***name***`)`** gives the functionality of **`ifdef`** and **`ifndef`** in an **`if`** directive.

## Defaults

By default,

- The command prefix is *"*`pragma `*"* (in upper or lower case)
- Unknown directives are left without comments
- **`Comment_before`** is set to "--"
- **`Comment_after`** is set to null
- Blanks are not allowed between the prefix and the command, except for the one blank already in the prefix

**NOTE**

This default mode differs slightly from the mode **`a.pp`** enters when the **`-lada`** option is given. Under that option, the prefix is set to "`pragma`", **`comment_before`** to "--", **`comment_after`** to null, and one or more spaces are allowed between the prefix and the command.

## Examples

Screen 4-5 and Screen 4-6 show possible contents of the **`.pprc`** and **`test.pp`** files, respectively.

```
pragma define long_form 1
pragma define short_form 2
pragma define ez_form 3
pragma define form short_form
```

**Screen 4-5. File .pprc**

```
     package tax_options is
pragma if form = short_form || form = ez_form
     itemize : boolean := FALSE;
pragma elsif form = long_form
     itemize : boolean := TRUE;
pragma else
     null;
pragma endif
     end tax_options;
```

**Screen 4-6. File test.pp**

If the compilation utility encounters a file that needs to be preprocessed, it automatically invokes **a.pp** with the following command line:

```
$ a.pp -lada test.pp test.a
```

and then compiles the file **test.a**. Screen 4-7 shows the contents of **test.a**.

```
       package tax_options is
-- pragma if form = short_form || form = ez_form
--        itemize : boolean := FALSE;
-- pragma elsif form = long_form
--        itemize : boolean := TRUE;
-- pragma else
--        null;
-- pragma endif
       end tax_options;
```

**Screen 4-7. File test.a**

Details on the interaction of MAXAda with **a.pp** are as follows.

- Invocations of **a.build** with a *unit_name* cause **a.build** to check all units required to make *unit_name*. If any out-of-date unit has a corresponding source file that ends with **.pp**, or the **-P** option was specified for that unit when it was introduced, it is preprocessed with **a.pp** prior to recompilation. The preprocessed source is sent to a temporary file which is removed after the recompilation. Any error messages refer to the **.pp** source file.
- The **-ev** compile option causes source errors (if any) to be embedded into the corresponding source file automatically. **a.build** will reprocess the source file if the user selects recompilation after entering the editor.
- Any MAXAda unit that was produced by a compilation involving **a.pp** will be preprocessed automatically if recompiled by **a.build**.

# a.release

## Display release installation information

The syntax of the **a.release** command is:

**a.release [***options***]**

The following represents the **a.release** options:

| Option | Meaning | Function |
|---|---|---|
| **-e** | env | Display the path of the selected environment |
| **-env** *env* | environment | Specify an environment pathname |
| **-H** | help | Display syntax and options for this function |
| **-n** | name | Display the name of the selected release |
| **-p** | path | Display the path to the selected release |
| **-q** | query | Display the selected environment and release |
| **-r** | remove | Remove the default release currently set for the invoking user |
| **-u** | user | Set the default release for the invoking user |
| **-rel** *release* | release | Specify a MAXAda release (other than the default release) |

If invoked without options, **a.release** lists all available release installations on the current host. For example,

```
$  a.release
```

provides output similar to the following:

```
The following releases are available on machine_name:

   Name              Path
   ----              ----
   ada95             /usr/ada/ada95
 * phase1            /usr/ada/phase1
   power_3.1         /usr/ada/power_3.1
   preval            /usr/ada/preval

(*) Designates the system default release

The predefined release installation, "default", is also available,
and refers to the system default release , phase1.
```

**Screen 4-8. a.release output**

The **-q** option displays the release for the specified environment (or the local environment if no environment is specified). For example,

```
$  a.release -q
```

in a MAXAda environment named **earth** provides the following output:

```
Release name   : phase1
Release path   : /usr/ada/phase1
Environment    : /env_name/earth
```

**Screen 4-9. a.release -q output**

**a.release** may be invoked with any combination of **-rel** and/or **-env** options. All remaining options are mutually exclusive, and may not be combined in a single invocation of **a.release**.

# a.resolve

### Resolve ambiguities created when a unit exists in multiple source files

The syntax of the **a.resolve** command is:

**a.resolve [***options***]** *unit-id*

The following represents the **a.resolve** options:

| Option | Meaning | Function |
|---|---|---|
| **-env** *env* | environment | Specify an environment pathname |
| **-H** | help | Display syntax and options for this function |
| **-l** | list | List the multiple source files in which unit is defined |
| **-r** *source_file* | file list | Resolve the ambiguity by selecting the unit from the *source_file*; other definitions are hidden |
| **-rel** *release* | release | Specify a MAXAda release (other than the default release) |
| **-v** | verbose | Display a message for each selected or hidden definition |

*unit-id* is defined by the following syntax:

*unit***[/***part***]**

where *part* is the **specification** or **body**; abbreviations are accepted.

Upon introducing a unit having the same name as a previously introduced unit, MAXAda labels both units as *ambiguous*. It will then refuse to perform any operations on either of the two versions, or on any units depending on the ambiguous unit. The user will be forced to choose which of the two units should actually exist in the environment by "removing" the other. This can be done using the **a.resolve** tool.

The **-r** option essentially "hides" the other units involved in the ambiguity.

See "a.hide" on page 4-27 for another way to resolve ambiguities and also "Hello Again... Ambiguous Units" on page 2-15 for an example of this type of scenario and its resolution.

# a.restore

### Restore a damaged environment

The syntax of the **a.restore** command is:

**a.restore [***options***]**

The following represents the **a.restore** options:

| Option | Meaning | Function |
|---|---|---|
| **-env** *env* | environment | Specify an environment pathname |
| **-H** | help | Display syntax and options for this function |
| **-rel** *release* | release | Specify a MAXAda release (other than the default release) |
| **-v** | verbose | Display each item restored |

In rare circumstances, an environment may become damaged. This is usually caused by a system crash or power failure that leaves files in an inconsistent state. MAXAda is unable to detect such situations, because its internal files may be corrupted in various ways. If tools consistently fail with unusual non-transient errors, and no other cause can be found for them (such as a full disk), it is possible that the environment was damaged. In that case, **a.restore** is able to recover the environment using backup information that is part of every environment. If possible, **a.restore** will restore the environment completely intact. However, if some of the backup information was damaged also, then some recompilation may be necessary for the units or partitions whose backup files were damaged and any other units or partitions that depend upon them. Cases where the backup information was damaged will be reported as warnings.

If executed when the current working directory is that of an environment, then it can be executed simply as:

**a.restore**

# a.rmenv

### Destroy an environment; compilation, linking, etc. no longer possible

The syntax of the **a.rmenv** command is:

**a.rmenv [***options***]** *environment_pathname*

The following represents the **a.rmenv** options:

| Option | Meaning | Function |
|---|---|---|
| **-env** *env* | environment | Specify an environment pathname |
| **-f** | force | Force an environment destruction, even if it or some portion of it does not exist |
| **-H** | help | Display syntax and options for this function |
| **-rel** *release* | release | Specify a MAXAda release (other than the default release) |

Removes an environment, including all units, their state information, and any partition definitions. The source files and any built partitions are left intact after this operation.

The **-f** option can be used to force an environment's destruction, even if some portion of it does not exist. For example, if the **a.mkenv** utility was interrupted during its execution (due to not enough disk space, power failure, etc.), the environment may not have been successfully created. If the environment cannot be recognized as valid, MAXAda will fail with a message similar to the following:

```
a.rmenv: fatal: invalid environment: /env_path/env_name
```

The **-f** option will force this environment to be removed, thereby overriding such error messages.

The environment can be re-created with **a.mkenv** (see page 4-53), but it will be empty and any state will have to be reconstructed by the user.

# a.rmsrc

### Remove knowledge of source files (and units therein) from the environment

The syntax of the **a.rmsrc** command is:

**a.rmsrc [***options***] [***source_file* **...]**

The following represents the **a.rmsrc** options:

| Option | Meaning | Function |
|---|---|---|
| **-all** | remove all | Remove all units in the current environment |
| **-env** *env* | environment | Specify an environment pathname |
| **-H** | help | Display syntax and options for this function |
| **-r** | remove | Remove the actual source files |
| **-rel** *release* | release | Specify a MAXAda release (other than the default release) |
| **-s** *file_list* | file list | *file_list* is assumed to be a list of files. When this option is given, **a.rmsrc** reads *file_list* and removes each file in the list<br><br>If − is given, **a.rmsrc** reads **stdin** instead |
| **-V** | very verbose | Echo removed units to **stdout** |
| **-v** | verbose | Display a message for each removed source file |

The **a.intro** tool can be used to re-associate the source files (and units therein) with the environment, but those units will be re-created in the `uncompiled` state.

# a.script

### Create script that will reproduce environment or part thereof

The syntax of the **a.script** command is:

**a.script [***options***] [***partition* **...]**

The following represents the **a.script** options:

| Option | Meaning | Function |
|---|---|---|
| **-active** | active | Script reproduces only active partitions |
| **-allparts** | all parts | Script reproduces all partitions |
| **-echo** | echo | Script echos additional progress information |
| **-env** *env* | environment | Specify an environment pathname |
| **-H** | help | Display syntax and options for this function |
| **-no_chmod** | no **a.chmod** | Script doesn't set environment permissions |
| **-no_mkenv** | no **a.mkenv** | Script doesn't reproduce environment or environment-wide options |
| **-o** *file* | output file | Create script in file *file* instead of standard output |
| **-pfile** *file* | parts file | Script reproduces only partitions listed in file *file* |
| **-rel** *release* | release | Specify a MAXAda release (other than the default release) |
| **-simple** | simple | Script issues one **a.intro** command per source file; script is slower but is easier to edit |

**a.script** generates a **sh(1)** script that can be used to re-create the current environment (or an environment specified with the **-env** option). The **sh** script, when executed, will create the environment, set up all the environment-wide options, set up the *Environment Search Path*, introduce all the same files as in the original environment, fetch all the same units, resolve all the same ambiguities, define all the same partitions, set the environment permissions, etc. The only difference between the environment created by the script and the one on which **a.script** was run is that nothing will be built in the one created by the script.

Normally, the generated script is written to standard output. The output can be redirected to a file or the **-o** option may be used to specify a filename. The specified file will be created with execute permissions.

Normally, the generated script creates the environment using the *same* release as the environment on which you ran **a.script**. The **-rel** option will override that and make the script create an environment using the release specified. This is quite useful for re-creating an existing environment after installing a new release of MAXAda.

Normally, the generated script ensures that the created environment has the same permissions as the one in which **a.script** was run. If **a.script** is run on a read-only environment, this could prove troublesome when a build is attempted in the created environ-

ment since the permissions on the created environment will be read-only as well. The **-no_chmod** option prevents the generated script from setting the environment permissions when creating the new environment so that a build may be performed in the created environment immediately.

The **-no_mkenv** skips creation of the environment. This allows the user to create the environment and set up the environment-wide options manually, only using the script to populate the environment.

Normally, the generated script issues only a small number of **a.intro** commands, specifying large numbers of source files to those few invocations. (In fact, most of the time, only a single **a.intro** command is necessary.) The **-simple** option issues one **a.intro** command per source file in the generated **sh** script. This allows the user to easily understand and modify the script after it has been generated. However, using this option results in a much slower-executing script.

The environment is created as an exact replica of the one on which **a.script** was run. If the source files were introduced with relative pathnames in the original environment, either the generated script should be executed in the same directory where **a.script** was originally run, or all the source files in the original environment should be copied to the directory where the **sh** script will be executed so that it has access to them. (Note that the source files are mentioned in the script relative to the location where the command was run, not relative to the environment.)

If partition names are passed on the command line to **a.script** or if any of the following options are specified,

- **-active** - to reproduce only active partitions
- **-allparts** - to reproduce all partitions
- **-pfile** *file* - to reproduce those partitions listed in file *file*

the generated script will avoid introducing, fetching or hiding any units in the generated environment, and will define only those partitions indicated by the above options and/or arguments. This is primarily useful for creating a new environment with all the same partition definitions as an earlier one if planning to add the original environment to the generated environment's path.

The **-echo** option makes the generated script emit more verbose information with respect to its progress when it is run.

In addition, the generated script may be executed with certain options. (See “Generated Script - Options” on page 4-91).

# Generated Script - Options

### Reproduce environment on which a.script was executed

The syntax for the script generated by **`a.script`** is:

***`generated-script`*** **`[`***`options`***`]`**

where ***`generated-script`*** is the name of the script generated by **`a.script`**.

The following represents the ***`generated-script`*** options:

| Option | Meaning | Function |
|---|---|---|
| **`-env`** *env* | environment | Specify an environment pathname |
| **`-f`** | force | Force environment creation, even if it or some portion of it already exists (similar to the **`-f`** option to **`a.mkenv`**) |
| **`-H`** | help | Display syntax and options for this function |
| **`-rel`** *release* | release | Specify a MAXAda release for environment creation |

# a.syntax

### Check the syntax of source files

The syntax of the **a.syntax** command is:

**a.syntax [***options***] [***source_file* **...]**

The following represents the **a.syntax** options:

| Option | Meaning | Function |
|---|---|---|
| **-e**[**e** \| **l** \| **L** \| **v**] | errors | Control error emission style:<br>**-e** list syntax errors for files **a.syntax** is unable to parse to **stdout** with related source lines<br>**-ee** embed syntax errors in files that **a.syntax** is unable to parse and invoke $EDITOR<br>**-el** list source files to **stdout**, interspersed with any syntax errors — only source files that **a.syntax** is unable to parse<br>**-eL** list source files to **stdout**, interspersed with any syntax errors — even source files that **a.syntax** is able to parse<br>**-ev** embed syntax errors in files that **a.syntax** is unable to parse and invoke **vi**<br>The default behavior is to list syntax errors to **stderr** with file name, and line and column number |
| **-env** *env* | environment | Specify an environment pathname |
| **-H** | help | Display syntax and options for this function |
| **-P** | preprocess | Preprocess source files before checking syntax |
| **-!P** | no preprocess | Do not preprocess source files (regardless of *source_file* extension) |
| **-rel** *release* | release | Specify a MAXAda release (other than the default release) |
| **-s** *file_list* | file list | *file_list* is assumed to be a list of files. When this option is given, **a.syntax** reads *file_list* and processes each file in the list<br>If **-** is given, **a.syntax** reads **stdin** instead |
| **-v** | verbose | Echo files as they are processed |

**a.syntax** automatically preprocesses files with a **.pp** extension, unless the **-!P** option is given. The **-P** option must be specified for files with an extension other than **.pp** that require preprocessing.

The error emission options allow you to process error messages in a number of ways. Syntactic errors in the file that **a.syntax** is parsing are listed to **stdout** when the **-e** option is specified. This lists only the erroneous lines with an explanation for each error.

More useful perhaps is the **-el** option which lists entire source files with errors to stdout with error messages interspersed at the positions where they occur. This option also lists

the line number for each line in the source file and displays a banner with the source file's name at the top of the listing. The **-eL** option provides the same functionality but will list the source file even if no errors have occurred.

The **-ev** option embeds the errors directly into the source file, and then opens the source file with the **vi** editor. Error messages are marked with the pattern ###, and the editor is positioned in the file with the cursor at the point of the first error. Each error is marked where it is found in the file and an explanation is given. Each error line is prefixed with --, which denotes an Ada comment so that the compiler can still process that file if the error messages have not been deleted. MAXAda prompts to recheck syntax when editing is completed. The **-ee** option provides the same functionality but opens the source file with the editor designated by the EDITOR environment variable.

The **-s** option takes as its argument a *file_list* containing the names of all the files to be processed by **a.syntax**. This is useful in order to check the syntax of many files at once. Each file must be on a separate line in the *file_list*.

> If **-** is specified for *file_list*, **a.syntax** uses input from **stdin**. This is provided mainly so that users can pipe output from another UNIX command to **a.syntax**.

# a.tags

### Generate a cross reference file

The syntax of the **a.tags** command is:

**a.tags [***options***]** *source_file* **...**

The following represents the **a.tags** options:

| Option | Meaning | Function |
|---|---|---|
| **-a** | append | Append to cross reference file |
| **-c** | ctags | Generate **ctags(1) tags** file (default behavior) |
| **-e** | emacs | Generate **etags(1) TAGS** file |
| **-env** *env* | environment | Specify an environment pathname<br>If the "**-env**" option is specified, the cross reference file is created in the environment specified by *env* |
| **-i** | infos | Suppress info messages |
| **-rel** *release* | release | Specify a MAXAda release (other than the default release) |
| **-s** *file_list* | file list | Read *file_list* for a list of files to process<br>If - is specified for *file_list*, read file list from **stdin** |
| **-t** | types | Create cross reference information for types also (default behavior with **-e**) |
| **-v** | vgrind | Generate **vgrind** index **tags** file |
| **-w** | warnings | Suppress warning messages |
| **-x** | cxref | Generate **cxref(1)** index to **stdout** |
| **-B** | backward | Record backward search patterns (**ctags** only) |
| **-F** | forward | Record forward search patterns (**ctags** only) |
| **-H** | help | Display syntax and options for this function |
| **-P** | preprocess | Preprocess source files |
| **-!P** | no preprocess | Do not preprocess source files (regardless of *source_file* extension) |
| **-V** | verbose | Mention source files as they are processed |

The **a.tags** command is analogous to the **ctags(1)** or **etags(1)** commands, providing cross referencing of Ada units and, optionally, of types as well. **a.tags** can be used to prepare an index of where and in which file a particular unit is defined. It also can be used to create a file usable by several editors (including **vi(1)** and **emacs(1)**) that allows a unit to be directly edited without knowing the file in which it is defined.

In Screen 4-10, the user is positioned in a MAXAda environment containing Ada source code and gets an indexed listing because of the **-x** option.

```
$ a.tags  -x  *.a
HANOI10 hanoi.aprocedure HANOI is
HANOI.DRAW_RING38 hanoi.aprocedure DRAW_RING
HANOI.DRAW_START72 hanoi.aprocedure DRAW_START is
HANOI.SOLVE93 hanoi.aprocedure SOLVE
TERMINAL2 termbody.apackage body TERMINAL is
s#TERMINAL2 termspec.package TERMINAL is
```

**Screen 4-10. a.tags -x Example**

The output gives the locations of packages, subprograms, tasks, protected units, entries, and generic units. The **-t** option adds types to the preceding 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 MAXAda environment as the previous example creates a file named **tags** containing the same type of information, but in a form readable by the **vi** editor. The **tags** file contains entries for types as well as units.

The command

```
$ a.tags  -t  -e  *.a
```

executed in the same MAXAda environment as the previous example creates a file named **TAGS** containing the same type of information, but in a form readable by the **emacs** editor. The **TAGS** file contains entries for types as well as units.

The command

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

then calls the **vi** text editor, locates the proper file, and places the cursor at the definition of the named type or unit ready for editing. While in **vi**, the command

```
:ta  TERMINAL
```

searches for the file in which `TERMINAL` is defined, enters the file, and again places the cursor at the named type or unit. Alternately, you may achieve the same effect by placing the cursor on the first character of a unit and then pressing the <CONTROL> key and the <]> key simultaneously.

The **emacs** command `find-tag`, normally bound to M-., will prompt for a unit name, and the location of a **TAGS** file the first time it is used. When they are entered, it will load the file containing the unit, and position the cursor on the line containing the declaration of the named unit or type.

Both Ada specifications and incomplete types are named by adding the prefix `s#` to the Ada name, bodies are named with the unmodified Ada name, and stubs for separates are named by adding the prefix `stub#` to the Ada name. These constructs are listed with their simple name (including the `s#` or `stub#` if present) only if that simple name is unique across all other tags. The fully expanded name is always given so that the user may search for either the simple name (if unique) or the fully expanded name.

Overloaded subprograms are *not* differentiated when generating **`ctags(1) tags`** output, or either **`vgrind`** or **`cxref(1)`** output. However, the tag can identify 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 can cause the pattern to match things other than the desired unit. Overloaded subprograms are differentiated when generating **`etags(1) TAGS`** output.

**NOTE**

When using **`vi -t`**, the desired unit or type must be in the same case (upper case or lower case) as it appeared in the source file, unless the **`vi`** `ignorecase` option is used. See the **`vi(1)`** man page.

# a.touch

## Make the environment consider a unit consistent with its source file's timestamp

The syntax of the **a.touch** command is:

**a.touch [***options***] [***unit-id ...***]**

The following represents the **a.touch** options:

| Option | Meaning | Function |
|---|---|---|
| **-env** *env* | environment | Specify an environment pathname |
| **-rel** *release* | release | Specify a MAXAda release (other than the default release) |
| **-v** | verbose | Display a message for each invalidated unit |
| **-H** | help | Display syntax and options for this function |
| **-pragma** *file* | config pragmas | Touch independent configuration pragmas from the given source *file* |
| **-source** *file* | source file | Touch all units in the specified file |

*unit-id* is defined by the following syntax:

*unit***[/***part***] | all[/***part***]**

where *part* is the **specification**, **body**, or **all**; abbreviations are accepted.

The **a.touch** tool is used to force a unit to be considered consistent with its source file, usually to keep it from being rebuilt by **a.build**. Note that it may still be considered inconsistent for other reasons, such as a required unit being changed.

The **a.invalid** tool is provided to allow the opposite functionality. See "a.invalid" on page 4-32 for details.

**NOTE**

The *file* specified by the **-pragma** option may only contain independent configuration pragmas.

# a.trace

### Format and display raw trace records

The syntax of the **a.trace** command is:

**a.trace [***options***]** *executable_file* **|** *trace_data_file* **[***task ...***]**

The following represents the **a.trace** options:

| Option | Meaning | Function |
|---|---|---|
| **-a** | ASCII | Dump ASCII trace records (read-only) |
| **-c** *file* | config | Use alternative trace config template |
| **-e** | events | Include symbolic event name with **-a** listing |
| **-env** *env* | environment | Specify an environment pathname |
| **-g** | ghost | Include ghost tasks (RTS internal tasks) |
| **-l** | list | List tasks found in data file |
| **-not_rts** | not_rts | The datafile was created by **ntraceud** without the Ada tracing runtime library (**-trace** or **-ntrace** options in the partition). Implies the **-r** option and precludes the **-a** option |
| **-r** | raw | Dump raw trace records (read-only) |
| **-rel** *release* | release | Specify a MAXAda release (other than the default release) |
| **-s** | stamp | Include timestamp in ASCII (**-a**) listing |
| **-sx** | stamp | Include raw interval_timer (hexadecimal) timestamp in ASCII (**-a**) listing |
| **-t** *file* | tasks | Only include tasks found in *file* |
| **-u** *file* | user | Include user table in *file* |
| **-v** | verbose | Include task control block addresses |
| **-w** | warning | Suppress warning messages |
| **-E** *file* | eventfile | Use alternative event map file |
| **-H** | help | Display syntax and options for this function |
| **-T** *file* | table | Use alternative trace table |
| **no options** | | Produce <prog>.ntrace.* files for **ntrace(1)** |

When "**-u** *file"* is used, event lookups will occur for all Ada events logged with the user_trace package using the "sub_id" as the table key. The format_table specified must be named "ada_user_trace".

See "Viewing Trace Events with a.trace" on page 11-24 for more details.

# Compile Options

The following options may be issued to **a.mkenv** and **a.options** (as well as **a.compile**).

See "Compile Options" on page 3-20 for a conceptual discussion of compile options within an environment.

| Option | Meaning | Function |
|---|---|---|
| **-e**[**e** \| **l** \| **L** \| **v**] | errors | Control error emission style:<br>**-e** list errors to **stdout**, with related source lines<br>**-ee** embed errors in source files and invoke $EDITOR<br>**-el** list source files to **stdout**, interspersed with any errors — only source files with errors<br>**-eL** list source files to **stdout**, interspersed with any errors — even source files with no errors<br>**-ev** embed errors in source files and invoke **vi**<br>The default behavior is to list errors to **stderr** with file name, and line and column number |
| **-g**[*level*] | debug level | Select debug level: 0 (none), 1 (lines) or 2 (full)<br>**-g** is equivalent to **-g2** |
| **-i** | info | Suppress information only messages |
| **-opp** | opportunism | Make opportunistic use of unit bodies to improve code optimization (beyond inlining) (Not supported in current release of MAXAda product) |
| **-sm** *share_mode* | share mode | Apply pragma SHARE_MODE(*share_mode*) |
| **-w** | warnings | Suppress warning and info messages |
| **-N** | not shared | Set default of pragma SHARE_BODY to FALSE |
| **-O**[*level*] | optimize | Select level of code optimization (1-3)<br>**-O** is equivalent to **-O2** |
| **-Q***keyword*[**=***value*] | qualifier | Specify a qualifier keyword |
| **-S** | suppress | Suppress checks (same as pragma SUPPRESS_ALL) |

## Negation (**!**)

Compile options may be preceded by **!** to negate them (e.g. **-!S**). This is useful for overriding a more general option that enabled them. See "Compile Options" on page 3-20.

Options which disallow arguments behave as though they were never specified when they are negated.

Options which allow arguments take on values that effectively disable them when negated:

| Negation | Equivalent to: |
|---|---|
| **-!g** | **-g0** |
| **-!O** | **-O1** |
| **-!sm** | **-sm non_shared** |

Negating a **-Q** option sets it to its default value. See "Qualifier Keywords (-Q options)" on page 4-105.

See "Compile Options" on page 4-99 for a complete list of compile options.

## Debug Level (-g[*level*])

This parameter controls the level of debug information generated for compilations in a given environment. This parameter can apply to a single unit or to an entire environment.

| | | |
|---|---|---|
| `none` | `(0)` | No debug information. Debugging tools requiring line numbers or symbolic information will not fully function on modules compiled at this debug level. |
| `lines` | `(1)` | Minimal level of debug information which provides line number information only. Debugging tools requiring symbolic information will not fully function on modules compiled at this debug level. |
| `full` | `(2)` | Full level of debug information. This is required for most debugging tools and packages to fully function. |

When new environments are created, the value of this parameter is set at the default value, `none (0)`.

If specified as a command-line parameter without a value (**-g**), the debug level is set to `full (2)`.

See "Debugging" on page 3-38 for more information.

See "Compile Options" on page 4-99 for a complete list of compile options.

## Opportunism (-opp)

Make opportunistic use of unit bodies to improve code optimization (beyond inlining). This option is not supported in the current release of MAXAda.

See “Compile Options” on page 4-99 for a complete list of compile options.

## Share Mode (`-sm`)

You control whether units are compiled for ordinary static linking (for use directly or in an archive) or as position independent code (for inclusion in a shared object). Since position independent code is not yet supported on RedHawk, the only possible value for this option is:

`non_shared` Compilations generate code that will be statically linked. Units with this share mode cannot be included in a shared object partition, but may be included in an archive or used directly in an active partition.

See “Compile Options” on page 4-99 for a complete list of compile options.

## Not Shared (`-N`)

The implementation-defined pragma `SHARE_BODY` indicates whether or not an instantiation is to be shared. For this release, the default for pragma `SHARE_BODY` is not to share any generics.

This option sets the default to not share any generics, but since this is already the default, it has no effect. The default is to share all generics that can be shared. This option negates this and sets the default to `FALSE`.

See “Pragma SHARE_BODY” on page M-129 for more information.

See “Compile Options” on page 4-99 for a complete list of compile options.

## Optimization Level (`-O[`*level*`]`)

The MAXAda compiler is capable of performing various levels of program object code optimization. There are three levels of optimization available: `MINIMAL` (**`-O1`**), `GLOBAL` (**`-O2`**), and `MAXIMAL` (**`-O3`**). Each higher level of optimization is a superset of the level of optimization which precedes it.

The quality of code produced by the compiler is representative of the level of optimization at which it was compiled.

- Optimization level `MINIMAL` produces less efficient code, but allows for faster compilation times and easier debugging.
- Level `GLOBAL` produces highly optimized code at the expense of greater compilation times.
- `MAXIMAL` is an extension of `GLOBAL` that is capable of producing even better code, but may change the behavior of the program in some cases. `MAXIMAL` attempts strength reduction operations that

may raise OVERFLOW_ERROR exceptions when dealing with values that approach the limits of the architecture of the machine.

The default for the optimization level is MINIMAL.

If specified as a command-line parameter without a value (**-O**), the optimization level is set to GLOBAL (**-O2**).

Table 4-4 shows these optimizations:

.

**Table 4-4. Levels of Optimization**

| OPTIMIZATIONS | MINIMAL (-O1) | GLOBAL (-O2) | MAXIMAL (-O3) |
|---|---|---|---|
| Short circuit boolean tests | * | * | * |
| Use of machine idioms | * | * | * |
| Literal pooling | * | * | * |
| Trivial constant folding | * | * | * |
| Binding of intermediate results to registers | * | * | * |
| Determination of optimal execution order | * | * | * |
| Simplification of algebraic expressions | * | * | * |
| Re-association of expressions to collect constants | * | * | * |
| Detections of unreachable instructions | * | * | * |
| Elimination of jumps to adjacent labels | * | * | * |
| Elimination of jumps over jumps | * | * | * |
| Replacement of a series of simple adjacent instructions by a single faster complex instruction | * | * | * |
| Constant folding | * | * | * |
| Elimination of unreachable code | | * | * |
| Insertion of zero trip tests | | * | * |
| Elimination of dead code | | * | * |
| Constant propagation | | * | * |
| Variable propagation | | * | * |
| Constraint propagation | | * | * |

**Table 4-4. Levels of Optimization (Cont.)**

| OPTIMIZATIONS | MINIMAL (-O1) | GLOBAL (-O2) | MAXIMAL (-O3) |
|---|---|---|---|
| Folding of control flow constructs with constant tests | | * | * |
| Elimination of local and global common sub-expressions | | * | * |
| Move loop invariant code out of loops | | * | * |
| Reordering of blocks to minimize branching | | * | * |
| Binding variables to registers | | * | * |
| Detection of uninitialized uses of variables | | * | * |
| Partial folding of Boolean expressions | | * | * |
| Direct branching to exception handlers | | * | * |
| Loop unrolling | | * | * |
| Register reallocation and redundant move elimination | | * | * |
| Instruction scheduling and reordering | | * | * |
| Comprehensive strength reduction | | | * |
| Test replacement | | | * |
| Induction variable elimination | | | * |
| Elimination of dead regions | | | * |

**NOTE**

Additional optimizations are performed when given the **-O3** option that do not get performed when the MAXIMAL OPT_LEVEL pragma is applied alone. This is also true with respect to the relationship between the **-O2** option and the GLOBAL OPT_LEVEL pragma. In order to take full advantage of optimization at a given level, it is recommended that the **-O** option be used instead of the pragmas.

Also, if pragma OPT_LEVEL is used to optimize code, only code within the scope of the pragma is optimized. See "Pragma OPT_LEVEL" on page M-122 for more information.

See "Compile Options" on page 4-99 for a complete list of compile options.

## Qualifier Keyword (-Q*keyword*[=*value*])

Qualifier keywords (or **-Q** options as they are more widely known) can be specified by using this option. See "Qualifier Keywords (-Q options)" on page 4-105 for a list of these options. Also, **a.options -HQ** provides this list.

See "Compile Options" on page 4-99 for a complete list of compile options.

## Suppress Checks (-S)

Suppresses all language-defined checks. Equivalent to pragma SUPPRESS_ALL.

See "Pragma SUPPRESS_ALL" on page M-133 for more information.

See "Compile Options" on page 4-99 for a complete list of compile options.

# Qualifier Keywords (-Q options)

| Keyword | Possible values | Default value |
|---|---|---|
| **inline_line_count** | (0 .. 4096) | 25 |
| **inline_nesting_depth** | (0 .. 50) | 3 |
| **inlines_per_compilation** | (0 .. 4096) | 75 |
| **inline_statement_limit** | (0 .. 16384) | 1000 |
| **opt_class** | (safe, unsafe, standard) | safe |
| **optimize_for_space** | (false, true) | false |
| **optimization_size_limit** | (0 .. 1000000) | 50000 |
| **objects** | (0 .. 10000) | 128 |
| **loops** | (0 .. 100) | 20 |
| **unroll_limit_const** | (0 .. 100) | 10 |
| **unroll_limit_var** | (0 .. 100) | 2 |
| **unroll_limit** | (0 .. 100) | 4 |
| **growth_limit** | (0 .. 10000) | 25 |
| **interesting** | (-2**31 .. (2**31)-1) | 0 |
| **target** | (ppc604) | ppc604 |
| **benchmark** | | |
| **invert_divides** | | |
| **no_component_reorder** | | |
| **no_multiply_add** | | |
| **noreorder** | | |
| **sync_volatile** | | |
| **warn_component_reorder** | | |

**inline_line_count**

The maximum number of statements allowed within an inline expanded subprogram. The default value is 25 lines. In other words, if the default is used, then only those subprograms which contain 25 or fewer lines will be expanded inline.

**inline_nesting_depth**

The maximum depth level of inline expanded subprograms. For example, if this value is 3, the compiler will perform nested inlines up to and including three levels deep. Any nested inline calls greater than three levels deep will not be expanded inline. The default value for this parameter is 3.

**inlines_per_compilation**

The maximum number of inline expansions that will be performed in a single compilation. Once this number of inline expansions has been performed for a given compilation, no other inline expansions will be performed by the compiler. The default value for this parameter is 75.

**inline_statement_limit**

The maximum number of Ada statements that will be inlined. When the running total of statements included within inline-expanded subprograms exceeds this limit, then all subsequent inline expansions will not be performed. The default value for this parameter is 1,000.

**opt_class**

Acceptable values for this parameter are `safe`, `unsafe`, and `standard`. Currently, `safe` and `standard` have the same effect. `safe` is the default value. If set to `unsafe`, additional optimizations will be performed that do not ensure that a program will perform correctly. (For instance, if set to `unsafe`, a loop test replacement may cause a program to loop infinitely).

**optimize_for_space**

A boolean value that determines whether *all* routines in a compilation will be optimized for space regardless of the values of other compiler directives. By default, this parameter is `false`.

**optimization_size_limit**

The maximum number of "expressions" that will be processed at the `GLOBAL` or `MAXIMAL` level of optimization. If this number of expressions is reached, the compiler performs all remaining optimization at level `MINIMAL`. The default value for this parameter is set at a relatively high number because the number of "expressions" processed during a compilation are not easily identified by inspection of the Ada source code. This parameter is best used as a ceiling to prevent the compiler from growing dangerously large (resulting in excessive swapping or perhaps the exhaustion of available system memory). The default value for this parameter is 50,000.

**objects**

The maximum number of objects (per routine) that will be optimized. An *object* is any scalar program variable or compiler-generated temporary variable that is a unique object in the eyes of the compiler. For example, if this number is set to 100, then only the 100 most-used objects in a given routine will be considered as "real" objects by the compiler. *Real objects* are the only objects taken into consideration by the optimizer when it comes time to perform optimizations such as copy propagation and dead-code elimination. By default, only the 128 most often used objects will be considered for optimizations.

**loops**

The maximum number of loops (per routine) that will be considered for optimization. Loop optimizations that occur at the higher levels of optimization are loop

unrolling, test replacement, strength reduction, and code motion. By default, only the 20 most deeply nested loops in a given routine will be optimized.

**unroll_limit_const=***N*

Limit the number of times a loop with a number of iterations known at compile time may be unrolled. For more information see the "Program Optimization" chapter of the *Compilation Systems Volume 2 (Concepts)* (0890460). *N* must be an integer greater than or equal to 0. The default on Series 6000 with global or maximal optimization is 10.

Note that while unrolling a loop body, the bounds of a small array may be exceeded within the copies, though the loop itself is iterated just a very few times. This will result in a "possibly exceeded array bounds" message during compilation, which will not appear if the unroll limit is set to less than the number of elements in the small array. This is an unlikely situation, and in any event the code is executed correctly.

**unroll_limit_var=***N*

Limit the number of times a loop with a number of iterations not known at compile time may be unrolled. For more information see the "Program Optimization" chapter of the *Compilation Systems Volume 2 (Concepts)* (0890460). *N* must be an integer greater than or equal to 0. The default on Series 6000 with global or maximal optimization is 2, as analysis indicates this is most profitable on Series 6000.

**unroll_limit**

This options determines the unroll limit for both constants and variables in the absence of either of **-Qunroll_limit_var** or **-Qunroll_limit_const**. Obviously, if either is specified, it overrides **-Qunroll_limit**. And if both are specified, **-Qunroll_limit** is completely ignored.

**growth_limit**

The growth limit parameter is a raw percentage that specifies the percentage increase allowed in a program's size due to the optimization performed on the program. By default, the combined effect of all optimizations which trade space for time cannot increase the size of a program by more than 25 percent. The raw percentage argument is an integer value that represents the percentage in size above 100 percent that the program may grow to be.

**interesting**

This option indicates that the default degree of interest for every object in the compilation shall be the specified value, unless the degree of interest for that object is overridden by a pragma INTERESTING in the source (see "Pragma INTERESTING" on page M-117).

**no_component_reorder**

Normally, the compiler reorders record components without representation clauses in order to better utilize memory (filling in holes in records caused by alignment, etc.). This behavior occurs even for unpacked types.

The **-Qno_component_reorder** option prevents such reordering.

**warn_component_reorder**

Normally, the compiler reorders record components without representation clauses in order to better utilize memory (filling in holes in records caused by alignment, etc.). This behavior occurs even for unpacked types.

The **-Qwarn_component_reorder** option causes the compiler to issue an info diagnostic when reordering does occur. See "Informational Messages" on page 3-31.

# Link Options

MAXAda supports a set of link options for each partition. These link options are persistent and may be specified using any of the methods discussed in "Link Options" on page 3-34.

| Option | Meaning | Function |
|---|---|---|
| **-ar=l***x* | archive | Pass a **-l***x* option to the system loader (**ld**), ensuring that it will be statically linked |
| **-bound** | bound | Set the default task weight to BOUND |
| **-elab_src** | elab source | Create a source file named "`.ELAB_`*main*`.a`" representing the elaboration of library units and execution of the main subprogram (where *main* is the unit designated as the main subprogram) |
| **-forgive** | forgive | Cause a partition to be linked despite the fact that some of its units are either not compiled or inconsistent |
| **-incr** | incremental | Allow the archive partition to be relinked incrementally |
| **-ld***file* | ld file | Pass an object file to the system loader (**ld**) |
| **-multiplexed** | multiplexed | Set the default task weight to MULTIPLEXED (multiplexed tasks are not supported in this release) |
| **-nosoclosure** | no so closure | Do not include full transitive closure of shared objects' units |
| **-skipobscurity** | skip obscurity | Skip obscurity checks |
| **-sl** | share link | Specifies a soft link from the shared object pathname to the output pathname |
| **-so=l***x* | shared object | Pass a **-l***x* option to the system loader (**ld**), ensuring that it will be dynamically linked |
| **-sp** *path* | share path | Set the shared object partition's pathname on the target system to *path* |
| **-trace[:***args***]** | trace | Activate tracing; *args* is a comma-separated list of the following options, abbreviations allowed (defaults in parentheses):<br><br>**enabled=**`true|false (true)`<br>**mechanism=**`internal[/default|rcim_tick]|`<br>`ntraceud (internal/default)`<br>**buffersize=***n* `(1000)`<br>**rtsinstrumentation=**`true|false (true)`<br>**elabinstrumentation=**`true|false (true)` |

## Share Path

The **-sp** option specifies the shared object partition's pathname on the target system. It does not cause the shared object to be created in the specified path; the shared object will still be built at the pathname specified for the target. However, all user programs created that require units from this shared object will expect the shared object to be in that location when they begin execution. The shared object must be placed at the *path* specified by **-sp** on the target system before any executables using it can be run.

With the **-sl** option, a soft link is created from the shared object's pathname to the output pathname. Using this option in conjunction with the **-sp** option removes the need for the shared object to be explicitly placed at the *path* specified by the **-sp** option.

See "Share Path" on page 3-14 for more information.

See "Link Options" on page 4-109 for a complete list of link options.

## Incrementally Updateable Partition

When **-incr** is specified on an archive partition, the result is an an incrementally updateable partition. If any units contained within this partition are changed, only those units will be updated when the partition is relinked. In order to reduce implementation overhead, the partition will be completely rebuilt if units that could have been included in the partition are removed from the environment.

The timestamp of the partition is used to determine which object files need to be replaced within it when the partition is relinked.

**WARNING**

The user must never change the timestamp of the target file for a partition configured with this option. If the target file's timestamp were changed and then relinked, the target file might contain stale object files.

See "Link Options" on page 4-109 for a complete list of link options.

## Tracing

The **-trace** option exists so that the linker can select an appropriate runtime library to link for tracing.

When linked with this tracing option, the resulting executable will generate tracing output when executed as specified by the attributes provided to this option. The syntax of this option is:

**-trace[:***args***]**

where *args* is a comma-separated list of the following options (defaults in parentheses):

```
enabled=true|false (true)
mechanism=internal[/default|rcim_tick]|
          ntraceud (internal/default)
buffersize=n (1000)
rtsinstrumentation=true|false (true)
elabinstrumentation=true|false (true)
```

Each of the above keywords may be abbreviated to any degree so long as its meaning remains unambiguous. Also, all the keywords are case-insensitive.

**NOTE**

The prefix **-trace** *is* case-sensitive like the other link options and so must be in lowercase.

The output may be analyzed with the **a.trace** utility or with the Concurrent NightTrace utility, **ntrace**, if it is available on your system. For more about tracing, see Chapter 11, specifically "Tracing Options" on page 11-14.

See "Link Options" on page 4-109 for a complete list of link options.

## Task Weight

The **-bound** and **-multiplexed** options set the task weight for the partitions to which they are applied, however multiplexed tasks are not supported in this release. These options override any other specifications such as those obtained from pragmas. For more information, see "Task Weights" on page 5-3 and "Pragma TASK_WEIGHT" on page 6-9.

See "Link Options" on page 4-109 for a complete list of link options.

## Shared Object Transitive Closure

Normally, if a shared object partition is included in another partition, then all units in that shared object partition are included, including any that might not be required because of normal Ada semantic dependence (see RM 10.1.4). As a result, further units may be included, and even further archive or shared object partitions (and this may propagate to even further units and partitions as a result of the latter). This is necessary to ensure that linker errors do not result because of undefined external symbols.

This behavior can be disabled with the **-nosoclosure** link option. If used, units in a shared object that are not required by Ada semantic dependence rules are not considered, nor are any units or partitions on which they might depend.

Caution must be exercised when using the **-nosoclosure** option, or linker errors may result. Even though any extra units in a shared object are not being considered, the shared object is being included, and those units cannot be separated from the shared object. So

any symbol references from those units must be satisfied with external symbol definitions or the system linker will produce undefined external symbol reference errors. The system linker option **-Znodefs** may be used to suppress those errors so long as they are never really used as the program executes. The link options would then include:

```
-nosoclosure -ld -Znodefs
```

**NOTE**

If **-Znodefs** was used to link, and any undefined external symbols really are referenced as the program executes, the program most probably will abort with SIGSEGV, SIGBUS, or SIGILL.

See "Link Options" on page 4-109 for a complete list of link options.

## Obscurity Checks

When a shared object partition is included in a program, it may include units in addition to those required by Ada semantics dependence (see RM 10.1.4). Also, because of shared object transitive closures (see "Shared Object Transitive Closure" on page 4-111), any units required by those additional units will be included. If the shared object partition is from a *foreign environment*, it is possible that the additional units may have been replaced in the current environment (or any environment nearer than the foreign one) with alternate versions. Those local (or nearer) versions are said to *obscure* the foreign versions.

Because shared objects must be included as a whole, those foreign versions must be included as well. **a.link** makes every possible effort to ensure that programs behave correctly in these cases, but there are a handful of cases where it is impossible, and **a.link** is forced to issue an error.

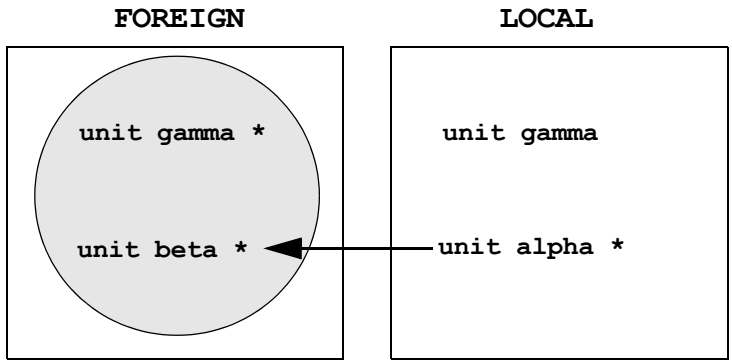
Please note in the following diagrams:

- The environments are named **LOCAL** and **FOREIGN**.
- The arrows a ---> b indicate that a requires b.
- The shaded areas indicate grouping in shared objects.
- The units marked with a ***** are included in the link.
- The unit named **alpha** is always the main unit, and is always in the *local environment*.
- The units marked with either **info** or **error** indicate that an info or error diagnostic is emitted for an obscurity associated with that unit.

### Case 1)

If the obscuring versions are not required by the program and the obscured versions are required only to satisfy the shared object transitive closure, **a.link** will include them, but they will never be used so they are harmless.

Example:

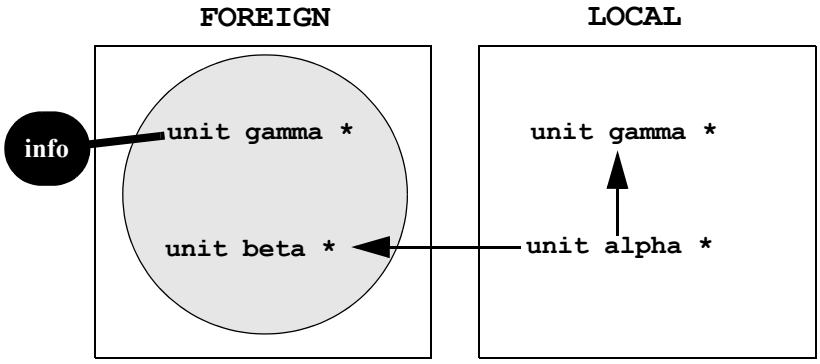



Because **gamma** is not required by Ada semantic dependence, the version in **LOCAL** will not be included in the link. The version in **FOREIGN** is never used but is included in the link because it is in the same shared object as **beta** (which is required by **alpha**).

### Case 2)

If the obscuring versions are required and the obscured versions are in required shared objects, then **a.link** will ensure via ordering of options to the system linker that the obscuring versions are used at run-time in preference to the obscured versions, even though both versions are present in the program. An informational diagnostic is emitted, but the program will work properly.

Example:

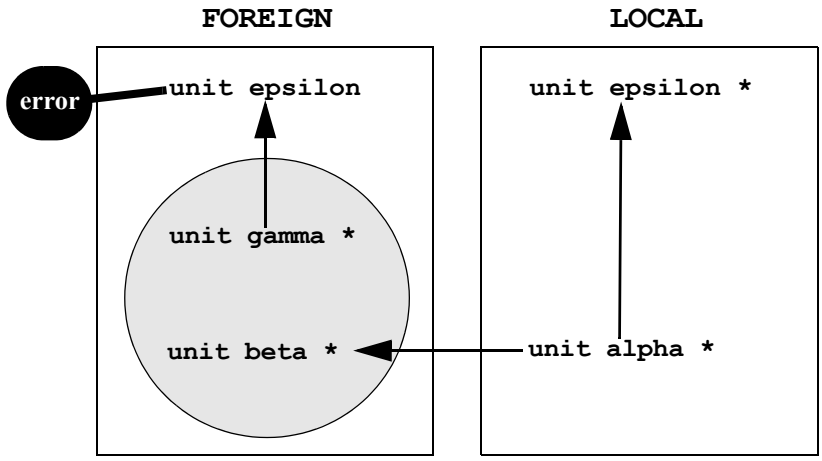



In this example, all the units will be included in the link, but MAXAda ensures that the version of **gamma** in **LOCAL** is used in preference to the version in **FOREIGN**.

### Case 3)

If the obscuring versions are required and the obscured versions are in non-shared objects, then **a.link** is forced to issue an error. This situation happens as follows:

Example:




In this case, Ada semantic dependence requires the version of **epsilon** from **LOCAL**. The shared object transitive closure (see "Shared Object Transitive Closure" on page 4-111) requires the version of **epsilon** from **FOREIGN**. Because they are both non-shared objects, it is impossible for them to coexist in the same program. So, **a.link** issues an error.

The **-skipobscurity** option will override this behavior and force the version of **epsilon** from **LOCAL** to be included in the program, and the version of **epsilon** from **FOREIGN** (and any units it requires that are not required by the version of **epsilon** in **LOCAL**) to be discarded. If the two versions of **epsilon** are not substantially different, such as when the version in **LOCAL** contains a simple bugfix and adds no new units, this will work as expected. If the two versions are substantially different, undefined external symbol references may result. In that case, the system linker option **-Znodefs** may be used to suppress those errors so long as the references are never really used as the program executes. The link options would then include:

```
-skipobscurity -ld -Znodefs
```

**NOTE**

If **-Znodefs** was used to link, and any undefined external symbols are referenced as the program executes, the program most probably will abort with `SIGSEGV`, `SIGBUS`, or `SIGILL`.

See "Link Options" on page 4-109 for a complete list of link options.

# 2
# Run-Time

**Replace with Part 2 tab**

# Part 2 - Run-Time

## Part 2 Run-Time

Chapter 5 Run-Time Concepts ..... 5-1

Chapter 6 Run-Time Configuration ..... 6-1

Chapter 7 Interrupt Handling ..... 7-1

# 5
# Run-Time Concepts


Tasking Model . . . . . 5-1
Features. . . . . 5-2
Performance . . . . . 5-2
Task Weights. . . . . 5-3
Bound Tasks . . . . . 5-3
Multiplexed Tasks . . . . . 5-3
Task Scheduling . . . . . 5-3
Task Time Slices . . . . . 5-3
Utilization of Multiple CPUs . . . . . 5-4
Ghost Tasks . . . . . 5-5
ADMIN Ghost Task . . . . . 5-5
TIMER Ghost Task . . . . . 5-5
Priorities . . . . . 5-5
OS Scheduling Policies . . . . . 5-7
Policy Selection by the Non-Tasking Run-Time. . . . . 5-8
Policy Selection by the Tasking Run-Time . . . . . 5-8
Restrictions for Priorities in the System.Interrupt_Priority Range . . . 5-9
Memory Management . . . . . 5-11
Text Memory . . . . . 5-11
Data Memory . . . . . 5-11
Collection Memory. . . . . 5-11
Stack Memory . . . . . 5-12
Other Memory . . . . . 5-12
Visibility of Memory . . . . . 5-13

# 5
# Run-Time Concepts

The MAXAda run-time system, also called the Ada Real-time Multiprocessor System (ARMS), is a flexible run-time system which has been designed to meet the needs of a wide range of Ada applications, including: time-sharing, low-priority, single threaded applications, multi-program shared-memory applications, and the most critical, real-time, multi-processor, multi-tasking applications.

It includes the following features:

- Implementation of all Ada language-defined run-time features
- Memory management
- Automatic distribution of tasks across CPUs
- True parallel task execution
- Predictable task scheduling
- Hardware and software interrupt handling
- Static and dynamic configuration control

## Tasking Model

The multithreaded, preemptive run-time executive supports standard Ada tasking as defined by ANSI/ISO/IEC-8652:1995.

Within this part of this manual, *program* or *application* refers to the `ENVIRONMENT` task (the `main` subprogram) and the entire set of Ada tasks that are included in the Ada program as defined by its dependencies (e.g., a single executable image on disk, such as **`a.out`**).

On RedHawk Linux targets, tasks are executed by a *clone*. A clone is an operating system process which shares almost all of its attributes with its parent, including: the address space, file descriptors, signal actions, etc. See **`clone(2)`** for more information.

The basic execution entity in the tasking model is a *server*. A server is an anonymous entity that actually executes on a CPU. Servers are implemented as clones. Servers are identified by entities called *server groups*, which are collections of one or more servers. Server groups are considered the execution resources that are available to Ada tasks. Server groups can be either named or anonymous, depending on their usage.

Ada tasks are assigned servers based on their task *weight*, which is either *bound* or *multiplexed*.

By default, all Ada tasks, including the ENVIRONMENT task (main subprogram), have unique clones dedicated for their execution. This is termed a completely *bound* configuration.

**NOTE**

Technically, the ENVIRONMENT task is not a clone, as it is the original operating system process that was created to represent the program (i.e. **exec(2)**). This distinction is not important with respect to this manual.

Alternatively, a completely *multiplexed* configuration specifies that all Ada tasks, including the ENVIRONMENT task (main subprogram), share the resources of a single pool which is served by a single clone. The number of clones which serve that pool is configurable.

**NOTE**

Multiplexed task weights are not currently implemented.

Many configuration options exist which provide for a mixture of the multiplexed and bound configuration models, even within a single application. See "Pragma TASK_WEIGHT" on page 6-9.

## Features

Tasking is implemented to meet the following requirements:

- Compliance with ANSI/ISO/IEC-8652:1995
- Highest possible performance
- Predictable task scheduling
- Sensible utilization of multiple CPUs
- Flexible tasking model configuration

## Performance

The run-time executive achieves its high performance task rendezvous speeds by minimizing kernel interaction during inter-clone communication and synchronization. Specialized kernel-free semaphores combined with low-contention, multithreaded use of system client/server services provide unequaled task performance.

# Task Weights

Every task in an Ada program has an attribute called its *weight*. There are two categories of weight: bound, and multiplexed.

## Bound Tasks

*Bound* tasks are served by anonymous server groups, each containing exactly one server. As each bound task is activated, its anonymous server group and server are created, and begin to execute the task. The newly created server group exists only to execute the single task for which it was created. It will never execute any other task. When the task terminates, its server group is destroyed. The servers contained in that group may be cached in the server cache for inclusion in other server groups later (see "Pragma SERVER_CACHE_SIZE" on page M-129) or simply destroyed. Server groups associated with bound tasks can be configured only by referencing their tasks.

## Multiplexed Tasks

Multiplexed task weights are not currently implemented.

# Task Scheduling

Ada tasks are cooperatively scheduled by the run-time executive and the real-time PowerMAX OS kernel. Task activation, rendezvous, and termination are implemented using real-time synchronization services designed for Ada tasking. Such scheduling adheres to the requirements set forth in RM D.2.1 (The Task Dispatching Model).

When the task dispatching policy is set to `FIFO_WITHIN_PRIORITIES`, scheduling occurs as per RM D.2.2. Other scheduling policies are described in "Pragma TASK_DISPATCHING_POLICY" on page 6-2.

## Task Time Slices

Apart from activation, rendezvous, abort, delay, termination, and priority, task scheduling is also dependent on its time slice. A task's time slice is determined by its quantum attribute. A *quantum* is the length of time an entity actually spends executing on an execution resource before begin preempted by other entities at the same priority waiting to run.

Under the `FIFO_WITHIN_PRIORITES` task dispatching policy, tasks quanta are required to be infinite (i.e. tasks are never preempted by other tasks or programs executing at the same priority).

Otherwise, the value of a task's quantum can be changed via pragmas (see "Pragma TASK_QUANTUM" on page 6-14) and run-time calls (see the `RUNTIME_CONFIGURATION` package in "vendorlib" on page 9-8).

The effect of task quanta are dependent on the task dispatching policy in effect, as discussed in the following sections.

## Utilization of Multiple CPUs

By default, servers are automatically distributed across all available CPUs on the system. However, applications are also provided precise control over server distribution with the concept of the CPU bias.

A *CPU bias* is a mask in which the relative bit number identifies a CPU number (LSB corresponds to CPU #0). For example:

| CPU Bias | Effect |
|---|---|
| 2#00000100# | Server will be bound to CPU #2 |
| 2#01000010# | Server allowed to execute on CPUs #6 & #1 |
| 2#11111111# | Server allowed to execute on all 8 CPUs |

Note that when more than 1 bit is set in a CPU bias, the kernel continually employs CPU load-balancing techniques and migrates the server to the least busy CPU specified in the bias.

If the application is utilizing physical "local memory" pools, the kernel's load-balancing algorithms will not automatically migrate tasks to CPUs without direct access to those physical pools. However, explicit task CPU bias specifications will allow such migration. See "Pragma MEMORY_POOL" on page M-120 for information about the implementation-defined pragma `MEMORY_POOL` and physical "local memory" utilization.

**NOTE**

Specifying a CPU bias of zero causes a run-time diagnostic to be emitted and preserves the CPU bias inherited from the environment.

See "Pragma TASK_CPU_BIAS" on page 6-12 and "Pragma GROUP_CPU_BIAS" on page 6-19 as well as **`cpu_bias(2)`** for more information on CPU biases.

**NOTE**

Hyper-threading is a feature of the Intel Pentium Xeon processor that allows for a single physical processor to appear to the operating system as two logical processors ("sibling CPUs"). It is important to note that hyper-threading affects CPU utilization. For example, if two tasks are scheduled with each one bound to a specific sibling CPU, each will be affected by the execution of the other since they share the same physical CPU. Refer to the section titled "Hyper-threading" in Chapter 2 of the *RedHawk™ Linux® User's Guide* (0898004).

# Ghost Tasks

*Ghost tasks* are tasks artificially created by the run-time executive for various internal purposes. They are solely for the use of the run-time executive and do not ever execute any user code. However, it is sometimes useful to know of their existence and to know what language constructs may cause them to exist. MAXAda also allows certain attributes associated with them to be configured as with ordinary tasks.

MAXAda currently has five kinds of ghost tasks:

- ADMIN (See “ADMIN Ghost Task” on page 5-5.)
- TIMER (See“TIMER Ghost Task” on page 5-5.)
- SHADOW (See “SHADOW Ghost Tasks” on page 7-4.)
- COURIER (See “COURIER Ghost Tasks” on page 7-3.)
- INTR_COURIER (See “INTR_COURIER and COURIER Ghost Tasks” on page 7-5.)

## ADMIN Ghost Task

The *ADMIN ghost task* exists only in programs that contain tasking (other than the `ENVIRONMENT` task). If it exists, it is a bound task that is responsible for the creation of all named server groups and for the creation of the `ENVIRONMENT` task. It also detects the termination of all other tasks and performs cleanup operations on those tasks, including deallocation of memory associated with those tasks.

## TIMER Ghost Task

The *TIMER ghost task* exists only in programs that contain multiplexed tasks (other than the `ENVIRONMENT` task). If it exists, it is a bound task that is responsible for all timing operations associated with multiplexed tasks. These operations include `delay` statements, `select` statements with `delay` alternatives, timed entry calls, and preemption based on time-slices. The TIMER task acts as an “alarm clock” that triggers rescheduling events when certain times have been reached because of these operations.

# Priorities

The Ada95 language defines priorities in terms of the discrete subtypes defined in the package `System`. The subtype `any_priority` spans the entire priority range supported by the implementation while the subtypes `priority` and `interrupt_priority` divide that range into standard user-level priorities and interrupt priorities (those which require the blocking of one or more interrupts).

Figure 5-1 is a graphical representation of the various RedHawk Linux scheduling policies and associated priority mappings. (Cross-hatched priority ranges are not available.)

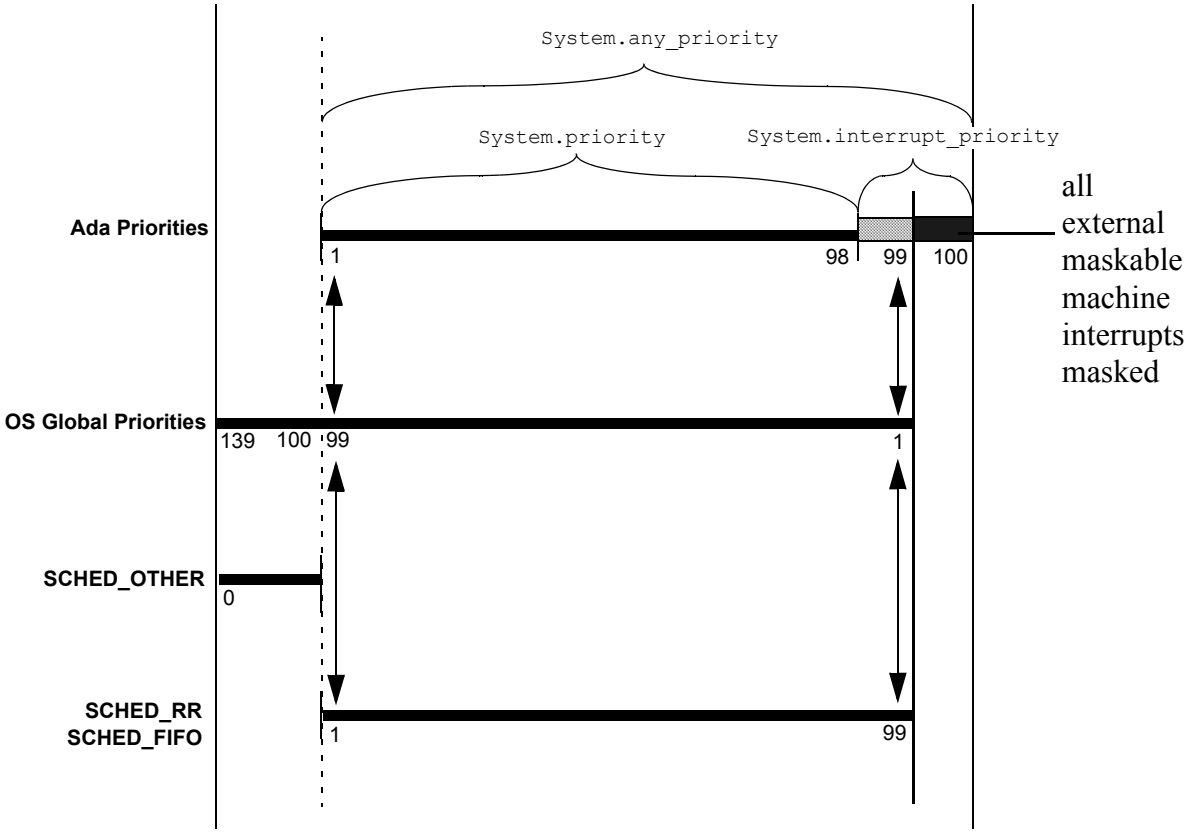



**Figure 5-1. Mapping of Various Priority Interpretations on RedHawk Linux**

RedHawk Linux offers three different POSIX-compliant scheduling policies; one for normal non-critical processes (SCHED_OTHER), and two fixed-priority policies for real-time applications (SCHED_RR and SCHED_FIFO).

The RedHawk kernel internally has a global priority range that includes values from 1..139. The priority ranges of each of the POSIX scheduling policies are distributed across the range of global priorities. The manner in which they are distributed is dependent on the policy.

Processes are scheduled by their global priority value, with 1 being the most urgent priority, and 139 the least urgent. Priorities within the global priority range 1..99 are reserved for the SCHED_RR and SCHED_FIFO policies. Priorities in the range 100..139 are reserved for the SCHED_OTHER policy.

SCHED_OTHER is the default policy; it employs a universal time-sharing algorithm designed to favor interactive processes. The range of SCHED_OTHER priorities is 0..0. A SCHED_OTHER process is initially given a global priority midway between the global priority range 100..139. The global priority is adjusted by the kernel as the process executes, but it will remain within the 100..139 range. The **nice(2)** value associated with such processes is related to its position within the 100..139 range.

SCHED_RR and SCHED_FIFO policies are most appropriate for processes with strict timing constraints. Priorities in the SCHED_RR and SCHED_FIFO policies are mapped into the global priority range 1..99 and remain fixed -- they are not adjusted by the kernel as the process executes. The mapping of these priorities is a reverse-linear relationship; SCHED_FIFO priority 99 is mapped to global priority 1, the most urgent, whereas SCHED_FIFO priority 1 is mapped to global priority 99, the least urgent in the real-time range.

The SCHED_RR policy differs from SCHED_FIFO policy only insomuch as SCHED_RR processes are round-robin scheduled. They will be preempted by other processes of the same priority once their quanta (a.k.a. time-slice) has expired. SCHED_FIFO processes are never preempted by other processes of the same priority due to time-slicing. They continue to execute on the CPU until they yield the CPU, require an unavailable system resource (e.g. a page needed to fetched from disk), or are preempted by higher priority processes.

The priority range defined by the Ada package System is 1..100. The System priorities in the range 1..99 are mapped to priorities in the POSIX scheduling policies based on the task dispatching policy (see "Policy Selection by the Tasking Run-Time" on page 5-8).

Use of System.interrupt_priority'last (100) is reserved for protected actions. All external maskable machine interrupts are masked during such actions. Programs which use this priority value must lock their address space in memory (e.g. pragma Pool_Lock_State(default,locked)) and must exercise exterme care inside protected actions. Misuse of this priority value can cause system panics and/or have significant effects on system performance and determinism.

By default, as per the Ada95 language standard, in the absence of a Priority pragma, all Ada tasks execute at priority 49.

Non-tasking Ada programs do not specifically set their priority in any way; they inherit the priority and scheduling policy of the invoking program, typically the shell.

See the *RedHawk Linux User's Guide* (0898004) for more detailed information about process scheduling and scheduling policies.

## OS Scheduling Policies

The operating system schedules *clones* based on their scheduling policy and priority.

The selection of the scheduling policy depends upon which run-time executive is used by the program:

- non-tasking run-time executive - (see "Policy Selection by the Non-Tasking Run-Time" on page 5-8)
- tasking run-time executive - (see "Policy Selection by the Tasking Run-Time" on page 5-8)

Once a task's priority has been determined, the run-time executive selects the most appropriate operating system scheduling policy for its server (clone). This selection depends upon the presence or absence of Ada tasking and other real-time Ada features

## Policy Selection by the Non-Tasking Run-Time

For programs that do not utilize any of the features that require the tasking run-time (see “Run-Time Systems” on page 1-5), the non-tasking run-time is employed. The non-tasking run-time does not alter the operating system scheduling policy or priority of the program in any way.

Note that technically, as required by the language, the main subprogram (`ENVIRONMENT` task) is still executing at the Ada priority which is midway within the range defined by `System.Priority`. However, with the non-tasking run-time, that priority has no relation to the operating system priority of the clone serving the main subprogram; it has no effect on the scheduling of the program with respect to other programs on the system. Thus for these “non-tasking” programs, the operating system scheduling policy and priority is determined by the spawning process, normally the shell, which usually selects the `SCHED_OTHER` (interactive) POSIX scheduling policy with an initial priority of zero.

## Policy Selection by the Tasking Run-Time

The tasking run-time assigns an operating system policy and priority to match the Ada priority of each task and the task dispatching policy in effect.

See “Run-Time Systems” on page 1-5 for the full set of features that require the tasking run-time.

The assignment of operating system class and priority is done in the following manner. When the task dispatching policy is:

- `FIFO_WITHIN_PRIORITIES`

  On RedHawk Linux systems, the `SCHED_FIFO` POSIX policy is selected. The priority of the task is mapped directly to the `SCHED_FIFO` priority.

  Priorities are *not* adjusted due to CPU utilization and tasks are never preempted by other tasks/programs executing at the same (or lower) priority.

- `ROUND_ROBIN_PRIORITIES`

  On RedHawk Linux systems, the `SCHED_RR` POSIX policy is selected. The priority of the task is mapped directly to the priority of the `SCHED_RR` policy.

  Tasks are time-sliced as per their quantum but their priority is *not* adjusted due to CPU utilization.

  See “Pragma TASK_QUANTUM” on page 6-14 for more information.

- `ROUND_ROBIN_ADJUSTABLE_PRIORITIES`

  On RedHawk Linux systems, the `SCHED_OTHER` (interactive) POSIX policy is selected.

  The priority of the task controls the initial **`nice`** value for the task. The full range of Ada task priorities 1..99 is mapped over the available **`nice`** range: -20..19. Thus the mapping is not 1:1.

The operating system adjusts priorities as the task executes based on CPU utilization and other factors.

- unspecified

  When the task dispatching policy is unspecified, it defaults to `FIFO_WITHIN_PRIORITIES`.

**NOTE**

Use of protected objects and the `CEILING_LOCKING` locking policy (the default locking policy - see "Pragma LOCKING_POLICY" on page 6-3), requires the task dispatching policy `FIFO_WITHIN_PRIORITIES` (see "Pragma TASK_DISPATCHING_POLICY" on page 6-2). Thus, the use of protected objects will cause selection of the operating system AD (Ada) scheduling class for all tasks.

With the tasking run-time, task priorities have a direct correspondence to operating system priorities (see Figure 5-1). As such, the process spawning the program has no effect on the priority of the main subprogram (`ENVIRONMENT` task), since the language requires it to execute midway within the range of `System.Priority` (unless otherwise specified by the user with a priority pragma). In other words, the command

```
nice -4 a.out
```

does not have an effect on a task's priority or the priority of the program as a whole. However, in anticipation of the need for priorities relative to that of the spawning process, the implementation-defined package `RUNTIME_CONFIGURATION` (see "vendorlib" on page 9-8) includes the constant `PRIORITY_OF_ENVIRONMENT`. That constant is elaborated during program start-up, before any user packages are elaborated. It can be used in a priority pragma to achieve an effect similar to **`nice -4`**. For example:

```
pragma TASK_PRIORITY (runtime_configuraiton.priority_of_environment-4);
```

would ensure that the task in question would execute at a operating system priority lower (by 4) than that of the spawning process.

### Restrictions for Priorities in the System.Interrupt_Priority Range

MAXAda does not allow application of Pragma `Interrupt_Priority` to normal tasks unless the value specified in the pragma is `Interrupt_Priority'First`. Tasks which execute at `System.Interrupt_Priority'First` are unrestricted.

Execution at higher priorities is restricted to:

- Protected subprograms and entries

Code executed at priorities higher than `System.Interrupt_Priority'First` is restricted as follows:

- May not enter the operating system kernel

- May not perform any tasking actions (other than protected subprogram calls and `Ada.Synchronous_Task_Control` calls)
- May not execute delay or asynchronous select statements
- May not cause machine exceptions (page faults, floating point machine exceptions, etc. Note that use of such `interrupt_priority` values causes the applications pages to be locked in memory by the Ada executive, thus pages faults would not occur except by unusual user interaction.)

**NOTE**

Use of restricted priorities should be done only in a controlled manner by those with an understanding of external interrupts and their interaction with the operating system kernel.

All maskable external interrupts are **masked** during execution of code at these restricted priorities.

Misuse of this capability may have significant impact on the execution of the system.

# Memory Management

The run-time system segments memory via the following classification:

- Machine instructions (text)
- Library-level variables (data)
- Collections
- Subprogram/task data (stack)
- Other

For each of the various types of memory region discussed here, the following attributes are configurable:

- Physical location (memory pool)
- Locking behavior (lock state)
- Cache mode
- Size and extensibility

See "Memory Attributes" on page 6-20 for details on configuration.

## Text Memory

Machine instructions, literals, and some constant data are allocated in statically sized segments commonly referred to as *text*. Text is typically allocated at the low end of the application's virtual address space (e.g., 0x1nnnnnnn). Generally, the size of text is determined statically by the linker.

## Data Memory

Library-level variables, such as those in library-level packages, are allocated in statically sized segments commonly referred to as *data*. Data is typically allocated in the middle of the application's virtual address space after the `ENVIRONMENT` task's stack segment (e.g., 0x3nnnnnnn). Generally, the size of data is determined statically by the linker.

## Collection Memory

The default collection, or default heap, is a region of memory used for designated objects of user-defined access types, dynamically sized objects, internal run-time structures, etc. The maximum size of the default collection may be virtually unlimited or may be specified statically. If unlimited, the heap will grow as required by the application. Default collection addresses are assigned dynamically by the operating system and tend to be at the

high end of the application's virtual address space (e.g., 0xbnnnnnn). The default collection is created by the run-time system. Its extensibility and size are configurable.

Additional collections are allocated to implement user-defined access types that have specific size requirements (e.g., use of `'STORAGE_SIZE` on an access type). Such collections are allocated dynamically when the corresponding access type is elaborated. If the access type is defined within a task or subprogram, the collection is allocated out of memory associated with the task or subprogram's stack. If the access type is defined in a library level package, the collection is allocated out of new memory at an address dynamically assigned by the operating system. Usually, the system automatically reclaims memory locations associated with collections allocated out of stack frames when those stack frames are exited. (See "Memory Attributes" on page 6-20). Memory associated with other heaps is reclaimed only when the application exits.

## Stack Memory

Subprogram and task data, including temporary variables generated by the compiler, are allocated and freed in stack frames associated with subprograms and task bodies as they are executed. Each task has a limit imposed by the run-time system on the total amount of stack space available for its use (except for the `ENVIRONMENT` stack, which may be virtually unlimited in size). All stack size limits are configurable.

The stack associated with the `ENVIRONMENT` task is allocated by the operating system at program start-up time and is the only stack that can grow dynamically. Hence, it is the only task that can have an `UNLIMITED` stack size. This merely indicates that the size of the stack is not limited by the MAXAda compilation system. The stack still obeys the RLIMIT_STACK limit imposed by the operating system. See **`getrlimit(2)`** and **`setrlimit(2)`**, or the shell special command **`ulimit`** for details on determining and affecting this limit. (Note that not all shells support setting the stack limit.)

It is possible to change the memory aspects of the `ENVIRONMENT` task stack but only the amount that is currently allocated by the operating system. Therefore, the size of the `ENVIRONMENT` task stack must be specified by the user before attempting to modify any of these aspects. If not specified, the `ENVIRONMENT` stack size will be set to 1 Mb.

The `ENVIRONMENT` task's stack is typically allocated in the middle of the application's virtual address space before the data segment (e.g., 0x2fnnnnnn).

Stacks associated with tasks are allocated and freed dynamically by the run-time system (during creation and termination) out of the default collection. As such, stack addresses tend to be at the high end of the application's virtual address space (e.g., 0xbnnnnnnn).

## Other Memory

Other memory may be part of the application's address space, due to the application's use of pragmas, packages, or tools.

## Visibility of Memory

All tasks in an application have actual access to all memory locations in the application's virtual address space. Visibility to these memory locations is limited programmatically by the compiler's enforcement of the Ada language rules. However, through use of `unchecked_conversion`, pragma `SUPPRESS`, erroneous programming, or other mechanisms outside the scope of the Ada language, every task has the ability to read (and perhaps modify) any memory location within the application's virtual address space.

# 6
# Run-Time Configuration


General Pragmas . . . . . 6-1
Pragma RUNTIME_DIAGNOSTICS . . . . . 6-1
Pragma MAP_FILE . . . . . 6-2
Pragma QUEUING_POLICY . . . . . 6-2
Pragma TASK_DISPATCHING_POLICY . . . . . 6-2
Pragma LOCKING_POLICY . . . . . 6-3
Pragma SERVER_CACHE_SIZE . . . . . 6-4
Task and Group Configuration Concepts . . . . . 6-4
Task Names and Default Settings . . . . . 6-4
Task Specifiers in Task Pragmas . . . . . 6-5
Group Names and Default Settings . . . . . 6-7
Group Specifiers in Group Pragmas . . . . . 6-8
Task Attributes . . . . . 6-9
Pragma TASK_WEIGHT . . . . . 6-9
Pragma TASK_PRIORITY . . . . . 6-11
Pragma TASK_CPU_BIAS . . . . . 6-12
Pragma TASK_QUANTUM . . . . . 6-14
Pragma TASK_HANDLER . . . . . 6-15
Group Attributes . . . . . 6-18
Pragma GROUP_PRIORITY . . . . . 6-18
Pragma GROUP_CPU_BIAS . . . . . 6-19
Pragma GROUP_SERVERS . . . . . 6-19
Memory Attributes . . . . . 6-20
Pool Specifiers . . . . . 6-21
Pragma MEMORY_POOL . . . . . 6-23
Pragma POOL_CACHE_MODE . . . . . 6-25
Pragma POOL_LOCK_STATE . . . . . 6-25
Pragma POOL_SIZE . . . . . 6-26
Pragma POOL_PAD . . . . . 6-28
Protected Object Attributes . . . . . 6-28
Pragma PROTECTED_PRIORITY . . . . . 6-28

# 6
# Run-Time Configuration

Although Appendix M discusses all pragmas, it focuses on pragmas that influence the software development environment, compiling, and linking. This chapter discusses pragmas that affect configuration of the whole run-time system, task execution, and memory utilization. It also provides some information about the underlying implementation of tasking and memory resources.

## General Pragmas

The following pragmas affect the run-time system as a whole:


- Pragma RUNTIME_DIAGNOSTICS (see page 6-1)
- Pragma MAP_FILE (see page 6-2)
- Pragma QUEUING_POLICY (see page 6-2)
- Pragma TASK_DISPATCHING_POLICY (see page 6-2)
- Pragma LOCKING_POLICY (see page 6-3)
- Pragma SERVER_CACHE_SIZE (see page 6-4)


## Pragma RUNTIME_DIAGNOSTICS

The implementation-defined pragma `RUNTIME_DIAGNOSTICS` may occur in any declarative part. It controls whether or not the run-time emits warning diagnostics.

**`pragma RUNTIME_DIAGNOSTICS (`***boolean***`);`**

*boolean*

A static boolean enumeration literal. `TRUE` means run-time warning diagnostics will be emitted. `FALSE` means run-time warning diagnostics will not be emitted. The default is `TRUE`. At run-time, you can specify this value via a call to `Runtime_Configuration.Set_Runtime_Diagnostics`.

See “General Pragmas” on page 6-1 for a list of other pragmas that affect the run-time system as a whole. In addition, “RM Annex L: Pragmas” on page M-104 lists all implementation-dependent and implementation-defined pragmas.

## Pragma MAP_FILE

The implementation-defined pragma MAP_FILE may occur in any declarative part. It causes the linker to automatically emit at link time a map file containing an ASCII description of pragma entries and comments that define the layout of the file. This file is useful with the **a.map** tool described in "a.map" on page 4-47. If this pragma is absent, then no map file is produced.

**pragma MAP_FILE (***file_name***);**

*file_name*

A static string of non-zero length specifying the name of the map file.

See "General Pragmas" on page 6-1 for a list of other pragmas that affect the run-time system as a whole. In addition, "RM Annex L: Pragmas" on page M-104 lists all implementation-dependent and implementation-defined pragmas.

## Pragma QUEUING_POLICY

The implementation-dependent pragma QUEUING_POLICY may occur as a configuration pragma. It sets the entry queuing policy.

**pragma QUEUING_POLICY (***policy_identifier***);**

*policy_identifier*

The keyword FIFO_QUEUING means the entry queuing policy as defined in the Ada 95 Reference Manual section D.4.

PRIORITY_QUEUING means the entry queuing policy as defined in Ada 95 Reference Manual section D.4.

The default is FIFO_QUEUING.

See "General Pragmas" on page 6-1 for a list of other pragmas that affect the run-time system as a whole. In addition, "RM Annex L: Pragmas" on page M-104 lists all implementation-dependent and implementation-defined pragmas.

## Pragma TASK_DISPATCHING_POLICY

This implementation-dependent pragma TASK_DISPATCHING_POLICY may occur as a configuration pragma. It sets the task dispatching policy.

**pragma TASK_DISPATCHING_POLICY (***policy_identifier***);**

*policy_identifier*

The keyword FIFO_WITHIN_PRIORITIES indicates the task dispatching policy as defined in the Ada 95 Reference Manual section D.2.2.

In addition, other policies are implemented as defined below:

`ROUND_ROBIN_PRIORITIES`

This policy is the same as `FIFO_WITHIN_PRIORITIES`, except that time-slicing occurs.

`ROUND_ROBIN_ADJUSTABLE_PRIORITIES`

This policy is similar to `ROUND_ROBIN_PRIORITIES`, except that priorities are adjusted by the operating system based on CPU utilization. This policy has a drastic effect on the relation of Ada priorities to System priorities. See "Priorities" on page 5-5 for more information.

By default, programs without tasks (other than the **`ENVIRONMENT`** task), without protected objects, and without implementation-defined memory configuration pramgas (e.g. `MEMORY_POOL`, `POOL_LOCK_STATE`, etc.) have a task dispatching policy of `ROUND_ROBIN_ADJUSTABLE_PRIORITIES`. All other programs have a default task dispatching policy of `FIFO_WITHIN_PRIORITIES`.

The implementation requires that the task dispatching policy be `FIFO_WITHIN_PRIORITIES` if the program contains any protected objects with a locking policy of `CEILING_LOCKING` (which is the only locking policy currently implemented).

See "General Pragmas" on page 6-1 for a list of other pragmas that affect the run-time system as a whole. In addition, "RM Annex L: Pragmas" on page M-104 lists all implementation-dependent and implementation-defined pragmas.

## Pragma LOCKING_POLICY

This implementation-dependent pragma `LOCKING_POLICY` may occur as a configuration pragma. It sets the protected object locking policy.

**`pragma LOCKING_POLICY (`***policy_identifier***`);`**

*policy_identifier*

The keyword `CEILING_LOCKING` indicates the protected object locking policy as defined in the Ada 95 Reference Manual section D.3.

The default locking policy is `CEILING_LOCKING`. This is currently the only locking policy that is implemented.

When the `CEILING_LOCKING` policy is in use and a protected action is underway for a specific protected object, attempts by other tasks (on other CPUs) to start a protected action on the same protected object will keep their CPUs busy (i.e. other tasks spin waiting to start the protected action on that protected object).

If the locking policy is explicitly specified or the program contains protected objects, the implementation requires that the task dispatching policy be `FIFO_WITHIN_PRIORITIES`. If the task dispatching policy has not explicitly been set, the implementation will automatically set it to `FIFO_WITHIN_PRIORITIES` if the program contains protected objects.

See “General Pragmas” on page 6-1 for a list of other pragmas that affect the run-time system as a whole. In addition, “RM Annex L: Pragmas” on page M-104 lists all implementation-dependent and implementation-defined pragmas.

## Pragma SERVER_CACHE_SIZE

The implementation-defined pragma `SERVER_CACHE_SIZE` may occur in any declarative part. It sets the size of the server cache. The server cache contains execution servers that are currently unneeded by the application, but which can be placed back into service when they become necessary. These include the anonymous servers for terminated bound tasks, as well as servers from server groups which were reduced in size by the run-time. (For more information about bound tasks, see “Pragma TASK_WEIGHT” on page 6-9.)

**pragma SERVER_CACHE_SIZE (***cache_size***);**

*cache_size*

A static, non-negative number specifying the maximum number of servers allowed in the cache (i.e., the server cache size). The default is 8. This value can also be set at run-time via a call to `Runtime_Configuration.Set_Server_Cache_Size`. See the specification of `Runtime_Configuration` in **vendorlib**.

See “General Pragmas” on page 6-1 for a list of other pragmas that affect the run-time system as a whole. In addition, “RM Annex L: Pragmas” on page M-104 lists all implementation-dependent and implementation-defined pragmas.

# Task and Group Configuration Concepts

## Task Names and Default Settings

To make good use of task pragmas, it is necessary to understand some terminology.

**ENVIRONMENT** *task*

At start-up, the run-time creates this one task that performs library-level package elaboration and executes the main program.

**DEFAULT** *pseudo task*

This non-executing pseudo task sometimes provides default task-attribute values for other tasks. The user may change these default values with task pragmas or with calls to routines in package `Runtime_Configuration`. See the specification of `Runtime_Configuration` in **vendorlib**.

*ghost task*

An automatically generated overhead task. Ghost tasks are described in "Ghost Tasks" on page 5-5.

For any actual task (excluding objects of task types) or the ADMIN or TIMER ghost task, if a configuration pragma is omitted for that task, the value specified for the **DEFAULT** pseudo task is used instead.

For objects of task types, the following steps indicate the search order for configuration pragma values.

1. If the object is a variable and the pragma exists for that variable, that pragma is used.
2. If the pragma exists for its task type, that pragma is used.
3. If the task type is a derived type, the pragma of the nearest ancestor type is used if found.
4. If no such pragma is found, the **DEFAULT** pseudo task is checked for the pragma, and that pragma is used if found.
5. If no pragma has been found, the default value is used.

The same steps take place simultaneously for any SHADOW, COURIER, and INTR_COURIER ghost tasks associated with a user's real task or with a user's protected attachments. (See "Ghost Tasks" on page 5-5.)

## Task Specifiers in Task Pragmas

The following task specifiers appear in task pragmas.

*task_specifier*

::= {*ordinary_task* | *ghost_task* | **ENVIRONMENT** | **SPEC**}

*ordinary_task*

::= {*task_type*_name | *task_variable*_name | **DEFAULT**}

*ghost_task*

::= {*companion_ghost_task* | *companion_po_ghost_task* | **ADMIN** | **TIMER**}

*companion_ghost_task*

::= {*shadow_ghost* | *courier_ghost* | *intr_courier_ghost*}

*companion_po_ghost_task*

::= {*shadow_po_ghost* | *courier_po_ghost* | *intr_courier_po_ghost*}

*shadow_ghost*

::= *ordinary_task*, **SHADOW**, *task_entry*

*courier_ghost*

::= *ordinary_task*, **COURIER**, *task_entry*

*intr_courier_ghost*

::= *ordinary_task*, **INTR_COURIER**, *task_entry*

*shadow_po_ghost*

::= *protected_procedure*_handler, **SHADOW** [, *attachment*_index]

*courier_po_ghost*

::= *protected_procedure*_handler, **COURIER** [, *attachment*_index]

*intr_courier_po_ghost*

::= *protected_procedure*_handler, **INTR_COURIER** [, *attachment*_index]

*task_entry*

::= {*entry_name* | **DEFAULT**}

**ADMIN**

The pragma sets the task attribute to the specified value for the ADMIN task. For more information about the ADMIN task, see "ADMIN Ghost Task" on page 5-5.

**COURIER**

The pragma sets the task attribute to the specified value for the **COURIER** task. See also "COURIER Ghost Tasks" on page 7-3 for more information.

**DEFAULT**

The pragma sets the task attribute to the specified value for the **DEFAULT** pseudo task, and therefore for all tasks, unless otherwise specified for a task.

**ENVIRONMENT**

The pragma sets the task attribute to the specified value for the **ENVIRONMENT** task.

**INTR_COURIER**

The pragma sets the task attribute to the specified value for the **INTR_COURIER** task. See also "INTR_COURIER and COURIER Ghost Tasks" on page 7-5 for more information.

**SHADOW**

The pragma sets the task attribute to the specified value for the **SHADOW** task. See also "SHADOW Ghost Tasks" on page 7-6 for more information.

**SPEC**

The pragma must occur in the declarative part of a task specification. It then applies to all tasks identified with that specification.

**`TIMER`**

The pragma sets the task attribute to the specified value for the TIMER task. For more information about the TIMER task, see "TIMER Ghost Task" on page 5-5.

*attachment*`_index`

The pragma is associated with the ghost task that corresponds to that particular attachment numbered in textual order. If no *attachment*`_index` is specified, the pragma selects all ghost tasks corresponding to all attachments on the specified handler.

Only positive integer literals or the identifier `DYNAMIC` is allowed.

If the identifier `DYNAMIC` is specified as the *attachment*`_index`, the pragma is associated with the ghost task that corresponds to the dynamic attachment on the protected procedure handler, if it exists.

*protected_procedure*`_handler`

A protected procedure to which either pragma INTERRUPT_HANDLER or ATTACH_HANDLER applies. See Section C.3.1 of the Ada 95 Reference Manual.

*task_type*`_name`

The pragma applies to all task objects of that task type, regardless of where the task objects are actually declared, unless overridden for derived types or task variables.

*task_variable*`_name`

The pragma must appear in the same declarative part as the declaration of the task variable. In this case, the pragma affects the task attribute of the specified task, regardless of any other task pragmas associated with defaults or task specifications.

**NOTE**

Pragmas `TASK_WEIGHT`, `TASK_PRIORITY`, `TASK_CPU_BIAS`, `TASK_QUANTUM`, and `TASK_HANDLER` let you omit the *task_specifier*. This has the same effect as **`SPEC`**.

## Group Names and Default Settings

To make good use of group pragmas, it is necessary to understand some terminology.

*server group*

Server groups allow users to restrict the resources their tasks use. These groups are designated by simple identifiers and are defined when they are used. However, they are not Ada program entities. They cannot be referenced anywhere except within the appropriate pragmas. In fact, they exist in a namespace which is separate from the Ada language's namespaces. This separate namespace is completely flat. That is, there is no hierarchical nesting to the namespace based on the units in which these pragmas appear. The same group name can be specified in two separate and unrelated units, and it will indicate the same group. See "Tasking Model" on page 5-1 for more information about server groups.

**PREDEFINED** *group*

At start-up, the run-time creates this one **PREDEFINED** group that includes and executes the **ENVIRONMENT** task. By default, the **DEFAULT** pseudo task is also in this group.

**DEFAULT** *pseudo group*

This pseudo group provides default group-attribute values for other groups that omit any group configuration pragmas. The user may change these default values with group pragmas or with calls to routines in package `Runtime_Configuration`.

To add tasks to any group, see "Pragma TASK_WEIGHT" on page 6-9.

## Group Specifiers in Group Pragmas

The following server group specifiers appear in group pragmas.

*group_spec* **::= {DEFAULT | PREDEFINED |** *group_name***}**

**DEFAULT**

The pragma sets the group attribute for the **DEFAULT** pseudo group, and therefore for all groups, to the specified value.

**PREDEFINED**

The pragma sets the group attribute for the **PREDEFINED** group to the specified value.

*group_name*

The pragma applies only to the group specified by *group_name*.

# Task Attributes

Users can control the execution of tasks: specifically, tasks' scheduling priority, time-slice duration, physical CPU binding, and weight. Control may be static through implementation-defined pragmas, and may be changed dynamically via supplied routines in the `Runtime_Configuration` package. See the specification of `Runtime_Configuration` in **`vendorlib`**.

In addition, the user may specify a procedure to be called for a task that is terminating because of an unhandled exception.

The task attribute pragmas can be applied to any user task. There are certain restrictions on tasks within generic units, however. The task attribute pragmas may be applied to such tasks, but they cannot be applied to a task in a particular instantiation of the generic. The pragma must be applied to the task in the generic, and the effect of the pragma will extend to *all* instantiations of that generic. Finally, note that task attribute pragmas applied to tasks in generic units cannot be changed via the **`a.map`** tool, as can other task attribute pragmas. See "a.map" on page 4-47 for more details.

The following pragmas are associated with task attributes:


- Pragma TASK_WEIGHT (see page 6-9)
- Pragma TASK_PRIORITY (see page 6-11)
- Pragma TASK_CPU_BIAS (see page 6-12)
- Pragma TASK_QUANTUM (see page 6-14)
- Pragma TASK_HANDLER (see page 6-15)


## Pragma TASK_WEIGHT

The implementation-defined pragma `TASK_WEIGHT` specifies the weight of a task.

**`pragma TASK_WEIGHT (`***weight***`[, `** *task_specifier* **`]);`**

*weight*

`BOUND`

*Bound tasks* are served by an anonymous group, distinct from all other groups, containing a single server.

`MULTIPLEXED`, *group_spec*

*Multiplexed tasks* are served by named groups, specified by *group_spec*, and are associated with multiple servers. Multiplexed tasks are not supported in this release.

*group_spec*

See "Group Specifiers in Group Pragmas" on page 6-8. These server groups are configured via other pragmas. (See "Group Attributes" on page 6-18.)

For more information about task weights, see "Task Weights" on page 5-3.

*task_specifier*

If specified, then the pragma must appear in the same declarative part as the referenced task.

If *task_specifier* is omitted, then the pragma must occur in the declarative part of a task specification. It then applies to all tasks identified with that specification.

See "Task Specifiers in Task Pragmas" on page 6-5.

The weight of default tasks can be overridden by certain link options.

**-bound** overrides as:

```
pragma TASK_WEIGHT (BOUND, DEFAULT);
```

**-multiplexed** overrides as:

```
pragma TASK_WEIGHT(MULTIPLEXED, PREDEFINED, DEFAULT);
pragma GROUP_SERVERS(1, PREDEFINED);
```

In the absence of any such link options, by default, the following pragmas apply:

```
pragma TASK_WEIGHT (BOUND, DEFAULT);
```

In other words, by default, the task weight for all tasks, including the **ENVIRONMENT** task, is bound. Of course, these defaults are overridden by user-specified pragmas or link options. For more information about link options, see "a.link" on page 4-33 and "Link Options" on page 4-109.

**NOTE**

This pragma will not be accepted for any ghost tasks. SHADOW ghost tasks have no associated weight. COURIER and INTR_COURIER ghost tasks are always bound. The ADMIN and TIMER ghost tasks, if they exist, are always bound.

See "Task Attributes" on page 6-9 for a list of other pragmas associated with task attributes. In addition, "RM Annex L: Pragmas" on page M-104 lists all implementation-dependent and implementation-defined pragmas.

## Pragma TASK_PRIORITY

The implementation-defined pragma `TASK_PRIORITY` is primarily used to set the task scheduling priority. For a bound task, it also sets the operating system scheduling priority of the bound task's anonymous group.

**`pragma TASK_PRIORITY (`***scheduling_priority* **`[,`** *task_specifier* **`]);`**

*scheduling_priority*

A required integer expression, possibly a program variable, specifying the scheduling priority. It should be in the range `System.Priority'Range`.

Values greater than `System.Priority'Last` will be truncated to `System.Priority'Last` by the run-time executive.

Values less than `0` are considered to be values relative to `System.Priority'Last+1`. The following pragmas are equivalent:

```
pragma TASK_PRIORITY (System.Priority'Last);
pragma TASK_PRIORITY (-1);
```

For information about priority values, see "Task Scheduling" on page 5-3.

*task_specifier*

If specified, then the pragma must appear in the same declarative part as the referenced task.

If *task_specifier* is omitted, then the pragma must occur in the declarative part of a task specification. It then applies to all tasks identified with that specification.

For information about task specifiers, see "Task Specifiers in Task Pragmas" on page 6-5.

See "Task Names and Default Settings" on page 6-4 to find out how a task without an explicit pragma `TASK_PRIORITY` setting gets its scheduling priority. Specifically, if no `TASK_PRIORITY` or `PRIORITY` pragma has been applied to a task and no `TASK_PRIORITY` pragma has been applied with a task specifier of **`DEFAULT`**, then a task's priority is inherited from its creator, as per RM D.1(19).

As discussed in "Task Scheduling" on page 5-3, the *task scheduling priority* of a task determines how the Ada run-time selects tasks for execution within a group. Similarly, the *operating system scheduling priority* determines how the real-time kernel selects task groups for execution.

As previously mentioned, for a bound task, this pragma sets the operating system scheduling priority of the bound task's anonymous group. The sequence:

```
pragma TASK_WEIGHT (BOUND, t);
pragma TASK_PRIORITY (prio, t);
```

is equivalent to:

```
pragma GROUP_SERVERS (1, anon_group_spec);
pragma TASK_WEIGHT (MULTIPLEXED, anon_group_spec, t);
pragma GROUP_PRIORITY (prio, anon_group_spec);
pragma TASK_PRIORITY (prio, t);
```

As specified in the Ada 95 Reference Manual section D.1(19), if a pragma Priority does not apply to the main subprogram, the initial base priority of the **ENVIRONMENT** task is System.Default_Priority.

**NOTE**

If an application is linked with the tasking run-time, the operating system priority associated with that task is also System.Default_Priority. If the application is not linked with the tasking run-time, then the operating system priority is inherited from the environment that invoked the application (usually the shell).

Unless otherwise specified, the default value of *scheduling_priority* for the other ghost tasks is as follows:

```
pragma TASK_PRIORITY(DEFAULT, SHADOW, DEFAULT, -1);
pragma TASK_PRIORITY(DEFAULT, COURIER, DEFAULT, -1);
pragma TASK_PRIORITY(DEFAULT, INTR_COURIER, DEFAULT, -1);
pragma TASK_PRIORITY(TIMER, -1);
```

Use of this pragma requires the CAP_SYS_NICE capability (see “Capabilities” on page 1-3).

Pragma TASK_PRIORITY differs from the language-defined pragma PRIORITY in that it can be applied to entities that pragma PRIORITY cannot; for example, individual task objects, implementation-defined tasks, etc.

**NOTE**

The task scheduling priority can also be set at run-time via a call to Ada.Dynamic_Priorities.Set_Priority. See the specification of Ada.Dynamic_Priorities in **predefined**.

See “Task Attributes” on page 6-9 for a list of other pragmas associated with task attributes. In addition, “RM Annex L: Pragmas” on page M-104 lists all implementation-dependent and implementation-defined pragmas.

## Pragma TASK_CPU_BIAS

The implementation-defined pragma TASK_CPU_BIAS provides for the binding of bound tasks to individual CPUs or a set of CPUs, associating a CPU bias with one or more bound

tasks. This is necessary because pragma GROUP_CPU_BIAS is not available for bound tasks (see "Pragma GROUP_CPU_BIAS" on page 6-19 for more information).

**pragma TASK_CPU_BIAS (***cpu_bias* **[,** *task_specifier* **]);**

*cpu_bias*

A required CPU bias, possibly a program variable, specifying CPUs that are valid for the machine configuration where the application will run. See "Utilization of Multiple CPUs" on page 5-4.

For information about CPU biases, see "Utilization of Multiple CPUs" on page 5-4.

*task_specifier*

If specified, then the pragma must appear in the same declarative part as the referenced task.

If *task_specifier* is omitted, then the pragma must occur in the declarative part of a task specification. It then applies to all tasks identified with that specification.

For information about task specifiers, see "Task Specifiers in Task Pragmas" on page 6-5.

See "Task Names and Default Settings" on page 6-4 to find out how a task without an explicit pragma TASK_CPU_BIAS setting gets its CPU bias.

The sequence:

```
pragma TASK_WEIGHT (BOUND, t);
pragma TASK_CPU_BIAS (bias, t);
```

is equivalent to:

```
pragma GROUP_SERVERS (1, anon_group_spec);
pragma TASK_WEIGHT (MULTIPLEXED, anon_group_spec, t);
pragma GROUP_CPU_BIAS (bias, anon_group_spec);
```

With the judicious use of pragmas MEMORY_POOL, TASK_CPU_BIAS, and GROUP_CPU_BIAS, an Ada application can take full advantage of all the CPU and memory resources of Series 6000 systems. See "Pragma MEMORY_POOL" on page 6-23 and "Pragma GROUP_CPU_BIAS" on page 6-19 for more information.

Use of this pragma requires the CAP_SYS_NICE capability (see "Capabilities" on page 1-3).

**NOTE**

The CPU bias can also be set at run-time via a call to Runtime_Configuration.Set_Task_CPU_Bias. See the specification of Runtime_Configuration in **vendorlib**.

See “Task Attributes” on page 6-9 for a list of other pragmas associated with task attributes. In addition, “RM Annex L: Pragmas” on page M-104 lists all implementation-dependent and implementation-defined pragmas.

**NOTE**

Hyper-threading is a feature of the Intel Pentium Xeon processor that allows for a single physical processor to appear to the operating system as two logical processors (“sibling CPUs”). It is important to note that hyper-threading affects CPU utilization. For example, if two tasks are scheduled with each one bound to a specific sibling CPU, each will be affected by the execution of the other since they share the same physical CPU. Refer to the section titled “Hyper-threading” in Chapter 2 of the *RedHawk Linux User's Guide* (0898004).

## Pragma TASK_QUANTUM

The implementation-defined pragma `TASK_QUANTUM` is used to set the task quantum for multiplexed tasks, and the operating system quantum for the anonymous server group of bound tasks.

**`pragma TASK_QUANTUM (`***quantum* **`[,`** *task_specifier* **`]);`**

*quantum*

A non-zero number, possibly a program variable, of 100Hz clock ticks.

*task_specifier*

If specified, then the pragma must appear in the same declarative part as the referenced task.

If *task_specifier* is omitted, then the pragma must occur in the declarative part of a task specification. It then applies to all tasks identified with that specification.

For information about task specifiers, see “Task Specifiers in Task Pragmas” on page 6-5.

See “Task Names and Default Settings” on page 6-4 to find out how a task without an explicit pragma `TASK_QUANTUM` setting gets its quantum.

The *task quantum* of a task determines how often the Ada run-time preempts tasks executing within a group. Similarly, the *operating system quantum* determines how the real-time kernel preempts task groups executing on a physical CPU.

Use of this pragma requires the `CAP_SYS_NICE` capability (see “Capabilities” on page 1-3).

**NOTE**

The following task dispatching policies will cause all task quanta specified to be ignored:

- FIFO_WITHIN_PRIORITIES

The FIFO policies require that all task quanta are infinite. Use ROUND_ROBIN policies when task time-slicing is desired. Note the use of protected objects with a locking policy of CEILING_LOCKING (currently the only locking policy implemented) requires the FIFO_WITHIN_PRIORITIES task dispatching policy.

**NOTE**

The task quantum can also be set at run time via a call to `Runtime_Configuration.Set_Task_Quantum`. See the `Runtime_Configuration` specification in **`vendorlib`**.

**NOTE**

Tasks quanta values are mapped to step values supported by the operating system. Issue the following command:

```
run --quantum=list
```

to see the step values available.

See "Task Attributes" on page 6-9 for a list of other pragmas associated with task attributes. In addition, "RM Annex L: Pragmas" on page M-104 lists all implementation-dependent and implementation-defined pragmas.

## Pragma TASK_HANDLER

The implementation-defined pragma `TASK_HANDLER` calls the specified procedure when the task to which it is applied completes because of an unhandled exception.

**`pragma TASK_HANDLER (`** *handler*`_name` **`[,`** *task_specifier* **`]);`**

*handler*`_name`

The handler must denote a library-level procedure. It must be either parameterless or contain only a single formal parameter of mode **`in`** and of type `Ada.Exceptions.Exception_Occurrence`.

If the procedure contains a formal parameter of type `Ada.Exceptions.Exception_Occurrence`, then the actual value of this parameter

will be the `Exception_Occurrence` for the exception that caused the termination of the task.

*task_specifier*

If specified, then the pragma must appear in the same declarative part as the referenced task.

If *task_specifier* is omitted, then the pragma must occur in the declarative part of a task specification. It then applies to all tasks identified with that specification.

For information about task specifiers, see "Task Specifiers in Task Pragmas" on page 6-5.

If a task to which this pragma is applied is about to complete because of an unhandled exception, then the denoted procedure will be called by the task before that task completes.

This pragma is especially useful when applied to the **`ENVIRONMENT`** task. It will be called for any unhandled exception that would cause completion of the ENVIRONMENT task, and thus of the application.

It is also especially useful when applied to the **`DEFAULT`** task. It will be called for any unhandled exception that would cause completion of any task which otherwise happens silently without any notification to the user.

Consider the following example. The task `first_task` will raise a `Constraint_Error` when it executes its code. Because there is no exception handler in the task itself, the procedure `handler` specified by pragma `TASK_HANDLER` is called. (This procedure appears below and also utilizes the formal parameter of type `Ada.Exceptions.Exception_Occurrence`.) Any processing with respect to this unhandled exception may occur in this procedure before the task completes.

```
with ada.text_io;
with handler;

procedure test_handler is
--
   task my_task is
      entry start;
   end my_task;

   task body my_task is
      subtype scale is integer range 1..10;
      i : scale;
   begin
      accept start do
         ada.text_io.put_line ("my_task: in rendezvous");
         i := scale'last;
         i := i + 1; -- will raise a constraint error
         ada.text_io.put_line ("This line won't be printed");
      end start;
   end my_task;
--
begin
--
```

```
      ada.text_io.put_line ("test_handler: starting");
      begin
         my_task.start;
      exception
      when others =>
         null;
      end;
      ada.text_io.put_line ("test_handler: exiting");
   --
   end test_handler;

   pragma task_handler (handler, default);


   -- and the handler itself...

   with ada.text_io;
   with ada.exceptions;
   with ada.task_identification;

   procedure handler (occurrence :
   ada.exceptions.exception_occurrence) is
   begin
      ada.text_io.put_line ("handler: Exception """ &
                            ada.exceptions.exception_name(occurrence) &
                            """ terminated """ &
                            ada.task_identification.image(
                            ada.task_identification.current_task) &
                               """.");
   end handler;
```

The output from running the `test_handler` procedure is as follows:

```
test_handler: starting
my_task: in rendezvous
handler: Exception "CONSTRAINT_ERROR" terminated
"test_handler.my_task".
test_handler: exiting
```

**WARNING**

Be cautious when using packages within a handler that may not be elaborated at the time the handler is called. For instance, in the above example, if procedure `handler` is called before `Ada.Text_IO` is elaborated, a `PROGRAM_ERROR` exception may be raised and handled by this same procedure, resulting in an infinite loop. This can be remedied by using the `write` function of the `POSIX_1003_1` binding in the handler instead of calling `Ada.Text_IO.put_line`.

See “Task Attributes” on page 6-9 for a list of other pragmas associated with task attributes. In addition, “RM Annex L: Pragmas” on page M-104 lists all implementation-dependent and implementation-defined pragmas.

# Group Attributes

Users can control the operating system scheduling priority, physical CPU binding, and number of servers in a group. Control may be static through implementation-defined pragmas or through the run-time configuration package, and may be changed dynamically via supplied routines that interface to the run-time executive.

The following pragmas manage group attributes:

- Pragma GROUP_PRIORITY (see page 6-18)
- Pragma GROUP_CPU_BIAS (see page 6-19)
- Pragma GROUP_SERVERS (see page 6-19)

## Pragma GROUP_PRIORITY

The implementation-defined pragma `GROUP_PRIORITY` may occur in any declarative part. It specifies the operating-system scheduling priority of all the servers in a given group. It does not specify the task scheduling priority of particular tasks within the group. If this pragma is not specified for a particular group, the group acquires the operating-system scheduling priority of the environment that spawned it.

**pragma GROUP_PRIORITY (***scheduling_priority***, *group_spec***);**

*scheduling_priority*

A static integer expression specifying the operating system scheduling priority. It is in the range `0..Max_Priority`, as defined by the package `Runtime_Configuration`. A run-time call to `Runtime_Configuration.Set_Group_Priority` can also be used to set this value. See the `Runtime_Configuration` specification in **vendorlib**.

Values greater than `Max_Priority` will be truncated to `Max_Priority` by the run-time executive.

Values less than 0 are considered to be values relative to `Max_Priority`+1.

For information about priority values, see “Task Scheduling” on page 5-3.

*group_spec*

For information about group specifiers, see “Group Specifiers in Group Pragmas” on page 6-8.

Use of this pragma requires the `CAP_SYS_NICE` capability (see “Capabilities” on page 1-3).

See "Group Attributes" on page 6-18 for a list of other pragmas that manage group attributes. In addition, "RM Annex L: Pragmas" on page M-104 lists all implementation-dependent and implementation-defined pragmas.

## Pragma GROUP_CPU_BIAS

The implementation-defined pragma GROUP_CPU_BIAS may occur in any declarative part. It specifies the CPU bias for all the servers in a given group. If this pragma is not specified for a particular group, the default bias is acquired from the environment, which indicates any CPUs.

**pragma GROUP_CPU_BIAS (***cpu_bias***,** *group_spec***);**

*cpu_bias*

A static CPU bias specifying CPUs that are valid for the machine configuration where the application will run. See "Utilization of Multiple CPUs" on page 5-4 for more information about CPU biases. At run time, this value can be set with a call to Runtime_Configuration.Set_Group_CPU_Bias. See the Runtime_Configuration specification in **vendorlib**.

For information about CPU biases, see "Utilization of Multiple CPUs" on page 5-4.

*group_spec*

For information about group specifiers, see "Group Specifiers in Group Pragmas" on page 6-8.

With the judicious use of pragmas MEMORY_POOL, TASK_CPU_BIAS, and GROUP_CPU_BIAS, an Ada application can take full advantage of all the CPU and memory resources of Series 6000 systems. See "Pragma MEMORY_POOL" on page 6-23 and "Pragma TASK_CPU_BIAS" on page 6-12 for more information.

Use of this pragma requires the CAP_SYS_NICE capability (see "Capabilities" on page 1-3).

See "Group Attributes" on page 6-18 for a list of other pragmas that manage group attributes. In addition, "RM Annex L: Pragmas" on page M-104 lists all implementation-dependent and implementation-defined pragmas.

## Pragma GROUP_SERVERS

The implementation-defined pragma GROUP_SERVERS may occur in any declarative part. It controls the number of servers for a particular group, including the **PREDEFINED** group.

**pragma GROUP_SERVERS (***group_size***,** *group_spec***);**

*group_size*

A static non-negative number indicating the quantity of servers in a group. If, for *any* group, no `GROUP_SERVERS` pragma is specified, then the default size for that group is 1. At run time, a call to `Runtime_Configuration.Set_Group_Servers` can be used to set this value. See the `Runtime_Configuration` specification in **`vendor-lib`**.

*group_spec*

For information about group specifiers, see "Group Specifiers in Group Pragmas" on page 6-8.

See "Group Attributes" on page 6-18 for a list of other pragmas that manage group attributes. In addition, "RM Annex L: Pragmas" on page M-104 lists all implementation-dependent and implementation-defined pragmas.

# Memory Attributes

Memory attributes can be specified for any of the following classifications of memory:

- Machine instructions (text)
- Library-level variables (data)
- Collections
- Subprogram/task data (stack)

For each of the various types of memory region discussed here, the following attributes are configurable:

- Physical location (memory pool)
- Locking behavior (lock state)
- Cache mode
- Size and extensibility

The following pragmas modify memory attributes:

- Pragma MEMORY_POOL (see page 6-23)
- Pragma POOL_CACHE_MODE (see page 6-25)
- Pragma POOL_LOCK_STATE (see page 6-25)
- Pragma POOL_SIZE (see page 6-26)
- Pragma POOL_PAD (see page 6-28)

**NOTE**

If any memory attribute is specified for a region of memory that is normally dynamically allocated (collections and stacks), then those regions of memory continue to be dynamically allocated. However, those allocations do not come from the default collection, as would normally be the case. Furthermore, those regions of memory cannot ever be deallocated during the lifetime of the program.

## Pool Specifiers

The following memory pool specifiers appear in memory pool pragmas.

*pool_spec*

::= {*text_pool* | *stack_pool* | *data_pool* | *collection_pool* | *default_pool*}

*sizeable_spec*

::= {*stack_pool* | *collection_pool*}

*paddable_spec*

::= {*stack_pool*}

*default_pool*

::= **DEFAULT**

*text_pool*

::= **TEXT**

*stack_pool*

::= **STACK**, {*task_specifier*}

*data_pool*

::= **DATA**, {**PKG** | **DEFAULT**}

*collection_pool*

::= **COLLECTION**, {**DEFAULT** | *access_type*}

**DEFAULT**

This value means the *memory_spec* is applied to all memory in the program for which a specific memory pool was not already specified.

**TEXT**

For the entire text image (machine instructions), specify a value.

**STACK**

For a specific task, an object of a task type, the **ENVIRONMENT** task, or the **DEFAULT** pseudo task, the stack may be allocated out of dynamic pools bound to local or global memory.

In this form, the pragma may occur in any declarative part. If the second parameter is **ENVIRONMENT**, then the pragma affects the **ENVIRONMENT** task's stack. If the second parameter is **SPEC**, then the pragma must be immediately enclosed by a task specification and will affect all associated tasks. If the second parameter is **DEFAULT**, the pragma applies to all stack frames for all tasks not marked with their own explicit pragma `MEMORY_POOL` specification. If the second parameter is not any of these three keywords, then it must be the name of a task type or a task variable in the same declarative part.

There are certain restrictions on which tasks can be specified by **STACK** memory pool specifiers. Tasks within generic units may be specified. However, tasks in particular instantiations of a generic cannot. If a task in a generic unit is specified, the effect of the particular pragma in which it is specified will extend to that task in *all* instantiations of the generic. Finally, note that pragmas applied to tasks in generic units cannot be changed via the **a.map** tool, as can other memory pool pragmas. See "a.map" on page 4-47 for more details.

**DATA**

For the static memory associated with a specific package, or for all other packages, specify a value.

In this form the pragma must occur in the immediate declarative part of a library-level package specification, a library-level package body, or a library-level subprogram. If the second parameter is **PKG**, it must occur in the package specification or body. When in a package specification, the pragma affects all static data for the package specification and for the package body, unless another pragma is applied to the body. When in a package body, the pragma affects all static data for the package body, regardless of any pragmas associated with the package specification. When the second parameter is **DEFAULT**, the pragma affects all static data including memory associated with packages unless a specific pragma exists for a particular package.

**COLLECTION**

For the memory associated with an access type with a `'Storage_Size` clause, specify a value. When a pragma is applied to a COLLECTION, that collection is allocated from heap memory and can never be deallocated. It is recommended that this be done only in library-level packages.

In this form the pragma must occur in the same declarative part as the specification of the supplied *access_type*. The *access_type* must have a `'Storage_Size` length clause associated with it before the pragma is encountered. When the second parameter is DEFAULT, the pragma affects all dynamically allocated data including memory associated with collections unless a specific pragma exists for a particular collection.

# Pragma MEMORY_POOL

The implementation-defined pragma `MEMORY_POOL` is used to change physical memory pool attributes from their default values for a memory pool. The pragma affects the mapping of abstract memory to physical memory.

**`pragma MEMORY_POOL (`***pool_spec*`,` *memory_spec*`);`**

*pool_spec*

See "Pool Specifiers" on page 6-21 for more information.

*memory_spec*

*memory_spec* ::= { *global_spec* | *local_spec* | *physical_spec*}

Specifies new values for memory pool attributes. If the `MEMORY_POOL` pragma is not specified for a particular pool (or for the DEFAULT pool), the default value for the *memory_spec* for that pool is determined from the environment (see **`run(1)`**).

**`GLOBAL`**

Uses physical global memory.

**`LOCAL`**, *mp_cpu_bias* [, *hardness*]

Uses physical local memory.

*mp_cpu_bias*

*cpu_bias*

Identifies which physical local memory pool to utilize. Distinct physical local memory pools are identified by specifying a CPU bias which contains a (partial) list of CPU numbers corresponding to a CPU board. A CPU bias is a mask in which the relative bit number identifies a CPU number (LSB corresponds to CPU #0). Note that the *cpu_bias* must specify at least one CPU (cannot be zero). The *cpu_bias* is used to locate a CPU board's local memory pool.

The *cpu_bias* is searched starting with the LSB (least significant bit) and the first CPU specified by the bias determines which CPU board is selected.

For example, assume that a user provides a *cpu_bias* with bits that specified CPUs existing on two different CPU boards. In that case, the CPU board selected would be the board that holds the lowest numbered CPU.

**`HOME`**

Allocates the memory pool from the LOCAL memory associated with the CPU on which the appropriate task is running. For TEXT and DATA memory pools, the appropriate

task is the **ENVIRONMENT** task and the allocation occurs before the **ENVIRONMENT** task executes any Ada code. For COLLECTION memory pools, the appropriate task is the task that elaborates the access type associated with the memory pool and the allocation occurs at the time of that elaboration. For STACK memory pools, the appropriate task is the one that will be using the stack during its execution, and the allocation occurs when that task is created. In any of these cases, if a task migrates to another CPU after the allocation occurs, the memory will *not* also migrate.

*hardness*

Controls usage of physical global memory if insufficient physical local memory is available.

**PHYSICAL**, *address*

Uses a specified physical memory address. (Note that this can only be used for non-DEFAULT STACK, COLLECTION, or DATA pools.)

*address*

The physical memory address at which the first storage unit of the given pool should be located. It is the user's responsibility to ensure that actual physical memory of some kind is located at the given address and is of a sufficient size for the given pool.

On Series 6000 systems, there are two kinds of physical memory pools:

- Global memory (1 pool)
- Local memory (up to 4 pools, 1 per CPU board)

*Global memory* is available to all CPUs via a system-wide bus. *Local memory* is available to CPUs via a local bus physically located on the same CPU board as the local memory. Accessing local memory from a foreign board CPU is allowed but is extremely costly and should be prevented in all time-critical areas.

**NOTE**

RedHawk does not currently support Non Uniform Memory Architectures (NUMA).

Use of this pragma requires the `CAP_SYS_NICE` capability (see "Capabilities" on page 1-3).

See “Memory Attributes” on page 6-20 for a list of other pragmas that modify memory attributes. In addition, “RM Annex L: Pragmas” on page M-104 lists all implementation-dependent and implementation-defined pragmas.

## Pragma POOL_CACHE_MODE

The implementation-defined pragma `POOL_CACHE_MODE` defines the cache mode for a memory pool.

**`pragma POOL_CACHE_MODE (`***pool_spec*`,` *cache_mode*`);`**

*pool_spec*

See “Pool Specifiers” on page 6-21 for more information.

*cache_mode*

The optional *cache_mode* sets the specified system cache attribute on the associated memory pool (see the **`memadvise(2)`** service for more information). This parameter can be either `COPYBACK` or `NCACHE`.

`COPYBACK`

Use the operating system’s COPYBACK cache mode. In COPYBACK cache mode, only a single task is usually modifying a semi-private data area at any given point in time and other tasks will not read the update immediately. This mode does not cause a cache flush or memory bus access until another CPU reads the data

If there is no DEFAULT pool, this parameter value is `COPYBACK`.

See “Memory Attributes” on page 6-20 for a list of other pragmas that modify memory attributes. In addition, “RM Annex L: Pragmas” on page M-104 lists all implementation-dependent and implementation-defined pragmas.

## Pragma POOL_LOCK_STATE

The implementation-defined pragma `POOL_LOCK_STATE` defines the lock state of a memory pool.

**`pragma POOL_LOCK_STATE (`***pool_spec*`,` *lock_state*`);`**

*pool_spec*

See “Pool Specifiers” on page 6-21 for more information.

*lock_state*

The keyword `LOCKED` or `UNLOCKED`.

`LOCKED`

means the memory pages are physically locked in memory and cannot be swapped out by the operating system.

`UNLOCKED`

means the memory pages can be swapped out by the operating system.

The default is the value specified for the DEFAULT pool. If there is no DEFAULT pool, the default is `UNLOCKED`.

By default, all pages are unlocked. In contrast, if a program specifies

```
pragma POOL_LOCK_STATE (DEFAULT, LOCKED);
```

then by default, all pages are locked, even if allocated via user system calls.

If a program specifies that DATA, DEFAULT or COLLECTION, DEFAULT is to be locked in local memory, then task migrations to foreign CPU boards are inhibited.

Other actions cause memory to be locked as well, including:

- User invocation of system services such as **`plock(2)`**, **`mlock(2)`**, etc.

Use of this pragma to request page locking requires the `CAP_IPC_LOCK` capability (see "Capabilities" on page 1-3).

See "Memory Attributes" on page 6-20 for a list of other pragmas that modify memory attributes. In addition, "RM Annex L: Pragmas" on page M-104 lists all implementation-dependent and implementation-defined pragmas.

## Pragma POOL_SIZE

The implementation-defined pragma `POOL_SIZE` permits the setting of the size for a STACK or COLLECTION memory pool.

**`pragma POOL_SIZE (`***sizeable_spec*`, `*size_spec*`);`**

*sizeable_spec*

See "Pool Specifiers" on page 6-21 for more information.

*size_spec* ::= {*size* | `UNLIMITED`}

*size*

A static non-negative number that controls the amount of space allocated for an Ada program's use.

`UNLIMITED`

A value that is allowed only for the COLLECTION, DEFAULT and STACK, ENVIRONMENT memory pools.

This pragma, if specified for the STACK, DEFAULT pool, will not affect the size of the STACK, ENVIRONMENT pool. This is the only pragma where such a statement is true. The implementation is this way so that the stack size for the **ENVIRONMENT** task can continue to be UNLIMITED, which is its default value. This value can always be overridden explicitly, though.

If this pragma is not specified for the **ENVIRONMENT** task's STACK pool, the default value is UNLIMITED. If this pragma is not specified for a task type's STACK pool, the default value is the task type's 'Storage_Size value if it exists, and 20,480 otherwise. If no POOL_SIZE pragma is valid for a task object or a task other than the **ENVIRONMENT** task, the default value for that real task is 20,480. The default values for ghost tasks are as follows:

**Table 6-1. Stack Pool Sizes for Ghost Tasks**

| Shadow Type | Default Stack Size |
|---|---|
| SHADOW | N/A |
| COURIER | 10240 |
| INTR_COURIER | 10240 |
| ADMIN | 12800 |
| TIMER | 12800 |

If this pragma is unspecified for the COLLECTION, DEFAULT pool, its value is UNLIMITED. If this pragma is unspecified for any other COLLECTION pool, then its default value is the value of the 'Storage_Size attribute for the collection.

**WARNING**

A shell's default stack limit occasionally causes storage problems for the compiler and other large compiled programs because it may provide too little stack space for the **ENVIRONMENT** task (main program). To resolve these problems, users may need to alter the shell's stack limit and recompile.

Most Bourne shell implementations do not allow stack sizes to be modified.

To reset the default stack size in the either the bash or Korn shell, users execute the following shell command:

```
$ ulimit -s kbytes
```

The C shell allows its default stack size of 512K bytes to be reset as high as the default process size. To alter the default stack size for the C shell, users execute the following shell command:

```
$ limit stacksize number
```

See "Memory Attributes" on page 6-20 for a list of other pragmas that modify memory attributes. In addition, "RM Annex L: Pragmas" on page M-104 lists all implementation-dependent and implementation-defined pragmas.

## Pragma POOL_PAD

The implementation-defined pragma POOL_PAD sets the pad for a STACK memory pool.

```
pragma POOL_PAD (paddable_spec, size);
```

*paddable_spec* ::= {*stack_pool*}

See "Pool Specifiers" on page 6-21 for more information.

*size*

A non-negative number that controls the amount of additional pad after the stack size. This value has no meaning when the stack size for the same pool is UNLIMITED.

This additional space is intended only for use by the run-time system or for signal handlers. For ADMIN ghost tasks, the default is 12,800; otherwise, it is 8,192.

See "Memory Attributes" on page 6-20 for a list of other pragmas that modify memory attributes. In addition, "RM Annex L: Pragmas" on page M-104 lists all implementation-dependent and implementation-defined pragmas.

# Protected Object Attributes

## Pragma PROTECTED_PRIORITY

The implementation-defined pragma PROTECTED_PRIORITY sets the scheduling priority for a protected object. Protected object priority values determine the priorities of tasks during protected actions as described in sections D.1 and D.3 of the Ada 95 Reference Manual.

```
pragma PROTECTED_PRIORITY (scheduling_priority
                           [,protected_object_specifier]);
```

*scheduling_priority*

A required integer expression specifying the scheduling priority. It is in the range `System.Any_Priority'Range` as defined in the package `System`. See "Task Scheduling" on page 5-3 for more information.

Values that fall within `System.Interrupt_Priority'Range` will be truncated to the actual maximum interrupt priority allowed on the target system executing the program.

Values less than `0` are considered to be values relative to `System.Priority'Last+1`. The following pragmas are equivalent:

```
pragma PROTECTED_PRIORITY
    (System.Priority'Last);


pragma PROTECTED_PRIORITY (-1);
```

*protected_object_specifier* ::= *ordinary_protected_object*

The two-parameter form of pragma `PROTECTED_PRIORITY` must appear in the same declarative part as the referenced protected object.

The one-parameter form must appear within the protected object itself. The protected object is assumed to be that in whose context the pragma appears.

Priorities in excess of `System.Interrupt_Priority'First` will cause all code associated with the protected object to execute with all external maskable machine interrupts masked. See "Priorities" on page 5-5 for a discussion of interrupt level execution and associated restrictions.

The `PROTECTED_PRIORITY` pragma differs from the language-defined pragma `PRIORITY` in that it can be applied to additional entities that pragma `PRIORITY` cannot (e.g. protected objects themselves, implementation-defined tasks associated with protected object interrupt handlers, etc).

See "RM Annex L: Pragmas" on page M-104 for a list of all implementation-dependent and implementation-defined pragmas.

# 7
# Interrupt Handling


Software Interrupts . . . . . 7-2
COURIER Ghost Tasks . . . . . 7-3
SHADOW Ghost Tasks . . . . . 7-4
Hardware Interrupts . . . . . 7-4
INTR_COURIER and COURIER Ghost Tasks . . . . . 7-5
SHADOW Ghost Tasks . . . . . 7-6
Privileges for Unrestricted Hardware Interrupts . . . . . 7-6
Interrupt Attachments . . . . . 7-6
Package Ada.Interrupts.Names . . . . . 7-6
Package Ada.Interrupts.Services . . . . . 7-7
Task Executives via Protected Handlers . . . . . 7-7
Example . . . . . 7-7
Description of Example . . . . . 7-10

# 7
# Interrupt Handling

MAXAda supports both software and hardware interrupt handlers. Software interrupt handlers allow applications to recieve and process operating system signals (see **signal(2)**) as calls to protected procedures or task entries. Hardware interrupt handlers allow applications to receive and process machine-generated interrupts as calls to protected procedures or task entries. Hardware interrupts include: real-time clocks (RTC), edge-triggered interrupts (ETI), and all system, VME, and PCI interrupts.

A list of software and hardware interrupts are defined in the **predefined** package `ada.interrupts.names`. This package includes predefined values of type `ada.interrupts.interrupt_id` which represent the software interrupts (signals) and all the real-time clock and edge-triggered interrupts.

Additional implementation-defined support packages are provided in **vendorlib**:

- `ada.interrupts.distrib_control`

  The `ada.interrupts.distrib_control` package provides services for the configuration and manipulation of distributed devices associated with closely-coupled systems. See **rcim_distrib_intr(4)** for more information.

- `ada.interrupts.eti_control`

  The package `ada.interrupts.eti_control` provides services for the configuration, programming, and manipulation of edge-triggered interrupt devices. See **rcim_eti(4)** for more information.

- `ada.interrupts.names.services`

  The package `ada.interrupts.names.services` provides textual information on device names, CPU biases, and reserved names associated with interrupt devices and values of `ada.interrupts.interrupt_id`.

- `ada.interrupts.pig_control`

  The package `ada.interrupts.pig_control` provides services for the configuration and manipulation of programmable interrupt generation devices associated with closely-coupled systems. See **rcim_pig(4)** for more information.

- `ada.interrupts.rtc_control`

  The package `ada.interrupts.rtc_control` provides services for the configuration, programming, and manipulation of real-time clock devices. See **rcim_rtc(4)** for more information.

- `ada.interrupts.services`

  The package `ada.interrupts.services` provides for the encoding, enabling, and decoding of values of `ada.interrupts.interrupt_id`.

The recommended mechanism for handling interrupts in Ada programs is to use pragma `INTERRUPT_HANDLER`, pragma `ATTACH_HANDLER`, or `ada.interrupts.attach_handler` with protected procedures. However, MAXAda still supports the obsolescent form of interrupt handling via task entries. An alternative method for handling interrupts is to bypass the language defined mechanisms completely and interface directly to the operating system. In the case of software interrupts (signals), this is relatively easy and maintainable. However, in the case of hardware interrupts it is complex and dangerous; utilization of the language-defined and implementation-supported mechanism is highly recommended instead.

The following definitions, paraphrased from RM C.3(2), are presented as they are important in subsequent discussions.

> *Generation* of an interrupt is the event in the underlying hardware or system that makes the interrupt available to the program.
>
> *Delivery* [of an interrupt] is the action that invokes part of the program as response to the interrupt.
>
> An *occurrence* of an interrupt is separable into generation and delivery.
>
> Between generation and delivery, the interrupt is *pending*.
>
> When an interrupt is *blocked*, all generated instances of that interrupt are prevented from being delivered.

MAXAda considers delivery to be the execution of the protected handler associated with the interrupt (or the rendezvous with the task entry associated with the interrupt).

Unfortunately, the operating system utilities and services use similar terminology in a slightly different manner. Subsequent discussions within this chapter will use these terms as defined above (paraphrased from RM C.3(2)), not as they normally appear in system service descriptions.

# Software Interrupts

Software interrupts are based on the operating system concept of signals (see **`signal(2)`**).

The signals SIGFPE, SIGSEGV, and SIGADA are used by the run-time system. Table 7-1 shows the type of erroneous program behavior that can result from intercepting these signals with user-defined signal handlers.

**Table 7-1. Erroneous Behavior Due to User-Defined Signal Handlers**

| Signal Used | Erroneous Behavior |
|---|---|
| SIGFPE | certain types of numeric exceptions may no longer be detected |
| SIGSEGV | STORAGE_ERROR will no longer be raised when such a signal occurs |
| SIGRTMIN+15 (47) | task preemption may cease to function, and certain kinds of task interactions may fail |

The following set of software signals are reserved (and therefore not available to be attached to protected procedures):

- SIGRTMIN+15 (47)
- SIGKILL
- SIGSTOP
- Signal values above 64
- SIGSEGV (if the POSIX package is in use)
- SIGALRM (if the POSIX package is in use)
- SIGFPE
- SIGILL (if the POSIX package is in use)

## COURIER Ghost Tasks

After a signal is generated, it is scheduled by the operating system for interception by the MAXAda run-time system which addresses it properly for subsequent delivery. An implementation-provided ghost task, called a COURIER task, is responsible for delivery of the addressed signal. If the interrupt is currently blocked, then it is queued internally by the COURIER task and will be delivered subsequently. An interrupt would be blocked only if its associated protected object had an ongoing protected action (or if its associated task, in the obsolescent model, was not suspended at an (open) accept for its entry) or other tasks executing in the system.interrupt_priority range were using resources which prevented delivery (e.g. they were executing on the CPU at the time). Once a signal is intercepted by the run-time system, a subsequent attempt at delivery will be made; such signals are not lost, they are queued.

The time taken for addressing an interrupt is extremely short; it is unrelated to the delivery of the signal or the execution of the associated protected procedure itself. Once interception of the signal is scheduled by the operating system, the signal mask of the interceptee contains the signal number of interest; it is not cleared until interception is complete (see

**sigprocmask(2)**). If another instance of the same signal is generated while the interceptee has the signal number set in its signal mask, the operating system will either queue or discard the new signal. Whether or not such signals are discarded is dependent on how the signal was initiated (see **sigaction(2)**).

Therefore, when a protected procedure handler is executed, only signals associated with that handler are blocked. They are blocked in the RM sense that they cannot be delivered to the protected handler; they are not necessarily blocked in the operating system sense of signal blocking (see **sigprocmask(2)**). Note that the execution of the handler will not be interrupted by interception of that signal number; task dispatching rules coupled with the priority of the protected object ensure this.

## SHADOW Ghost Tasks

There is a ghost task associated with a software interrupt task handler, called a SHADOW task. It is not a physical task in any real sense. It merely acts as the virtual caller of the interrupt handler. It does not, however, physically execute on any server or CPU.

# Hardware Interrupts

Hardware interrupts are machine-generated interrupts. Machine interrupt handling is based on the specific driver associated with the device generating the interrupt. The interrupt courier task blocks in an **ioctl** call waiting for the interrupt to occur.

Machine interrupts are further divided into two categories: “restricted” and “unrestricted”. A “restricted” interrupt places restrictions upon the code executed by its handler. An “unrestricted” interrupt places no such restrictions; the handler may execute any legal Ada statement.

A handler which handles a “restricted” interrupt will execute at operating system interrupt priority level (IPL); these priorities correspond to the Ada priorities in the range:

```
interrupt_priority'first+1 .. interrupt_priority'last
```

as presented in “Priorities” on page 5-5. Such a handler must follow the restrictions indicated in “Restrictions for Priorities in the System.Interrupt_Priority Range” on page 5-9. Violating those restrictions almost always will result in a system hang or panic.

**NOTE**

Restricted interrupts are not currently supported on RedHawk.

However, use of protected action handlers whose ceiling priority exceeds `interrupt_priority'first` are supported. While such protected actions are executed, all external maskable machine interrupts are masked.

See "Restrictions for Priorities in the System.Interrupt_Priority Range" on page 5-9.

## INTR_COURIER and COURIER Ghost Tasks

An INTR_COURIER ghost task is provided by the implementation for each protected procedure handler or task which handles a machine interrupt.

When a machine interrupt is generated, the INTR_COURIER for the associated interrupt returns from its blocking **`ioctl`** call.

The INTR_COURIER has the following responsibilities:

1. It addresses the interrupt for subsequent delivery.
2. Depending on circumstances described below, it may deliver the interrupt itself or notify the COURIER task of the interrupt.

Addressing the interrupt for delivery is a very fast operation; it simply involves determining which handler is currently attached to the interrupt.

For protected procedure handlers, if the interrupt is designated as "restricted", then the INTR_COURIER will deliver the interrupt itself. Delivery for a protected procedure handler involves initiating a protected action which invokes the associated protected procedure. If a protected action cannot be initiated (because a protected action is already underway for the associated protected object), the INTR_COURIER will spin on its CPU waiting for the action to complete. Note that an appropriate choice of the ceiling priority associated with the protected object, in combination with task dispatching rules, ensures that the interrupt will not preempt a protected action for that protected object (otherwise the INTR_COURIER might spin forever). When handling a "restricted" machine interrupt, the user must ensure that the ceiling priority of the associated protected object matches (or exceeds) the priority of the interrupt; otherwise, when the attempt at delivery is made, the INTR_COURIER will cause a ceiling violation and the exception PROGRAM_ERROR will be raised. The exception will be handled by the INTR_COURIER task and the interrupt will be lost (a (suppressible) message is printed by the MAXAda run-time when this occurs). For "restricted" interrupts, it is important to understand that the computer system as a whole is prevented from handling machine interrupts of like or lower priority, on the same CPU, while the INTR_COURIER delivers the interrupt or when protected actions for the associated protected object are executed in general.

Alternatively, for protected procedure handlers, if the interrupt is "unrestricted", the INTR_COURIER will notify another ghost task, the COURIER task, who will deliver the interrupt. If another instance of the same interrupt is generated before the COURIER

delivers the previous interrupt, it will be queued by the MAXAda run-time system for subsequent delivery. Thus, the computer system as a whole is not prevented from handling machine interrupts of like or lower priority while the COURIER delivers an interrupt or during protected actions associated with the protected object.

For task handlers, the INTR_COURIER simply notifies the COURIER task of the interrupt unless the user has specified that the address space of the entire application is to be locked into memory (see "Pragma POOL_LOCK_STATE" on page 6-25). In this case, the INTR_COURIER attempts to obtain critical access to the task in a non-blocking manner. If it is able to obtain access to the task and the task is blocked with an open accept alternative for the entry, the INTR_COURIER delivers the interrupts; otherwise it notifies the COURIER task of the interrupt. If the COURIER task is already busy delivering a previous interrupt, then the interrupt is queued by the MAXAda run-time system for subsequent delivery.

## SHADOW Ghost Tasks

The SHADOW task associated with a hardware interrupt handler for a task entry serves the same purpose as that for a software interrupt handler. It acts as the virtual caller of the interrupt handler. It does not physically execute on any server or CPU.

### Privileges for Unrestricted Hardware Interrupts

The permissions on the associated device files must be readable and writable by the user process; e.g. **`/dev/rcim/rtc1`**.

## Interrupt Attachments

The language-defined package, `ada.interrupts`, is the basis for all interrupt attachments. It defines the type `interrupt_id`, which is an encoded integer which represents an interrupt. It also includes language-defined procedures for attaching and detaching interrupts.

MAXAda provides additional child packages to `ada.interrupts` which aid the user.

### Package Ada.Interrupts.Names

The package `ada.interrupts.names` provides predefined interrupt IDs for signals and commonly handled hardware devices. Note that the constants defined in that package may not have the same internal encoded value across architectures or systems. The constants are set by the MAXAda run-time system upon program elaboration.

These constants may be specified in `ATTACH_HANDLER` pragmas or calls to subprograms in the `ada.interrupts` package.

### Package Ada.Interrupts.Services

Since values of `ada.interrupt_id` are encoded integers, this package (subsequently referred to as `AIS` in this chapter), provides a mechanism for encoding the values based on the common identifier for the interrupt; either a signal number or a machine interrupt identifier.

To obtain an `interrupt_id` for a signal, invoke the routine `ais.encode_signal_interrupt_id` and supply the signal number. To obtain the signal number associated with a value of `interrupt_id`, first check that the interrupt ID is indeed an encoded signal via the function `ais.is_signal_interrupt_id` and then invoke `ais.decode_signal_interrupt_id`.

**NOTE**

Currently on RedHawk, the only machine devices for which MAXAda supports interrupt handling are those listed in `ada.interrupts.names`.

## Task Executives via Protected Handlers

Protected objects naturally lend themselves to providing efficient task scheduling. The following example utilizes the receipt of an interrupt to begin the execution of a frame in a cyclic scheduler.

```
package executive is
     protected executive is
          procedure interrupt;
          entry wait_for_interrupt;
     private
          execute : boolean := false;
     end executive;
end executive;

package body executive is
     protected body executive is
          procedure interrupt is
          begin
               execute := true;
          end interrupt;
          entry wait_for_interrupt when execute is
          begin
               if wait_for_interrupt'count = 0 then
                    execute := false;
               end if;
          end wait_for_interrupt;
     end executive;
end executive;
```

### Example

A complete example is provided to illustrate this concept:

```
with system ;
generic
   type tasks is (<>) ;
package cyclic_scheduler is
--
   type cycles is mod 2**32 ;
   type cycle_counts is array (tasks) of cycles ;

   protected type scheduler (priority : system.interrupt_priority) is
      entry start_cycle (tasks) (overran, finished : out boolean) ;
      procedure interrupt ;
      procedure shut_down ;
      function current_cycle return cycles ;
   private
      pragma interrupt_handler (interrupt) ;
      pragma interrupt_priority (priority) ;
      cycle  : cycles := 0 ;
      counts : cycle_counts := (others => 0) ;
      stop   : boolean := false ;
   end scheduler ;
--
end cyclic_scheduler ;

package body cyclic_scheduler is
--
   protected body scheduler is
   --
      procedure interrupt is
      begin
         cycle := cycle + 1 ;
      end interrupt ;

      entry start_cycle (for t in tasks) (overran, finished : out boolean)
      when counts(t) /= cycle or stop is
      begin
         if stop then
            finished := true ;
            overran  := false ;
         else
            finished  := false ;
            counts(t) := counts(t) + 1 ;
            overran   := counts(t) /= cycle ;
         end if ;
      end start_cycle ;

      procedure shut_down is
      begin
         stop := true ;
      end shut_down ;

      function current_cycle return cycles is
      begin
         return cycle ;
      end current_cycle ;
   --
   end scheduler ;
--
end cyclic_scheduler ;

with cyclic_scheduler ;
with system ;
package scheduler_example is
--
   type tasks is (cpu_0, cpu_1) ;
```

```
   task type t (id : tasks; priority : system.priority; cpu_bias : integer)
is
      pragma task_priority (priority) ;
      pragma task_cpu_bias (cpu_bias) ;
   end t ;

   package sched is new cyclic_scheduler (tasks) ;

   scheduler : sched.scheduler (interrupt.priority'first) ;
   done      : boolean := false ;
--
end scheduler_example ;

with ada.text_io ;
package body scheduler_example is
--
   workload : integer := 10_0000 ;

   procedure work is
       x : long_float := 0.0 ;
   begin
      for i in integer range 1..workload loop
         x := x * x ;
      end loop ;
   end work ;

   task body t is
      overran  : boolean ;
      finished : boolean ;
   begin
      loop
         scheduler.start_cycle(id) (overran, finished) ;
         exit when overran or else finished ;
         work ;
      end loop ;
      if overran then
         ada.text_io.put_line ("Task " & tasks'image(id) & " overran") ;
      end if ;
      ada.text_io.put_line ("Task " & tasks'image(id) & " complete.") ;
   end t ;
--
end scheduler_example ;

with ada.interrupts ;
with ada.interrupts.names ;
with ada.interrupts.rtc_control ;
with ada.interrupts.services ;
with ada.text_io ;
procedure scheduler_example.main is
--
   package ai renames ada.interrupts ;

   cpu0_task   : t (id => cpu_0, priority => -10, cpu_bias => 2#0001#) ;
   cpu1_task   : t (id => cpu_1, priority => -11, cpu_bias => 2#0010#) ;
   rtc         : ai.rtc_control.rtc_id ;
--
begin
--
   ai.attach_handler (scheduler.interrupt'access,
                      ai.names.rtc2c1) ;

   rtc := ai.rtc_control.configure_rtc (cycle_time_msec => 16.6666666,
                                        id              => ai.names.rtc2c1)
;
```

```
   ai.rtc_control.start_rtc (rtc) ;
   loop
      exit when done ;
      delay 1.0 ;
      ada.text_io.put_line ("cycles =" &
                            sched.cycles'image(scheduler.current_cycle));
   end loop ;
   ai.rtc_control.stop_rtc (rtc) ;

   delay 0.020 ;

   scheduler.shut_down ;
--
end scheduler_example.main ;
```

### Description of Example

In the example above, a protected object is used to coordinate the cyclic scheduling of multiple tasks using a real-time clock as an external timing source.

It is handy to run the **a.monitor** tool to track the execution of the test. Invoke **a.monitor** with the name of the program file for the test (e.g. **a.monitor a.out**). Select the Tasks menu item from the View menu to track the number of interrupts delivered and the status of each task.

Receipt of the machine interrupt associated with expiration of the clock defines the start of a cycle. In the example, all tasks are scheduled to start execution at the beginning of a cycle.

Each task registers with the protected object when it is ready via a protected entry call to `start_cycle`. The tasks will block on that entry call until the beginning of the next cycle.

An entry family is used simply to detect cycle overruns for the tasks; otherwise a single entry might be used.

Under normal operation (i.e. if there are no overruns), all the tasks in the scheduler will be blocked on their entry call to `start_cycle`. Upon receipt of an interrupt, a protected action is started and the protected procedure `interrupt` is called, incrementing the current cycle count `cycle` by one.

As part of finishing the protected action (immediately after returning from procedure `interrupt`), the entry queues are services (see RM 9.5.3(13)). All tasks which blocked on the entry `start_cycle` before the interrupt occurred are released (since the entry barrier condition will now evaluate to TRUE (see RM 9.5.3(7)).

Inside the entry body, overruns for each task are detected by comparing the current cycle number to the cycle number when that task was last released.

Finally, when all tasks have been released, the protected action completes.

Note that if one of the tasks is released, completes its processing, and then makes another entry call to `start_cycle` before all the other tasks have been released, it will remain queued until the next interrupt occurs due to the barrier condition for that entry index (i.e. `counts(id)` will equal `cycle` until the next interrupt occurs).

The priority and cpu_bias of the tasks are set using pragmas and per-object expressions.

The priority of the protected object is set similarly.

# 3
# General Features

**Replace with Part 3 tab**

# Part 3 - General Features

## Part 3 General Features

Chapter 8 Shared Memory and Process Communication ..... 8-1

Chapter 9 Support Packages ..... 9-1

# 8
# Shared Memory and Process Communication


Shared Memory . . . . . 8-1
Shared Packages . . . . . 8-1
Pragma SHARED_PACKAGE . . . . . 8-1
Restrictions on Contents of Shared Packages . . . . . 8-4
Characteristics of Shared Packages . . . . . 8-4
Shared Package Semaphores . . . . . 8-5

# 8
# Shared Memory and Process Communication

This chapter describes how to use MAXAda to communicate between distinct processes. Through the use of implementation-defined pragmas and attributes, a user can write programs in the Ada programming language that interface to objects in other programs. Some of these other programs may even be written in languages other than Ada. These communications are provided only through implementation-defined features.

## Shared Memory

With the use of pragma SHARED_PACKAGE, Ada programs can interface to separate programs, possibly running on different CPUs on a multiple-CPU system.

This communication is achieved internally by utilizing shared memory services, such as, **shmget(2)**, **shmat(2)**, etc.

## Shared Packages

MAXAda has provided an implementation-defined pragma SHARED_PACKAGE. This provides for the sharing and communication of Ada objects in library-level packages between distinct Ada programs.

All variables declared in the specification of a package marked with pragma SHARED_PACKAGE (henceforth referred to as a *shared package*) are allocated in shared memory that is created and maintained by the implementation. As such, all Ada programs that reference the shared packages can communicate through variables in the specifications of those packages. Note that variables declared in the body of a shared package are *not* shared. Any objects declared in specifications of packages nested within shared packages are also shared as part of the same shared memory segment.

See also “4.1.4(12) Implementation-defined attributes” on page M-13 for more information related to pragma SHARED_PACKAGE.

## Pragma SHARED_PACKAGE

The implementation-defined pragma SHARED_PACKAGE provides for the sharing and communication of data declared within the specification of library-level packages.

Its syntax is:

**pragma SHARED_PACKAGE[("***params***")];**

*params*

an optional argument, that, if specified, must be a string constant containing a comma-separated list of system shared-segment configuration parameters, as defined below

The SHARED_PACKAGE pragma must appear within the specification of the library-level package. The pragma may also be repeated in the package body to allow the user to override the shared memory configuration parameters that were associated with the pragma in the specification. However, the pragma still affects only objects declared in the specification of the package.

The following is a list of the shared-segment configuration parameters that pragma SHARED_PACKAGE may accept:

key=*name*

Identifies the system shared-segment key to be used in subsequent **shmget(2)** system calls. These calls are done automatically by the implementation in configuring the shared segment.

*name* is considered to be the name of an existing file. This filename will then be translated to a shared segment key using the **ftok(3C)** service. Note that relative pathnames may be specified but will cause key translation to be dependent on the user's current working directory when program execution is initiated. If *name* is a numeric literal (a decimal integer or Ada octal- or hexadecimal-based literal), MAXAda interprets this as the actual system key, and does not translate it using the **ftok** service.

If no key is specified, MAXAda creates an empty file by the name:

*{absolute MAXAda environment path}*/.ada/shmem/*package_name*

and uses that file as the key for the SHARED_PACKAGE pragma.

ipc=(IPC_CREAT, IPC_EXCL, IPC_PRIVATE)

Allows the user to specify details about the initialization of the shared segment. By default, MAXAda applies ipc=(IPC_CREAT) to the shared package, thereby creating the shared segment if it did not previously exist. If any ipc parameters are given, they entirely replace the default ipc specification.

SHM_RDONLY

Specifies that the segment is available only for READ operations. MAXAda defaults shared package segments to READ/WRITE.

**CAUTION**

The current shared memory implementation does not allow the use of the 'LOCK and 'UNLOCK attributes with a SHM_RDONLY shared memory segment. Any use of these attributes with a package marked SHM_RDONLY will raise PROGRAM_ERROR at run time. See "Shared Package Semaphores" on page 8-5.

mode=*n*

Where *n* is assumed to be an octal number defining the access to the shared segment. By default, MAXAda applies mode=644 to the shared package, (owner read/write, group read, other read). The specified value for mode is ORed into the *shmflgs* parameter that MAXAda uses for the **shmget(2)** call. Additional bits can be supplied via mode to control caching, etc. (e.g., "mode = 8#200644#" would specify SHM_COPYBACK, as well as the 644 mode). For more information, see the *PowerMAX OS Programming Guide*.

SHM_LOCAL

Requests that pages for the shared segment be allocated from the local memory pool. If a program attempts to attach to a segment which has been allocated from local memory on a different CPU, then the attachment will fail. See **shmget(2)**.

SHM_LOCK

Specifies that virtual memory pages be locked into physical memory at program start-up time. Doing this makes these pages immune to swapping.

SHM_HARD

When used in conjunction with SHM_LOCAL, specifies that pages for the shared segment *must* be allocated from the local memory pool. If pages are not available from local memory then the signal SIGSEGV is delivered to the process. See **shmget(2)**.

no_bsem

Prohibits the use of the shared package lock attributes 'LOCK and 'UNLOCK. In shared packages marked with this parameter, binary semaphore space is not initialized in the shared memory segment. Any attempt to make use of the lock attributes in a shared package marked with no_bsem will raise PROGRAM_ERROR at run time. Unlike RDONLY shared packages, packages marked by no_bsem have READ/WRITE capability.

bind=*n*

Where *n* is assumed to be an octal number. The segment will be attached to the physical memory address specified by *n*. This parameter requires the CAP_IPC_LOCK capability (see "Capabilities" on page 1-3).

A detailed explanation of the IPC and SHM flags, and access modes may be found in the following man pages: **shmget(2)**, **ipcs(1)**, **ipcrm(1)**, and **chmod(1)**.

## Restrictions on Contents of Shared Packages

The implementation restricts the kinds of objects that can be declared in a shared package. Objects that cannot be declared in a shared package include:

- Unconstrained or dynamically sized objects
- Access type objects
- Generic instantiations

If any of these restrictions are violated, a warning message is issued and the package is not shared. These restrictions apply to nested packages as well. Note that if a nested package violates one of the preceding restrictions, it prevents the sharing of all enclosing packages as well.

Task objects are allowed within shared packages, however, the tasks as well as the data defined within those tasks are not shared.

Packages that require initialization should not be marked with the pragma unless the user is prepared to deal with concurrency issues. The compiler does not reject the pragma in these cases; however, every program that uses the shared package will initialize it during program elaboration. Initialization can occur as a result of an explicit initialization by the user (e.g., `a : integer := 54 ;`) or implicitly due to an object's representation (an array or record with gaps). The compiler issues a warning message in either case.

## Characteristics of Shared Packages

With the valid application of pragma `SHARED_PACKAGE` to a library-level package, the following assumptions can be made about the objects declared in the specification of the package:

- The lifetime of such objects can be greater than the lifetime defined by the complete execution of a single program.
- The lifetime of such objects is guaranteed to extend from the elaboration of the shared package by the first concurrent program until the termination of execution of the last concurrent program.
- A program that elaborates a shared package inherits the state of the objects within it, if their lifetime, as defined before, has not expired.

In the preceding assumptions, a *concurrent program* is defined to be any Ada program that elaborates the body of a shared package, whose span of execution, from elaboration of such a package to termination, overlaps that of another such program.

In actuality, the shared memory segments created by these programs remain even after the last concurrent program has exited. The values of objects within these segments remain valid until the segment is destroyed, or until the system is rebooted. Segments may be explicitly destroyed through the shared memory service **`shmctl(2)`**, to which an interface is provided in the MAXAda package `shared_memory_support`. Alternatively, the user may obtain information about active shared memory segments through the **`ipcs(1)`** utility. These segments may be removed via the **`ipcrm(1)`** utility.

Objects declared in shared packages that have not been implicitly or explicitly initialized may have invalid representations if of a scalar type, or may be abnormal otherwise. It is a bounded error for a program to evaluate an object with an invalid representation, and it is erroneous for a program to evaluate an abnormal object. This implementation does not prevent these evaluations. See RM 13.9.1(9) for more details.

The preceding discussion describes the intent that several Ada programs may begin, continue and complete their execution simultaneously, with the contents of the variables in the shared packages consistent with the execution of those programs.

The association of a system shared memory segment with the shared package occurs during the elaboration of the package body. If this association should fail due to system shared memory constraints, access, or improper use of shared memory configuration parameters, an error message is issued and the `PROGRAM_ERROR` exception is raised.

**WARNING**

If the **`shmbind(2)`** attempt fails due to `EBUSY`, the implementation will ignore the error and continue, assuming that another program has already bound the segment to the desired location. Shared memory segments bound to physical memory should be freed manually by the user via **`ipcrm(1)`**.

**CAUTION**

By default, every shared package that is available for `READ/WRITE` has a binary semaphore initialized which starts 12 bytes before the end of the segment and extends to the end of the segment. If a shared package is bound to a device using the `bind=` parameter, be aware that the contents of these bytes may change if the `'LOCK` and `'UNLOCK` attributes are utilized. The only exceptions are those shared packages which are defined as `SHM_RDONLY` or those marked by the `no_bsem` parameter. In these cases, the semaphore space is not initialized, but it is still present.

## Shared Package Semaphores

Because programs may wish to define critical sections to reference and update variables within the shared packages, MAXAda has provided semaphore operations, `P'LOCK` and `P'UNLOCK`, with which this can be accomplished.

The following programs illustrate a use of pragma `SHARED_PACKAGE`, `'LOCK` and `'UNLOCK`.

**Example:**

```
--
-- shared_data.a
--
```

```
package shared_data is
--
   -- Data definitions
   initialization_complete : boolean ;
   writer_count : integer ;

   -- Message Buffer Definitions
   subtype message_range is integer range 0..20 ;
   message : array (message_range) of string (1..3) ;
   message_index : integer ;
   --
   pragma shared_package ;
--
end shared_data ;
package body shared_data is

begin
--
   -- Every program which uses this shared package
   -- will execute this code at elaboration time.
   --
   -- This holds all programs (they all wait till this flag is true)
   initialization_complete := false ;
--
end shared_data ;

--
-- init.a
--
with shared_data ;

procedure init is

begin
--
   shared_data.message_index := -1 ;
   shared_data.writer_count := 0 ;
--
end init ;

--
-- starter.a
--
with shared_data ;

procedure starter is

begin
--
   shared_data.initialization_complete := true ;
--
end starter ;

--
-- writer.a
--
with ada.command_line;
with shared_data ;

procedure writer is
   index : integer ;

begin
--
   -- Increment the writer count
```

```
   shared_data'lock ;
   shared_data.writer_count := shared_data.writer_count - 1 ;
   shared_data'unlock ;

   -- Wait for starter program
   while not shared_data.initialization_complete loop
      delay 1.0;
   end loop ;

   -- Allocate slots in the shared message buffer and fill them in
   while shared_data.message_index < shared_data.message'last loop
   --
      -- Lock the package
      shared_data'lock ;

      -- Reserve this index
      if shared_data.message_index >= shared_data.message'last then
         exit ; -- Might have changed already
      end if ;
      shared_data.message_index := shared_data.message_index + 1 ;
      index := shared_data.message_index ;

      -- Unlock the package
      shared_data'unlock ;

      -- Write the argument supplied to this routine to the buffer
      shared_data.message(index) := ada.command_line.argument(1)(1..3) ;

      -- Waste some time
      delay 1.0 ;
   --
   end loop ;

   -- Tell the reader we are done
   shared_data'lock ;
   shared_data.writer_count := shared_data.writer_count + 1 ;
   shared_data'unlock ;
--
end writer ;


--
-- reader.a
--
with ada.text_io ;
with shared_data ;

procedure reader is

begin
--
   -- Wait for the initialization program to complete
   while not shared_data.initialization_complete loop
      delay 1.0 ;
   end loop ;

   -- Wait for all writers to finish
   while shared_data.writer_count < 0 loop
      delay 1.0 ;
   end loop ;


   -- Write out the messages
   for index in shared_data.message_range loop
      ada.text_io.put_line (shared_data.message(index)) ;
```

```
   end loop ;
--
end reader ;
```

Introduce the source files (your environment should already exist - if not, create one with **a.mkenv**):

```
$ a.intro shared_data.a  init.a  starter.a  writer.a
    reader.a
```

Now create an active partition for each of the units:

```
$ a.partition -create active init
$ a.partition -create active writer
$ a.partition -create active reader
$ a.partition -create active starter
```

Now build all of the partitions:

```
$ a.build -allparts
```

From the shell, invoke the programs in the following order. Note that the **&** character instructs the shell to execute the program in the background.

```
$ init
$ writer one &
$ writer two &
$ writer thr &
$ reader &
$ starter
```

The **reader** program will wait until all **writer** programs have finished and then print the contents of the message buffer. The message buffer will reflect the fact that all three **writer**s are writing simultaneously.

# 9
# Support Packages


Supplied Environments . . . . . 9-5
predefined . . . . . 9-6
vendorlib . . . . . 9-8
bit_ops . . . . . 9-9
ada.exceptions.addresses . . . . . 9-9
ada.numerics.constants . . . . . 9-10
ada.real_time.local . . . . . 9-10
runtime_configuration . . . . . 9-10
shared_memory_support . . . . . 9-10
system.addresses . . . . . 9-11
system.information . . . . . 9-11
system.storage_pools.standard . . . . . 9-11
system.storage_pools.standard.objects . . . . . 9-11
publiclib . . . . . 9-11
c_to_ada_types . . . . . 9-12
character_type . . . . . 9-12
curses . . . . . 9-12
qsort . . . . . 9-12
rtdm . . . . . 9-12
real_time_data_monitoring . . . . . 9-12
deprecated . . . . . 9-13
obsolescent . . . . . 9-13
posix_1003.1 . . . . . 9-14
posix_1003_1 . . . . . 9-14
posix_1003.5 . . . . . 9-15
sockets . . . . . 9-16
sockets . . . . . 9-16
general . . . . . 9-16
night_trace_bindings . . . . . 9-17
timers . . . . . 9-17

# 9
# Support Packages

MAXAda supplies a number of environments containing various packages that can be used for program development.

**Table 9-1. Support environments**

| Keyword | Environment |
|---|---|
| **predefined** | /usr/ada/*rel_name*/predefined |
| **vendorlib** | /usr/ada/*rel_name*/vendorlib |
| **publiclib** | /usr/ada/*rel_name*/publiclib |
| **rtdm** | /usr/ada/*rel_name*/rtdm |
| **deprecated** | /usr/ada/*rel_name*/deprecated |
| **obsolescent** | /usr/ada/*rel_name*/obsolescent |
| **posix_1003.1** | /usr/ada/*rel_name*/bindings/posix_1003.1 |
| **posix_1003.5** | /usr/ada/*rel_name*/bindings/posix_1003.5 |
| **sockets** | /usr/ada/*rel_name*/bindings/sockets |
| **general** | /usr/ada/*rel_name*/bindings/general |

where *rel_name* is the name of the MAXAda release.

Table 9-2 lists the MAXAda support packages and the environments in which they are contained.

**Table 9-2. Support packages**

| package | environment |
|---|---|
| ada | **predefined** |
| ada.calendar | **predefined** |
| ada.characters | **predefined** |
| ada.characters.handling | **predefined** |
| ada.characters.latin_1 | **predefined** |
| ada.command_line | **predefined** |
| ada.direct_io | **predefined** |
| ada.dynamic_priorities | **predefined** |

**Table 9-2. Support packages (Cont.)**

| package | environment |
|---|---|
| ada.exceptions | **predefined** |
| ada.exceptions.addresses | **vendorlib** |
| ada.finalization | **predefined** |
| ada.float_text_io | **predefined** |
| ada.float_wide_text_io | **predefined** |
| ada.integer_text_io | **predefined** |
| ada.integer_wide_text_io | **predefined** |
| ada.interrupts | **predefined** |
| ada.interrupts.distrib_control | **vendorlib** |
| ada.interrupts.eti_control | **vendorlib** |
| ada.interrupts.names | **predefined** |
| ada.interrupts.names.services | **vendorlib** |
| ada.interrupts.pig_control | **vendorlib** |
| ada.interrupts.rtc_control | **vendorlib** |
| ada.interrupts.services | **vendorlib** |
| ada.io_exceptions | **predefined** |
| ada.long_float_text_io | **predefined** |
| ada.long_float_wide_text_io | **predefined** |
| ada.numerics | **predefined** |
| ada.numerics.constants | **vendorlib** |
| ada.numerics.discrete_random | **predefined** |
| ada.numerics.elementary_functions | **predefined** |
| ada.numerics.float_random | **predefined** |
| ada.numerics.generic_elementary_functions | **predefined** |
| ada.numerics.long_elementary_functions | **predefined** |
| ada.real_time | **predefined** |
| ada.real_time.local | **vendorlib** |
| ada.sequential_io | **predefined** |
| ada.short_integer_text_io | **predefined** |
| ada.short_integer_wide_text_io | **predefined** |
| ada.storage_io | **predefined** |
| ada.streams | **predefined** |
| ada.streams.stream_io | **predefined** |
| ada.strings | **predefined** |
| ada.strings.bounded | **predefined** |
| ada.strings.fixed | **predefined** |
| ada.strings.maps | **predefined** |

**Table 9-2. Support packages (Cont.)**

| package | environment |
|---|---|
| ada.strings.maps.constants | **predefined** |
| ada.strings.unbounded | **predefined** |
| ada.strings.wide_bounded | **predefined** |
| ada.strings.wide_fixed | **predefined** |
| ada.strings.wide_maps | **predefined** |
| ada.strings.wide_maps.wide_constants | **predefined** |
| ada.strings.wide_unbounded | **predefined** |
| ada.synchronous_task_control | **predefined** |
| ada.tags | **predefined** |
| ada.task_attributes | **predefined** |
| ada.task_identification | **predefined** |
| ada.text_io | **predefined** |
| ada.text_io.text_streams | **predefined** |
| ada.tiny_integer_wide_text_io | **predefined** |
| ada.unchecked_conversion | **predefined** |
| ada.unchecked_deallocation | **predefined** |
| ada.wide_text_io | **predefined** |
| ada.wide_text_io.text_streams | **predefined** |
| binary_semaphores | **vendorlib** |
| bit_ops | **vendorlib** |
| c_to_ada_types | **publiclib** |
| calendar | **obsolescent** |
| character_type | **publiclib** |
| client_server_services | **vendorlib** |
| curses | **publiclib** |
| cyclic_scheduler | **vendorlib** |
| direct_io | **obsolescent** |
| distrib_services | **vendorlib** |
| eti_services | **vendorlib** |
| fbsched | **vendorlib** |
| indivisible_operations | **vendorlib** |
| interfaces | **predefined** |
| interfaces.c | **predefined** |
| interfaces.c.pointers | **predefined** |
| interfaces.c.strings | **predefined** |
| interfaces.restricted_fortran | **predefined** |
| interfaces.unchecked_c | **predefined** |

**Table 9-2. Support packages (Cont.)**

| package | environment |
|---|---|
| interval_timer | **vendorlib** |
| io_exceptions | **obsolescent** |
| machine_code | **obsolescent** |
| night_trace_bindings | **general** |
| posix | **posix_1003.5** |
| posix_1003_1 | **posix_1003.1** |
| posix_calendar | **posix_1003.5** |
| posix_configurable_file_limits | **posix_1003.5** |
| posix_configurable_system_limits | **posix_1003.5** |
| posix_file_locking | **posix_1003.5** |
| posix_file_status | **posix_1003.5** |
| posix_files | **posix_1003.5** |
| posix_group_database | **posix_1003.5** |
| posix_io | **posix_1003.5** |
| posix_local_signals | **posix_1003.5** |
| posix_permissions | **posix_1003.5** |
| posix_process_environment | **posix_1003.5** |
| posix_process_identification | **posix_1003.5** |
| posix_process_primitives | **posix_1003.5** |
| posix_process_primitives.local | **posix_1003.5** |
| posix_process_times | **posix_1003.5** |
| posix_signals | **posix_1003.5** |
| posix_supplement_to_ada_io | **posix_1003.5** |
| posix_terminal_functions | **posix_1003.5** |
| posix_unsafe_process_primitives | **posix_1003.5** |
| posix_user_database | **posix_1003.5** |
| qsort | **publiclib** |
| real_time_data_monitoring | **rtdm** |
| rescheduling_control | **vendorlib** |
| rtc_services | **vendorlib** |
| runtime_configuration | **vendorlib** |
| sequential_io | **obsolescent** |
| shared_memory_support | **vendorlib** |
| sockets | **sockets** |
| spin_locks | **vendorlib** |
| system | **predefined** |
| system.address_to_access_conversions | **predefined** |

**Table 9-2. Support packages (Cont.)**

| package | environment |
|---|---|
| system.addresses | **vendorlib** |
| system.information | **vendorlib** |
| system.machine_code | **predefined** |
| system.storage_elements | **predefined** |
| system.storage_pools | **predefined** |
| system.storage_pools.standard | **vendorlib** |
| system.storage_pools.standard.objects | **vendorlib** |
| task_synchronization | **vendorlib** |
| tasking_semaphores | **vendorlib** |
| text_io | **obsolescent** |
| timers | **general** |
| unchecked_conversion | **obsolescent** |
| unchecked_deallocation | **obsolescent** |
| user_trace | **vendorlib** |
| user_trace.raw | **vendorlib** |
| usermap_support | **vendorlib** |

# Supplied Environments

The following environments are supplied with MAXAda:


- "predefined" on page 9-6
- "vendorlib" on page 9-8
- "publiclib" on page 9-11
- "rtdm" on page 9-12
- "deprecated" on page 9-13
- "obsolescent" on page 9-13
- "posix_1003.1" on page 9-14
- "posix_1003.5" on page 9-15
- "sockets" on page 9-16
- "general" on page 9-16

# predefined

MAXAda provides the Predefined Language Environment (**predefined**) which contains packages as defined in Annex A of the Ada 95 Reference Manual. According to the Reference Manual, the library units listed in this Annex "shall be provided by every implementation".

**Table 9-3. predefined environment**

| package |
|---|
| ada |
| ada.calendar |
| ada.characters |
| ada.characters.handling |
| ada.characters.latin_1 |
| ada.command_line |
| ada.direct_io |
| ada.dynamic_priorities |
| ada.exceptions |
| ada.finalization |
| ada.float_text_io |
| ada.float_wide_text_io |
| ada.integer_text_io |
| ada.integer_wide_text_io |
| ada.interrupts |
| ada.interrupts.names |
| ada.io_exceptions |
| ada.long_float_text_io |
| ada.long_float_wide_text_io |
| ada.numerics |
| ada.numerics.discrete_random |
| ada.numerics.elementary_functions |
| ada.numerics.float_random |
| ada.numerics.generic_elementary_functions |
| ada.numerics.long_elementary_functions |
| ada.real_time |
| ada.sequential_io |

**Table 9-3. predefined environment (Cont.)**

| package |
|---|
| ada.short_integer_text_io |
| ada.short_integer_wide_text_io |
| ada.storage_io |
| ada.streams |
| ada.streams.stream_io |
| ada.strings |
| ada.strings.bounded |
| ada.strings.fixed |
| ada.strings.maps |
| ada.strings.maps.constants |
| ada.strings.unbounded |
| ada.strings.wide_bounded |
| ada.strings.wide_fixed |
| ada.strings.wide_maps |
| ada.strings.wide_maps.wide_constants |
| ada.strings.wide_unbounded |
| ada.synchronous_task_control |
| ada.tags |
| ada.task_attributes |
| ada.task_identification |
| ada.text_io |
| ada.text_io.text_streams |
| ada.tiny_integer_wide_text_io |
| ada.unchecked_conversion |
| ada.unchecked_deallocation |
| ada.wide_text_io |
| ada.wide_text_io.text_streams |
| interfaces |
| interfaces.c |
| interfaces.c.pointers |
| interfaces.c.strings |
| interfaces.restricted_fortran |
| interfaces.unchecked_c |

**Table 9-3. predefined environment (Cont.)**

| package |
|---|
| system |
| system.address_to_access_conversions |
| system.machine_code |
| system.storage_elements |
| system.storage_pools |

## vendorlib

This environment contains packages that do not collectively represent an Ada binding but serve as a collection of utility packages and thin bindings to Concurrent-specific services.

**Table 9-4. vendorlib environment**

| package |
|---|
| bit_ops |
| ada.exceptions.addresses |
| ada.numerics.constants |
| ada.real_time.local |
| runtime_configuration |
| shared_memory_support |
| system.addresses |
| system.information |
| system.storage_pools.standard |
| system.storage_pools.standard.objects |
| The following packages are discussed in Chapter 7, "Interrupt Handling" |
| ada.interrupts.distrib_control |
| ada.interrupts.eti_control |
| ada.interrupts.names.services |
| ada.interrupts.pig_control |
| ada.interrupts.rtc_control |
| ada.interrupts.services |

**Table 9-4. vendorlib environment (Cont.)**

| package |
|---|
| The following packages are discussed in Chapter 10, "Real-Time Extensions" |
| binary_semaphores |
| client_server_services |
| cyclic_scheduler |
| distrib_services |
| eti_services |
| fbsched |
| indivisible_operations |
| interval_timer |
| rescheduling_control |
| rtc_services |
| spin_locks |
| task_synchronization |
| tasking_semaphores |
| user_trace |
| user_trace.raw |
| usermap_support |

### bit_ops

The `bit_ops` package consists of bit_manipulation routines for type `integer`.

### ada.exceptions.addresses

The `ada.exceptions.addresses` package contains two functions that deal with exceptions:

- `originating_instruction`

  This function returns the address of the instruction which raised the exception associated with the supplied exception occurrence.

- `propagation_map`

  This function returns the list of instructions associated with the most recently raised exception in the calling task. The first entry in the list corresponds to the address of the instruction which raised the exception. Subsequent entries refer to instruction addresses along which the exception was propagated (or reraised) before reaching a handler.

  This function need not be called directly from the handler; it always reports on the last exception raised by the calling task.

### ada.numerics.constants

The `ada.numerics.constants` package contains constants whose values have been taken from the following sources:

1. *CRC Handbook of Tables for Mathematics*, Fourth Ed., Robert C. Weast (ed.), 1970, The Chemical Rubber Co.
2. Knuth, Donald E., 'Fundamental Algorithms', Vol. 1 of '*The Art of Computer Programming*', 2nd ed., 1973. (Appendix B).
3. Davis, Harold T. and Fisher, Vera J., 'Arithmetical Tables', Vol. III of '*Tables of the Mathematical Functions*', The Principia Press, Texas, 1962.
4. Fletcher, A. et al., '*An Index of Mathematical Tables*', Scientific Computing Service Limited, London, 1962.

Where values exist in more than one source, such values have been cross checked. In all cases, such values agree except for possibly a value of one in the last digit. In such cases of difference, the higher value is used, under the assumption that it is a rounded value and that the lower value is a truncated value.

### ada.real_time.local

The `ada.real_time.local` package contains two convenient functions that convert values of type `Ada.Real_Time.Time_Span` to `Long_Float`.

The values returned are in units of seconds.

```
function to_seconds (t : time_span) return long_float;
function fast (t : time_span) return long_float;
```

The `fast` function utilizes an algorithm that does the conversion more efficiently than the `to_seconds` function, but it loses precision for large time spans.

### runtime_configuration

The `runtime_configuration` package provides support for the retrieval and modification of certain run-time attributes.

### shared_memory_support

The `shared_memory_support` package contains Ada types, subprogram definitions, and interfaces to aid the user in manually interfacing to the shared memory system services.

This includes:

- System *defines* and records layouts as defined by the C programming language include files **<sys/shm.h>** and **<sys/ipc.h>**.
- Interface specifications to shared memory system calls: **shmget(2)**, **shmat(2)**, **shmctl(2)**, **shmdt(2)**.

### system.addresses

The `system.addresses` package provides routines to convert between integer types and `system.address`, associate a variable with a machine register, and associate a variable with a location in physical (machine; not virtual!) memory. It also includes other conversion, arithmetic, and comparison functions.

### system.information

The `system.information` package is a thick/abstract binding to the **`sysinfo(2)`** service. See the specification of `system.information` for more information.

### system.storage_pools.standard

The `system.storage_pools.standard` package contains the standard storage pool types used by the implementation for access types without `'Storage_Pool` clauses.

The `Predefined_Storage_Pool` type is used for those access types for which no `'Storage_Size` clause is present. There is a single object of this type (`system.storage_pools.standard.objects.predefined`). It is erroneous to attempt to create any other object of this type.

The `Collection_Storage_Pool` type is used for those access types for which a `'Storage_Size` clause is present. The implementation creates a distinct object of this type for each such access type. It is illegal to attempt to create any object of this type directly. However, it is possible to reference the implementation-created object via the `'Storage_Pool` attribute.

### system.storage_pools.standard.objects

The `system.storage_pools.standard.objects` package contains the predefined storage pool object used by the implementation for access types with neither `'Storage_Pool` nor `'Storage_Size` clauses.

## publiclib

The **`publiclib`** environment contains general-purpose, public-domain Ada packages. Note that Concurrent neither owns nor supports any of the packages in **`publiclib`**; these packages are provided as a courtesy to users.

**Table 9-5. publiclib environment**

| package |
|---|
| `c_to_ada_types` |

**Table 9-5. publiclib environment (Cont.)**

| package |
|---|
| character_type |
| curses |
| qsort |

### c_to_ada_types

This package provides Ada type definitions for C types (`int` , `char`, `bool`, ...).

### character_type

This package contains commonly used routines that test and manipulate characters (`isalpha`, `isupper`, `islower`, ...).

### curses

The `curses` package provides an Ada interface to the **`curses(3X)`** library containing terminal information and screen-manipulation routines.

For more information about using routines in the `curses` package, refer to the *Character User Interface Programming* manual.

### qsort

A generic sort. The generic implements Knuth's Algorithm Q [Knuth, "Searching and Sorting", *The Art of Computer Programming*, Volume 3, Addison-Wesley, ppg 116-7].

The only parameter to the instantiated procedure is the array to be sorted.

## rtdm

This environment contains a package which provides a flexible interface to the key features of data monitoring.

**Table 9-6. rtdm environment**

| package |
|---|
| real_time_data_monitoring |

### real_time_data_monitoring

This package contains subprograms that allow you to specify executable programs that contain Ada, C, or FORTRAN variables to be monitored, obtain and modify the values of

selected variables by specifying their names, and obtain such information about the variables as their virtual addresses, types, and sizes.

The interface provided allows for viewing and modifying data objects without knowledge a priori of the set of data objects or their data type. The current implementation supports a limited set of data items, including:

- integer objects
- floating point objects
- fixed point objects
- enumeration objects
- array components
- record fields
- pointers
- limited expressions involving pointer indirection

The `real_time_data_monitoring` package also makes use of an `interest_threshold` to filter out less interesting data items when using `get`, `set` or `list` activities. If the interest level of a data item is lower than the `interest_threshold` of its associated `program_descriptor`, it is as if that data item did not exist.

Interest levels for particular data items are set using the implementation-defined pragma INTERESTING (see "Pragma INTERESTING" on page M-117) or the `-Qinteresting` compilation option (see "Qualifier Keywords (`-Q` options)" on page 4-105). By default, all data items have an interest level of zero.

This information is only useful if full debug information is enabled (see "Pragma DEBUG" on page M-109 or "Debug Level (`-g[level]`)" on page 4-100.

For more information about using routines in the `real_time_data_monitoring` package, see the *Data Monitoring Reference Manual*.

## deprecated

This environment contains packages which are provided for compatibility with previous releases only. This environment, and all the packages in it, will be removed in a future release of MAXAda.

## obsolescent

The **`obsolescent`** environment contains those packages whose functionality is largely redundant with other features defined in the Ada 95 Reference Manual. Use of these features is not recommended in newly written programs.

Descriptions of these features of the language can be found in Annex J of the Ada 95 Reference Manual.

**Table 9-7. obsolescent environment**

| package |
|---|
| calendar |
| direct_io |
| io_exceptions |
| machine_code |
| sequential_io |
| text_io |
| unchecked_conversion |
| unchecked_deallocation |

# posix_1003.1

This environment contains a package that provides a thin Ada binding to all header files and subprograms defined in the IEEE-Std-1003.1 (POSIX 1003.1) and IEEE-Std-1003.1b (POSIX 1003.1b) standards for UNIX operating systems.

**Table 9-8. posix_1003.1 environment**

| package |
|---|
| posix_1003_1 |

### posix_1003_1

The `posix_1003_1` package provides a thin Ada binding to all header files and subprograms defined in the IEEE-Std-1003.1 (POSIX 1003.1) and IEEE-Std-1003.1b (POSIX 1003.1b) standards for UNIX operating systems.

Certain services available in POSIX 1003.1 require use of additional OS support libraries (e.g. **`librt.a`**, **`libpthread.a`**). The easiest way to ensure that the required library is included in the link is to put the following pragma in any Ada unit that uses these services:

```
pragma linker_options ("-lpthread") ;
```

It is not included automatically because it would introduce unnecessary overhead in applications which do not use such services.

## posix_1003.5

This environment contains packages whose specifications were extracted from *IEEE Std 1003.5-1992, IEEE Standard for Information Technology--POSIX Ada Language Interfaces--Part 1: Binding System Application Program Interface*, copyright (c)1992 by the Institute of Electrical and Electronics Engineers, Inc. The IEEE Std 1003.5-1992 must be used in conjunction with these package specifications in order to claim conformance.

The Concurrent implementation of this binding is fully compliant with the standard and thus allows the user to write fully compliant Ada applications.

**Table 9-9. posix_1003.5 environment**

| package |
|---|
| posix |
| posix_calendar |
| posix_configurable_file_limits |
| posix_configurable_system_limits |
| posix_file_locking |
| posix_file_status |
| posix_files |
| posix_group_database |
| posix_io |
| posix_local_signals |
| posix_permissions |
| posix_process_environment |
| posix_process_identification |
| posix_process_primitives |
| posix_process_primitives.local |
| posix_process_times |
| posix_signals |
| posix_supplement_to_ada_io |
| posix_terminal_functions |
| posix_unsafe_process_primitives |
| posix_user_database |

This environment contains an implementation of POSIX.5. In addition to the packages defined in that standard, a child package of `posix_process_primitives` is provided:

```
package posix_process_primitives.local is
--
   type a_procedure is access procedure (void : integer);

   procedure set_child_process_callback (
                       template : in out process_template;
                       routine  : in a_procedure;
                       param    : in integer := 0);
--
end posix_process_primitives.local;
```

Use of `set_child_process_callback` will result in the supplied user routine to be called after the process forks but before it execs. The value of `param` is provided on the call.

This is a convenient way to set process attributes that are not addressed by the POSIX.5 standard.

## sockets

This environment contains a package that provides a direct thin Ada binding to UNIX sockets.

**Table 9-10. sockets environment**

| package |
|---|
| sockets |

### sockets

This package provides a direct thin Ada binding to sockets.

## general

This environment contains Ada bindings to some general purpose services including an Ada interface to some of the system timing devices and a thin binding to the NightTrace service routines.

**Table 9-11. general environment**

| package |
| --- |
| night_trace_bindings |
| timers |

### night_trace_bindings

This package contains a "thin/abstract" binding to the Ntrace service routines as described in **ntrace(3x)**.

### timers

This package contains interfaces to various OS timings services.

# 4
# Real-Time Features

**Replace with Part 4 tab**

# Part 4 - Real-Time Features

Part 4 Real-Time Features

Chapter 10 Real-Time Extensions ..... 10-1

Chapter 11 Real-Time Event Tracing ..... 11-1

Chapter 12 Real-Time Monitoring ..... 12-1

# 10
# Real-Time Extensions


Mutual Exclusion Interfaces . . . . . 10-1
Spin Locks . . . . . 10-1
Binary Semaphores . . . . . 10-2
Tasking Semaphores . . . . . 10-4
Task Synchronization . . . . . 10-6
Cyclic Scheduling . . . . . 10-6
User Trace . . . . . 10-8
Low-Level Interfaces . . . . . 10-8
Indivisible Operations . . . . . 10-8
Rescheduling Control . . . . . 10-10
Client-Server Services . . . . . 10-11
Usermap Support . . . . . 10-11

# 10
# Real-Time Extensions

This chapter describes a variety of Ada extensions to MAXAda that can be utilized with Concurrent real-time services. Because the majority of the interfaces to real-time services are C language library routines, MAXAda provides an Ada interface to these services through packages in the **vendorlib** environment. These packages provide a complete Ada binding to the real-time library routines and data structures for the real-time services mentioned before. This capability provides users with an Ada interface to real-time services without having to go outside of the Ada language.

The following real-time packages are available in **vendorlib**:

```
binary_semaphores
client_server_services
cyclic_scheduler
distrib_services
eti_services
fbsched
indivisible_operations
rescheduling_control
rtc_services
spin_locks
task_synchronization
tasking_semaphores
user_trace
user_trace.raw
usermap_support
```

## Mutual Exclusion Interfaces

### Spin Locks

The `spin_locks` package provides an efficient and reliable means of performing busy-wait mutual exclusion between two or more programs. A spin lock must be allocated within a shared memory region (a MAXAda shared package, for instance) for it to be visible to more than one program. See “Shared Memory” on page 8-1.

Spin locks are most efficient when the critical sections they guard are short. I/O operations, system calls, and extended computations should be avoided when spin locks are locked.

Special privileges are needed to use this package because it makes use of the `rescheduling_control` package. Refer to "Rescheduling Control" on page 10-10 for proper use of the `rescheduling_control` package.

Use of this package requires the `CAP_SYS_RAWIO` capability (see "Capabilities" on page 1-3).

## Binary Semaphores

The `binary_semaphores` package provides an efficient means of performing sleepy-wait mutual exclusion between two or more *programs* using the PowerMAX OS client/server synchronization services. The "sleep" operation is performed using the priority inheritance protocol which limits the length of time a high-priority process must wait for a low-priority process to release the semaphore.

Once a semaphore is locked, the locking thread may perform any actions desired, including I/O and system calls.

The most common means of using this package is to declare a shared package containing an object of type `binary_semaphores.semaphore`. For example,

```
with binary_semaphores;
package sync_package is
--
  pragma elaborate_body;
  --
  -- A binary semaphore for arbitration of exclusive access to
  -- some_unnamed_shared_resource shared among multiple programs.
  --
  sema : binary_semaphores.semaphore;

  --
  -- This flag is set to TRUE when the first program calls
  -- sema_init to initialize sema.
  --
  initialized : boolean;

  pragma shared_package;
  pragma pool_lock_state (default, locked);
 --
  end sync_package;

package body sync_package is
begin
--
    sync_package'lock;

    if not initialized then
       binary_semaphores.sema_init (sema);
       initialized := True;
    end if;

    sync_package'unlock;
--
end sync_package;
```

Programs may specify `sync_package` and `binary_semaphores` in `with` clauses to obtain mutually exclusive access to any shared resource.

The following operations on binary semaphores are provided:

```
--
-- Initialize a semaphore.
--
procedure sema_init (sema : in out semaphore);

--
-- Lock a semaphore. If the semaphore is already locked, the
-- caller blocks until the lock can be gained and the owner of the
-- lock will inherit its priority.
--
procedure sema_lock (sema : in out semaphore);

--
-- Unlock a semaphore. If the calling process has had its priority
-- adjusted through priority inheritance this call may result in a
-- rescheduling operation.
--
procedure sema_unlock (sema : in out semaphore);

--
-- Return true if the semaphore is currently locked.
--
procedure sema_is_locked (sema : in out semaphore; result : out boolean);
--
-- Destroy the semaphore. Any waiting threads will be released
-- and will have semaphore_error raised within them. This operation
-- is provided to aid in recovery, and is not needed for deallocation
-- or other such ”normal” circumstances.
--
procedure sema_destroy (sema : in out semaphore);
```

The `sema_init` procedure must be called with any semaphore prior to calling `sema_lock`, `sema_unlock`, or `sema_is_locked`. In the preceding example, the package body provides for the initialization of the semaphore at elaboration time. All programs that specify the shared package containing the binary semaphore in a `with` clause will execute this elaboration code, so an “initialized” flag is provided in the package body. In this way, the first program to elaborate the package will initialize the semaphore, and others will use its pre-existing state. The shared package `'LOCK` and `'UNLOCK` attribute procedures are used to arbitrate access to the initialization flag. This is the recommended procedure for initializing binary semaphores.

Once a semaphore has been initialized, `sema_lock` and `sema_unlock` may be used to obtain and release the semaphore. The `sema_is_locked` procedure is provided as a means of detecting a locked semaphore without blocking, but it should be noted that blocking may still occur if another program obtains the lock after a call to `sema_is_locked`, but before a subsequent call to `sema_lock`.

The exception, `binary_semaphores.SEMAPHORE_ERROR`, may be raised under the following circumstances:

- If any one of `sema_lock`, `sema_unlock`, `sema_is_locked` is called with an uninitialized semaphore.
- If any one of `sema_lock`, `sema_unlock`, `sema_is_locked` is called with a destroyed semaphore.
- The `sema_lock` procedure is called with a semaphore owned by a thread that no longer exists.

- The `sema_lock` procedure is called with a semaphore owned by a thread with a different real or effective user id.
- The `sema_unlock` procedure is called with a semaphore owned by another thread.
- The `sema_unlock` procedure is called and the waking service returns an error status when attempting to wake waiting threads.

Use of this package requires the `CAP_SYS_RAWIO` capability (see "Capabilities" on page 1-3).

**NOTE**

On RedHawk, the maximum number of waiting processes is bounded by a constant in the private part of the package specification.

If this limit is insufficient, copy both the specification and body source files to another environment, increase the limit, and compile the package locally.

## Tasking Semaphores

The `tasking_semaphores` package provides a simple and fast means of performing sleepy-wait mutual exclusion between two or more Ada *tasks* within a single Ada program.

Once a semaphore is locked, the locking task may perform any actions desired, including I/O and system calls.

The semaphore defined by this package is valid only within the context of a single Ada program, and therefore, is meaningless to other processes that might have access to the semaphore object if it is placed in shared memory. For multi-programming binary semaphores, see "Binary Semaphores" on page 10-2.

The following operations on tasking semaphores are provided:

```
--
-- Initialize a semaphore.
--
procedure sema_init (sema : out semaphore);
--
-- Lock a semaphore. If the semaphore is already locked, the
-- caller blocks until the lock can be gained.
--
procedure sema_lock (sema : in semaphore);

--
-- Unlock a semaphore.
--
procedure sema_unlock (sema : in semaphore);

--
-- Return true if the semaphore is currently locked.
```

```
--
function sema_is_locked  (sema : in semaphore) return boolean;

--
-- Return true if the semaphore is currently locked.
--
procedure sema_is_locked  (sema : in semaphore; result : out boolean);
--
-- Destroy the semaphore. Any waiting tasks will be released
-- and will have tasking_error raised within them. This operation is
-- provided to aid in recovery, and is not needed for deallocation
-- or other such "normal" circumstances.
--
procedure sema_destroy  (sema : in out semaphore);
```

The `sema_init` procedure should be called once per program. For semaphores defined in a package, it is recommended that `sema_init` be called in the elaboration code portion of the package body. If this is done, it is also recommended to include the following line after the `context_clause` portion of the package body:

```
pragma ELABORATE (tasking_semaphores)
```

For example:

```
with tasking_semaphores ;
package sync_package is
   sema : tasking_semaphores.semaphore ;
end sync_package ;

with tasking_semaphores ;
pragma elaborate (tasking_semaphores) ;
package body sync_package is
begin
   tasking_semaphores.sema_init (sema) ;
end sync_package ;
```

Once a semaphore has been initialized, `sema_lock` and `sema_unlock` may be used to obtain and release the semaphore. The `sema_is_locked` function is provided as a means of detecting a locked semaphore without blocking, and the `sema_is_locked` procedure is provided to remain as compatible as possible with the `binary_semaphores` package. Note that blocking may still occur if another task obtains the lock after a call to `sema_is_locked`, but before a subsequent call to `sema_lock`.

The exception, `tasking_semaphores.SEMAPHORE_ERROR`, may be raised under the following circumstances:

- If any one of `sema_lock`, `sema_unlock`, `sema_is_locked` is called with an uninitialized semaphore.
- If any one of `sema_lock`, `sema_unlock`, `sema_is_locked` is called with a destroyed semaphore.

# Task Synchronization

The `task_synchronization` package provides an extremely efficient mechanism for synchronizing tasks. Synchronization of two tasks is naturally expressed within the Ada language as task rendezvous. Synchronization of more than two tasks becomes more complicated. The `task_synchronization` package provides a method for easily synchronizing more than two tasks.

Use of this package requires the `CAP_SYS_RAWIO` capability (see "Capabilities" on page 1-3).

# Cyclic Scheduling

The real-time features provided with the run-time executive make cyclic scheduling designs extremely efficient and easy to implement.

Using the `rtc_control` package, cyclic scheduling can be implemented with three Ada statements:

1. Configure clock at desired frequency and attach to task entry

```
use system;
clock : address := rtc_control.configure_clock
 (cycle_time_msec => 16.6,
  device_name => "/dev/rrtc/0c2");
for entry_1 use at clock;
```

2. Start the clock

```
rtc_control.start_clock (clock);
```

3. Stop the clock

```
rtc_control.stop_clock (clock);
```

The generic package `cyclic_scheduler` supplied with MAXAda automatically multi-threads a set of work loads and schedules them at a specified frequency. Features include:

- Automatic distribution of work loads across CPUs
- Specification of timing source (RTC) and simulation rate
- Binding of work loads to CPUs
- Specification of individual periods for each work load
- Specification of individual priorities for each work load
- Overrun detection/handling
- Start/stop simulation

**NOTE**

The `cyclic_scheduler` included with MAXAda is not related to the Concurrent frequency-based scheduler (FBS). Any similarities between the two interfaces is purely coincidental.

Use of this package requires the following capabilities: CAP_SYS_NICE and CAP_SYS_RAWIO (see "Capabilities" on page 1-3).

Following is an example program that makes use of the `cyclic_scheduler` package.

### Example:

```
with cyclic_scheduler;


procedure cyclic_example is
--
   procedure foo;
   procedure bar;
   procedure print;

   type workloads is new integer range 1..3;
   type workload_info is array (workloads) of integer;

   package simulation is new cyclic_scheduler (
      workload_id    => workloads,
      workload_info  => workload_info,
      cycle_duration =>  500.0,
      first_cycles   => (2, 2, 1),
      periods        => (2, 2, 2),
      workload_1     => foo,
      workload_2     => bar,
      workload_3     => print
                                              );

   initial_value : constant := 2 ** 9;
   foo_value     : integer := initial_value;
   bar_value     : integer := initial_value;

   procedure foo is separate;
   procedure bar is separate;
   procedure print is separate;
--
begin
--
   simulation.executive.enable;
   delay 10.0;
   simulation.executive.disable;
   simulation.executive.termination;
--
end cyclic_example;

separate (cyclic_example)
procedure foo is
begin
   foo_value := foo_value * 2;
end foo;

separate (cyclic_example)
procedure bar is
```

```
begin
   bar_value := bar_value / 2;
end bar;

with ada.text_io;
separate (cyclic_example)
procedure print is
begin
   ada.text_io.put_line ("foo_value =" & integer'image(foo_value));
   ada.text_io.put_line ("bar_value =" & integer'image(bar_value));
end print;
```

# User Trace

See "user_trace package" on page 11-3 for information about the `user_trace` tracing package.

See "user_trace.raw package" on page 11-5 for information about the `user_trace.raw` tracing package.

# Low-Level Interfaces

## Indivisible Operations

The `indivisible_operations` package provides subprograms that implement indivisible operation for process synchronization. For maximum efficiency, they are implemented as compiler intrinsics, whenever possible.

The memory location associated with the formal parameter `memory` in the supplied `test_and_set`, `fetch_and_store`, `fetch_and_add`, `fetch_and_increment`, and `fetch_and_decrement` subprograms, is modified according to the respective subprogram being invoked.

```
with system ;

package indivisible_operations is
--
   pragma preelaborate ;

   --
   -- The memory location associated with the supplied operand is set to 1
   -- and the previous value is returned as a boolean (TRUE if 1, FALSE if
   -- 0).  Only values of 0 and 1 are expected; other values will yield
   -- erroneous results.
   --
   function test_and_set( memory : boolean )        return boolean ;
   function test_and_set( memory : tiny_integer )   return boolean ;
   function test_and_set( memory : short_integer )  return boolean ;
   function test_and_set( memory : integer )        return boolean ;
   function test_and_set( memory : system.address ) return boolean ;
```

```
--
-- These forms work the same as the function forms above, except that
-- they return the result of the test in the out parameter "result".
--
procedure test_and_set( memory : in out boolean ;
                        result :    out boolean ) ;
procedure test_and_set( memory : in out tiny_integer ;
                        result :    out boolean ) ;
procedure test_and_set( memory : in out short_integer ;
                        result :    out boolean ) ;
procedure test_and_set( memory : in out integer ;
                        result :    out boolean ) ;
procedure test_and_set( memory : in out system.address ;
                        result :    out boolean ) ;


--
-- The variable "memory" is set to 1 and the previous value of "memory"
-- is returned in the out parameter "result".
--
procedure fetch_and_store( memory : in out boolean ;
                           result :    out boolean ) ;
procedure fetch_and_store( memory : in out tiny_integer ;
                           result :    out tiny_integer ) ;
procedure fetch_and_store( memory : in out short_integer ;
                           result :    out short_integer ) ;
procedure fetch_and_store( memory : in out integer ;
                           result :    out integer ) ;
procedure fetch_and_store( memory : in out system.address ;
                           result :    out system.address ) ;


--
-- The variable "memory" is set to the value in the parameter "value"
-- and the previous value of "memory" is returned in the out parameter
-- "result".
--
procedure fetch_and_store( memory : in out boolean ;
                           value  : in     boolean ;
                           result :    out boolean ) ;
procedure fetch_and_store( memory : in out tiny_integer ;
                           value  : in     tiny_integer ;
                           result :    out tiny_integer ) ;
procedure fetch_and_store( memory : in out short_integer ;
                           value  : in     short_integer ;
                           result :    out short_integer ) ;
procedure fetch_and_store( memory : in out integer ;
                           value  : in     integer ;
                           result :    out integer ) ;
procedure fetch_and_store( memory : in out system.address ;
                           value  : in     system.address ;
                           result :    out system.address ) ;

--
-- The variable "memory" is incremented by the value in the parameter
-- "value" and the previous value of "memory" is returned in the out
-- parameter "result".
--
procedure fetch_and_add( memory : in out integer ;
                         value  : in     integer ;
                         result :    out integer ) ;

--
-- The variable "memory" is incremented (or decremented) by 1 and the
```

```
   -- previous value of "memory" is returned in the out parameter "result".
   --
   procedure fetch_and_increment( memory : in out integer ;
                                 result :    out integer ) ;
   procedure fetch_and_decrement( memory : in out integer ;
                                 result :    out integer ) ;

   --
   -- The variable "memory" is incremented (or decremented) by 1.
   --
   procedure increment( memory : in out integer ) ;
   procedure decrement( memory : in out integer ) ;
--
private
--
   pragma dont_elaborate ;

   pragma import (intrinsic, test_and_set) ;
   pragma import (intrinsic, fetch_and_store) ;
   pragma import (intrinsic, fetch_and_add) ;
   pragma import (intrinsic, fetch_and_increment) ;
   pragma import (intrinsic, fetch_and_decrement) ;
   pragma import (intrinsic, increment) ;
   pragma import (intrinsic, decrement) ;
--
end indivisible_operations;
```

It is not possible to perform an indivisible operation on a bit location at a bit offset. Only byte, short and word operations are allowed. If the operand of an operation is the component of a record or array, the user must take proper steps to align the field using a representation clause.

The operations available in the `indivisible_operations` package are highly dependent on the target architecture, and as such, may not be available in the same form on other architectures.

## Rescheduling Control

The `rescheduling_control` package is an interface to the operating system rescheduling control services. Control over rescheduling is useful for low-level synchronization services. For example, a non-priority-inverting and very fast implementation of busy-wait mutual exclusion may be implemented by combining rescheduling control with a target machine `test_and_set` instruction.

See the **vendorlib** package `spin_locks` for an example. See “Spin Locks” on page 10-1.

The `rescheduling_control` package allows Ada applications to set a flag which signals the operating system not to perform a context switch.

Use of this package requires the `CAP_SYS_RAWIO` capability (see “Capabilities” on page 1-3).

All Ada tasking programs have a rescheduling variable registered with the operating system. (The run-time system creates, registers, and initializes this variable.) This package

provides an interface to the rescheduling services for manipulation of that rescheduling variable.

## Client-Server Services

The `client_server_services` package provides an interface to the client/server communication services. These services use a priority inheritance protocol to implement efficient and deterministic client and server interactions. They may be used to create a variety of "higher-level" inter-process communication protocols.

**NOTE**

These services should be used only very carefully with the Ada run-time executive. The **`server_wake1(2)`** and **`server_wakevec(2)`** should *not* be issued unless the user is certain that the processes being waked contain absolutely *no* tasking or are already blocked (via the **`server_block(2)`** service). See "Client/Server Services" on page A-3 for more details.

Use of rescheduling variables in combination with this package requires the `CAP_SYS_RAWIO` capability (see "Capabilities" on page 1-3).

## Usermap Support

The `usermap_support` package provides an abstract binding to the **`usermap(2)`** system service (which maps memory pages of an executing program into the address space of the calling process).

This package automatically tracks calls to **`usermap(2)`** in an effort to minimize it's use; each call to **`usermap(2)`** will allocate a new attachment to the calling processes address space, regardless of whether an attachment to the same target address was made earlier. (i.e. the OS doesn't do the bookkeeping).

This package deals with unsigned address arithmetic.

This package also allows the user to check-in a range of address that have already been mapped into the address space by some other means (**`shmat(2)`** perhaps), for consideration on subsequent `usermap_support.usermap` calls.

# 11
# Real-Time Event Tracing


Specifying Trace Events . . . . . 11-1
Predefined Trace Events . . . . . 11-2
Library Unit Elaboration . . . . . 11-2
User-Defined Trace Events . . . . . 11-2
user_trace package . . . . . 11-3
Specification . . . . . 11-4
Usage . . . . . 11-5
user_trace.raw package . . . . . 11-5
Specification . . . . . 11-7
NightTrace Binding . . . . . 11-8
Specification . . . . . 11-9
Usage . . . . . 11-12
NightView Debugger . . . . . 11-13
Tracing Options . . . . . 11-14
Tracing Options - Examples . . . . . 11-17
Logging Trace Events . . . . . 11-19
Logging Mechanisms . . . . . 11-19
Ada Executive . . . . . 11-19
Trace Buffer . . . . . 11-20
Timing Source . . . . . 11-20
NightTrace Daemon . . . . . 11-21
Log Files . . . . . 11-22
Viewing Trace Events . . . . . 11-23
User Table . . . . . 11-23
Viewing Trace Events with a.trace . . . . . 11-24
Viewing Trace Events with NightTrace . . . . . 11-25
Creating the NightTrace Configuration File . . . . . 11-25
Modifying the NightTrace Configuration File . . . . . 11-26

# 11
# Real-Time Event Tracing

Real-time event tracing is one way to debug and analyze the performance of Ada applications, including multi-tasking applications. It allows the user to gather information about important events in an application, such as event occurrences, timings, and data values.

MAXAda has established two types of trace events: predefined trace events and user-defined trace events. This chapter will discuss these in further detail, including how trace events are specified. In addition, examples of usage are provided.

Tracing behavior is controlled via the **`-trace`** link option to **`a.partition`**. Details about this option and its associated attributes are presented in this chapter.

Also discussed are the available mechanisms used for logging trace events as well as the utilities used to view the resultant log files, including the MAXAda **`a.trace`** utility and the NightTrace tool.

## Specifying Trace Events

A *trace point* is a location within an application at which information is logged. The information logged is termed a *trace event*. At a minimum, it includes a trace event ID number and a timestamp; it may be accompanied by additional data as well.

Logging of a trace event is done via a procedure call to one of various tracing mechanisms. These mechanisms are discussed in “Logging Mechanisms” on page 11-19.

A special version of the Ada runtime executive (see “Ada Executive” on page 11-19) is provided with a significant number of trace points which log *predefined trace events*. These events describe the execution of the user's Ada application in terms of tasking, interrupt handling, exception occurrence and handling, protected actions, and elaboration of library units.

The specification of certain attributes to the **`-trace`** link option causes the selection of a version of the Ada runtime executive which contains these trace points. See “Tracing Options” on page 11-14 for details.

Additionally, the user may define trace points in his source code. At each *user-defined trace point*, the user must provide an event ID and optional data arguments. The exact time that each trace point is encountered is included in the trace event.

A single clock is utilized to timestamp the events so that, regardless of the logging method chosen, all trace events may be combined and sorted chronologically by analysis tools.

Trace points might be placed at:

- suspected bug locations

- process, subprogram, or loop entry and exits
- timing points
- sychronization points/multi-process interaction
- endpoints of atomic operations
- endpoints of shared memory access code

Careful trace point placement may aid you in identifying patterns and anomalies in your application.

## Predefined Trace Events

Predefined trace events are generated by tracing versions of the Ada runtime executive and by library elaboration code generated for the `ENVIRONMENT` task. They typically describe execution in terms of tasking, interrupt handling, exception occurrence and handling, and protected actions.

Setting the **`rtsinstrumentation`** attribute of the **`-trace`** link option to `true` for a partition generates predefined trace events. See "Tracing Options" on page 11-14 for details.

### Library Unit Elaboration

A pair of trace events (entry and exit) for the elaboration of every library unit in the partition may be generated by setting the **`elabinstrumentation`** attribute of the **`-trace`** link option to `true`. See "Tracing Options" on page 11-14 for details.

**NOTE**

The user may wish to increase the length of the trace buffer used for logging trace events if there are a large number of library units to be traced. (See "Trace Buffer" on page 11-20 for more information).

## User-Defined Trace Events

User-defined trace events generate information at specified points within the source code that are of particular interest to the user. By placing these trace points strategically, users can determine locations of suspected bugs, values of certain variables, and errors in timing or synchronization. Patterns of irregular or erroneous behavior of an application can be discovered by careful placement of the trace points.

Trace points are selected and placed within the source code. The source code is then recompiled, the application is relinked and executed, and the resultant trace event file is analyzed.

MAXAda supports three methods of establishing user-defined trace points:

- the MAXAda-supplied `user_trace` package
- bindings to the NightTrace services
- support of trace points embedded by the NightView debugger

## user_trace package

The `user_trace` package consists of all the procedures and functions necessary for placing user-defined trace points within source code and generating a resultant trace event file to be later viewed and analyzed. This package is supplied with MAXAda in the **vendorlib** environment in the file **user_trace.a**. See Chapter 9, "Support Packages" for more information about the **vendorlib** environment.

The `user_trace` package can be used independently of any other vendor-supplied tracing mechanism, and the trace events generated by this package can be viewed and analyzed by tools supplied with MAXAda, specifically the **a.trace** utility.

**NOTE**

The `user_trace.raw` package is provided to allow users to easily switch between the `user_trace` and `night_trace_bindings` packages. See “user_trace.raw package” on page 11-5 for more information.

## Specification

The specification of the `user_trace` package is:

```
with interval_timer ;
with ada.task_identification ;

package user_trace is
--
   --
   -- trace_mode
   --
   --    Specifies whether trace points logging is enabled or disabled.
   --
   type trace_mode is (DISABLED, ENABLED) ;

   --
   -- trace_user_type
   --
   --    Defines a type whose value must be zero.  This distinguishes
   --    user trace points from internal trace points.  Trace log
   --    entries for user trace points are marked with "USER_TRACE".
   --
   type trace_user_type is range 0..0 ;

   --
   -- user_trace_event_number
   --
   --    Defines the value of all trace events logged via this
   --    package (see ntrace(1)).
   --
   user_trace_event_number : constant := 4402 ;

   --
   -- trace_sub_id
   --
   --    Defines a broad integer range of values representing
   --    user events / trace_points.  These values are echoed
   --    in the trace log.
   --
   subtype trace_sub_id is natural range 0..2**16-1 ;

   --
   -- log
   --
   --    These overloaded procedures cause individual trace points
   --    to be logged (assuming that the current mode is set to enabled).
   --
   procedure log (sub_id : trace_sub_id) renames pp.log0 ;

   procedure log (sub_id : trace_sub_id ; data1 : integer) renames pp.log1 ;

   procedure log (sub_id : trace_sub_id ; data1 : integer ;
                                          data2 : integer) renames pp.log2 ;

   --
   -- buffer_length
   --
   --    The length of the trace buffer for each task (# of entries)
   --    is specified by the following value.  Buffer allocation
   --    occurs during task creation if tracing is ENABLED, or on
   --    the first "set_trace_mode" call that ENABLE tracing for a
   --    task.  Each trace buffer entry consumes approximately 24 bytes.
   --
   --    The default value is 1000 entries.
   --
   trace_buffer_length : integer ;

   --
   -- set_trace_mode
   --
   --    By default, the trace mode for the environment task is
   --    DISABLED.  When linking with the tracing runtime system
   --    the trace mode is automatically set to ENABLED.
   --
   --    When a task is created, its trace mode is inherited from
   --    its parent task (or environment task).
   --
   --    In the event that tracing is specified for a particular
```

```
--    task for the first time via "set_trace_mode", the trace
--    buffer is allocated at that time.  STORAGE_ERROR is raised
--    if that allocation fails.
--
--    The "function" form of this utility returns the previous
--    trace mode for the specified task.
--
procedure set_trace_mode_all (mode : in trace_mode) ;
procedure set_trace_mode (mode    : in trace_mode ;
                          task_id : in ada.task_identification.task_id :=
                                       ada.task_identification.null_task_id);
function swap_trace_mode (mode    : in trace_mode ;
                          task_id : in ada.task_identification.task_id :=
                                       ada.task_identification.null_task_id)
                             return trace_mode ;

--
-- dump
--
--    This routine can be called from any task and results in the
--    trace records for ALL tasks (active and terminated) to be
--    dumped.
--
procedure dump ;
```

### Usage

The call to log a user trace event using the `user_trace` package might look like:

```
user_trace.log (sub_id => 47, data1 => x, data2 => y) ;
```

This example uses named notation to specify the parameters in the procedure call.

A NightTrace event with a trace event ID of `4402` is logged for every invocation of `user_trace.log`, with the following correspondence:

| `user_trace` parameter | NightTrace Expression |
|---|---|
| `sub_id` | `arg(2)` |
| `data1` | `arg(3)` |
| `data2` | `arg(4)` |

**NOTE**

The NightTrace exrepssion `arg(1)` will hold the `task_id` of the task that issued the trace point.

## user_trace.raw package

The `user_trace.raw` package is provided to allow users to easily switch between the `user_trace` and `night_trace_bindings` packages. (See “user_trace package” on page 11-3 and “NightTrace Binding” on page 11-8 for more information on these packages.)

The `user_trace.raw` package differs from the parent `user_trace` package only in that it does absolutely *no* transformation on the event or arguments passed to these routines.

Consider a scenario where the user wants to use **ntraceud** and the night_trace_bindings to log and capture data. If the user then switches to the internal tracing mechanism (using the **-trace** link option to **a.partition**), the arguments in the resultant trace events will differ (see below). The user_trace.raw package is supplied to conveniently switch between the bindings; the user only needs to change the with and the use clauses, since trace_event is defined for both.

The mapping between the parameters to user_trace.raw.trace_event and NightTrace expressions is:

| user_trace.raw parameter | NightTrace Expression |
|---|---|
| event | id |
| arg (or arg1) | arg(1) |
| arg2 | arg(2) |

whereas the mapping between user_trace.log parameters and NightTrace expressions is:

| user_trace parameter | NightTrace Expression |
|---|---|
| implicitly generated event_id of 4402 | id |
| implicitly generated task_id | arg(1) |
| sub_id | arg(2) |
| data1 | arg(3) |
| data2 | arg(4) |

See “user_trace package” on page 11-3 for more information on the abovementioned transformations.

### Specification

The specification of the `user_trace.raw` package is:

```
package user_trace.raw is
--
   subtype event_type is integer ; -- Must be in range 0..4095

   procedure trace_event (event  : event_type) ;

   procedure trace_event (event  : event_type ;
                          arg    : integer) ;

   procedure trace_event (event  : event_type ;
                          arg    : long_float) ;

   procedure trace_event (event  : event_type ;
                          arg1   : integer ;
                          arg2   : integer) ;
--
end user_trace.raw ;
```

## NightTrace Binding

NightTrace is an interactive debugging and performance analysis tool that is available separately from MAXAda. NightTrace allows users to generate user-defined trace events by making certain NightTrace procedure and function calls within their source code. After the events are generated, NightTrace allows the users to display the trace event information as numerical statistics and as graphical images.

The framework of the graphical display can be configured by the user for more meaningful analysis of the information generated by the trace events.

MAXAda provides a thin binding to the NightTrace services. This binding can be found in the MAXAda-supplied **`general`** environment in the file **`night_trace.a`**. See Chapter 9, "Support Packages" for more information about the **`general`** environment.

**NOTE**

Use of the `night_trace_bindings` package precludes the use of any other MAXAda tracing mechanisms. This binding can not be used in conjunction with either the **`-trace`** or **`-ntrace`** link options, the `user_trace` package, or the **`a.trace`** utility.

## Specification

The specification of the NightTrace binding is:

```
--
-- This package contains a "thin/abstract" binding to the Ntrace
-- service routines as described in ntrace(3x).
--
-- Descriptions of the service routines are identical to those
-- found in the system documentation (e.g. man ntrace(3x)),
-- except that the subprograms have been specified using
-- Ada's descriptive style: overloading, enumerated error codes,
-- strong typing.
--
-- There is no overhead incurred due to the Ada bindings,
-- except for the trace_start and trace_open_thread routines
-- which translate an Ada string into a form suitable for their
-- respective system services.
--

package night_trace_bindings is
--
   type ntrace_error is (
      NTNOERROR,
      NTIO,
      NTNODAEMON,
      NTNOTRACEFILE,
      NTINVALID,
      NTPERMISSION,
      NTALREADY,
      NTNOSHMID,
      NTRESOURCE,
      NTFLUSH,
      NTINIT,
      NTMAPSPLREG,
      NTMAPTIMER,
      NTLOSTDATA,
      NTEXISTS,
      NTBUSY,
      NTPGLOCK,
      NTNOMEM,
      NTMAPCLOCK,
      NTBADVERSION,
      NTFILETRASHED,
      NTLISTEN
                        ) ;

   type event_type is range 0..4095 ;

   type ntclock_t is (NT_USE_ARCHITECTURE_CLOCK, NT_USE_RCIM_TICK_CLOCK) ;
   for ntclock_t use (NT_USE_ARCHITECTURE_CLOCK => 0,
                      NT_USE_RCIM_TICK_CLOCK    => 1) ;
   for ntclock_t'size use 32 ;


   --
   -- Private Bindings
   --
   --    Do not reference specifications within "private_package".
   --
   package private_bindings is
      function  arg   (e : event_type; arg : integer)      return ntrace_error ;
      function  flt   (e : event_type; arg : float)        return ntrace_error ;
      function  flt2  (e : event_type; a1,a2 : float)      return ntrace_error ;
      function  dbl   (e : event_type; arg : long_float)   return ntrace_error ;
      function  dbl2  (e : event_type; a1,a2 : long_float) return ntrace_error ;
      procedure parg  (e : event_type; arg : integer) ;
      procedure pflt  (e : event_type; arg : float) ;
      procedure pflt2 (e : event_type; a1,a2 : float) ;
      procedure pdbl  (e : event_type; arg : long_float) ;
      procedure pdbl2 (e : event_type; a1,a2 : long_float) ;
      function  arg4  (e : event_type; a1,a2,a3,a4:integer) return ntrace_error;
      procedure parg4 (e : event_type; a1,a2,a3,a4:integer) ;
      function  dis   (e : event_type; f   : event_type)   return ntrace_error ;
      function  ena   (e : event_type; f   : event_type)   return ntrace_error ;
      procedure pdis  (e : event_type; f   : event_type) ;
      procedure pena  (e : event_type; f   : event_type) ;
   private
      pragma import (C, arg,   "", "trace_event_arg") ;
```

```
   pragma import (C, flt,   "", "trace_event_flt") ;
   pragma import (C, flt2,  "", "trace_event_two_flt") ;
   pragma import (C, dbl,   "", "trace_event_dbl") ;
   pragma import (C, dbl2,  "", "trace_event_two_dbl") ;
   pragma import (C, parg,  "", "trace_event_arg") ;
   pragma import (C, pflt,  "", "trace_event_flt") ;
   pragma import (C, pflt2, "", "trace_event_flt") ;
   pragma import (C, pdbl,  "", "trace_event_dbl") ;
   pragma import (C, pdbl2, "", "trace_event_two_dbl") ;
   pragma import (C, arg4,  "", "trace_event_four_arg") ;
   pragma import (C, parg4, "", "trace_event_four_arg") ;
   pragma import (C, dis,   "", "trace_disable_range") ;
   pragma import (C, pdis,  "", "trace_disable_range") ;
   pragma import (C, ena,   "", "trace_enable_range") ;
   pragma import (C, pena,  "", "trace_enable_range") ;
end private_bindings ;

--
-- Administrative: start, end, etc
--
function  trace_begin       (trace_file   : string ;
                             buffer_size  : integer := 1024 * 16 ;
                             use_spl      : boolean := true ;
                             use_resched  : boolean := false ;
                             lock_pages   : boolean := true ;
                             clock        : ntclock_t :=
                                 NT_USE_ARCHITECTURE_CLOCK;
                             shmid_perm   : integer := 8#666# ;
                             inherit      : boolean := true)
                             return ntrace_error ;
procedure trace_begin       (trace_file   : string ;
                             buffer_size  : integer := 1024*16 ;
                             use_spl      : boolean := true ;
                             use_resched  : boolean := false ;
                             lock_pages   : boolean := true ;
                             clock        : ntclock_t :=
                                 NT_USE_ARCHITECTURE_CLOCK;
                             shmid_perm   : integer := 8#666# ;
                             inherit      : boolean := true) ;

function  trace_end                                     return ntrace_error ;
procedure trace_end ;

function  trace_open_thread (threadname   : string)     return ntrace_error ;
procedure trace_open_thread (threadname   : string) ;

function  trace_close_thread                            return ntrace_error ;
procedure trace_close_thread ;

function  trace_flush                                   return ntrace_error ;
procedure trace_flush ;

function  trace_trigger                                 return ntrace_error ;
procedure trace_trigger ;

--
-- Logging Trace Events
--
function  trace_event         (event      : event_type) return ntrace_error ;
procedure trace_event         (event      : event_type) ;
pragma import (C, trace_event) ;

function  trace_event         (event      : event_type ; arg  : integer)
   return ntrace_error renames private_bindings.arg ;

procedure trace_event         (event      : event_type ; arg  : integer)
   renames private_bindings.parg ;

function  trace_event         (event      : event_type ; arg  : long_float)
   return ntrace_error renames private_bindings.dbl ;

procedure trace_event         (event      : event_type ; arg  : long_float)
   renames private_bindings.pdbl ;

function  trace_event         (event      : event_type ;
                               arg1       : integer ;
                               arg2       : integer ;
                               arg3       : integer ;
                               arg4       : integer)
   return ntrace_error renames private_bindings.arg4 ;
```

```
   procedure trace_event         (event      : event_type ;
                                  arg1       : integer ;
                                  arg2       : integer ;
                                  arg3       : integer ;
                                  arg4       : integer)
      renames private_bindings.parg4 ;

   --
   -- Enable/Disable Trace Events
   --
   function  trace_enable        (event      : event_type) return ntrace_error ;
   procedure trace_enable        (event      : event_type) ;
      pragma import (C, trace_enable) ;

   function  trace_enable        (event_low  : event_type ;
                                  event_high : event_type)
      return ntrace_error renames private_bindings.ena ;
   procedure trace_enable        (event_low  : event_type ;
                                  event_high : event_type)
      renames private_bindings.pena ;

   function  trace_disable       (event      : event_type) return ntrace_error ;
   procedure trace_disable       (event      : event_type) ;
      pragma import (C, trace_disable) ;

   function  trace_disable       (event_low  : event_type ;
                                  event_high : event_type)
      return ntrace_error renames private_bindings.dis ;
   procedure trace_disable       (event_low  : event_type ;
                                  event_high : event_type)
      renames private_bindings.pdis ;

   function  trace_enable_all                                return ntrace_error ;
   procedure trace_enable_all ;

   function  trace_disable_all                               return ntrace_error ;
   procedure trace_disable_all ;

   -- Deprecated interfaces
   function  trace_start       (trace_file   : string)     return ntrace_error ;
   procedure trace_start       (trace_file   : string) ;

--
private
--
   for ntrace_error use (
      NTNOERROR         =>  0,
      NTIO              =>  1,
      NTNODAEMON        =>  2,
      NTNOTRACEFILE     =>  3,
      NTINVALID         =>  4,
      NTPERMISSION      =>  5,
      NTALREADY         =>  6,
      NTNOSHMID         =>  7,
      NTRESOURCE        =>  8,
      NTFLUSH           =>  9,
      NTINIT            => 10,
      NTMAPSPLREG       => 11,
      NTMAPTIMER        => 12,
      NTLOSTDATA        => 13,
      NTEXISTS          => 14,
      NTBUSY            => 15,
      NTPGLOCK          => 16,
      NTNOMEM           => 17,
      NTMAPCLOCK        => 18,
      NTBADVERSION      => 19,
      NTFILETRASHED     => 20,
      NTLISTEN          => 21
                ) ;
   for ntrace_error'size use 32 ;

   for event_type'size use 32 ;

   pragma import (C, trace_enable_all) ;
   pragma import (C, trace_disable_all) ;
   pragma import (C, trace_flush) ;
   pragma import (C, trace_close_thread) ;
   pragma import (C, trace_end) ;
   pragma import (C, trace_trigger) ;
--
end night_trace_bindings ;
```

### Usage

Follow these steps to use the NightTrace binding:

1. Edit an Ada application and insert calls to the NightTrace services using the `night_trace_bindings` package. This makes it possible to log user-defined trace events at user-defined trace points. (See **`ntrace(3X)`**.) Some sample calls might be:

   ```
   retval := night_trace_bindings.trace_begin ("my_trace_file");
   retval := night_trace_bindings.trace_open_thread (trace_thread);
   retval := night_trace_bindings.trace_event (event_id, data);
   retval := night_trace_bindings.trace_close_thread;
   retval := night_trace_bindings.trace_end;
   ```

   Placement of `trace_begin` and `trace_open_thread` calls is critical to the tracing strategy of Ada tasking programs. Performing these calls before task elaboration causes all tasks to log to the same thread name. This can be done with the following type of statement:

   ```
   Trace_begin_stat : ntrace_error := trace_begin("tracefile") ;
   Trace_open_thread_stat : ntrace_error := trace_open_thread ("my_prog");
   ```

2. Make the bindings visible, compile, and link the application. For example,

   ```
   $ a.path -a general
   $ a.build main
   ```

3. Start **`ntraceud`**, the NightTrace user daemon, to capture trace events in a trace-event log file. (Note: this file should have the same name as the file specified in the `trace_start` call.) (See **`ntraceud(1)`**.) For example,

   ```
   $ ntraceud  my_trace_file
   ```

4. Run the application and simultaneously log trace-event information into a file. For example,

   ```
   $ a.out
   ```

5. Stop **`ntraceud`**, the NightTrace user daemon, when the application completes. For example,

   ```
   $ ntraceud  --quit  my_trace_file
   ```

6. From an X server, set the `DISPLAY` environment variable to the server name. This needs to be done only once per login. An example of setting this variable in the Bourne shell for a terminal named “eagle” follows:

   ```
   $ DISPLAY=eagle:0.0
   $ export DISPLAY
   ```

7. From an X server, view the trace-event information from the trace-event log file with **`ntrace`**, the NightTrace graphical display tool. (See **`ntrace(1)`**.) For example,

   ```
   $ ntrace my_trace_file
   ```

See the *NightTrace User's Guide* for further information.

## NightView Debugger

The NightView Debugger provides a means of modifying an executable program so that it logs trace events.

See the *NightView User's Guide* for details.

# Tracing Options

Tracing behavior is controlled via the **-trace** link option to **a.partition**. See "Link Options" on page 4-109 for more information.

Examples of usage are provided in "Tracing Options - Examples" on page 11-17.

The format for this option is:

**-trace[:***attributes***]**

where *attributes* is a comma-separated list of the following (the defaults appear in parentheses):

**enabled**=true|false (true)
**mechanism**=internal[**/**default|rcim_tick]|ntraceud
(internal/default)
**buffersize**=*n* (1000)
**rtsinstrumentation**=true|false (true)
**elabinstrumentation**=true|false (true)

Note that any of the keywords for the above attributes can be abbreviated to their shortest non-ambiguous form.

The following steps will help determine which *attributes* are necessary to obtain the desired tracing behavior.

1. Determine whether tracing should be activated for this executable.

   **-trace**

   When this option is specified, tracing support will be included in the output file, allowing the logging of trace events.

   If no attributes are specified, the default values for the attributes are used. In particular, tracing is automatically enabled. To override this default, you may set the **enabled** attribute for the **-trace** option to `false`.

**NOTE**

If the partition is linked with the **-trace** option, tracing may be subsequently enabled or disabled at runtime without the need for relinking by calling `user_trace.set_trace_mode` or `user_trace.set_trace_mode_all`. See "user_trace package" on page 11-3 for more information.

In addition, tracing may be enabled or disabled without the need for relinking by using **a.map**. See "a.map" on page 4-47 for more information.

If tracing support is not desired, then there is no need to specify the **-trace** option to **a.partition**. However, if tracing is subsequently desired, the program must be relinked with this option.

2. Determine whether tracing should be enabled.

   **enabled=**`true | false (true)`

   This attribute allows the user to control initially whether or not trace events are to be logged to a trace buffer. When the **-trace** option is specified, logging is automatically enabled.

   **enabled=**`true`

   > Enables logging of predefined and user-defined trace events to the trace buffer. (See "Predefined Trace Events" on page 11-2 and "User-Defined Trace Events" on page 11-2 for more information.)

   **enabled=**`false`

   > Disables logging of predefined and user-defined trace events.

   Tracing may be enabled or disabled at runtime by calling `user_trace.set_trace_mode` or `user_trace.set_trace_mode_all`. (See "user_trace package" on page 11-3 for more information.)

   **NOTE**

   > If the partition was linked with the **-trace** option, tracing may be enabled or disabled without the need for relinking by using **a.map**. (See "a.map" on page 4-47 for more information.)

3. Determine the mechanism used to log trace events.

   **mechanism=**`internal[/[default|rcim_tick]] | ntraceud`
   `(internal/default)`

   Real-time event tracing can be performed by either the MAXAda-supplied executive (`internal`) or the NightTrace daemon (`ntraceud`). See "Logging Mechanisms" on page 11-19 for more information.

   **NOTE**

   > If the partition was linked with the **-trace** option, the mechanism may be specified without the need for relinking by using **a.map**. (See "a.map" on page 4-47 for more information.) This is only applicable if the executable was originally linked with the **mechanism** setting of `ntraceud`.

**mechanism=**internal

This attribute specifies that logging shall be performed by the Ada executive independent of the NightTrace product. See "Ada Executive" on page 11-19 for more information.

The Ada executive logs trace events to wraparound trace buffers in memory (one buffer per task). When a trace buffer becomes full, the newest trace events overwrite the oldest trace events in that buffer.

When the Ada executive is selected as the mechanism to log trace events, further attributes may be specified:

i. Select the mechanism used to determine timestamps for trace events. See "Timing Source" on page 11-20 for more information.

**mechanism=**internal/default

By default, when the internal mechanism is chosen, the timing device used to determine timestamps for trace events is the default high-precision clock for the architecture where the program runs; specifically, the interval timer (NightHawk 6000 Series), the Time Base Register (Power Hawk/PowerStack), or the Time Stamp Counter (TSC) for Pentium systems.

**mechanism=**internal/rcim_tick

If you are running on a closely-coupled system that has a Real-Time Clock and Interrupt Module (RCIM) attached, you may specify the synchronized tick clock on the RCIM as the trace timing source.

See the *Real-Time Clock and Interrupt Module User's Guide* (0891082) for more information about this device.

ii. Specify the size of the trace buffer that the Ada executive uses to log trace events. See "Trace Buffer" on page 11-20 for more information.

**buffersize=***n* (1000)

The length specified is the maximum number of trace events for each task that can be contained within the buffer. For instance, it may be desirable to specify a fairly large buffer length if there are in excess of 500 library units being traced. In such a case, the ENVIRONMENT task will have logged a minimum of 1000 trace events (an entry and exit of each library unit elaboration) before the main subprogram even executes.

If the partition was linked with the **-trace** option, the size of the trace buffer may be specified without the need for relinking by using **a.map**. (See "a.map" on page 4-47 for more information.)

**mechanism=**ntraceud

This attribute specifies that logging shall be performed via the NightTrace user daemon, **ntraceud**. This method allows greater flexibility, providing a number of options to tailor the tracing to the needs of the user. See "NightTrace Daemon" on page 11-21 for more information. Also, see **ntraceud(1)** and the *NightTrace User's Guide* (0890398) for more information about the NightTrace user daemon.

The NightTrace user daemon has its own option for selecting a timing source. See the section titled **Option to Select Timestamp Source (-clock)** in Chapter 4 of the *NightTrace User's Guide* (0890398).

In addition, the NightTrace user daemon has its own option for setting the shared memory buffer size. See the section titled **Option to Define Shared Memory Buffer Size (-memsize)** in Chapter 4 of the *NightTrace User's Guide* (0890398).

4. Determine whether runtime events are desired.

   **rtsinstrumentation=**true|false (true)

   When set to true, this attribute causes the tracing version of the Ada executive to generate predefined trace events as the application executes. These trace events describe execution mostly in terms of tasking, interrupt handling, exception occurrence and handling, and protected actions.

   See "Predefined Trace Events" on page 11-2 for more information.

5. Determine whether library unit elaboration events are desired.

   **elabinstrumentation=**true|false (true)

   When set to true, this attribute causes the generation of a pair of trace events (entry and exit) for the elaboration of every library unit in the partition.

**NOTE**

The user may wish to increase the length of the trace buffer used for logging trace events if there are a large number of library units to be traced. This is specified using the **buffersize** attribute of the **-trace** option.

## Tracing Options - Examples

The following are some examples of using the **-trace** link option:

**-trace:enabled=false**

Specifies that tracing is activated but no logging of trace points will occur (unless the user subsequently modifies the **enabled** setting via the **a.map** tool or calls the

`set_trace_mode` subprograms from the `user_trace` package at runtime - see “user_trace package” on page 11-3 for more information.).

In addition, the default settings for other trace attributes specify that library unit elaboration and runtime events would be traced, the tracing **`mechanism`** would be the Ada executive (i.e. not **`ntraceud`**) using the default timing source, and the size of the per-task trace buffer is 1000 (events).

**`-trace:rts=false,elab=true,mech=ntraceud`**

Specifies that tracing is activated and enabled and that library unit elaboration will be traced but runtime events will not.

In addition, the default settings for other trace attributes specify that the tracing mechansim is **`ntraceud`**; this requires the user to start the NightTrace user daemon (**`ntraceud`**) before executing the program being traced.

**`-trace:mech=internal/rcim_tick`**

Specifies that tracing is activated and enabled and that library unit elaboration and runtime events will be traced and that the tracing mechanism is the Ada executive (i.e. not **`ntraceud`**) using the RCIM synchronized tick clock as the timing source and the size of the per-task trace buffer is 1000 (events). Use of the RCIM sychronized tick clock as the timing source is required for subsequent trace analysis if multiple trace files from multiple single board computers are to be combined (which in turn requires that each of the single board computers have an RCIM all connected in the same chain).

# Logging Trace Events

This section discusses the available mechanisms used to log trace events and the resultant log files from the tracing activity.

## Logging Mechanisms

Logging of trace events is done by either the MAXAda-supplied executive or the NightTrace daemon.

### Ada Executive

The Ada executive can log predefined and user-defined trace events. It does not require the use of the NightTrace product.

This mechanism is specified by setting the **`mechanism`** attribute of the **`-trace`** option to `internal`. See "Tracing Options" on page 11-14 for details.

Because of the **`-trace`** link option (and the default values for its associated attributes) in the following example, predefined trace events will be generated by the Ada executive in addition to any user-defined `user_trace` events (see "Tracing Options" on page 11-14 for details). The **`a.partition`** command which sets the **`-trace`** option is not required if only user-defined `user_trace` events are desired. However, in such a case, the **`enabled`** attribute defaults to `false` so no trace events will be generated until the mode is changed (either via the **`a.map`** tool or at runtime via a call to `user_trace.set_trace_mode` or `user_trace.set_trace_mode_all`).

#### Example

1. Introduce the source file

   ```
   $ a.intro some_tasking_program.a
   ```

2. Create the partition

   ```
   $ a.partition -create active some_tasking_program
   ```

3. Select Ada executive logging

   ```
   $ a.partition -oset "-trace" some_tasking_program
   ```

4. Build the partition

   ```
   $ a.build some_tasking_program
   ```

5. Invoke the application

```
$ some_tasking_program
```

**NOTE**

The **-trace** option could be specified at the same time the partition is created. The command would look like:

```
$ a.partition -create active -oset "-trace"
some_tasking_program
```

### Trace Buffer

The Ada executive logs trace events to a separate wraparound trace buffer in memory. When a trace buffer is full, the newest trace events overwrite the oldest trace events in that buffer.

Each task has its own trace buffer in memory so there is never any buffer contention when logging trace events. The default size of these buffers is 1000 entries (approximately 24 bytes per entry) and may be configured by setting the `trace_buffer_length` variable in the `user_trace` package. These buffers are allocated during task creation. A `storage_error` exception is raised if the allocation fails.

Additionally, the size of the buffer can be specified using the **buffersize** attribute to the **-trace** option. See “Tracing Options” on page 11-14 for details.

**NOTE**

If the partition was linked with the **-trace** option, the size of the trace buffer may be specified without the need for relinking by using **a.map**. (See “a.map” on page 4-47 for more information.)

The trace events are dumped only when specified by the user or when the application exits. They are then dumped to a trace file. Because the user controls when the events are written to the trace file, there is no extraneous disk activity. See "Log Files" below for more information about these trace files.

### Timing Source

By default, the timing device used to determine timestamps for trace events is the default high-precision clock for the architecture where the program runs; specifically, the interval timer (NightHawk 6000 Series), the Time Base Register (Power Hawk/PowerStack), or the Time Stamp Counter (TSC) for Pentium systems.

If you are running on a closely-coupled system that has a Real-Time Clock and Interrupt Module (RCIM) attached, you may specify the synchronized tick clock on the RCIM as the trace timing source. See the *Real-Time Clock and Interrupt Module User’s Guide* (0891082) for more information about this device.

The timing source can be specified using the **mechanism** attribute to the **-trace** option. See "Tracing Options" on page 11-14 for details.

## NightTrace Daemon

The NightTrace user daemon, **ntraceud**, can log predefined and user-defined trace events. It is part of the NightTrace product, which is sold separately from MAXAda.

This mechanism is selected by setting the **mechanism** attribute of the **-trace** option to `ntraceud` (see "Tracing Options" on page 11-14 for details). All trace events will be logged to a singular global memory buffer controlled by **ntraceud**. It is the **ntraceud** tool itself that dumps the actual trace events from the global memory buffer to the actual trace file.

**NOTE**

The NightTrace daemon, **ntraceud**, must be invoked before the Ada application is run. In addition, **ntraceud** must be terminated when the application completes. See the example below.

The NightTrace daemon, **ntraceud**, allows greater flexibility using the buffers and trace files by providing a number of options to tailor the tracing to the needs of the user. See **ntraceud(1)** for more information about the NightTrace user daemon

### Example

1. Introduce the source file

   ```
   $ a.intro some_tasking_program.a
   ```

2. Create the partition

   ```
   $ a.partition -create active some_tasking_program
   ```

3. Select NightTrace user daemon logging

   ```
   $ a.partition -oset "-trace:mech=ntraceud"
   some_tasking_program
   ```

4. Build the partition

   ```
   $ a.build some_tasking_program
   ```

5. Invoke the NightTrace user daemon (note the file specified)

   ```
   $ ntraceud some_tasking_program.trace.data
   ```

6. Invoke the application

   ```
   $ some_tasking_program
   ```

7. After the application finishes, stop the NightTrace user daemon

   ```
   $ ntraceud --quit some_tasking_program.trace.data
   ```

**NOTE**

The selection of the NightTrace user daemon could be made at the same time the partition is created. The command would look like:

```
$ a.partition -create active -oset
"-trace:mech=ntraceud" some_tasking_program
```

## Log Files

Two files are created with respect to trace event logging, regardless of which method of logging is chosen:

1. *program_name*`.trace.data`

   This binary trace event file contains sequences of trace event information that the application logged.

2. *program_name*`.trace.tables`

   This ASCII configuration file contains dynamically generated string tables of textual information about the tasks in the Ada application.

# Viewing Trace Events

Once the trace events have been logged, they must be viewed for analysis. MAXAda provides two methods of viewing the trace event log files. MAXAda supplies a utility, **`a.trace`**, for viewing the trace event logging results. In addition, the results may be viewed using the NightTrace product.

Refer to Figure 11-1 to see how each method is used for viewing trace events.

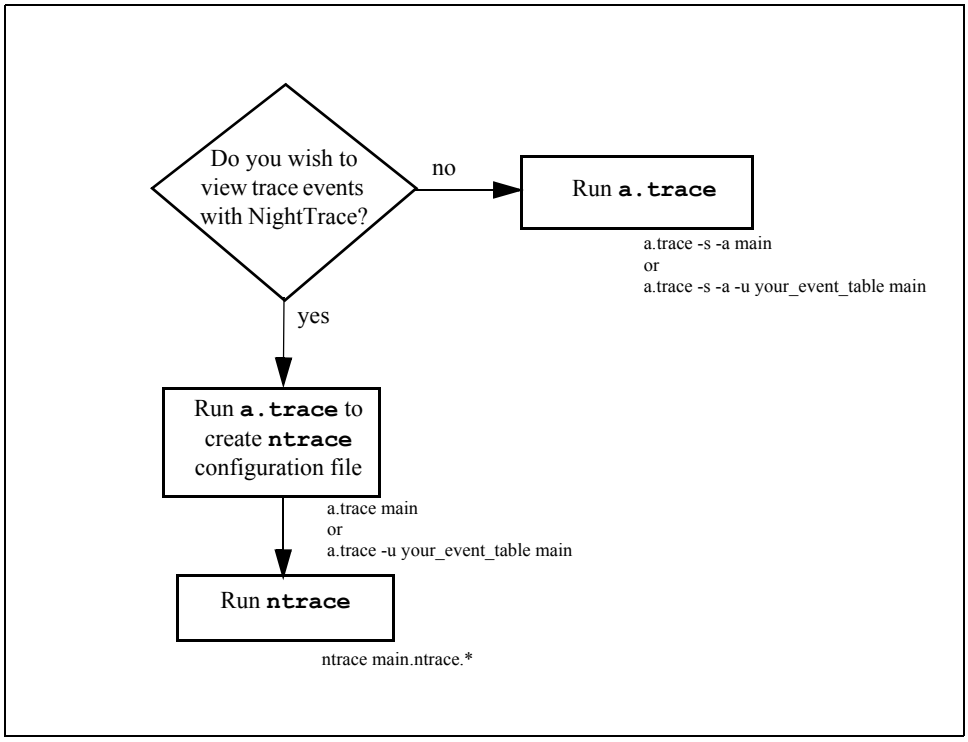



**Figure 11-1. Viewing Trace Events**

# User Table

A *user table* contains a format table that associates specific `user_trace` trace events with particular character strings. The user table can be used:

- by **`a.trace`** when viewing trace events or
- by **`a.trace`** when creating the configuration files needed by NightTrace

The **`-u`** option to **`a.trace`** is used to specify the name of the file containing the user table.

The user table uses each trace event `sub_id` as a table key, associating a particular character string with each event. Formatting for these character strings and optional values

displayed within the string can be specified for each trace event. Events not specified in the user table are displayed using the formatting for the `default_item`.

See the *NightTrace User's Guide* (0890398) for details on format tables and specifying these values.

**NOTE**

The format table *table_name* specified must be named `ada_user_trace`.

An example of the contents of a simple user table file might look like:

```
format_table (ada_user_trace) = {
        default_item = "<who knows>" ;
        item = 1 , "Start it: data1=%d, data2=%d", "arg3", "arg4" ;
        item = 2 , "End it: data1=%d, data2=%d", "arg3", "arg4" ;
} ;
```

When viewed, trace events with a `sub_id` of `1` will produce a string similar to the following:

```
Start it: data1=2, data2=2
```

containing the particular runtime values for `data1` and `data2` at the time the trace event was logged.

## Viewing Trace Events with a.trace

MAXAda provides the **`a.trace`** utility as a stand-alone method of viewing Ada executive predefined trace events as well as user-defined `user_trace` trace events. There is no requirement for either NightTrace or for an X server. The resultant listing of trace events is displayed in ASCII format in chronological order. High-level symbolic information, trace-event time stamps, and raw-trace dumps are also available.

**`a.trace`** uses *program_name*`.trace.data` and *program_name*`.trace.tables`. See “Log Files” on page 11-22 for more information about these files.

To view trace event information in ASCII:

**`a.trace -a`** *program_name*`.trace.data`

or optionally,

**`a.trace -a`** *program_name*

**NOTE**

The **.trace.data** extention is optional when invoking **a.trace**. **a.trace** will append **.trace.data** to the *program_name* when invoked if it is not specified.

Additionally, a user table file can be specified by using the **-u** option to **a.trace**. For example:

**a.trace -u** *user_table_file* **-a** *program_name*

will display the `user_trace` trace events according to the formatting specified in the format table contained in *user_table_file*. See "User Table" on page 11-23 for more details.

See "a.trace" on page 4-98 for more information about this utility and its options.

# Viewing Trace Events with NightTrace

In order to view trace events with the NightTrace graphical display utility, **ntrace**, a configuration file must be created for use with NightTrace. The MAXAda utility **a.trace** creates this NightTrace configuration file for viewing predefined trace events and user-defined `user_trace` trace events with **ntrace**.

## Creating the NightTrace Configuration File

The MAXAda-supplied utility, **a.trace** uses the *program_name***.trace.data** file from the trace event logging to create the files needed by NightTrace before it can display graphically the information obtained from the tracing. See "Log Files" on page 11-22 for more information about this file.

**a.trace** takes the *program_name***.trace.data** file as its only argument to generate the necessary files. (Note: no options should be specified to **a.trace** when generating the NightTrace files.)

To create the files needed for NightTrace to view the tracing information, issue the following command:

**a.trace** *program_name***.trace.data**

Optionally, a user table file can be specified by using the **-u** option to **a.trace**. For example:

**a.trace -u** *user_table_file program_name*

will use the formatting specified in the format table contained in *user_table_file* when creating the NightTrace configuration file. See "User Table" on page 11-23 for more details.

Either of these commands creates the following two files:

1. *program_name*`.ntrace.data`

   This file is a hard link to *program_name*`.trace.data`. See "Log Files" on page 11-22 for more information about this file.

2. *program_name*`.ntrace.config`

   This file contains string tables, format tables, and a NightTrace display page, including descriptions of NightTrace display objects for this application's trace events. It combines information from the `template` and `table` files in the `sup/trace` directory and the file created by the execution of the application, *program_name*`.trace.data`.

These two files then are given as input to the NightTrace graphical display tool, `ntrace`, to view the trace event information generated by the run of the application.

The tracing information from the application, *program_name*, can then be viewed using `ntrace` by issuing the following command:

`ntrace` *program_name*`.ntrace.*`

## Modifying the NightTrace Configuration File

The NightTrace configuration file, *program_name*`.ntrace.config`, may be modified and reused on subsequent tracings of *program_name*. This holds true only for programs which create their tasks in a deterministic order.

**NOTE**

For programs which do not create their tasks in a deterministic order, the internal representation of specific Ada tasks may change with each run, thereby invalidating a previously created (and perhaps modified) configuration file

In cases such as this, the configuration file, *program_name*`.ntrace.config`, must be recreated by executing `a.trace` on the latest trace event files. See "Creating the NightTrace Configuration File" on page 11-25 for details.

If `a.trace` is executed on the latest trace event files, modifications to the previous configuration files will be not be retained.

# 12
# Real-Time Monitoring


Data Monitoring . . . . . 12-1
Compiling . . . . . 12-1
Eligible Data Objects . . . . . 12-1
Eligible Data Types . . . . . 12-2
real_time_data_monitoring Package . . . . . 12-2
Task Monitoring . . . . . 12-3
a.monitor . . . . . 12-4
Menu Bar . . . . . 12-6
File . . . . . 12-6
View . . . . . 12-8
Options . . . . . 12-8
Task Bar . . . . . 12-11
Display Area . . . . . 12-13
Tasks . . . . . 12-13
Memory . . . . . 12-17
System . . . . . 12-20

# 12
# Real-Time Monitoring

Real-time monitoring involves observing and changing the values of program variables and displaying task states and system utilization. This chapter describes the **a.monitor** real-time monitoring utility.

## Data Monitoring

This section describes how to use MAXAda to perform data monitoring for debugging real-time applications. Currently, several ways are available in which to use the data monitoring capabilities. The following sections include information for:

- Compiling Ada source code for data monitoring
- Eligibility of data objects for data monitoring
- Using the **rtdm** `Real_Time_Data_Monitoring` package

In order to monitor programs whose effective user ID differs from the monitoring process, the `CAP_SYS_ADMIN` capability is required (see "Capabilities" on page 1-3).

### Compiling

Ada source code must be compiled with the **-g** option or pragma `DEBUG` to allow the resulting program to produce symbolic debug information required for data monitoring.

### Eligible Data Objects

The implementation of data monitoring supports monitoring and modifying only those data objects that have static addresses, such as the variables declared in an Ada library-level package. Variables declared in Ada procedures or tasks, or objects in an access type's collection, are allocated dynamically, and are, therefore, ineligible for data monitoring.

The following criteria are used to determine if a data object is eligible for data monitoring:

- The compilation unit containing the object must be a library-level package specification or body. Objects declared in nested packages inside a library-level package are also eligible.
- The object must *not* be declared in a generic or in the instantiation of a generic.

- The object must have a size and representation which is statically determined at compile time.
- The object may be declared in a library-level package marked with pragma SHARED_PACKAGE. (See "Pragma SHARED_PACKAGE" on page M-131.)

## Eligible Data Types

The following data types are eligible for data monitoring:

- Any integer, fixed-point or floating-point type or subtype.
- Any character, Boolean or enumeration type or subtype.
- Access types.
- Array and record types (for records with variant parts, only components that have a statically determined component offset are eligible).
- Task types are *not* eligible types.

## real_time_data_monitoring Package

See "rtdm" on page 9-12 for an overview of this package.

A detailed description of all interfaces and services can be found in the *Data Monitoring Reference Manual* (0890493).

# Task Monitoring

This section describes how to use MAXAda to perform real-time task monitoring for debugging real-time applications. No special options or pragmas are required in the program to be monitored. Task monitoring is accomplished through use of the **`a.monitor`** tool.

# a.monitor

The MAXAda **a.monitor** utility provides users with a full-screen real-time program monitor. It provides an interactive menu interface that allows users to cyclically monitor task and memory information  Currently, **a.monitor** can only monitor Ada tasking programs.

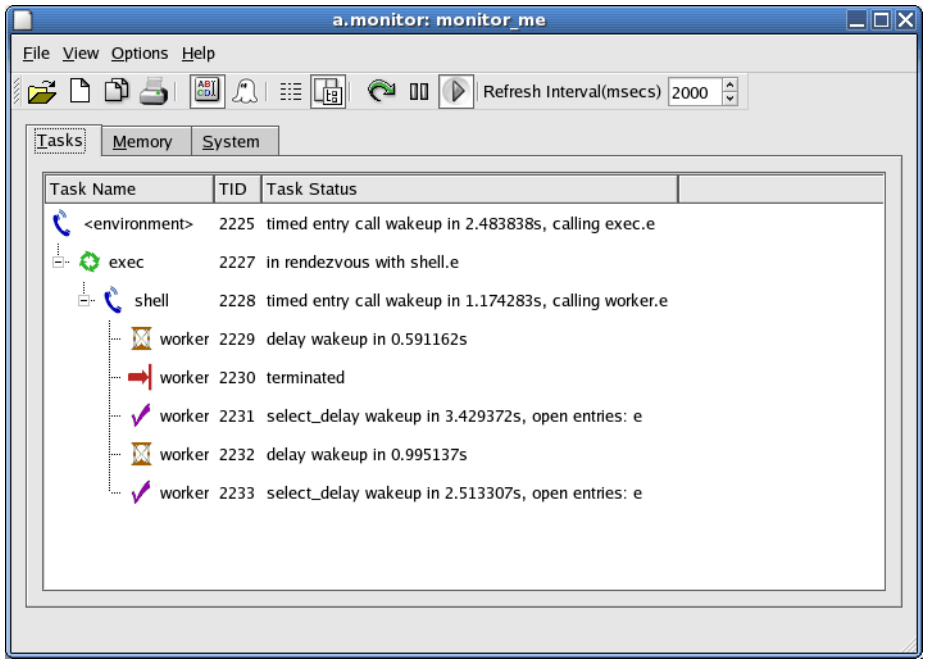



**Figure 12-1.  a.monitor**

The **a.monitor** utility:

- is non-intrusive.  It operates independently of target applications.
- can monitor an Ada program in real-time by displaying system utilization and the activities of individual Ada tasking threads of execution
- is a stand-alone monitor that does not require an entire compilation environment

Use of **a.monitor** requires the `CAP_SYS_ADMIN` privilege (see “Capabilities” on page 1-3) when monitoring processes whose effective user IDs are other than that of the user invoking **a.monitor**.

**a.monitor** can also be used in a non-graphical mode in situations where a graphical display is unavailable or unwanted.  Depending on supplied options, **a.monitor** will generated standard ASCII text to **stdout**.  See  “a.monitor” on page 4-55 for command-line syntax and a list of options.

The **`a.monitor`** graphical user interface consists of the following components:

- Menu Bar (see “Menu Bar” on page 12-6)
- Task Bar (see “Task Bar” on page 12-11)
- Display Area (see “Display Area” on page 12-13)

## Menu Bar

The **a.monitor** menu bar provides access to the following menus:

- File (see "File" on page 12-6)
- View (see "View" on page 12-8)
- Options (see "Options" on page 12-8)

### File

The File menu allows the user to specify which program to monitor in the current window and provides the user with the option to create either a new window in which to monitor a different program or a clone window in which to monitor a different view of the current program. The user may also print a snapshot of the data displayed in the Display Area to a printer or to a file.

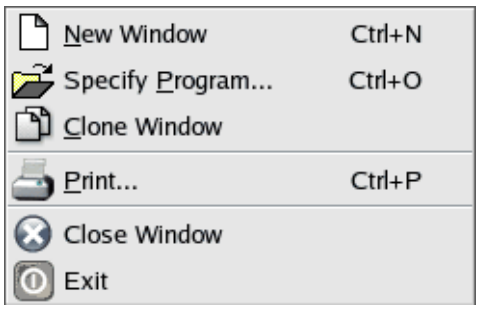



**Figure 12-2. a.monitor - File menu**

#### New Window

The New Window item creates a new window that initially has all the same attributes of the window from which New Window is initiated. However, unlike Clone Window, changes to the program specification do not affect the parent window. Thus, New Window can be used to monitor an additional program simultaneously. Use of Clone Window subsequently from a New Window will create a new *clone window group*.

#### Specify Program...

This menu item launches the Program Selection dialog which instructs **a.monitor** as to which program should be monitored. Programs may be specified on the **a.monitor** command line or selected via this dialog.

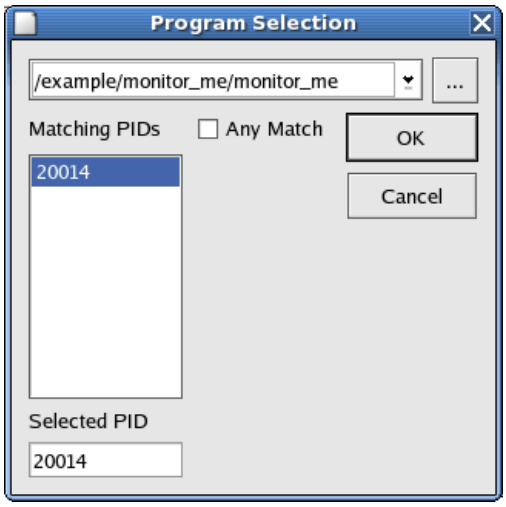



**Figure 12-3. Program Selection dialog**

The Program Selection dialog presents a standard file browser for selection of the executable file representing your program. The lower portion of the dialog provides for selection of an operating system process ID (PID). The Matching PIDs list is populated with PIDs of currently executing processes which match the executable file selection. You can select any of the PIDs from that list, type in a specific PID in the Selected PID field, or check the Any Match checkbox. When Any Match is selected, **a.monitor** will choose one of the existing PIDs or, if no processes currently exist that match the executable file name, **a.monitor** will enter *scan mode*.

*Scan mode* is a state in which **a.monitor** stops displaying program information because the program being monitored has exited or does not exist. In this state, **a.monitor** periodically scans the system for a matching PID. Once a PID is located, *monitoring mode* is automatically re-initiated.

In *monitoring mode*, **a.monitor** continues to iteratively display information on the program until the user exits the tool, the program exits, or the user specifies a new program/process. Thus **a.monitor** can be invoked with a program file name and left running to monitor all subsequent invocations of the program.

### Clone Window

The Clone Window item creates a new window with the same attributes of the window from which Clone Window is initiated. It retains the program specification relationship with its parent window, but all other changes to the window remain properties of the clone. Clones of clones are all members of a *clone window group* - they all share the same Program Specification. If a new program is selected in any window of a *clone window group*, then it instantly applies to all windows in the *clone window group*.

#### Print...

Opens the Setup Printer dialog allowing the user to print a snapshot of the data in the Display Area (see "Display Area" on page 12-13) to a printer or to a file.

#### Close Window

The Close Window item closes the current window. It does not affect any other open windows. If the current window is the only window open, **a.monitor** exits.

#### Exit

Exits the **a.monitor** tool.

### View

The View menu allows the user to select the type of information displayed by **a.monitor**.

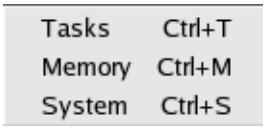



**Figure 12-4. a.monitor - View menu**

#### Tasks

Selecting the Tasks view displays information about the state of all active tasks in the program. See "Tasks" on page 12-13.

#### Memory

When the Memory view is selected, **a.monitor** provides information about the virtual address space of the program. See "Memory" on page 12-17.

#### System

Selecting the System view displays system information such as the TID, PID, CPU bias, and scheduling priority for all active tasks. See "System" on page 12-20.

### Options

The Options menu provides the user with options regarding the amount of information displayed in the Display Area (see "Display Area" on page 12-13), whether ghost tasks are displayed, and whether the information in the Tasks view (see "Tasks" on page 12-13) should be displayed in columnar format or in a hierarchical manner.

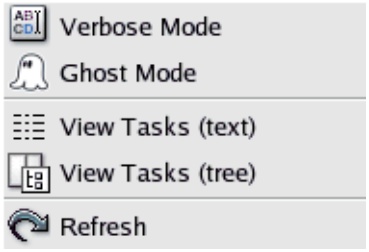



**Figure 12-5. a.monitor - Options menu**

### Verbose Mode

The Verbose Mode item toggles the amount of infomation displayed in the Display Area (see "Display Area" on page 12-13). When Verbose Mode is selected, the following information is displayed:

- time-out values for delay, timed entry call, and timed accept statements
- open entry names for entry call, accept, and select statements
- interrupt counts for interrupt attachments to tasks or protected objects

### Ghost Mode

Use of the Ghost Mode item toggles whether *ghost tasks* are displayed in the Display Area (see "Display Area" on page 12-13). Ghost tasks are tasks created by the MAXAda run-time system for interrupt handling and delivery as well as administrative actions. When Ghost Mode is selected, ghost tasks are included in the Display Area When Ghost Mode is not selected, they are omitted from the display. See "Ghost Tasks" on page 5-5 for more information.

### View Tasks (text)

Displays information on the Tasks page of the Display Area (see "Display Area" on page 12-13) in columnar format as shown in Figure 12-6:

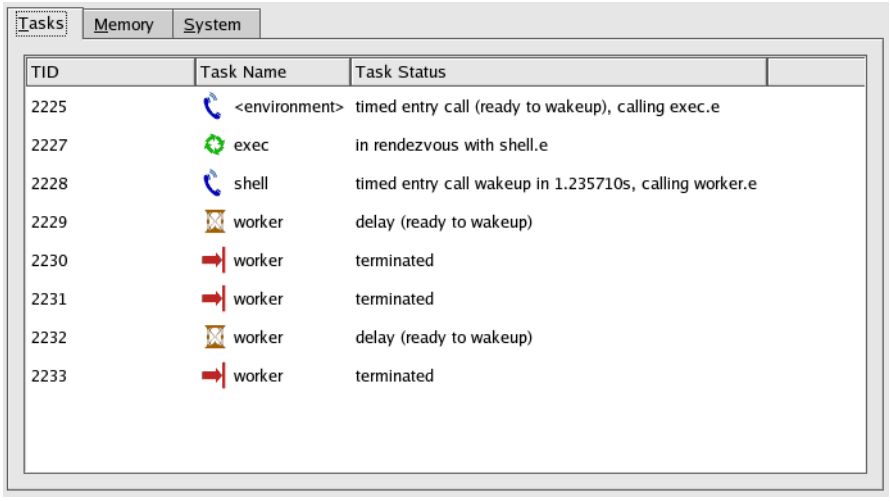



**Figure 12-6. a.monitor - View Tasks (text)**

### View Tasks (tree)

Displays information on the Tasks page of the Display Area (see "Display Area" on page 12-13) in a hierarchical manner illustrating graphically the relationships between the tasks. This is shown in Figure 12-6:

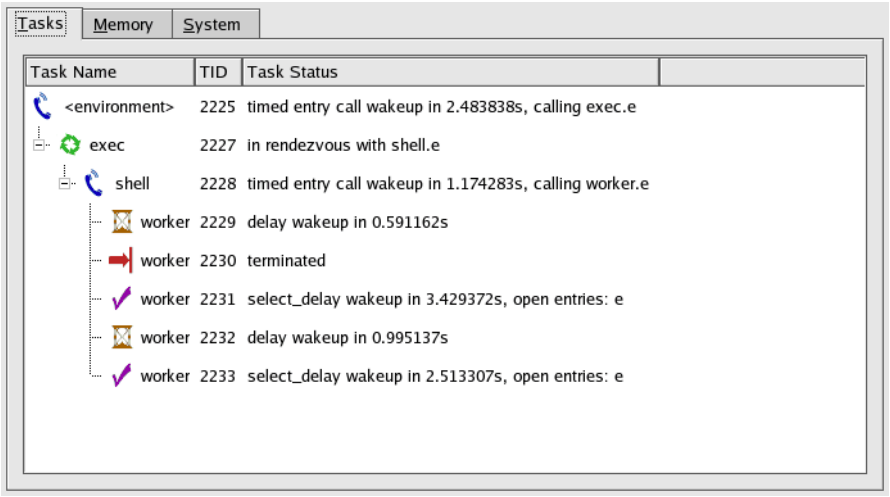



**Figure 12-7. a.monitor - View Tasks (tree)**

### Refresh

Updates the Display Area (see "Display Area" on page 12-13) with the most recent information.

## Task Bar

**`a.monitor`** provides a task bar for quick access to some of the more commonly-used features of this utility.


**Figure 12-8. a.monitor Task Bar**

The following describe each item on the task bar:

### Specify Program

Launches the Program Selection dialog which instructs **`a.monitor`** as to which program should be monitored.

See “Specify Program...” on page 12-6 for more information.

### New Window

The New Window item creates a new window that initially has all the same attributes of the window from which New Window is initiated. However, unlike Clone Window, changes to the program specification do not affect the parent window. See “New Window” on page 12-6 for more information.

### Clone Window

The Clone Window item creates a new window with the same attributes of the window from which Clone Window is initiated. It retains the program specification relationship with its parent window, but all other changes to the window remain properties of the clone.

See “Clone Window” on page 12-7 for more information.

### Print

Opens the Setup Printer dialog allowing the user to print a snapshot of the data in the Display Area (see “Display Area” on page 12-13) to a printer or to a file.

### Verbose Mode

The Verbose Mode item toggles the amount of infomation displayed in the Display Area (see “Display Area” on page 12-13).

See “Verbose Mode” on page 12-9 for more information.

### Ghost Mode

Use of the Ghost Mode item toggles whether *ghost tasks* are displayed in the Display Area (see "Display Area" on page 12-13).

See "Ghost Mode" on page 12-9 for more information.

### View Tasks (text)

Displays information on the Tasks page of the Display Area (see "Display Area" on page 12-13) in columnar format.

See "View Tasks (text)" on page 12-9 for more information.

### View Tasks (tree)

Displays information on the Tasks page of the Display Area (see "Display Area" on page 12-13) in a hierarchical manner illustrating graphically the relationships between the tasks.

See "View Tasks (tree)" on page 12-10 for more information.

### Refresh

Updates the Display Area (see "Display Area" on page 12-13) with the most-recent information.

### Pause Data

Temporarily halts updating of the information in the Display Area (see "Display Area" on page 12-13).

### Action

Resumes updating of the information in the Display Area (see "Display Area" on page 12-13).

### Refresh Interval

Duration of time (in milliseconds) before updating the information in the Display Area (see "Display Area" on page 12-13).

## Display Area

The bottom portion of the **a.monitor** GUI is divided into three pages that show different views of the information being monitored.

These views are:

- Tasks (see page 12-13)
- Memory (see page 12-17)
- System (see page 12-20)

### Tasks

When the Tasks menu item is selected from the View menu of **a.monitor** (see "View" on page 12-8), information is provided about the state of all active tasks in the program.

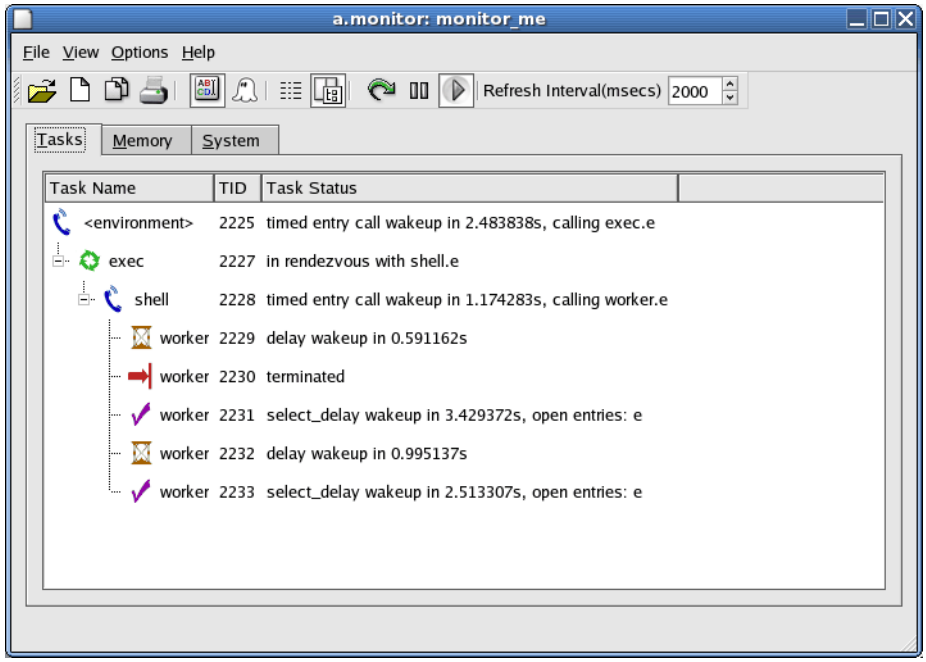



**Figure 12-9. a.monitor - Tasks view**

**NOTE**

The information in the Display Area (see "Display Area" on page 12-13) can be sorted by clicking on the desired column header.

Three columns of information are presented.

### TID

The task identifier (TID) is the process ID (PID) of the displayed task. Some ghost tasks do not have processes associated with them; their TID will be zero. Otherwise, a TID uniquely identifies a task.

### Task Name

The Task Name provides the name of the task using the simple name of the task object from the user's source code. For allocators or components of composite types, the simple name of the task type is used.

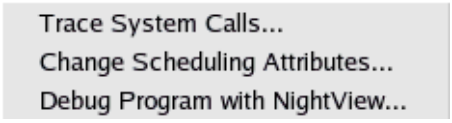
The name <*environment*> is used to identify the *environment task* (as per section 10.2 of the *Ada Reference Manual*).

*Ghost tasks* also include notations within <*angle-brackets*> to identify a specific type of ghost task.

In addition, an icon precedes the task name indicating the state of that particular task:

| | |
|---|---|
|  | executing |
|  | calling |
|  | rendezvous |
|  | ready |
|  | timeout |
|  | waiting |

Right-clicking on a particular task name presents the following menu:



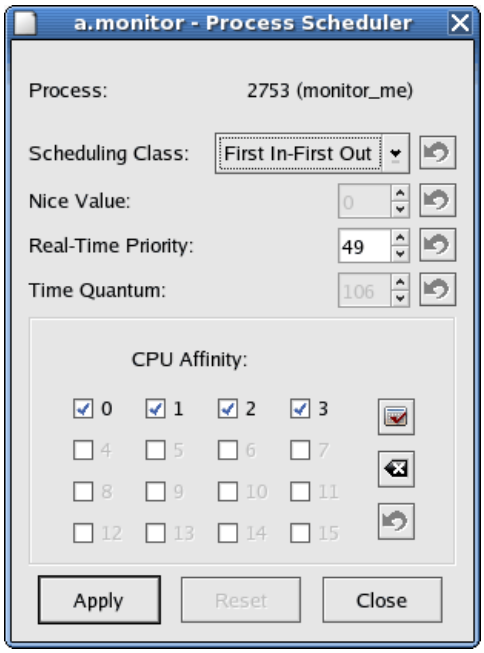

These menu items are described below:

### Trace System Calls...

Selecting this menu item launches a scrollable text dialog which shows system call activity (using the **`strace`** utility) of an individual task.

### Change Scheduling Attributes...

Displays the following dialog, allowing the user to change scheduling attributes of an individual task including the POSIX scheduling policy, scheduling priority, and the CPU affinity.




### Debug Program with NightView...

Selecting this menu item launches the NightView debugger which will automatically attach to the process. NightView fully supports debugging multi-

tasking programs. NightView attaches to all tasks in the program; therefore, selecting this menu item for a single task affects the entire program.

### Task Status

The Task Status column provides a description of the state of the task in terms relating to activities defined specifically by the Ada language. Typically these includes delay statements, task and protected object entry calls, and accept and select statements. When a task is described as *executing*, it is executing in the Ada language sense - not blocked on a resource or timing event directly related to an Ada language statement. This does not necessarily imply that the task is actually executing on a CPU. It may be executing on a CPU, blocked in a system call or may be available for execution by a CPU which is currently executing another task or process.

**NOTE**

The task information screen is available only for Ada tasking programs. It will display:

```
Not an Ada program.  No tasks available
```

if the program is not an Ada tasking program, has not yet been fully elaborated, or has had all symbolic information stripped from the executable file.

## Memory

When the Memory menu item is selected from the View menu of **a.monitor** (see "View" on page 12-8), information is provided about the virtual address space of the program.

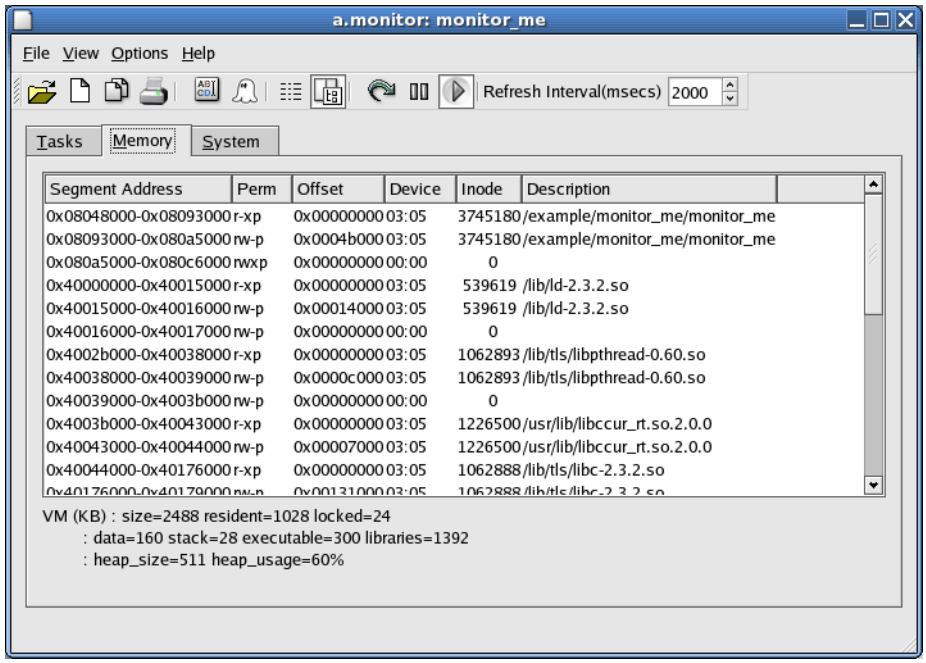



**Figure 12-10. a.monitor - Memory view**

**NOTE**

The information in the Display Area (see "Display Area" on page 12-13) can be sorted by clicking on the desired column header.

For each segment of the virtual address space, the following information is displayed:

### Segment Address

The Segment Address column shows the virtual address range of a segment of memory that shares the set of attributes in the remaining columns.

### Perm

The Perm column describes the permissions associated with the address segment, using the notation `rwxp` which indicates whether the segment is readable, writable, executable, private (or shared). Thus a description of `r-xp` would indicate that the

segment is readable, not writable, executable (typically containing machine instructions), and is private (not shared).

#### Offset

The Offset column indicates the offset, if any, from the base object associated with the segment. This is most often a segment of a file which has been memory mapped. Examples include the `.text` and `.data` segments of an executable file.

#### Device

The Device column indicates the device major and minor numbers associated with the file memory mapped to the specified segment. The major and minor numbers are displayed in hexadecimal format.

#### Inode

The Inode column indicates the `inode` of the corresponding file, if any, which is memory mapped to the specified segment.

#### Description

The Description column includes a textual description of the segment, when available. Segments for which information is available typically include the pathname of the executing program, pathnames to any required shared libraries, and names of Ada packages and collections. Ada package names are not available unless they have memory-related pragmas associated with them.

The summary statistics sections of the screen are also dynamically updated. They include:

#### Virtual Size

The size in KB of all virtual pages associated with the program. Pages included in this statistic may be currently swapped out and therefore not currently allocated to physical memory.

#### Resident Size

The size in KB of all pages within the program currently occupying physical memory.

#### Locked Size

The size in KB of all pages locked into physical memory.

#### Data Size

The size in KB of all pages associated with data, collections, stack, as well as anonymous pages **`mmaped`** into the address space (e.g. **`/dev/zero`**), not including any static data pages associated with required shared libraries.

### Stack Size

The size in KB of the *environment task* stack. This does not include stacks allocated for Ada tasks. If the program includes memory-related pragmas which select a non-standard set of attributes for the *environment task* stack, additional space is allocated for that stack and it is not included in this statistic.

### Executable Size

The size in KB of all executable pages associated with the program, not including executable pages from any required shared libraries.

### Library Size

The size in KB of all pages associated with shared libraries required by the program.

### Heap Size
### Heap Usage

For Ada programs, information on the state of the main collection (often referred to as a *heap*), which is used to allocate space for dynamically sized variables, Ada allocators, and Ada task stacks. It includes the size in KB of the heap and the percentage of the heap that has been utilized. The heap may grow in size as the program executes, unless the user has put a specific limit on it via memory-related pragmas.

### System

When the System menu item is selected from the View menu of **a.monitor** (see “View” on page 12-8), system information such as the TID, PID, CPU bias, and scheduling priority is provided for all active tasks.

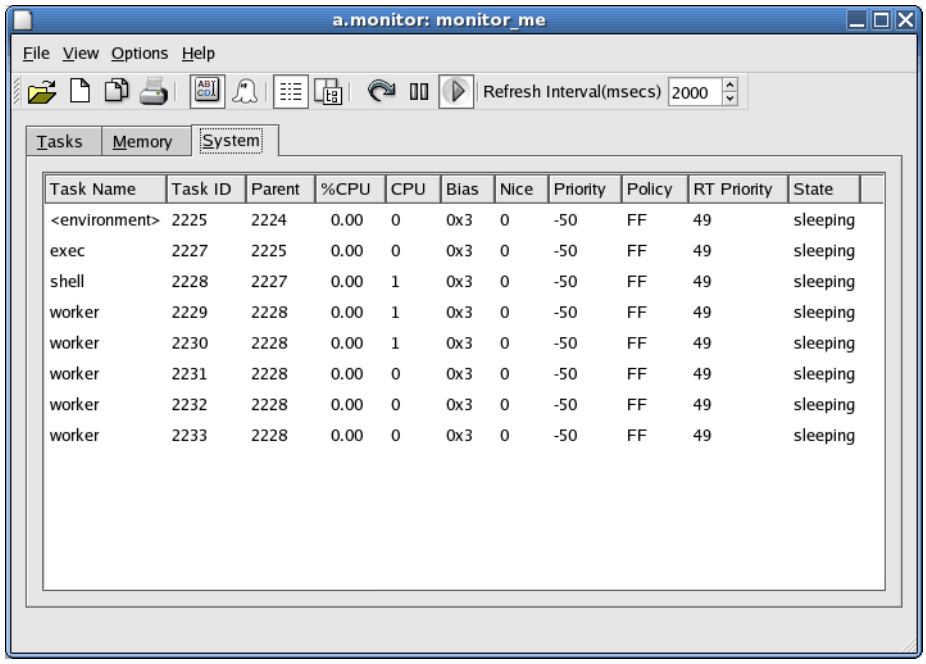
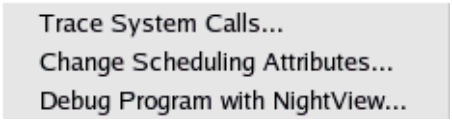



**Figure 12-11. a.monitor - System view**

**NOTE**

The information in the Display Area (see “Display Area” on page 12-13) can be sorted by clicking on the desired column header.

Right-clicking on a particular task name presents the following menu:



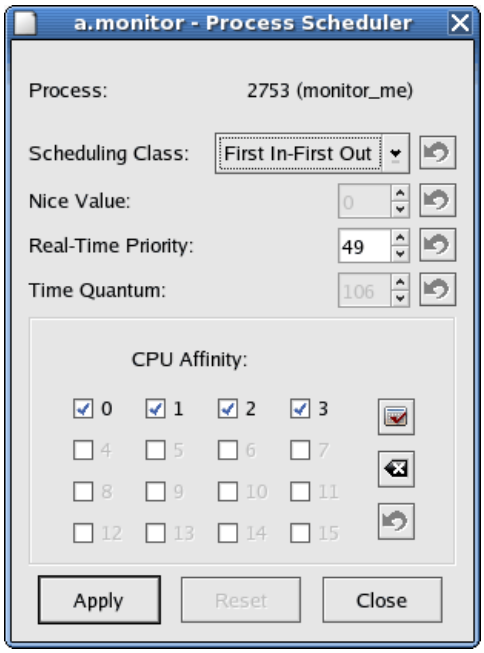

These menu items are described below:

### Trace System Calls...

Selecting this menu item launches a scrollable text dialog which shows system call activity (using the **`strace`** utility) of an individual task.

### Change Scheduling Attributes...

Displays the following dialog, allowing the user to change scheduling attributes of an individual task including the POSIX scheduling policy, scheduling priority, and the CPU affinity.




### Debug Program with NightView...

Selecting this menu item launches the NightView debugger which will automatically attach to the process. NightView fully supports debugging multi-

tasking programs. NightView attaches to all tasks in the program; therefore, selecting this menu item for a single task affects the entire program.

# 5
# Appendixes and Index

**Replace with Part 5 tab**

# Part 5 - Appendixes, Glossary, and Index

Part 5 Appendixes and Index

Appendix A Troubleshooting ..... A-1

Appendix B MAXAda Configuration ..... B-1

Appendix C Ada NightView..... C-1

Appendix M Implementation-Defined Characteristics.....M-1

# A
# Troubleshooting


Configuration Errors . . . . . A-1
System Configuration . . . . . A-1
Application Configuration . . . . . A-2
Using Tasks to Multithread Algorithms . . . . . A-2
User Errors . . . . . A-2
Concurrent Access . . . . . A-2
Hung Processes . . . . . A-3
Client/Server Services . . . . . A-3
Run-Time Diagnostics . . . . . A-4
Run-Time Diagnostic Messages . . . . . A-4
Compiler Errors . . . . . A-6

# A
# Troubleshooting

Typically, problems can be categorized into configuration errors and user errors.

## Configuration Errors

Configuration issues may involve many areas of the computer system depending on the features that are used. These areas include: kernel, administrative, hardware, and application software.

### System Configuration

As described in Appendix B, Ada applications require special privileges in order to execute. See "MAXAda Configuration" on page B-1 for details on kernel and privilege configuration.

If an invalid system configuration exists or sufficient privileges cannot be granted, the following scenarios are possible:

- Program fails to initialize
  - Due to system constraints on number of processes
  - Due to system constraints on the amount of real memory or swap space (see the system administrator).
  - Due to insufficient capabilities. See "Capabilities" on page 1-3.
- Task activation raises `TASKING_ERROR`
  - Due to system constraints on total number of processes system-wide.
  - Due to system constraints on total number of processes per-user.
  - Due to invalid hardware interrupt requests (bad device, device busy, bad vector number, etc.)
- Task elaboration diagnostics issued
  - Due to insufficient capability (e.g., pragma `TASK_PRIORITY` requires `CAP_SYS_NICE`; pragma `TASK_CPU_BIAS` requires `CAP_SYS_NICE`)
  - Due to invalid hardware configuration (e.g., pragmas `MEMORY_POOL` or `TASK_CPU_BIAS` specifying CPUs which do not exist)

## Application Configuration

Typical application configuration problems are those associated with the size of the environment task stack and default collection. By default there is no run-time-enforced limit on the size of the environment task stack or the default collection; however, absolute limits may be specified. When absolute limits are specified, allocation of the associated memory pages occurs at program start-up time. This is often advantageous for real-time applications as all pages can be faulted in and locked in memory, if needed.

For information on how to change these two values, see "Pragma POOL_SIZE" on page 6-26 and the `RUNTIME_CONFIGURATION` specification in **`vendorlib`**.

Another typical problem: a user-defined task may attempt to exceed its statically determined maximum stack size, resulting in `STORAGE_ERROR` being raised within the task. This often results in the task simply completing (assuming the user does not supply a handler for the `STORAGE_ERROR` exception). The application may then appear to hang if it is expecting activity from the task.

### Using Tasks to Multithread Algorithms

With bound tasks (or tasks within a group which contains multiple servers), Ada tasks can be used as an easy and effective method to multithread parallel algorithms. (E.g., one might use multiple tasks to implement a quick sort on large amounts of data.) When using this method, applications should take into account the stack requirements of their algorithms. The maximum size of a task stack frame may be set using the language-defined method of applying a `'STORAGE_SIZE` length clause to the task type specification, or via pragma `POOL_SIZE` (See "Pragma POOL_SIZE" on page 6-26).

# User Errors

Other than normal application errors, this chapter attempts to describe typical errors associated with utilizing parallel programming language techniques.

## Concurrent Access

In a multi-programmed model, (e.g., multiple non-tasking programs), access to variables that are shared between multiple processes is usually explicitly defined by the user. When programming with tasks, the user does not need to take extra steps to share data between them. Because the user does not need to explicitly mark items to be shared, it is easier to overlook concurrency problems.

Users should take steps to define critical sections where necessary. Critical sections can be defined with task rendezvous and/or protected types.

Additionally, there may be occasions where tasks poll variables which are being modified by other tasks. By default, if these variables are in library-level packages, the MAXAda

compiler ensures that all modifications are eventually stored to memory. However, the compiler is free to keep local copies of the variables in registers for short periods of time. When dealing with local (stack-based) variables, the compiler has even more freedom for register allocation.

In cases where the application needs to ensure that all references and modifications are applied *immediately* to memory, the user should apply implementation-defined pragma `VOLATILE`. (See "Pragma VOLATILE" on page M-135.)

Concurrency issues are not isolated to variables, but also apply to Ada I/O operations, interface routines, and programming on the whole.

## Hung Processes

In addition to tasking deadlock conditions, applications may hang.

For example, if an Ada task is aborted due to some non-Ada event (e.g., segmentation fault, abort external from Ada, etc.) then the run-time executive may still consider the task to be in a runnable state when the actual task is no longer executing.

This situation may be accompanied with an error message from the run-time executive if tasking operations are requested by some other task. However, an error message may not occur if tasking operations are quiescent.

## Client/Server Services

The run-time executive utilizes the client/server services of the operating system to implement task activation, rendezvous, and completion.

The services it uses are:

- **`server_block(2)`**
- **`server_wake1(2)`**
- **`server_wakevec(2)`**

Users may use these operating services directly; however, care must be taken so as not to interfere with the run-time executive.

A simple rule should be followed:

Only issue a **`server_wake1(2)`** or **`server_wakevec(2)`** call if the process(es) that are to wake up contain absolutely *no* tasking or are indeed already blocked by a user's call to the respective blocking service (**`server_block(2)`**).

If the user were to issue a **`server_wake1(2)`** call on a process that was currently blocked in the run-time executive on a **`server_block(2)`** call, it would wake up prematurely and would probably execute erroneously (or abort).

# Run-Time Diagnostics

Run-time diagnostic messages may be issued due to invalid system configuration, invalid user requests (pragmas, hardware interrupts, etc.), abnormal task termination, and internal executive failures.

All diagnostics are accompanied by a severity level, a message type, and a text description. The diagnostic severity level is either: information, warning, fatal, or panic.

Panic and fatal diagnostics occur when at least one task has terminated abnormally preventing the entire application from normal termination. Panic diagnostics are also used in cases where the integrity of the run-time system has been violated. (These are internal errors.)

Warning diagnostics may or may not prevent the completion of the operation causing them. If associated with task activation, `TASKING_ERROR` is raised when appropriate in the activator.

Run-time diagnostic messages are written to `stderr` and may be suppressed via pragma `RUNTIME_DIAGNOSTICS` (See "Pragma RUNTIME_DIAGNOSTICS" on page 6-1) or via calls available in the `RUNTIME_CONFIGURATION` package (See the specification for `RUNTIME_CONFIGURATION`).

If internal errors occur, contact Concurrent Customer Support.

## Run-Time Diagnostic Messages

Message:

```
configuration: unable to lock pages in memory, memcntl(2)
failed.
```

Cause/Correction:

Insufficient physical memory. Adjust local/global bindings. Add memory.

Message:

```
configuration: bind of package to memory pool failed, mmap(2).
```

Cause/Correction:

Invalid CPU bias for configuration. Insufficient local memory. Change pragma.

Message:

```
configuration: specified bias applies to multiple local memory
pools.
```

Cause/Correction:

CPU bias does not identify a single CPU board. Change pragma.

Message:

```
configuration: unable to set task cpu bias (mpadvise(3C));

configuration: unable to set task priority;

configuration: unable to set task quantum;
```

Cause/Correction:

Invalid configuration. Invalid access. **`errno`** supplied with message.

Message:

```
configuration: unable to set cpu_bias = 128, mpadvise: errno =
21
```

Cause/Correction:

Specification of inactive CPUs. Insufficient privilege. **`errno`** supplied. Change pragma.

Message:

```
panic: ...
```

Cause/Correction:

Abnormal termination of task. Internal errors.

# Compiler Errors

Message(s):

```
fatal: unable to open ...; errno = 24 (Too many open files)

fatal: unable to create ...; errno = 24 (Too many open files)

fatal: unable to determine current directory; errno = 24 (Too
many open files)

fatal: cannot allocate DWARF memory: open() failure; errno = 24
(Too many open files)

fatal: ...; errno = 24 (Too many open files)
```

Cause/Correction:

The tool has exceeeded the maximum allowable number of file descriptors. That maximum must be increased. If the user has sufficient privileges, this can be done with the **`ulimit -n`** command. To determine the current maximum number of file descriptors:

```
# ulimit -n
64
```

To increase the number to 256 file descriptors:

```
# ulimit -n 256
```

Message(s):

```
unit_name is damaged: net (net_name) is older than expected
(unit_time)

unit_name is damaged: net (net_name) is missing

unit_name is damaged: backup (backup_name) is older than
expected (unit_time)

unit_name is damaged: backup (backup_name) is missing

unit_name is damaged: nonshared object (object) is older than
expected (unit_time)

unit_name is damaged: nonshared object (object) is missing

unit_name is damaged: shared object (object) is older than
expected (unit_time)

unit_name is damaged: shared object (object) is missing
```

Cause/Correction:

Something has happened to the internal representation of the unit.

Run **a.build** in the home environment for the damaged units, so that it can correct any such problems.

Message(s):

```
TEXT_IO not defined
```

Cause/Correction:

This package is now a child of the package `Ada` and resides in the **predefined** environment. See "predefined" on page 9-6 for more information about this environment. However, MAXAda allows you to continue using the package `Text_IO`. You may do so by adding the **obsolescent** environment to your path. See "a.path" on page 4-74 and also "obsolescent" on page 9-13 for more information.

# B
# MAXAda Configuration

Capabilities . . . . . B-1

# B
# MAXAda Configuration

The MAXAda tools and run-time utilize sensitive real-time system services that require special capabilities that are not generally available to all users and processes. The manner in which privileges are granted to users and processes depends on the specific security configuration of the system.

Additionally, the operating system kernel may need to be reconfigured to activate features utilized by the run-time.

## Capabilities

Table B-1 shows the features, pragmas, and tools that require capabilities.

**Table B-1. Required Capabilities**

| Capability | Feature(s) Requiring Capability |
|---|---|
| CAP_SYS_NICE | Pragmas TASK_CPU_BIAS, GROUP_CPU_BIAS, MEMORY_POOL, PRIORITY, TASK_PRIORITY, INTERRUPT_PRIORITY; packages cyclic_scheduler, ada.dynamic_priorites, runtime_configuration |
| CAP_SYS_RAWIO | All tasking programs, packages rescheduling_control, task_sychronization, spin_locks. |
| CAP_IPC_LOCK | Pragma POOL_LOCK_STATE |
| CAP_SYS_ADMIN | Use of the real_time_data_monitoring package or the **a.monitor** tools when the effective user ID of the program being monitored differs from that of the monitoring program. |

# C
# Ada NightView


Hints for Debugging Ada Programs with NightView . . . . . C-1
Tasking Programs . . . . . C-1
Debugging Context . . . . . C-2
Exception Handling and Interception . . . . . C-3
Generics . . . . . C-4
General NightView Operational Hints . . . . . C-4
Listing Source, Packages, and Subprograms . . . . . C-4
Disassembly . . . . . C-5
Interest Threshold . . . . . C-5
Expression Evaluation Syntax . . . . . C-5

# C
# Ada NightView

This appendix discusses debugging Ada programs with the NightView symbolic debugger. Specific topics include:

- Debugging tasking programs
- Understanding the debugging context, NightView's **`select-context`** command, and MAXAda-supplied macros
- Debugging with exception handling and interception using NightView's **`handle/exception`** and **`x`** commands
- Debugging generics
- Debugging overloaded subprograms
- Listing source, packages, and subprograms
- Disassembling
- Evaluating expressions

For more information about NightView, see the *NightView User's Guide* and NightView's on-line help.

## Hints for Debugging Ada Programs with NightView

The "Debugging Ada Programs" section of the *NightView User's Guide* provides an overview of this topic. The following section contains hints that may also be useful, but they should not be considered a substitute for the *NightView User's Guide*.

### Tasking Programs

First, some review on the implementation of Ada tasks.

Ada tasks are implemented by the run-time system in various manners that are dependent on pragmas and linker options you specify.

A task has a weight which is one of:

- Bound
- Multiplexed
- Passive

A *bound task* always has a single clone associated with it. As such, a task's context (register set) is always represented by the registers associated with its clone.

A *multiplexed task* is served by one or more clones. At any point in time, the task's context may be represented by either:

- A server clone which is actively serving the task
- Its task control block (TCB), if no clone is actively serving the task
  (Note that the term *actively* means that the clone is currently acting on behalf of the assigned task; it does not necessarily mean that the clone is actually executing on a CPU.)

A *passive task* is served by the task it is in rendezvous with. A passive task's context is represented either by its TCB or by the clone that is currently serving it.

Consult Chapter 5 for more information on the run-time tasking implementation.

## Debugging Context

When the operating system gives the debugger control of a process due to a debugging event (e.g., hitting a breakpoint, receiving a signal, etc.), all clones in the process are stopped by the debugger. If multiple task processes stop at once, NightView writes messages for each one, and makes one the current context.

The NightView Data Window allows you to display variables, registers, and tasks.

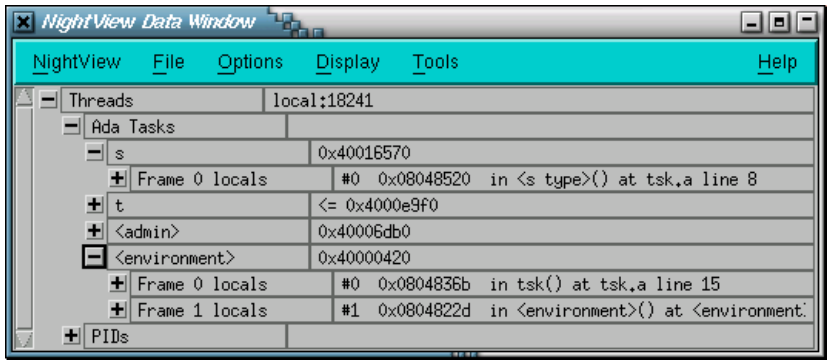



**Figure 2-1. NightView Data Window**

The list of active tasks includes a description of the name of the task, its Task Control Block (TCB) address, and, optionally, its stack walkback.

You can switch to the context of any task in the list by right-clicking on the "+" or "-" sign to the left of the task name and choosing Select Frame.

The NightView source display will change to the topmost stack frame for the newly selected task. All references to process state are now in the context of the selected task.

Alternatively, you can use the **select-context** command. (The **select-context** command may be abbreviated as **sel**.)

The formats of the command include:

```
select-context task=expr
select-context pid=pid
select-context default
```

The first form of the command sets the current context to that associated with the task as specified by *expr*. The second form of the command sets the context to that associated with the process as specified by *pid*. The third form of the command sets the context to that which was presented when the debugger was last given control of the process.

The first form of the command is usually appropriate for tasking programs. In this form, *expr* should be an Ada expression that identifies a task object (task variable, task body proper, or an expression which evaluates to a task type). Alternatively, *expr* can be an integer literal that identifies the address of the Task Control Block (TCB) for the task of interest. The TCB address of a task is available through:

- MAXAda `runtime_configuration.current_task` subprogram call
- MAXAda tool **a.monitor**
- the description supplied for each task in the Data Window when threads are selected to be displayed (by choosing the Threads menu item from the Display menu)

The MAXAda-supplied tool, **a.monitor**, provides dynamic snapshots of all the tasks associated within a single process. Such snapshots include the task ID of each active task. The task ID is actually the clone's PID. By running **a.monitor** in parallel with NightView, you can always determine the ${*taskid*} of any active task in the process as well as its Ada "task status".

Once the debugging context has been changed, you can peruse stack frames, view variables, set breakpoints, etc.

For a more complete description, see **select-context** in the *NightView User's Guide*.

## Exception Handling and Interception

Exceptions raised in your program can be intercepted by the debugger before being delivered to the process. The NightView **handle/exception** command is effective in doing this.

Note that the exception will be delivered to the process regardless of whether it is intercepted by the debugger. Interception simply delays delivery until the process is resumed again.

The **handle/exception** command allows for intercepting all exceptions or specific exceptions you specified. For more information, see **handle** and **info exception** in the *NightView User's Guide*.

## Generics

NightView does not currently fully support debugging generic instantiations. For example, you cannot set a breakpoint on a routine by specifying the name of a generic subprogram or the file/line-number of a file that contains a generic body. Similarly, you cannot view variables that are associated with portions of shared generic instantiations.

The debugger does support setting breakpoints on the instantiated unit name of a generic.

Once a breakpoint has been reached inside a generic body, single stepping, advancing, and setting breakpoints via line number function correctly.

## General NightView Operational Hints

The following is not meant as complete description of NightView commands, rather as some helpful hints about useful commands for Ada programmers. Consult the *NightView User's Guide* for more information.

### Listing Source, Packages, and Subprograms

In Ada, more so than in other languages, programmers tend to think of source code via its corresponding unit name (package, procedure, function) rather than by the name of the source file containing the source code.

To list the names of functions, subroutines, or Ada unit names recorded in the debug tables, use the following NightView command:

**`info functions`** *regexp*

To list the source for a specific unit, simply supply the unit name.

**Examples:**

| | |
|---|---|
| **`list text_io'spec`** | Lists the specification of `text_io` |
| **`list system`** | Lists the body of `system` |
| **`list calendar.local_time.clock`** | Lists the body of the subprogram `clock` inside the package body `local_time` inside the package body `calendar` |
| **`list 45:my_unit_name`** | Lists line 45 of the source file containing the unit `my_unit_name` |

To list a file by file name, the file name should be a quoted string. Otherwise, the string may be ambiguous. For example:

```
list "myfile.a"
```

## Disassembly

Users can select the Disassembly Preferred menu item from the View menu in the NightView Debug Window to aid in assembly-level debugging. In disassembly-preferred mode, the NightView debug source display shows disassembly when possible. If the debugger tries to show a file that does not have any corresponding instructions, then it will show the file with no disassembly.

NightView also offers a mixed-preferred mode in which the debugger shows a line of source followed by the instructions that correspond to that line. Source lines that do not produce code are not shown. Only one source line is shown for each group of instructions, so statements that span lines are only partially shown. This mode is selected by choosing the Mixed Preferred menu item from the View menu in the NightView Debug Window.

## Interest Threshold

By default, NightView considers locations in the program which don't have debug information associated with them to be uninteresting.

As such, it attempts to automatically back up to a stack frame which is associated with a routine with debug information.

You can change this behavior by setting the interest threshold to zero:

```
interest threshold 0
```

See information about the **interest** comand in the "Command-Line Interface" chapter of the *NightView User's Guide* (0890395) for more information.

## Expression Evaluation Syntax

NightView automatically parses commands and expressions you supply based on the effective language setting. The **set-language** command allows you to let NightView automatically determine the language based on the current debugging context (**set-language auto**) or allows you to specify a constant value. To specify Ada as the default language, use the command:

```
set-language ada
```

For example, the following command has two drastically different effects based on the effective language setting:

**print** *variable* **= 3**

If the effective language is Ada, the preceding command prints `TRUE` or `FALSE`, depending on whether *variable* has the value `3` or not. However, if the effective language is C, the preceding command assigns the value `3` to *variable*!

In many other cases, if you attempt a command using Ada syntax and the effective language setting is not Ada, the debugger does not recognize the syntax and issues an error.

For example, if the effective language setting is C, the following commands have the following effect:

| | |
|---|---|
| **print** *variable'address* | Illegal syntax |
| **print** *package_name.variable_name* | *Package_name* is not a record |

Ensure that your language setting is as you desire. You can set the language to **auto** in which case the debugger automatically changes the language to that of the routine where the process is stopped.

# M
# Implementation-Defined Characteristics


RM Chapter 1: General . . . . . M-2
RM 1.1.2 Structure . . . . . M-2
RM 1.1.3 Conformity of an Implementation with the Standard . . . . . M-2
RM 1.1.4 Method of Description and Syntax Notation . . . . . M-4
RM Chapter 2: Lexical Elements . . . . . M-5
RM 2.1 Character Set . . . . . M-5
RM 2.2 Lexical Elements, Separators, and Delimiters . . . . . M-5
RM 2.8 Pragmas . . . . . M-6
RM Chapter 3: Declarations and Types . . . . . M-9
RM 3.5 Scalar Types . . . . . M-9
RM 3.5.2 Character Types . . . . . M-9
RM 3.5.4 Integer Types . . . . . M-9
RM 3.5.5 Operations of Discrete Types . . . . . M-10
RM 3.5.6 Real Types . . . . . M-10
RM 3.5.7 Floating Point Types . . . . . M-11
RM 3.5.9 Fixed Point Types . . . . . M-11
RM 3.6.2 Operations of Array Types . . . . . M-12
RM 3.9 Tagged Types and Type Extensions . . . . . M-12
RM Chapter 4: Names and Expressions . . . . . M-13
RM 4.1.4 Attributes . . . . . M-13
RM 4.3.1 Record Aggregates . . . . . M-16
RM Chapter 5: Statements . . . . . M-17
RM Chapter 6: Subprograms . . . . . M-18
RM Chapter 7: Packages . . . . . M-19
RM Chapter 8: Visibility Rules . . . . . M-20
RM Chapter 9: Tasks and Synchronizations . . . . . M-21
RM 9.6 Delay Statements, Duration, and Time . . . . . M-21
RM 9.10 Shared Variables . . . . . M-22
RM Chapter 10: Program Structure and Compilation Issues . . . . . M-23
RM 10.1 Separate Compilation . . . . . M-23
RM 10.1.4 The Compilation Process . . . . . M-23
RM 10.1.5 Pragmas and Program Units . . . . . M-24
RM 10.2 Program Execution . . . . . M-25
RM 10.2.1 Elaboration Control . . . . . M-29
RM Chapter 11: Exceptions . . . . . M-30
RM 11.4.1 The Package Exceptions . . . . . M-30
RM 11.5 Suppressing Checks . . . . . M-31
RM Chapter 12: Generic Units . . . . . M-32
RM Chapter 13: Representation Issues . . . . . M-33
RM 13.1 Representation Items . . . . . M-33
RM 13.2 Pragma Pack . . . . . M-34
RM 13.3 Representation Attributes . . . . . M-35
Address Attributes . . . . . M-35
Alignment Attributes . . . . . M-36
Size Attributes for Objects . . . . . M-39
Size Attributes for Subtypes . . . . . M-40
Component_Size Attributes . . . . . M-42

External_Tag Attributes . . . . . M-43
RM 13.4 Enumeration Representation Clauses . . . . . M-43
RM 13.5.1 Record Representation Clauses . . . . . M-44
RM 13.5.2 Storage Place Attributes . . . . . M-46
RM 13.5.3 Bit Ordering . . . . . M-46
RM 13.7 The Package System . . . . . M-47
RM 13.7.1 The Package System.Storage_Elements . . . . . M-47
RM 13.8 Machine Code Insertions . . . . . M-48
Pentium . . . . . M-48
Pentium Instruction Set . . . . . M-49
Pentium Register Set . . . . . M-54
Pentium Address Modes . . . . . M-54
Pentium Machine Code Example . . . . . M-55
RM 13.9 Unchecked Type Conversions . . . . . M-55
RM 13.11 Storage Management . . . . . M-59
RM 13.11.2 Unchecked Storage Deallocation . . . . . M-62
RM 13.12 Pragma Restrictions . . . . . M-62
RM 13.13.2 Stream-Oriented Attributes . . . . . M-63

RM Annex A: Predefined Language Environment . . . . . M-64
RM A.1 The Package Standard . . . . . M-64
RM A.3.2 The Package Characters.Handling . . . . . M-65
RM A.4.4 Bounded-Length String Handling . . . . . M-65
RM A.5.1 Elementary Functions . . . . . M-65
RM A.5.2 Random Number Generation. . . . . M-66
RM A.5.3 Attributes of Floating Point Types . . . . . M-68
RM A.7 External Files and File Objects . . . . . M-68
RM A.9 The Generic Package Storage_IO . . . . . M-71
RM A.10 Text Input-Output . . . . . M-71
RM A.10.7 Input-Output of Characters and Strings. . . . . M-72
RM A.10.9 Input-Output for Real Types . . . . . M-72
RM A.13 Exceptions in Input-Output . . . . . M-72
RM A.15 The Package Command_Line . . . . . M-73
RM Annex B: Interface to Other Languages. . . . . M-74
RM B.1 Interfacing Pragmas . . . . . M-74
RM B.2 The Package Interfaces. . . . . M-78
RM B.3 Interfacing with C. . . . . M-79
RM B.4 Interfacing with COBOL . . . . . M-81
RM B.5 Interfacing with Fortran . . . . . M-81
RM Annex C: Systems Programming . . . . . M-83
RM C.1 Access to Machine Operations . . . . . M-83
RM C.3 The Package Interrupts . . . . . M-84
RM C.3.1 Protected Procedure Handlers . . . . . M-87
RM C.3.2 The Package Interrupts . . . . . M-87
RM C.4 Preelaboration Requirements . . . . . M-87
RM C.5 Pragma Discard_Names . . . . . M-88
RM C.7.1 The Package Task_Identification. . . . . M-88
RM C.7.2 The Package Task_Attributes . . . . . M-89
RM Annex D: Real-Time Systems . . . . . M-91
RM D.1 Task Priorities. . . . . M-91
RM D.2.1 The Task Dispatching Model . . . . . M-91
RM D.2.2 The Standard Task Dispatching Policy . . . . . M-91
RM D.3 Priority Ceiling Locking. . . . . M-92
RM D.4 Entry Queuing Policies. . . . . M-93
RM D.6 Preemptive Abort . . . . . M-93
RM D.7 Tasking Restrictions . . . . . M-94
RM D.8 Monotonic Time . . . . . M-95
RM D.9 Delay Accuracy . . . . . M-97
RM D.12 Other Optimizations and Determinism Rules . . . . . M-97
RM Annex G: Numerics . . . . . M-98
RM G.1 Complex Arithmetic . . . . . M-98
RM G.1.1 Complex Types . . . . . M-98
RM G.1.2 Complex Elementary Functions. . . . . M-99
RM G.2 Numeric Performance Requirements . . . . . M-99
RM G.2.1 Model of Floating Point Arithmetic. . . . . M-100
RM G.2.3 Model of Fixed Point Arithmetic. . . . . M-100
RM G.2.4 Accuracy Requirements for the Elementary Functions . . . . . M-100
RM G.2.6 Accuracy Requirements for Complex Arithmetic . . . . . M-101
RM Annex J: Obsolescent Features. . . . . M-102
RM J.7.1 Interrupt Entries . . . . . M-102
RM Annex K: Language-Defined Attributes . . . . . M-103

RM Annex L: Pragmas . . . . . M-104
Pragma ALL_CALLS_REMOTE - (not yet supported) . . . . . M-106
Pragma ASSIGNMENT . . . . . M-106
Pragma ASYNCHRONOUS - (not yet supported) . . . . . M-106
Pragma ATOMIC . . . . . M-106
Pragma ATOMIC_COMPONENTS . . . . . M-107
Pragma ATTACH_HANDLER . . . . . M-107
Pragma CONTROLLED . . . . . M-107
Pragma CONVENTION. . . . . M-108
Pragma DATA_RECORD - (obsolete) . . . . . M-109
Pragma DEBUG . . . . . M-109
Pragma DEPRECATED_FEATURE . . . . . M-110
Pragma DISCARD_NAMES . . . . . M-110
Pragma DONT_ELABORATE . . . . . M-110
Pragma ELABORATE . . . . . M-111
Pragma ELABORATE_ALL . . . . . M-111
Pragma ELABORATE_BODY . . . . . M-111
Pragma EXPORT . . . . . M-111
Pragma EXTERNAL_NAME - (obsolete). . . . . M-112
Pragma FAST_INTERRUPT_TASK . . . . . M-113
Pragma GROUP_CPU_BIAS . . . . . M-113
Pragma GROUP_PRIORITY. . . . . M-113
Pragma GROUP_SERVERS . . . . . M-114
Pragma IMPLICIT_CODE. . . . . M-114
Pragma IMPORT . . . . . M-114
Pragma INLINE . . . . . M-115
Pragma INSPECTION_POINT - (not yet supported) . . . . . M-116
Pragma INTERESTING. . . . . M-117
Pragma INTERFACE - (obsolete) . . . . . M-117
Pragma INTERFACE_NAME - (obsolete) . . . . . M-117
Pragma INTERFACE_OBJECT - (obsolete) . . . . . M-118
Pragma INTERFACE_SHARED - (obsolete) . . . . . M-118
Pragma INTERRUPT_HANDLER . . . . . M-118
Pragma INTERRUPT_PRIORITY. . . . . M-118
Pragma LINK_OPTION - (obsolete) . . . . . M-119
Pragma LINKER_OPTIONS . . . . . M-119
Pragma LIST . . . . . M-119
Pragma LOCKING_POLICY . . . . . M-120
Pragma MAP_FILE . . . . . M-120
Pragma MEMORY_POOL. . . . . M-120
Pragma NORMALIZE_SCALARS - (not yet supported) . . . . . M-121
Pragma OPT_FLAGS. . . . . M-121
Pragma OPT_LEVEL. . . . . M-122
Pragma OPTIMIZE . . . . . M-122
Pragma PACK. . . . . M-123
Pragma PAGE. . . . . M-123
Pragma PASSIVE_TASK - (obsolete) . . . . . M-123
Pragma POOL_CACHE_MODE . . . . . M-124
Pragma POOL_LOCK_STATE . . . . . M-124
Pragma POOL_PAD. . . . . M-124
Pragma POOL_SIZE . . . . . M-124
Pragma PREELABORATE . . . . . M-125
Pragma PRIORITY . . . . . M-125
Pragma PROTECTED_PRIORITY . . . . . M-125

Pragma PURE . . . . . M-127
Pragma QUEUING_POLICY . . . . . M-127
Pragma REMOTE_CALL_INTERFACE - (not yet supported) . . . . . M-127
Pragma REMOTE_TYPES - (not yet supported) . . . . . M-127
Pragma RESTRICTIONS . . . . . M-128
Pragma RETURN_CONVENTION . . . . . M-128
Pragma REVIEWABLE - (not yet supported) . . . . . M-129
Pragma RUNTIME_DIAGNOSTICS . . . . . M-129
Pragma SERVER_CACHE_SIZE . . . . . M-129
Pragma SHARE_BODY . . . . . M-129
Pragma SHARE_MODE . . . . . M-130
Pragma SHARED - (obsolete) . . . . . M-131
Pragma SHARED_PACKAGE . . . . . M-131
Pragma SHARED_PASSIVE - (not yet supported) . . . . . M-131
Pragma SPECIAL_FEATURE . . . . . M-131
Pragma STORAGE_SIZE . . . . . M-132
Pragma SUPPRESS . . . . . M-132
Pragma SUPPRESS_ALL . . . . . M-133
Pragma TASK_CPU_BIAS . . . . . M-133
Pragma TASK_DISPATCHING_POLICY . . . . . M-133
Pragma TASK_HANDLER . . . . . M-134
Pragma TASK_PRIORITY . . . . . M-134
Pragma TASK_QUANTUM . . . . . M-134
Pragma TASK_WEIGHT . . . . . M-135
Pragma TDESC . . . . . M-135
Pragma TRAMPOLINE . . . . . M-135
Pragma VOLATILE . . . . . M-135
Pragma VOLATILE_COMPONENTS . . . . . M-136

# M
# Implementation-Defined Characteristics

This appendix describes amendments to the Ada 95 Reference Manual for the Ada Programming Language, ANSI/ISO/IEC-8652:1995.

The appendix is organized parallel to the RM, with one section for each RM chapter. Headings specify RM subsections and paragraphs where appropriate.

In the following text, syntactic categories, such as *range_constraint* and *unit*, are italicized.

# RM Chapter 1: General

## RM 1.1.2 Structure

Implementation Advice

**1.1.2(37)** **Whether or not each recommendation given in Implementation Advice is followed**

Each recommendation given in Implementation Advice is listed within this appendix parallel to its appropriate section in the RM. Each of these sections contains the heading IMPLEMENTATION ADVICE under which is listed each specific paragraph that appears in the RM and whether or not MAXAda has followed the given advice.

## RM 1.1.3 Conformity of an Implementation with the Standard

Implementation Requirements

**1.1.3(3)** **Capacity limitations of the implementation**

- Source files are limited to 4,294,967,295 lines

  Excessively long subprogram bodies or declarative areas requiring generation of code can result in long compiler times or even exhaustion of internal code generator limits. A practical limit of 100,000 lines per source file is recommended.

- Source lines are limited to 500 characters

- Rooted names are limited in length such that no symbol name may exceed 800 characters in length. Symbols name lengths are usually similar in length to rooted names, but will include additional characters. They always include an '`A_`' prefix. If the rooted name contains any overloadable entities other than a child unit name, then the name will also contain overload resolution substrings of the form '`llllllSccc`' or '`llllllBccc`' for each such overloadable entity. The sequence '`llllll`' is the line number and the sequence '`ccc`' is the column number of the declaration of the overloadable entity. The line number and column number will not contain leading zeros and so will be variable length. They will usually be much shorter than these worst-case examples.

- The number of individual declarative items associated with a single subprogram body, entry body, or task body (including declarations within nested packages) is limited by the addressing modes used for the target machine architecture.

  The actual limitation is dependent on the ordering and size of the individual declarative items in each body; the limitation could be as small as ~3,000 items or as large as ~16,000 items.

- Static data is limited only by any virtual memory limitations enforced by the operating system, and by limitations of the system linker.

- Subtypes and objects are limited to 2,147,483,520 (256 * 1024 * 1024 * 8 - 128) bits in size. (records, arrays, protected types, and objects)

**1.1.3(6)** **Variations from the standard that are impractical to avoid given the implementation's execution environment**

MAXAda contains no variations from the standard that are impractical to avoid given the implementation's execution environment.

**1.1.3(10)** **Which code_statements cause external interactions**

Any code statement containing Pentium instructions which affect memory or the machine state might cause external interactions.

See the *IA-32 Intel Architecture Software Developer's Manual Volume 2* for details.

## Documentation Requirements

**1.1.3(18)** **Certain aspects of the semantics are defined to be either implementation defined or unspecified. In such cases, the set of possible effects is specified, and the implementation may choose any effect in the set. Implementations shall document their behavior in implementation-defined situations, but documentation is not required for unspecified situations. The implementation-defined characteristics are summarized in Annex M.**

This appendix documents the implementation-defined characteristics addressed in the Ada 95 Reference Manual as summarized in Annex M as well as items referenced under the headings:

- Implementation Advice
- Implementation Permissions
- Documentation Requirements

Such behavior is documented in subsequent sections of this appendix corresponding to the appropriate RM section. Further discussion of a particular section can be found under its corresponding heading NOTES.

**1.1.3(19)** **The implementation may choose to document implementation-defined behavior either by documenting what happens in general, or by providing some mechanism for the user to determine what happens in a particular case.**

This appendix documents all implementation-defined characteristics addressed in the Ada 95 Reference Manual, as noted above.

## Implementation Advice

**1.1.3(20)** **If an implementation detects the use of an unsupported Specialized Needs Annex feature at run time, it should raise Program_Error if feasible.**

MAXAda follows this advice.

**1.1.3(21)** **If an implementation wishes to provide implementation-defined extensions to the functionality of a language-defined library unit, it should normally do so by adding children to the library unit.**

MAXAda follows this advice.

## RM 1.1.4 Method of Description and Syntax Notation

Implementation Advice

**1.1.4(12)** **If an implementation detects a bounded error or erroneous execution, it should raise Program_Error.**

MAXAda follows this advice.

# RM Chapter 2: Lexical Elements

## RM 2.1 Character Set

### Static Semantics

**2.1(4)** **The coded representation for the text of an Ada program.**

MAXAda provides the full *graphic_character* textual representation for programs.

Any character in row 00 of the ISO 10646-1 BMP except ESC (decimal value 27) is represented as a single byte whose value is the character's cell-octet within row 00 of the BMP.

Any other character in the ISO 10646-1 BMP is represented as the character ESC followed by two *extended_digit* characters which encode the hexadecimal number representing the character's row-octet within the BMP and two more *extended_digit* characters which encode the hexadecimal number representing the character's cell-octet within that row.

**2.1(14)** **The control functions allowed in comments**

The following control characters are allowed in comments:

| Control character | Decimal value |
|---|---|
| HT (horizonal tab) | 9 |
| LF (line feed) | 10 |
| VT (vertical tab) | 11 |
| FF (form feed) | 12 |
| CR (carriage return) | 13 |

## RM 2.2 Lexical Elements, Separators, and Delimiters

### Static Semantics

**2.2(2)** **The representation for an end of line.**

Each line is terminated by a line feed (`LF`) or vertical tab (`VT`) character.

### Implementation Requirements

**2.2(15)** **Maximum supported line length and lexical element length.**

Source lines may contain up to 500 characters, including the terminator. All variable-length Ada elements, such as identifiers and literals, may extend up to the full 499-character limit. The fully expanded symbol name is limited to 800 characters.

# RM 2.8 Pragmas

## Implementation Permissions

**2.8(14) Implementation-defined pragmas.**

Implementation-defined pragmas are listed along with Language-Defined Pragmas in "RM Annex L: Pragmas" on page M-104.

Specifically, they are:


| | |
|---|---|
| Pragma ASSIGNMENT | page M-106 |
| Pragma DATA_RECORD - (obsolete) | page M-109 |
| Pragma DEBUG | page M-109 |
| Pragma DEPRECATED_FEATURE | page M-110 |
| Pragma DONT_ELABORATE | page M-110 |
| Pragma EXTERNAL_NAME - (obsolete) | page M-112 |
| Pragma FAST_INTERRUPT_TASK | page M-113 |
| Pragma GROUP_CPU_BIAS | page M-113 |
| Pragma GROUP_PRIORITY | page M-113 |
| Pragma GROUP_SERVERS | page M-114 |
| Pragma IMPLICIT_CODE | page M-114 |
| Pragma INTERESTING | page M-117 |
| Pragma INTERFACE - (obsolete) | page M-117 |
| Pragma INTERFACE_NAME - (obsolete) | page M-117 |
| Pragma INTERFACE_OBJECT - (obsolete) | page M-118 |
| Pragma INTERFACE_SHARED - (obsolete) | page M-118 |
| Pragma LINK_OPTION - (obsolete) | page M-119 |
| Pragma MAP_FILE | page M-120 |
| Pragma MEMORY_POOL | page M-120 |
| Pragma OPT_FLAGS | page M-121 |
| Pragma OPT_LEVEL | page M-122 |
| Pragma PASSIVE_TASK - (obsolete) | page M-123 |
| Pragma POOL_CACHE_MODE | page M-124 |
| Pragma POOL_LOCK_STATE | page M-124 |
| Pragma POOL_PAD | page M-124 |
| Pragma POOL_SIZE | page M-124 |
| Pragma PROTECTED_PRIORITY | page M-125 |
| Pragma RETURN_CONVENTION | page M-128 |
| Pragma RUNTIME_DIAGNOSTICS | page M-129 |
| Pragma SERVER_CACHE_SIZE | page M-129 |
| Pragma SHARE_BODY | page M-129 |
| Pragma SHARE_MODE | page M-130 |
| Pragma SHARED - (obsolete) | page M-131 |

| | |
|---|---|
| Pragma SHARED_PACKAGE | page M-131 |
| Pragma SPECIAL_FEATURE | page M-131 |
| Pragma SUPPRESS_ALL | page M-133 |
| Pragma TASK_CPU_BIAS | page M-133 |
| Pragma TASK_HANDLER | page M-134 |
| Pragma TASK_PRIORITY | page M-134 |
| Pragma TASK_QUANTUM | page M-134 |
| Pragma TASK_WEIGHT | page M-135 |
| Pragma TDESC | page M-135 |
| Pragma TRAMPOLINE | page M-135 |


## Implementation Advice

**2.8(16)** **Normally, implementation-defined pragmas should have no semantic effect for error-free programs; that is, if the implementation-defined pragmas are removed from a working program, the program should still be legal, and should still have the same semantics.**

The following implementation-defined pragmas can have a semantic effect on error-free programs; their removal from a working program could have a semantic effect.

- `DONT_ELABORATE`
- `GROUP_PRIORITY`
- `IMPLICIT_CODE`
- `MEMORY_POOL`
- `OPT_FLAGS`
- `OPT_LEVEL`
- `POOL_SIZE`
- `PROTECTED_PRIORITY`
- `RUNTIME_DIAGNOSTICS`
- `SHARED_PACKAGE`
- `SPECIAL_FEATURE`
- `SUPPRESS_ALL`
- `TASK_HANDLER`
- `TASK_PRIORITY`
- `TASK_WEIGHT`
- `TDESC`

**2.8(17)** **Normally, an implementation should not define pragmas that can make an illegal program legal, except as follows:**

**2.8(18)** **A pragma used to complete a declaration, such as a pragma Import;**

MAXAda follows this advice.

**2.8(19)** **A pragma used to configure the environment by adding, removing, or replacing library_items.**

MAXAda follows this advice.

## Static Semantics

**2.8(27)** **Effect of pragma Optimize.**

The implementation-dependent pragma `OPTIMIZE` is recognized by the implementation but does not have an effect in this release.

# RM Chapter 3: Declarations and Types

## RM 3.5 Scalar Types

Dynamic Semantics

**3.5(37)** **The sequence of characters of the value returned by S'Image when some of the graphic characters of S'Wide_image are not defined in Character**

For `S'Image`, when some of the characters in `S'Wide_Image` are not defined in `Character`, the sequence of characters returned is the same as that returned by `S'Wide_Image` except that each character not defined in `Character` is replaced with a space character.

## RM 3.5.2 Character Types

Implementation Advice

**3.5.2(5)** **If an implementation supports a mode with alternative interpretations for Character and Wide_Character, the set of graphic characters of Character should nevertheless remain a proper subset of the set of graphic characters of Wide_Character. Any character set ``localizations'' should be reflected in the results of the subprograms defined in the language-defined package Characters.Handling (see A.3) available in such a mode. In a mode with an alternative interpretation of Character, the implementation should also support a corresponding change in what is a legal identifier_letter.**

MAXAda does not support a mode with alternative interpretations for `Character` and `Wide_Character`; therefore, this advice is not relevant.

## RM 3.5.4 Integer Types

Implementation Permissions

**3.5.4(25)** **The predefined integer types declared in Standard**

Four predefined integer types are declared in Standard:

```
type integer is range -2_147_483_648 ..
2_147_483_647;
type long_integer is range -2_147_483_648 ..
2_147_483_647;
type short_integer is range -32768 .. 32767;
type tiny_integer is range -128 .. 127;
```

**3.5.4(26)** **Any nonstandard integer types and the operators defined for them**

MAXAda does not define any nonstandard integer types.

Implementation Advice

**3.5.4(28)** **An implementation should support Long_Integer in addition to Integer if the target machine supports 32-bit (or longer) arithmetic. No other named integer subtypes are recommended for package Standard. Instead, appropriate named integer subtypes should be provided in the library package Interfaces (see B.2).**

MAXAda does not follow this advice.

MAXAda provides the following predefined integer types in the package `Standard`:

```
tiny_integer
short_integer
long_integer
```

Removal of these types from the package Standard would put an undue burden on users which were familiar with legacy Ada products from Concurrent and other companies which provide these definitions.

Users are advised to remove uses of such types and utilize those provided in the package `Interfaces`.

**3.5.4(29)** **An implementation for a two's complement machine should support modular types with a binary modulus up to System.Max_Int*2+2. An implementation should support a nonbinary modulus up to Integer'Last.**

MAXAda follows this advice.

## RM 3.5.5 Operations of Discrete Types

Implementation Advice

**3.5.5(8)** **For the evaluation of a call on S'Pos for an enumeration subtype, if the value of the operand does not correspond to the internal code for any enumeration literal of its type (perhaps due to an uninitialized variable), then the implementation should raise Program_Error. This is particularly important for enumeration types with noncontiguous internal codes specified by an enumeration_representation_clause.**

MAXAda follows this advice.

## RM 3.5.6 Real Types

Implementation Permissions

**3.5.6(8)** **Any nonstandard real types and the operators defined for them**

There are not any nonstandard real types defined in MAXAda.

# RM 3.5.7 Floating Point Types

## Legality Rules

**3.5.7(7)** **What combinations of requested decimal precision and range are supported for floating point types**

MAXAda provides two floating-point types in addition to `universal_real`: `FLOAT` and `LONG_FLOAT`.

| Type | Precision | Range |
|---|---|---|
| `FLOAT` | 6 decimal digits | -3.40282e+38 .. 3.40282e+38 |
| `LONG_FLOAT` | 15 decimal digits | -1.79769313486232e+308 .. 1.79769313486232e+308 |

## Implementation Permissions

**3.5.7(16)** **The predefined floating point types declared in Standard**

Two predefined floating point types are declared in Standard:

```
type float is digits 6 ;
type long_float is digits 15 ;
```

## Implementation Advice

**3.5.7(17)** **An implementation should support Long_Float in addition to Float if the target machine supports 11 or more digits of precision. No other named floating point subtypes are recommended for package Standard. Instead, appropriate named floating point subtypes should be provided in the library package Interfaces (see B.2).**

MAXAda follows this advice.

`FLOAT` is implemented in MAXAda with 6 digits of precision.

`LONG_FLOAT` is implemented in MAXAda with 15 digits of precision.

There are no other floating point types defined in the package Standard.

# RM 3.5.9 Fixed Point Types

## Legality Rules

**3.5.9(8)** **The small of an ordinary fixed point type**

MAXAda defines the *small* of an ordinary fixed point type to be the largest power of two less than or equal to the delta.

**3.5.9(10)** **What combinations of small, range, and digits are supported for fixed point types**

MAXAda defines the allowable values for a *small* to be between 2**-26 and 2**1024.

MAXAda defines the allowable *range* for a given *small* to be:

(-2.0**31)**small* .. ((2.0**31)+1)**small*

MAXAda does not support decimal types; therefore, *digits* is not supported.

## RM 3.6.2 Operations of Array Types

Implementation Advice

**3.6.2(11)** **An implementation should normally represent multidimensional arrays in row-major order, consistent with the notation used for multidimensional array aggregates (see 4.3.3). However, if a pragma Convention(Fortran, ...) applies to a multidimensional array type, then column-major order should be used instead (see B.5, ``Interfacing with Fortran'').**

MAXAda follows this advice.

## RM 3.9 Tagged Types and Type Extensions

Static Semantics

**3.9(10)** **The result of Tags.Expanded_Name for types declared within an unnamed block_statement**

The result is the expanded name of the first subtype of the type of the prefix subtype with an automatically generated anonymous block id inserted at the place of the unnamed block statement.

# RM Chapter 4: Names and Expressions

## RM 4.1.4 Attributes

Implementation Permissions

**4.1.4(12)** **Implementation-defined attributes**

- MAXAda has defined the following attributes for use in conjunction with the implementation-defined pragma `SHARED_PACKAGE` (see "Pragma SHARED_PACKAGE" on page M-131):

```
P'Key
P'SHM_ID
P'Lock
P'Unlock
```

  where the prefix `P` denotes a package marked with pragma `SHARED_PACKAGE`.

  The `'Key` attribute is an overloaded function without parameters that returns the key used to identify the system shared segment associated with the package. One specification of the function returns the predefined type `String` and returns a value specifying the file name used in the key translation (**`ftok(3C)`**). If an integer literal key was specified in the pragma `SHARED_PACKAGE` parameters, this function returns a null string. The other specification of the function returns the predefined type `universal_integer`, and returns a value specifying the translated integer key. The latter form of the function raises the predefined exception `Program_Error` if the shared package body has not yet been elaborated.

  The `'SHM_ID` attribute is a function without parameters that returns the shared memory segment identifier for the system shared memory segment associated with the shared package `P`. This identifier corresponds to the identifier that is returned by the shared memory service **`shmget(2)`** upon creation of the shared package.

  The `'SHM_ID` attribute raises `Program_Error` if the call to **`shmget`** failed when the segment associated with the shared package `P` was created.

  The `'Lock` and `'Unlock` attributes are procedures without parameters that manipulate the "state" of a shared package. MAXAda defines all shared packages to have two states: `Lock`ed and `Unlock`ed. Upon return from the `'Lock` procedure, the state of the package will be `Lock`ed. If upon invocation, `'Lock` finds the state already `Lock`ed, it waits until it becomes `Unlock`ed before altering the state and returning. `'Unlock` sets the state of the package to `Unlock`ed and then returns. At the point of unlocking the package, if another process waiting in the `'Lock` procedure has a more favorable operating system priority, the system immediately schedules its execution.

  Note that if `'Lock` is waiting, it may be interrupted by the MAXAda run-time system's time slice for tasks which may cause another task within the process to become active. Eventually, MAXAda will again transfer control to the `'Lock` procedure in the original task, and it will continue waiting or return to the task.

The state of the package is meaningful only to the `'Lock` and `'Unlock` attribute procedures that set and query the state. A `Lock`ed state ***does not prevent concurrent access*** to objects in the shared package. These attributes provide indivisible operations only for the setting and testing of implicit semaphores that could be used to control access to shared package objects.

**CAUTION**

The current shared memory implementation does not allow the use of the `'Lock` and `'Unlock` attributes with a `SHM_RDONLY` shared memory segment or a shared package marked with the `no_bsem` parameter.

- MAXAda has defined the following attribute for use in machine code insertions:

  ```
  X'Ref
  ```

  where X denotes an object or label.

  See "RM 13.8 Machine Code Insertions" on page M-48 for details about this attribute.

- MAXAda has defined the following attribute:

  ```
  X'Addr
  ```

  For a prefix that denotes an object, program unit, or label, it behaves exactly as does the `'Address` attribute defined in RM 13.3(10-11). For a prefix that denotes an exception, it denotes the first of the storage elements allocated to `X`. The value is equivalent to an unchecked conversion to `System.Address` of `X'Identity`, but without the semantic dependence on `Ada.Exceptions`.

  `Addr` may not be specified via an `attribute_definition_clause`.

- MAXAda has defined the following attributes:

  ```
  P'Unrestricted_Access
  X'Unrestricted_Access
  ```

  For a prefix `P` that denotes a subprogram, all rules and semantics that apply to `P'Access` apply also to `P'Unrestricted_Access`, except that it is as if `P` were declared immediately within a library package. In other words, accessibility checks are not performed, and it is possible to pass a more-nested subprogram as a parameter to a less-nested subprogram. An attempt to call a dereferenced more-nested subprogram that is no longer in scope is erroneous, and it is the programmer's responsibility to ensure that this does not happen.

  For a prefix `X` that denotes an aliased view of an object, all rules and semantics that apply to `X'Unchecked_Access` apply also to `X'Unrestricted_Access`.

- MAXAda has defined the following attribute:

  ```
  S'Has_Tag
  ```

  For a prefix `S` that is a formal subtype, it yields `True` if the actual subtype corresponding to `S` is a tagged record type or a derivation of a type whose private view is non-tagged but whose full view is tagged; otherwise it yields `False`. The value of this attribute is of the predefined type `Boolean`.

- MAXAda has defined the following attribute:

  ```
  S'Tagged
  ```

  For a prefix `S` that is a formal subtype, it yields `True` if the actual subtype corresponding to `S` is a tagged record type; otherwise it yields `False`. The value of this attribute is of the predefined type `Boolean`.

- MAXAda has defined the following attribute:

  ```
  S'Has_Discriminants
  ```

  For a prefix `S` that is a formal subtype, it yields `True` if the actual subtype corresponding to `S` has discriminants; otherwise it yields `False`. The value of this attribute is of the predefined type `Boolean`.

- MAXAda has defined the following attribute:

  ```
  S'Part_Has_Tag
  ```

  For a prefix `S` that is a formal subtype, it yields `True` if the actual subtype corresponding to `S` is a composite type with any part which is a tagged record type or a derivation of a type whose private view is non-tagged but whose full view is tagged; otherwise it yields `False`. The value of this attribute is of the predefined type `Boolean`.

- MAXAda has defined the following attribute for future use:

  ```
  S'Internal_Tag
  ```

  where S denotes a subtype of a tagged type.

  Its meaning is currently undefined.

- MAXAda also supports the following attributes for backward compatibility with Ada 83. A warning will be issued when any of these attributes are used:

  ```
  Emax
  Epsilon
  Large
  Mantissa
  Safe_Emax
  Safe_Large
  Safe_Small
  ```

# RM 4.3.1 Record Aggregates

Dynamic Semantics

**4.3.1(19)** **For the evaluation of a record_component_association_list, any per-object constraints (see 3.8) for components specified in the association list are elaborated and any expressions are evaluated and converted to the subtype of the associated component. Any constraint elaborations and expression evaluations (and conversions) occur in an arbitrary order, except that the expression for a discriminant is evaluated (and converted) prior to the elaboration of any per-object constraint that depends on it, which in turn occurs prior to the evaluation and conversion of the expression for the component with the per-object constraint.**

In order to support efficient renaming of dynamic non-dependent subcomponents of record objects, the implementation will reorder subcomponents of record objects and aggregates in memory such that no data for any dynamic component whose subtype does not depend upon a discriminant ever follows data for a subcomponent whose subtype depends upon a discriminant. In order to also minimize the memory size and execution time involved in elaborating record objects and aggregates, the implementation will evaluate `record_component_association_lists` in the order in which the component data is physically laid out in memory.

The order of evaluation of expressions involved in a `record_component_association_list` may be different than the textual order of the expressions in the user's source code. Source code should not be written so as to implicitly depend upon the order of evaluation of the expressions in an aggregate or extension aggregate, or the order of evaluation of component initialization expressions for a default-initialized record object.

# RM Chapter 5: Statements

There are no MAXAda amendments to Chapter 5 of the RM.

# RM Chapter 6: Subprograms

There are no MAXAda amendments to Chapter 6 of the RM.

# RM Chapter 7: Packages

There are no MAXAda amendments to Chapter 7 of the RM.

# RM Chapter 8: Visibility Rules

There are no MAXAda amendments to Chapter 8 of the RM.

# RM Chapter 9: Tasks and Synchronizations

## RM 9.6 Delay Statements, Duration, and Time

### Legality Rules

**9.6(6)** **Any implementation-defined time types**

There are no implementation-defined time types in MAXAda. MAXAda supports `Ada.Calendar.Time` and `Ada.Real_Time.Time` only.

### Dynamic Semantics

**9.6(20)** **The time base associated with relative delays**

The time base associated with relative delay statements is the system's notion of GMT at the time the statement is executed. The time base is therefore independent of local time, inasmuch as the system's clock can accurately determine GMT. Thus, a relative delay statement with a value of 3600.0 (seconds) issued just before switching to or from daylight savings time would indeed delay for (approximately) 3600.0 physical seconds. However, the time base for `Ada.Calendar.Time` may be adjusted via user interaction or by system daemons which attempt to synchronize system time with an external source. The time base for `Ada.Real_Time.Time` is not adjusted after system boot time.

**9.6(23)** **The time base of the type Calendar.Time**

The time base of the type `Calendar.Time` is the system's notion of GMT. The time base is therefore independent of local time, inasmuch as the system's clock can accurately determine GMT. Therefore, values of type `Calendar.Time` are interchangeable across time zones (with other MAXAda Calendar.Time values). Local time affects only the splitting and forming of values of type `Calendar.Time` via the `Calendar.Split` and `Calendar.Time_Of` subprograms. The time base may be adjusted due to user interaction or by system daemons which attempt to synchronize system time with an external source.

**9.6(24)** **The timezone used for package Calendar operations**

The package `Calendar` gets the time zone information from the system configuration (See **`tzselect(1)`**) which can be overridden with the `TZ` environment variable (See **`environ(5)`**).

### Implementation Permissions

**9.6(28)** **An implementation may define additional time types (see D.8).**

MAXAda supports two time types: `Ada.Calendar.Time` and `Ada.Real_Time.Time`. MAXAda also supports subtypes of the supported time types, as well as types derived from a supported time type.

**9.6(29)** **Any limit on delay_until_statements of select_statements**

The expression in a `delay_until_statement` of a `select_statement` may not specify a time in excess of a value corresponding to approximately 19 Jan 2038.

Implementation Advice

**9.6(30)** **Whenever possible in an implementation, the value of Duration'Small should be no greater than 100 microseconds.**

MAXAda follows this advice.

The value of `duration'small` is 61.035 microseconds (or 2**-14 seconds).

**9.6(31)** **The time base for delay_relative_statements should be monotonic; it need not be the same time base as used for Calendar.Clock.**

MAXAda follows this advice.

The time base for `delay_relative` statements is the same time base as used for `Ada.Real_Time.Clock`, which is monotonic. As explained above, the time base for `Ada.Real_Time.Clock` and `Ada.Calendar.Clock` are initially the same (at system boot time) but `Ada.Calendar.Clock`'s base can be adjusted by user interaction or system daemons (and therefore may not be monotonic).

## RM 9.10 Shared Variables

Static Semantics

**9.10(1)** **Whether or not two nonoverlapping parts of a composite object are independently addressable, in the case where packing, record layout, or Component_Size is specified for the object**

These are not independently addressable in the MAXAda implementation.

# RM Chapter 10: Program Structure and Compilation Issues

## RM 10.1 Separate Compilation

**10.1(2)** **The representation for a compilation**

A compilation may be:

- The portion of an ASCII source file containing a single compilation unit together with any preceding configuration pragmas (even configuration pragmas that are not immediately preceding),

OR

- The entire source file for an ASCII source file that contains only configuration pragmas.

Implementation Permissions

**10.1(4)** **Any restrictions on compilations that contain multiple compilation_units**

Compilations may not contain multiple compilation units. This should not pose considerable hardship, however, because multiple compilation units within a particular source file may be compiled as distinct compilations. See "a.build" on page 4-3 for more information.

## RM 10.1.4 The Compilation Process

**10.1.4(3)** **The mechanisms for creating an environment and for adding and replacing compilation units**

MAXAda uses **`a.mkenv`** to create an environment. MAXAda will set up its internal directory structure for that environment within the current, or a specified, directory. For more information, see "a.mkenv" on page 4-53.

The tool **`a.rmenv`** is provided to remove an existing environment. It removes an environment, including all units, their state information, and any partition definitions. The source files and any built partitions are left intact after this operation. See "a.rmenv" on page 4-87 for more details.

This implementation requires that a unit be *introduced* to an environment before it can be used in any way. Compilation units are introduced using the **`a.intro`** tool. See "a.intro" on page 4-30.

After having been introduced, though, a unit is still not visible (i.e. it has still not been added to the environment) until it has been compiled successfully.

Further, if multiple versions of the same unit are introduced, possibly from different source files, none are visible (i.e. appropriate removals are performed such that none

exist in the environment) until the user has manually resolved the ambiguity in favor of one of the versions.

To be precise, our environment is defined to include those units which are introduced to the environment, which have been compiled, and which are still semantically consistent, unless those units are obscured by other units. Units can be obscured in the following cases:

1. Having been manually *hidden*. Units can be hidden from the environment using the **`a.hide`** utility. More information about this tool can be found on page 4-27.

2. Having been hidden by a resolution in favor of another version of that unit. MAXAda provides the **`a.resolve`** tool as one way of resolving an ambiguity between units. See "Ambiguous Units" on page 3-10 for a more detailed discussion. For an example of this situation and its resolution, see "Hello Again... Ambiguous Units" on page 2-15.

3. Being a body for which a specification of the same name is already introduced to the environment, where the body cannot possibly be a completion for the specification (e.g. the specification is a package, whereas the body is a subprogram).

We consider this functionality desirable because it detects situations which, in real programs, are most probably user errors. It occasionally happens in large source trees that the same unit will be declared twice in two different source files. Certain compilation systems may arbitrarily select one to be compiled, perhaps without any indication that such an arbitrary choice had been made. Our system detects this case and forces the user to choose the intended version.

Further, the accidental introduction of a unit which causes an ambiguity can be resolved in favor of the original version without damaging the consistency of any units in the environment which might have depended upon the original version.

Finally, the accidental introduction of a subprogram body will not affect the consistency of any non-subprogram declaration with the same name unless specifically desired by the user. Once again, this permits the user to select the non-subprogram declaration as the correct version, without damaging the consistency of any units in the environment which might have depended upon it.

Units can also be removed from the environment completely using the **`a.rmsrc`** tool. This tool removes knowledge of source files (and units therein) from the environment. The syntax and usage of this tool can be found on page 4-88.

## RM 10.1.5 Pragmas and Program Units

### Implementation Permissions

**10.1.5(9)** **An implementation may place restrictions on configuration pragmas, so long as it allows them when the environment contains no library_items other than those of the predefined environment.**

Configuration pragmas that appear in a compilation with no `compilation_units` may only be successfully compiled when all units local to an environment are either uncompiled or inconsistent. See "Configuration Pragmas" on page 3-9 for more information.

# RM 10.2 Program Execution

## Post-Compilation Rules

**10.2(2) The manner of explicitly assigning library units to a partition**

Library units are explicitly assigned to a partition using the **`-set`** or **`-add`** options to **`a.partition`**.

Both options take a parameter which is a list of units that are to be included (or excluded) from a specified partition. There is also a way to include units that are directly or indirectly required by a given unit.

The **`-set`** option assigns the units in this list to the partition specified, removing any other units that may have previously been assigned to the partition.

The **`-add`** option assigns the units in this list to the partition, retaining any other units that may have been previously been assigned to the partition.

In addition, the **`-del`** option to **`a.partition`** is provided to remove specified units from a given partition.

A complete description of **`a.partition`** can be found on page 4-62.

**10.2(2) The implementation-defined means, if any, of specifying which compilation units are needed by a given compilation unit**

There are no implementation-defined means of specifying which compilation units are needed by a given compilation unit.

However, it is possible for the implementation to require that certain units be consistently compiled even though they will not be elaborated by the **`ENVIRONMENT`** task of an active partition.

If a unit is required by an active partition but the user specifies (via a link rule or dependent partition list) that the link method for that unit should be **`shared_object`**, then the unit will be utilized via a shared object partition, rather than being included directly in the active partition. As a result, for the purposes of the active partition, any other units included in the shared object partition are required to be consistently compiled, so that the shared object partition can be consistently linked. However, these other units will not be elaborated by the active partition.

**NOTE**

The active partition can be defined not to need any otherwise unneeded units in a required shared object partition by use of the **`-nosoclosure`** link option. However, extreme caution is recommended so that attempts to link the partition do not result in undefined symbols.

**10.2(7) The manner of designating the main subprogram of a partition**

The main subprogram of a partition is specified by using the **`-main`** option to **`a.partition`**. In the absence of an explicitly supplied **`-main`** option, if the partition has the same name as a library subprogram in the environment, that subprogram is assumed to be the main subprogram. Otherwise, no main subprogram is assumed and one must be explicitly specified using this option, if desired.

See "a.partition" on page 4-62 for details.

**10.2(18) The order of elaboration of library_items**

The order is determined with respect to the rules specified in Section 10.2 of the Ada 95 Reference Manual.

The order of elaboration of `library_items` for a particular partition may be obtained by invoking the **`a.link`** tool with the **`-E`** option for that partition. See "a.link" on page 4-33 for details. In addition, the **`-elab_src`** link option will produce similar results when the partition is built. See "Link Options" on page 4-109 for more information.

MAXAda obeys all the elaboration order requirements specified in RM 10.2.1. In addition, it attempts to automatically detect cases where elaboration order requirements were not specified but probably were desired. These automatically-detected elaboration order requirements are secondary to those specified by the language. If a conflict should arise, **`a.link`** will issue a warning and will obey the language requirements. Furthermore, if two automatically detected elaboration order requirements conflict, **`a.link`** will issue a warning and will select one arbitrarily. Conflicts arising solely from automatically-detected elaboration order requirements will never cause a partition to fail to link.

Whenever MAXAda automatically detects an elaboration order requirement that is not already specified by a pragma, it will issue an informational diagnostic suggesting that a pragma probably is desired. The insertion of the suggested pragma will increase the probability that the unit will work successfully with other compilers. Also, if two automatically-detected elaboration order requirements conflict, but it is known that an acceptable elaboration order does exist, the elaboration order can be specified by selecting and following the appropriate suggestion and ignoring the other.

When the execution for any of the following constructs can occur as part of the elaboration of a library unit, MAXAda assumes that an elaboration order requirement identical to the presence of a pragma `Elaborate_All` is desired:

- call to a subprogram
- call to a protected operation

- creation of a task object
- evaluation of a 'Access attribute whose prefix is a subprogram or protected operation
- call to a task entry

This first three items correspond to the elaboration checks required by RM 3.11(9-12). The item related to `'Access` is present to provide safety for calls to dereferenced access-to-subprogram objects. The item related to task entry calls is present to provide safety against deadlock or inconsistent rendezvous behavior with a task whose body has not been elaborated and therefore has not been activated. For calls, the elaboration order requirement indicates the library unit that contains the callable entity. For the creation of a task object, it indicates the library unit that contains the task type. For evaluation of `'Access` attributes, it indicates the library unit that contains the callable entity denoted by the prefix.

When the execution for the following construct can occur as part of the elaboration of a library unit, MAXAda assumes that an elaboration order requirement identical to the presence of a pragma `Elaborate` is desired:

- instantiation of a generic

This corresponds to the elaboration check required by RM 3.11(13). The elaboration order requirement indicates the library unit that contains the generic. In addition, the content of an instance of a generic is checked for any of the constructs listed above.

If the elaboration of a library unit includes constructs that are only executed conditionally, MAXAda assumes the worst: that all the constructs present can be executed. So, it assumes elaboration order requirements for all the possible executions.

Unfortunately, it is impossible to detect automatically all elaboration order requirements. In particular, the execution of the following constructs as part of the elaboration of a library unit probably will require a particular elaboration order, but the nature of that requirement cannot be determined automatically:

- dispatching call
- a library unit whose elaboration calls one of its own subunits which calls another of its subunits which executes any of the constructs listed above.

The execution in a task declared at library level of the any of the constructs listed earlier also can cause an elaboration order requirement. These are not detected automatically because, in general practice, library level tasks are written to postpone their execution until after the elaboration of library units.

**10.2(21) Parameter passing and function return for the main subprogram**

A main subprogram may not have any formal parameters and therefore no actual parameters are provided.

A main subprogram may be either a procedure or a function returning `Standard.Integer` (predefined type).

Unless overridden, the result of the call to a function main subprogram is used as the exit status of the program.

Upon program termination, the exit status is determined by the first applicable following rule:

- If the `Ada.Command_Line.Set_Exit_Status` procedure was called, the program's exit status is the last value used in a call to this procedure.
- If the main subprogram propagated an (unhandled) exception to the environment task, the exit status is the value 42, as required by the POSIX 1003.5 standard.
- If the main subprogram was a procedure which returned normally, the exit status is `Ada.Command_Line.Success`, which is the value 0.
- If the main subprogram was a function which returned normally, the exit status is the result of the call to that main subprogram.

**10.2(24) The mechanisms for building and running partitions**

The **`a.build`** utility is provided for building partitions. A single partition may be built by specifying its name to **`a.build`** or all partitions may be built by using the **`-allparts`** option. For more information, see "a.build" on page 4-3.

Active partitions may be run by specifying the executable's name on the command line (either the partition name itself or the output file name passed to **`a.build`** with the **`-o`** option). Nonactive partitions (**`archive`** and **`shared_object`**) cannot be executed independently but rather are utilized by active partitions.

## Dynamic Semantics

**10.2(25) The details of program execution, including program termination**

The execution of an Ada program (whose main procedure is written in Ada) includes the following steps:

1. Allocation of resources by the operating system required for execution, including internal operating system tables, virtual memory for program instructions and data, etc.
2. Execution is then begun at the start address of the program, or, for programs which utilize dynamically linked libraries, initially at the dynamic linker followed by the start address of the program (the start address is a symbol named `__start`).
3. Initialization of system libraries and user-defined `.init` routines then occurs.
4. Initialization of the run-time system then takes place.
5. The partition is then executed by calling its environment task.
6. After the environment task completes (and assuming it has not been terminated directly by the operating system or by direct user action via an operating system service (e.g. **`exit(2)`**)), the run-time system is finalized.

7. Finalization of system libraries and (non-Ada) user-defined `.fini` routines then occurs.
8. Execution of the program is then completed via the operating system service **`exit(2)`**.

Implementation Permissions

**10.2(28)** **The semantics of any nonactive partitions supported by the implementation**

**`archive`** and **`shared_object`** are two nonactive partitions supported by MAX-Ada. Neither of these types of partitions can be elaborated or executed independently. They are associated with an **`active`** partition at static link time (for an **`archive`** partition) or dynamic link time (for a **`shared_object`** partition). The **`active`** partition is responsible for elaboration and execution of any units in an **`archive`** or **`shared_object`** partition.

## RM 10.2.1 Elaboration Control

Implementation Advice

**10.2.1(12)** **In an implementation, a type declared in a preelaborated package should have the same representation in every elaboration of a given version of the package, whether the elaborations occur in distinct executions of the same program, or in executions of distinct programs or partitions that include the given version.**

MAXAda follows this advice.

# RM Chapter 11: Exceptions

## RM 11.4.1 The Package Exceptions

Static Semantics

**11.4.1(10)** **The information returned by Exception_Message**

The function `Exception_Message` returns a string containing the reason for the exception and a reference to the section in the Ada 95 Reference Manual from which it was derived.

The implementation-defined function `Originating_Instruction` in the package `Ada.Exceptions.Addresses` provides the address of the instruction which caused the associated exception to be raised. The implementation-defined function `Ada.Exceptions.Addresses.Propagation_Map` provides instruction addresses associated with the propagation of the associated exception.

**11.4.1(12)** **The result of Exceptions.Exception_Name for types declared within an unnamed block_statement**

The unnamed block_statement is given an artificial name of the form:

BLOCK__M*number*

where *number* is assigned in an arbitrary order for each declare block in the unit.

Consider the following example,

```
procedure foo is
        procedure bar is
        begin
            myname:
            declare
                this_except:exception;
            begin
                :
                :
            end
            declare
                this_except:exception;
            begin
                :
                :
            end
        end bar;
end foo;
```

In this example, there is a named block_statement and an unnamed block_statement. The exception in the named block_statement has a fully expanded name of `foo.bar.myname.this_except`. The exception in the unnamed block_statement has a fully expanded name of

`foo.bar.BLOCK__M1.this_except` (where the *number* `1` has been arbitrarily assigned).

**11.4.1(13)** **The information returned by Exception_Information**

The function `Exception_Information` returns a string containing the `Exception_Name`, `Exception_Message`, and the value of the program counter where the exception occurred.

Implementation Advice

**11.4.1(19)** **Exception_Message (by default) and Exception_Information should produce information useful for debugging. Exception_Message should be short (about one line), whereas Exception_Information can be long. Exception_Message should not include the Exception_Name. Exception_Information should include both the Exception_Name and the Exception_Message.**

The function `Exception_Message` returns a string containing the reason for the exception and a reference to the section in the Ada 95 Reference Manual from which it was derived.

The function `Exception_Information` returns a string containing the `Exception_Name`, `Exception_Message`, and the value of the program counter where the exception occurred. Additional information can be obtained via the implementation-defined function `Propagation_Map` in the package `Ada.Exceptions.Addresses`.

## RM 11.5 Suppressing Checks

Implementation Permissions

**11.5(27)** **Implementation-defined check names**

There are no implementation-defined check names in addition to those defined by the RM.

Implementation Advice

**11.5(28)** **The implementation should minimize the code executed for checks that have been suppressed.**

MAXAda does not strictly follow this advice.

In general, MAXAda will minimize code executed for checks that have been suppressed, but not always. Specifically, when a pragma `Suppress` is applied to a specific named entity, MAXAda does NOT minimize such code (i.e. the pragma has no effect in such circumstances).

# RM Chapter 12: Generic Units

There are no MAXAda amendments to Chapter 12 of the RM.

# RM Chapter 13: Representation Issues

## RM 13.1 Representation Items

### Implementation Permissions

**13.1(20)** **The interpretation of each aspect of representation**

**Any restrictions placed upon representation items**

*Coding* aspect of enumeration literals of an enumeration subtype:

See “RM 13.4 Enumeration Representation Clauses” on page M-43.

*Controlled* aspect of an access type:

See “RM 13.11 Storage Management” on page M-59.

*Convention* aspect of an object or subtype:

See “RM B.1 Interfacing Pragmas” on page M-74.

*Exported* aspect of an object:

See “RM B.1 Interfacing Pragmas” on page M-74.

*Imported* aspect of an object:

See “RM B.1 Interfacing Pragmas” on page M-74.

*Layout* aspect of records and record extensions:

See “RM 13.5.1 Record Representation Clauses” on page M-44.

*Packing* aspect of a type:

See “RM 13.2 Pragma Pack” on page M-34.

*Alignment* aspect of a subtype:

See “Notes” on page M-37.

### Implementation Advice

**13.1(21)** **The recommended level of support for all representation items is qualified as follows:**

**13.1(22)** **An implementation need not support representation items containing nonstatic expressions, except that an implementation should support a representation item for a given entity if each nonstatic expression in the representation item is a name that statically denotes a constant declared before the entity.**

MAXAda does not follow this advice in this release for nonstatic expressions that are names denoting a constant declared before the entity.

**13.1(23)** **An implementation need not support a specification for the Size for a given composite subtype, nor the size or storage place for an object (including a component) of a given composite subtype, unless the constraints on the subtype and its composite subcomponents (if any) are all static constraints.**

MAXAda follows this advice.

**13.1(24)** **An aliased component, or a component whose type is by-reference, should always be allocated at an addressable location.**

MAXAda follows this advice.

## RM 13.2 Pragma Pack

Implementation Advice

**13.2(6)** **If a type is packed, then the implementation should try to minimize storage allocated to objects of the type, possibly at the expense of speed of accessing components, subject to reasonable complexity in addressing calculations.**

MAXAda follows this advice.

**13.2(7)** **The recommended level of support for pragma Pack is:**

**13.2(8)** **For a packed record type, the components should be packed as tightly as possible subject to the Sizes of the component subtypes, and subject to any record_representation_clause that applies to the type; the implementation may, but need not, reorder components or cross aligned word boundaries to improve the packing. A component whose Size is greater than the word size may be allocated an integral number of words.**

MAXAda follows this advice. In addition, MAXAda may reorder components if a representation clause does not fully specify the layout of the record.

**13.2(9)** **For a packed array type, if the component subtype's Size is less than or equal to the word size, and Component_Size is not specified for the type, Component_Size should be less than or equal to the Size of the component subtype, rounded up to the nearest factor of the word size.**

MAXAda follows this advice. The implementation attempts to pack components of composite types as tightly as possible, except when alignment restrictions apply.

# RM 13.3 Representation Attributes

## Address Attributes

### Implementation Advice

**13.3(14)** **For an array X, X'Address should point at the first component of the array, and not at the array bounds.**

MAXAda follows this advice.

**13.3(15)** **The recommended level of support for the Address attribute is:**

**13.3(16)** **X'Address should produce a useful result if X is an object that is aliased or of a by-reference type, or is an entity whose Address has been specified.**

MAXAda follows this advice.

**13.3(17)** **An implementation should support Address clauses for imported subprograms.**

MAXAda follows this advice.

**13.3(18)** **Objects (including subcomponents) that are aliased or of a by-reference type should be allocated on storage element boundaries.**

MAXAda follows this advice.

**13.3(19)** **If the Address of an object is specified, or it is imported or exported, then the implementation should not perform optimizations based on assumptions of no aliases.**

MAXAda follows this advice.

### Notes

The implementation supports `Address` attribute definition clauses for variables, constants, and task entries.

For variables and constants, both logical and machine addresses are supported. A *logical address* refers to a virtual memory address in the execution program's address space. A *machine address* refers to a physical memory address.

**Logical Address Clauses**

- The function `Virtual_Address` is defined in the package `System.Addresses` to provide conversion from `Integer` values to `Address` values for virtual addresses only.
- Both static and variable logical addresses are supported.
- The value supplied to the address clause must be a valid logical address in the user's program.

**Machine Address Clauses**

- When a machine address is desired, the expression supplied on the address clause ***must*** be an invocation of the function `Machine_Address`, found in the implementation-defined package `System.Addresses`. Any other expression supplied to the address clause will cause it to be interpreted as a virtual address.
- Both static and variable machine addresses are supported.
- If the argument to `Machine_Address` is an integer literal, then static address translation can occur, thereby removing any additional overhead involved in accessing the variable at run time.
- In order to use machine address clauses, you must have permission to read and write the file **`/dev/mem`**.

**WARNING**

It is the user's responsibility to ensure that the supplied address is a valid physical memory address.

Memory copies done through address clauses will require a bus access for each word.

## Alignment Attributes

### Implementation Advice

**13.3(29)** **The recommended level of support for the Alignment attribute for subtypes is:**

**13.3(30)** **An implementation should support specified Alignments that are factors and multiples of the number of storage elements per word, subject to the following:**

**13.3(31)** **An implementation need not support specified Alignments for combinations of Sizes and Alignments that cannot be easily loaded and stored by available machine instructions.**

MAXAda follows this advice.

Alignments of 0, 1, 2, 4, 8, and 16 bytes are supported. An `Alignment` of 0 is used for non-aligned (or bit-aligned) subtypes. An `Alignment` of 0 means that the object is not necessarily aligned on a storage element boundary (RM 13.3(24)).

For stack objects, 4 is the maximum alignment supported.

MAXAda disallows combinations of `Size` and `Alignment` for stand-alone objects when not permitted by the target architecture. For example, the PowerPC architecture requires at least 4-byte alignment of floating point objects. Such restrictions will be enforced by the compiler.

**13.3(32)** **An implementation need not support specified Alignments that are greater than the maximum Alignment the implementation ever returns by default.**

MAXAda follows this advice.

Alignments greater than 16 bytes will not be supported for a subtype.

**13.3(33)** **The recommended level of support for the Alignment attribute for objects is:**

**13.3(34)** **Same as above, for subtypes, but in addition:**

**13.3(35)** **For stand-alone library-level objects of statically constrained subtypes, the implementation should support all Alignments supported by the target linker. For example, page alignment is likely to be supported for such objects, but not for subtypes.**

MAXAda does not follow this advice.

MAXAda provides support for 0, 1, 2, 4, 8, and 16 byte alignments. Page alignment for objects is not supported.

## Notes

`Alignment` is a property of a subtype. If a subtype requires alignment, then the address of any object of the subtype modulo the alignment is required to be zero. This is equivalent to the address being an integer multiple of the `Alignment` (if non-zero). For example, if an object's subtype has an `Alignment` of 4 (i.e. it is word aligned), its address must be a multiple of 4 bytes. The address `16#7fff439c#` is a multiple of 4 bytes, so it is word aligned. However, the address `16#7fff439b#` is not, because `16#7fff439b# mod 4` is 3.

Ideally, computer hardware and compilers would allow the size and alignment of any subtype to be any number of bits with any alignment. Unfortunately, allowing this in general can produce very slow code, so compilers impose some minimal restrictions, and make additional default choice to produce code with will execute much faster. For the PowerPC architecture, MAXAda imposes restrictions on the alignment of several classes of types.

MAXAda defines two distinct alignment concepts for each subtype: optimal alignment, and minimal alignment. The *optimal alignment* is the smallest alignment supported by the underlying hardware efficiently. The *minimal alignment* is the absolutely smallest alignment supported for the subtype. If smaller than the optimal alignment, its use often will result in inefficient code.

The following table summarizes the optimal and minimal alignments for each class of types:

**Table M-1. Alignment Restrictions**

| Class of type | Optimal (default) alignment | Minimal alignment |
|---|---|---|
| discrete and fixed point, representable in 1 - 8 bits | 1 | 0 |
| discrete and fixed point, representable in 9 - 16 bits | 2 | 0 |
| discrete and fixed point, representable in 17 - 32 bits | 4 | 0 |
| floating point, single precision (e.g. `Float`) | 4 | 4 |
| floating point, double precision (e.g. `Long_Float`) | | 4 |
| access | 4 | 4 |
| class-wide tagged | 4 | 4 |
| other composite | see below | see below |

**NOTE**

Notwithstanding the above table, the minimal alignment of an atomic or by-reference subtype is equal to its optimal alignment. Furthermore, for an aliased or atomic object, its subtype is treated as though its minimal alignment was equal to its optimal alignment.

When a subtype is packed, its default `Alignment` is equal to its minimal alignment. Otherwise, its default `Alignment` is equal to its optimal alignment.

### Discrete and fixed point

MAXAda imposes no minimal alignment restrictions on any discrete or fixed point subtypes. Components of these subtypes may be aligned to any arbitrary bit with record representation clauses, and arrays of these subtypes can be packed perfectly with attribute definition clauses or `pragma Pack`.

### Floating point

MAXAda imposes a minimal alignment of 4 bytes (a word) for floating point subtypes. The PowerPC architecture imposes a large penalty loads and stores of misaligned objects of floating point subtypes, so allowing smaller alignments is counterproductive.

The optimal alignment for single precision (32 bit) floating point subtypes (e.g. `Float`) also is 4 bytes.

The optimal alignment for double precision (64 bit) floating point subtypes (e.g. `Long_Float`) is 4 bytes.

#### Access

MAXAda imposes a minimal alignment of 4 bytes (a word) for access subtypes, in order to ensure fast dereference operations. The optimal alignment also is 4 bytes.

#### Composite

Except for class-wide tagged subtypes (see below), the minimal and optimal alignments are determined, respectively, by the largest minimal and optimal alignments of all the component and subcomponent subtypes. For example, if a record subtype contains an object whose subtype is an array of an access subtype, then the array subtype has both a minimal and optimal alignment of 4 bytes, and the record subtype has both a minimal and optimal alignment of at least 4 bytes (although the alignments could be larger because of other components).

Composite subtypes may contain implementation-defined components which affect alignment, also. The following classes of types contain implementation-defined components with minimal and optimal alignment of 4 bytes, causing a minimal and optimal alignment of the composite subtype of at least 4 bytes:

- record types with components or subcomponents of dynamic size (e.g. array components with variable or discriminant bounds)
- task types (contain a pointer to a task control block)
- protected types (contain a pointer to runtime protected information)
- tagged types (contain a tag component)
- controlled types (contain a tag component)

#### Class-wide tagged

An object nominally of a class-wide tagged subtype may actually denote an object of the root tagged type of the class, or of any type derived (directly or indirectly) from the root. Because an extension of the root is capable of adding components of any subtype, the minimum alignment for any class-wide subtype is the largest `Alignment` allowed for any subtype, 4 bytes.

## Size Attributes for Objects

Static Semantics

**13.3(41)** **Size may be specified for stand-alone objects via an attribute_definition_clause; the expression of such a clause shall be static and its value nonnegative.**

The expression of an `attribute_definition_clause` specifying the `Size` of a first subtype or object must have a value in the range 0 .. (2 ** 31) -1.

If an attempt is made to specify a size larger than (2 ** 31) - 1, a compilation error will occur.

The `Component_Size` `attribute_definition_clause` is restricted similarly.

Implementation Advice

**13.3(42)** **The recommended level of support for the Size attribute of objects is:**

**13.3(43)** **A Size clause should be supported for an object if the specified Size is at least as large as its subtype's Size, and corresponds to a size in storage elements that is a multiple of the object's Alignment (if the Alignment is nonzero).**

MAXAda does not follow this advice.

A size clause will be supported for an object if all of the following apply:

- The `Size` is at least as large as the object subtype's `Size`
- The `Size` is a multiple of the object subtype's *minimal* alignment (if non-zero), which may allow a `Size` other than a multiple of the object's `Alignment`
- For an aliased or floating point object, the `Size` is exactly its subtype's `Size`

## Size Attributes for Subtypes

Static Semantics

**13.3(48)** **The meaning of Size for indefinite subtypes**

If the prefix of a `Size` attribute reference denotes a specific indefinite subtype, then such a `Size` attribute reference will return the maximum possible size for an object of that prefix subtype.

If the prefix of a `Size` attribute reference denotes a class-wide subtype, then such a `Size` attribute reference will return the `Size` of the subtype at the root of the class. Note that the `Size` attribute is not defined for class-wide subtypes of the form `S'Class`, so this implementation-defined behavior applies only to named class-wide subtypes.

If the size of any subtype is not representable because its representation would exceed the word size of the target machine, then `Constraint_Error` will be raised.

The expression of an `attribute_definition_clause` specifying the `Size` of a first subtype or object must have a value in the range 0 .. (2 ** 31) -1.

If an attempt is made to specify a size larger than (2 ** 31) - 1, a compilation error will occur.

## Implementation Advice

**13.3(50)** **If the Size of a subtype is specified, and allows for efficient independent addressability (see 9.10) on the target architecture, then the Size of the following objects of the subtype should equal the Size of the subtype:**

**13.3(51)** **Aliased objects (including components).**

MAXAda follows this advice.

**13.3(52)** **Unaliased components, unless the Size of the component is determined by a component_clause or Component_Size clause.**

MAXAda follows this advice.

**13.3(53)** **A Size clause on a composite subtype should not affect the internal layout of components.**

MAXAda follows this advice.

**13.3(54)** **The recommended level of support for the Size attribute of subtypes is:**

**13.3(55)** **The Size (if not specified) of a static discrete or fixed point subtype should be the number of bits needed to represent each value belonging to the subtype using an unbiased representation, leaving space for a sign bit only if the subtype contains negative values. If such a subtype is a first subtype, then an implementation should support a specified Size for it that reflects this representation.**

MAXAda follows this advice.

**13.3(56)** **For a subtype implemented with levels of indirection, the Size should include the size of the pointers, but not the size of what they point at.**

MAXAda follows this advice.

The implementation does not implement any subtypes with implicit levels of indirection. Therefore no reference to the `Size` of a subtype will return a pointer or offset size.

## Notes

MAXAda supports the `Size` attribute definition clause fully for all discrete, fixed point, and composite subtypes. For floating point and access subtypes, a `Size` must conform to a supported machine representation; alternate or packed representations are not supported. The following table shows the required `Size` for each restricted class of type:

| Class of type | Required `Size` |
|---|---|
| Floating point, single precision (e.g. `Float`) | 32 |
| Floating point, double precision (e.g. `Long_Float`) | 64 |

| Class of type | Required `Size` |
|---|---|
| Access-to-object | 32 |
| Access-to-subprogram, protected | 64 |
| Access-to-subprogram, not protected | 96 |

## Component_Size Attributes

Implementation Advice

**13.3(71)** **The recommended level of support for the Component_Size attribute is:**

**13.3(72)** **An implementation need not support specified Component_Sizes that are less than the Size of the component subtype.**

MAXAda does not follow this advice.

MAXAda supports a `Component_Size` that is less than the `Size` of the component subtype's `Size`, as long as it is at least as large as the default `Size` that MAXAda would choose for the component subtype, and none of the restrictions in the following section apply.

**13.3(73)** **An implementation should support specified Component_Sizes that are factors and multiples of the word size. For such Component_Sizes, the array should contain no gaps between components. For other Component_Sizes (if supported), the array should contain no gaps between components when packing is also specified; the implementation should forbid this combination in cases where it cannot support a no-gaps representation.**

MAXAda does not follow this advice.

MAXAda supports a `Component_Size` only if all of the following apply:

- The `Component_Size` is at least as large as the component subtype's `Size`
- The `Component_Size` is an integer multiple of the component subtype's *minimal* alignment (if non-zero), which may allow a `Size` other than a multiple of the subtype's `Alignment`
- For an aliased or floating point component subtype, the `Component_Size` is exactly its subtype's `Size`

For a `Component_Size` which is a factor or multiple of the word size (1, 2, 4, 8, 16, or integer multiples of 32 bits), MAXAda will not place gaps between components.

For any other `Component_Size`, MAXAda will not place gaps between components when packing also is specified (e.g. via pragma `Pack`).

## External_Tag Attributes

Static Semantics

**13.3(75)** **The default external representation for a type tag**

The default external_tag representation of a tagged subtype is the expanded name of the first subtype of the type of the prefix subtype with the following implementation-defined names inserted:

- At the place of an unnamed block statement, or package elaboration block, an automatically generated anonymous block id.
- At the place of an accept statement, an automatically generated unique accept statement id.
- At the place of an unnamed loop statement, an automatically generated anonymous loop id.
- At the place of an Others exception handler, an automatically generated anonymous clause id.
- At the place of an inline expanded subprogram call, an automatically generated inline label id.
- At the place of an overloaded subprogram, an automatically generated overload resolution suffix is appended to the subprogram's name.

The implementation-defined names allow unique identification of tagged types defined within the associated language constructs.

Implementation Requirements

**13.3(76)** **What determines whether a compilation unit is the same in two different partitions**

If a compilation unit is not recompiled between building two different partitions that utilize it, it is considered “the same” compilation unit in both partitions. For purposes relating to the formation of the external tag of tagged types declared in such compilation units, the restrictions are not as stringent. The user can be assured that the external tag will be formed in the same manner for compilation units in multiple partitions if the source text of the compilation unit (and all compilation units upon which it depends) are identical, the compilation options are identical, the configuration pragmas in effect are identical, the target architectures are identical, and the version of the compiler is identical.

# RM 13.4 Enumeration Representation Clauses

Implementation Advice

**13.4(9)** **The recommended level of support for enumeration_representation_clauses is:**

**13.4(10)** **An implementation should support at least the internal codes in the range System.Min_Int..System.Max_Int. An implementation need not support enumeration_representation_clauses for boolean types.**

MAXAda follows this advice.

MAXAda implements the recommended level of support for an `enumeration_representation_clause`:

- The implementation will support internal codes in the range `System.Min_Int .. System.Max_Int`. Internal codes outside the supported range will be rejected at compile time.
- `enumeration_representation_clauses` are not supported for boolean types in this release.

## RM 13.5.1 Record Representation Clauses

The simple expression following the keywords `at mod` in an alignment clause specifies the `Storage_Unit` alignment restrictions for the record and must be one of the following values: 0, 1, 2, 4, 8, or 16.

The simple expression following the keyword `at` in a component clause specifies the `Storage_Unit` (relative to the beginning of the record) at which the following `range` is applicable. The static range following the keyword `range` specifies the bit range of the component. Components may overlap word boundaries (4 `Storage_Unit`s).

A component clause applied to a component that is a composite type does not imply packing for that component. For such component types, the implementation requires that pragma `PACK` or a record representation clause be applied to the subtype of the component if packing beyond the component's default size is desired. No component may be given a component clause which specifies a component size smaller than the `Size` of the component's subtype.

### Implementation Permissions

**13.5.1(15)** **Implementation-defined components**

MAXAda generates implementation-defined components for the following classes of types:

- record types with components or subcomponents of dynamic size (e.g. array components with variable or discriminant bounds)
- task types (contain a pointer to a task control block)
- protected types (contain a pointer to runtime protected information)
- tagged types (contain a tag component)
- controlled types (contain a tag component)

However, no means of naming implementation-defined components is supported, and no support is provided for representing such components in a `component_clause` of a `record_representation_clause`.

**13.5.1(16)** **If a record_representation_clause is given for an untagged derived type, the storage place attributes for all of the components of the derived type may differ from those of the corresponding components of the parent type, even for components whose storage place is not specified explicitly in the record_representation_clause.**

MAXAda takes advantage of this permission.

Implementation Advice

**13.5.1(17)** **The recommended level of support for record_representation_clauses is:**

**13.5.1(18)** **An implementation should support storage places that can be extracted with a load, mask, shift sequence of machine code, and set with a load, shift, mask, store sequence, given the available machine instructions and run-time model.**

MAXAda follows this advice.

For a `component_clause` for a component with `Size` less than `System.Word_Size`, MAXAda will not permit the component to occupy bits from more than 4 storage units. Specifically, `R.C'Last_Bit` must be less than `System.Word_Size`.

An informational diagnostic is issued if it is determined that a component clause would force the generation of less that optimal loads or stores for a component. This is commonly caused by components with alignments which do not conform to the optimal alignment.

**13.5.1(19)** **A storage place should be supported if its size is equal to the Size of the component subtype, and it starts and ends on a boundary that obeys the Alignment of the component subtype.**

MAXAda follows this advice.

MAXAda places no restriction on the end of a storage place.

**13.5.1(20)** **If the default bit ordering applies to the declaration of a given type, then for a component whose subtype's Size is less than the word size, any storage place that does not cross an aligned word boundary should be supported.**

MAXAda follows this advice.

MAXAda does support storage places that cross word boundaries in some cases. See 13.5.1(18) on page M-45.

**13.5.1(21)** **An implementation may reserve a storage place for the tag field of a tagged type, and disallow other components from overlapping that place.**

MAXAda does not follow this advice.

**13.5.1(22)** **An implementation need not support a component_clause for a component of an extension part if the storage place is not after the storage places of all components of the parent type, whether or not those storage places had been specified.**

MAXAda follows this advice.

Notes

MAXAda supports a storage place only if all of the following apply:

- For a discrete or fixed point component, the size of its storage place is at least as large as the minimum size required to represent its base range
- For a composite or access component, the size of its storage place is at least as large as the component subtype's `Size`
- For a component with `Size` less than `System.Word_Size`, the storage place occupies bits from no more than 4 storage units (specifically, `R.C'Last_Bit` must be less than `System.Word_Size`)
- The `Size` is a multiple of the component subtype's *minimal* alignment (if non-zero)
- For an aliased or floating point component, the `Component_Size` is exactly its subtype's `Size`

## RM 13.5.2 Storage Place Attributes

Implementation Advice

**13.5.2(5)** **If a component is represented using some form of pointer (such as an offset) to the actual data of the component, and this data is contiguous with the rest of the object, then the storage place attributes should reflect the place of the actual data, not the pointer. If a component is allocated discontiguously from the rest of the object, then a warning should be generated upon reference to one of its storage place attributes.**

MAXAda follows this advice.

## RM 13.5.3 Bit Ordering

Static Semantics

**13.5.3(5)** **If Word_Size = Storage_Unit, the default bit ordering is implementation defined. If Word_Size > Storage_Unit, the default bit ordering is the same as the ordering of storage elements in a word, when interpreted as an integer.**

```
Storage_Unit  = 8 bits.
Word_Size     = 32 bits.
```

The implementation supports only the default bit ordering.

The default bit ordering is dependent upon the conventions of the target machine architecture.

MAXAda on the Pentium uses `Low_Order_First` ("little endian") bit ordering.

Implementation Advice

**13.5.3(7)** **The recommended level of support for the nondefault bit ordering is:**

**13.5.3(8)** **If Word_Size = Storage_Unit, then the implementation should support the nondefault bit ordering in addition to the default bit ordering.**

Since Word_Size /= Storage_Unit, this advice is not relevant.

## RM 13.7 The Package System

Static Semantics

**13.7(2)** **The contents of the visible part of package System and its language-defined children**

The following files contain the package `System` and its language-defined descendants. They can be found in **`/usr/ada/`***rel_name***`/predefined`** (where *rel_name* is the name of the MAXAda release).

- `System.a`
- `System.Storage_Elements.a`
- `System.Storage_Pools.a`
- `System.Address_To_Access_Conversions.a`
- `System.Machine_Code.a`

Implementation Advice

**13.7(37)** **Address should be of a private type.**

MAXAda follows this advice.

## RM 13.7.1 The Package System.Storage_Elements

Implementation Advice

**13.7.1(16)** **Operations in System and its children should reflect the target environment semantics as closely as is reasonable. For example, on most machines, it makes sense for address arithmetic to "wrap around." Operations that do not make sense should raise Program_Error.**

MAXAda follows this advice.

In particular, the address arithmetic operations in the `System.Addresses` package "wrap around".

# RM 13.8 Machine Code Insertions

Static Semantics

**13.8(7)** **The contents of the visible part of package System.Machine_Code, and the meaning of code_statements**

The following file can be found in **/usr/ada/***rel_name***/predefined** (where *rel_name* is the name of the MAXAda release).

- `System.Machine_Code.a`

**WARNING**

Inline expansion of machine-code procedures is supported, but the user should exercise caution. It is not recommended practice to inline-expand machine-code procedures, as the compiler does not track register uses and definitions made by machine-code procedures.

## Pentium

The general definition of the package `Machine_Code` provides an assembly language interface for the target machine including the record types needed in the code statement, an enumeration type containing all of 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`) is the implementation-defined attribute `'Ref`.

Machine-code statements accept operands of type `Operand`, a private type that forms the basis of all machine-code address formats for the target.

The general syntax for a machine_code instruction statement is:

```
code_n' (opcode, operand {, operand});
```

In the following example, `code_1` is a record 'format' whose first argument is an enumeration vaue of the type `Opcode`, followed by a single operand of type `Operand`:

```
code_1' (call, disp("some_routine"));
```

The opcode must be an enumeration literal (i.e. it cannot be an object, an attribute, or a rename). Valid opcodes are listed in “Pentium Instruction Set” on page M-49.

An operand can only be an entity defined in `Machine_Code`. The `'Ref` attribute denotes the effective address of the first of the storage units allocated to the object. For a label, it refers to the address of the machine code associated with the corresponding body or statement. The attribute is of type `Operand` defined in the package `Machine_Code` and is allowed only within a machine code procedure. `'Ref` is supported only for simple objects, formal parameters and labels declared immediately within the subprogram containing the reference. Using `'Ref` on a formal parameter forces the formal parameter to be stored in memory and reference via memory.

The general syntax for a machine_code data statement is:

```
data_aggregate' (format, value {, value});
```

In the following example, `data_n` is a record 'format' whose first argument is an enumeration value of the type `Data_Format`, followed by an array of operands of type `Operand_Seq`:

```
data_n' (short, (+1, +2, +3));
```

The syntax for a machine_code directive statement is:

```
directive' (string_literal);
```

The single argument is a string literal which provides for the capability to insert any text directly into the assembly stream for that routine.

### Pentium Instruction Set

The `Machine_Code` package supports the following Pentium opcodes:

| | | | |
|---|---|---|---|
| aaa | aad | aam | aas |
| adcb | adcl | adcw | addb |
| addl | addpd | addps | addsd |
| addss | addw | andb | andl |
| andnpd | andnps | andpd | andps |
| andw | arplw | boundl | boundw |
| bsfl | bsfw | bsrl | bsrw |
| bswapl | btcl | btcw | btl |
| btrl | btrw | btsl | btsw |
| btw | call | call_abs | callw |
| callw_abs | cbtw | clc | cld |
| clflush | cli | cltd | clts |
| cmc | cmovael | cmovaew | cmoval |
| cmovaw | cmovbel | cmovbew | cmovbl |
| cmovbw | cmovcl | cmovcw | cmovel |
| cmovew | cmovgel | cmovgew | cmovgl |
| cmovgw | cmovlel | cmovlew | cmovll |
| cmovlw | cmovnael | cmovnaew | cmovnal |
| cmovnaw | cmovnbel | cmovnbew | cmovnbl |
| cmovnbw | cmovncl | cmovncw | cmovnel |
| cmovnew | cmovngel | cmovngew | cmovngl |
| cmovngw | cmovnlel | cmovnlew | cmovnll |
| cmovnlw | cmovnol | cmovnow | cmovnpl |
| cmovnpw | cmovnsl | cmovnsw | cmovnzl |

| | | | |
|---|---|---|---|
| cmovnzw | cmovol | cmovow | cmovpel |
| cmovpew | cmovpl | cmovpol | cmovpow |
| cmovpw | cmovsl | cmovsw | cmovzl |
| cmovzw | cmpb | cmpeqpd | cmpeqps |
| cmpeqsd | cmpeqss | cmpl | cmplepd |
| cmpleps | cmplesd | cmpless | cmpltpd |
| cmpltps | cmpltsd | cmpltss | cmpneqpd |
| cmpneqps | cmpneqsd | cmpneqss | cmpnlepd |
| cmpnleps | cmpnlesd | cmpnless | cmpnltpd |
| cmpnltps | cmpnltsd | cmpnltss | cmpordpd |
| cmpordps | cmpordsd | cmpordss | cmppd |
| cmpps | cmpsb | cmpsd | cmpsl |
| cmpss | cmpsw | cmpunordpd | cmpunordps |
| cmpunordsd | cmpunordss | cmpw | cmpxchg8b |
| cmpxchgb | cmpxchgl | cmpxchgw | comisd |
| comiss | cpuid | cvtdq2pd | cvtdq2ps |
| cvtpd2dq | cvtpd2pi | cvtpd2ps | cvtpi2pd |
| cvtpi2ps | cvtps2dq | cvtps2pd | cvtps2pi |
| cvtsd2si | cvtsd2ss | cvtsi2sd | cvtsi2ss |
| cvtss2sd | cvtss2si | cvttpd2dq | cvttpd2pi |
| cvttps2dq | cvttps2pi | cvttsd2si | cvttss2si |
| cwtd | cwtl | daa | das |
| decb | decl | decw | divb |
| divl | divpd | divps | divsd |
| divss | divw | emms | enter |
| f2xm1 | fabs | fadd | faddl |
| faddp | fadds | fbldt | fbstpt |
| fchs | fclex | fcmovb | fcmovbe |
| fcmove | fcmovnb | fcmovnbe | fcmovne |
| fcmovnu | fcmovu | fcom | fcomi |
| fcomip | fcoml | fcomp | fcompl |
| fcompp | fcomps | fcoms | fcos |
| fdecstp | fdiv | fdivl | fdivp |
| fdivr | fdivrl | fdivrp | fdivrs |
| fdivs | ffree | fiadd | fiaddl |
| ficom | ficoml | ficomp | ficompl |
| fidiv | fidivl | fidivr | fidivrl |
| fild | fildl | fildll | fimul |

| | | | |
|---|---|---|---|
| fimull | fincstp | finit | fist |
| fistl | fistp | fistpl | fistpll |
| fisub | fisubl | fisubr | fisubrl |
| fld | fld1 | fldcw | fldenv |
| fldl | fldl2e | fldl2t | fldlg2 |
| fldln2 | fldpi | flds | fldt |
| fldz | fmul | fmull | fmulp |
| fmuls | fnclex | fninit | fnop |
| fnsave | fnstcw | fnstenv | fnstsw |
| fpatan | fprem | fprem1 | fptan |
| frndint | frstor | fsave | fscale |
| fsin | fsincos | fsqrt | fst |
| fstcw | fstenv | fstl | fstp |
| fstpl | fstps | fstpt | fsts |
| fstsw | fsub | fsubl | fsubp |
| fsubr | fsubrl | fsubrp | fsubrs |
| fsubs | ftst | fucom | fucomi |
| fucomip | fucomp | fucompp | fwait |
| fxam | fxch | fxrstor | fxsave |
| fxtract | fyl2x | fyl2xp1 | hlt |
| idivb | idivl | idivw | imulb |
| imull | imulw | inb | incb |
| incl | incw | inl | insb |
| insl | insw | int | int3 |
| into | invd | invlpg | inw |
| iret | iretd | ja | jae |
| jb | jbe | jc | jcxz |
| je | jecxz | jg | jge |
| jl | jle | jmp | jmp_abs |
| jmpw | jmpw_abs | jna | jnae |
| jnb | jnbe | jnc | jne |
| jng | jnge | jnl | jnle |
| jno | jnp | jns | jnz |
| jo | jp | jpe | jpo |
| js | jz | lahf | larl |
| larw | lcall | lcallw | ldmxcsr |
| ldsl | ldsw | lea | leal |
| leave | leaw | lesl | lesw |

| | | | |
|---|---|---|---|
| lfence | lfsl | lfsw | lgdt |
| lgsl | lgsw | lidt | ljmp |
| ljmpw | lldt | lmsw | lock |
| lodsb | lodsl | lodsw | loop_opcode |
| loope | loopne | loopnz | loopz |
| lret | lsll | lslw | lssl |
| lssw | ltr | maskmovdqu | maskmovq |
| maxpd | maxps | maxsd | maxss |
| mfence | minpd | minps | minsd |
| minss | movapd | movaps | movb |
| movd | movdq2q | movdqa | movdqu |
| movhlps | movhpd | movhps | movl |
| movlhps | movlpd | movlps | movmskpd |
| movmskps | movntdq | movnti | movntpd |
| movntps | movntq | movq | movq2dq |
| movsb | movsbl | movsbw | movsd |
| movsl | movss | movsw | movswl |
| movupd | movups | movw | movzbl |
| movzbw | movzwl | mulb | mull |
| mulpd | mulps | mulsd | mulss |
| mulw | negb | negl | negw |
| nop | notb | notl | notw |
| orb | orl | orpd | orps |
| orw | outb | outl | outsb |
| outsl | outsw | outw | packssdw |
| packsswb | packuswb | paddb | paddd |
| paddq | paddsb | paddsw | paddusb |
| paddusw | paddw | pand | pandn |
| pause | pavgb | pavgw | pcmpeqb |
| pcmpeqd | pcmpeqw | pcmpgtb | pcmpgtd |
| pcmpgtw | pextrw | pinsrw | pmaddwd |
| pmaxsw | pmaxub | pminsw | pminub |
| pmovmskb | pmulhuw | pmulhw | pmullw |
| pmuludq | popa | popad | popf |
| popfd | popl | popw | por |
| prefetchnta | prefetcht0 | prefetcht1 | prefetcht2 |
| psadbw | pshufd | pshufhw | pshuflw |
| pshufw | pslld | pslldq | psllq |

| | | | |
|---|---|---|---|
| psllw | psrad | psraw | psrld |
| psrldq | psrlq | psrlw | psubb |
| psubd | psubq | psubsb | psubsw |
| psubusb | psubusw | psubw | punpckhbw |
| punpckhdq | punpckhqdq | punpckhwd | punpcklbw |
| punpckldq | punpcklqdq | punpcklwd | pusha |
| pushad | pushf | pushfd | pushl |
| pushw | pxor | rclb | rcll |
| rclw | rcpps | rcpss | rcrb |
| rcrl | rcrw | rdmsr | rdpmc |
| rdtsc | rep_insb | rep_insl | rep_insw |
| rep_lodsb | rep_lodsl | rep_lodsw | rep_movsb |
| rep_movsl | rep_movsw | rep_outsb | rep_outsl |
| rep_outsw | rep_stosb | rep_stosl | rep_stosw |
| repe_cmpsb | repe_cmpsl | repe_cmpsw | repe_scasb |
| repe_scasl | repe_scasw | repne_cmpsb | repne_cmpsl |
| repne_cmpsw | repne_scasb | repne_scasl | repne_scasw |
| ret | rolb | roll | rolw |
| rorb | rorl | rorw | rsm |
| rsqrtps | rsqrtss | sahf | salb |
| sall | salw | sarb | sarl |
| sarw | sbbb | sbbl | sbbw |
| scasb | scasl | scasw | 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 | sfence | sgdt | shlb |
| shldl | shldw | shll | shlw |
| shrb | shrdl | shrdw | shrl |
| shrw | shufpd | shufps | sidt |
| sldtl | sldtw | smsw | sqrtpd |
| sqrtps | sqrtsd | sqrtss | stc |
| std | sti | stmxcsr | stosb |
| stosl | stosw | str | subb |

| | | | |
|---|---|---|---|
| subl | subpd | subps | subsd |
| subss | subw | sysenter | sysexit |
| testb | testl | testw | ucomisd |
| ucomiss | ud2 | unpckhpd | unpckhps |
| unpcklpd | unpcklps | verr | verw |
| wait | wbinvd | wrmsr | xaddb |
| xaddl | xaddw | xchgb | xchgl |
| xchgw | xlat | xlatb | xorb |
| xorl | xorpd | xorps | xorw |

## Pentium Register Set

The supported Pentium registers include:

- General purpose 32-bit registers

  `eax`, `ebx`, `ecx`, `edx`, `esp`, `ebp`, `esi`, `edi`

- General purpose 16-bit registers

  `ax`, `bx`, `cx`, `dx`, `sp`, `bp`, `si`, `di`

- General purpose 8-bit registers

  `al`, `ah`, `bl`, `bh`, `cl`, `ch`, `dl`, `dh`

- Floating point registers

  `st0`, `st1`, `st2`, `st3`, `st4`, `st5`, `st6`, `st7`

- 64-bit SIMD registers

  `mm0`, `mm1`, `mm2`, `mm3`, `mm4`, `mm5`, `mm6`, `mm7`

- 128-bit SIMD registers and control register

  `xmm0`, `xmm1`, `xmm2`, `xmm3`, `xmm4`, `xmm5`, `xmm6`, `xmm7`

- Segment Registers

  `cs`, `ds`, `ss`, `es`, `fs`, `gs`

## Pentium Address Modes

The supported Pentium addressing modes include:

Displacements/Comments:

```
function immed (value : integer) return operand;
function "+"(value : integer) return operand;
```

```
function "-"(value : integer) return operand;
```

References to symbols/routines:

```
function disp (name : string) return operand;
```

Effective Address = [ reg ] and [ reg ] + disp

```
function base (reg : operand) return operand;
function base (disp : integer;
               reg : operand) return operand;
```

Effective Address = [ reg ] * s and [ reg ] * s + disp

```
type scale is (one, two, four, eight);

function index      (reg  : operand;
                     s    : scale := one) return operand;
function index      (disp : integer;
                     reg  : operand;
                     s    : scale := one) return operand;
```

Effective Address = [ reg1 + reg2 * s ] and [ reg1 + reg2 * s ] + disp

```
function indr_index (reg1 : operand;
                     reg2 : operand;
                     s    : scale := one) return operand;
function indr_index (disp : integer;
                     reg1 : operand;
                     reg2 : operand;
                     s    : scale := one) return operand;
```

### Pentium Machine Code Example

The following example uses machine code to add two numbers and return the result:

```
function add (x,y : integer) return integer is
    pragma implicit_code(off);
begin
    -- First argument is at (esp)+4
    code_2' (movl, base(4,esp), eax);
    code_2' (addl, base(8,esp), eax);
    -- Return values for scalars go into eax
    code_0' (op => ret);
end add;
```

## RM 13.9 Unchecked Type Conversions

Dynamic Semantics

**13.9(11)** **The effect of unchecked conversion**

Unchecked Type Conversions are implemented both for cases which do and do not meet the criteria in RM 13.9(6-10). The behavior for both cases is described here. This behavior is consistent with the semantics described in RM 13.9(5) for those cases that do meet the criteria in RM 13.9(6-10), and is a reasonable behavior for other cases.

The implementation treats an unchecked conversion as if some number of bits of the representation of the source expression are interpreted as, or moved to a target object.

The source expression is the actual expression passed to the formal parameter, S, of the instantiation. The target object is the result returned by an instantiation of the unchecked conversion function.

If the target subtype is an unconstrained composite subtype, then the result will have the maximum size possible for any object of that type. Otherwise, the result will have the same size as the size of the target subtype. This is referred to as the target size.

If the target subtype of an unchecked conversion is indefinite, then the value of the source expression is interpreted as a value of the target subtype. It is the user's responsibility to ensure that the source expression is a valid representation of a value of the target subtype, and that the `Size` of the source expression is sufficient to represent a value of the target subtype. If not, `Storage_Error` may be raised when the result of the unchecked conversion is used, or else the value of the result may contain garbage data.

The size of the source expression is simply the size of the value of the actual expression. This is referred to as the source size.

The representation of the result will be determined by effectively moving N bits from the actual expression to the result object. The number of bits moved, N, is the smaller of the source size and the target size.

The implementation will take advantage of permission granted in 13.9(12), and return the result by reference when appropriate. However, the implementation currently goes beyond the granted permission and returns the result by reference even for by-copy types if the result is indistinguishable from returning the result by copy.

Additional explanation is necessary in cases where the source size and target size are not the same. The meaning of such conversions depends upon the class of types involved, as well as which of the source or target sizes is larger.

**Justification:**

The implementation considers all objects of elementary types to be "left-justified" within the storage allocated, and all objects of composite types to be "left-justified". If, for alignment reasons, an object is placed in storage which is larger than the object's `Size`, the representation of an object of an elementary type is placed in the least-significant bits of storage, left-justified, with any padding in the most-significant bits.

Likewise, should an object of a composite type be allocated storage which is larger than the object's `Size`, the representation is placed in the most-significant bits of storage, left-justified, with any padding in the least-significant bit.

### Elementary Type to Elementary Type Conversions:

For all elementary types, calls to instantiations of unchecked conversions are implemented using the most efficient block move instruction to move a 1, 2, 4, or 8 byte source expression object to the target object. However, a bit move will be used if the size aspect of the object's representation has been specified in a size attribute definition_clause to be a value which is not a power of two.

If the source size and target size differ, then the smaller size is used.

If the target size is larger than the source size, then the bits of the source object's representation are moved to the least-significant bits of the target object. If the target object's subtype is signed, then the most-significant bit of the source object's representation is sign-extended through the most-significant bits of the target object's representation. Otherwise, the most-significant bits of the target object's representation are zero-filled.

If the target size is smaller than the source size, then the least-significant bits of the source object's representation are moved to the target object.

### Composite Type to Composite Type Conversions:

All composite-to-composite type conversions occur transferring bits starting with the lowest addressable bit of the source object to bits starting at the lowest addressable bit of the target object.

If the source size and target size differ, then the smaller size is used.

If the target size is larger than the source size, then the remaining bits of the target object's representation are zero-filled.

If the target size is smaller than the source size, the highest addressable bits from the source object are discarded.

### Elementary Type to Composite Type Conversions:

Conversions from elementary types to composite types are implemented by moving least-significant bits of the representation of the source object to the lowest addressable bits of the target object.

If the source size and target size differ, then the smaller size is used.

If the target size is larger than the source size, then the remaining bits of the target object's representation are zero-filled.

If the target size is smaller than the source size, then the least-significant bits of the source object's representation are moved to the target object.

### Composite Type to Elementary Type Conversions:

Conversions from composite types to elementary types are implemented by moving the lowest addressable bits of the representation of the source object to the least-significant bits of the target object.

If the source size and target size differ, then the smaller size is used.

If the target size is larger than the source size, then the bits of the source object's representation are moved to the least-significant bits of the target object. If the target object's subtype is signed, then the most-significant bit of the source object's representation is sign-extended through the most-significant bits of the target object's representation. Otherwise, the most-significant bits of the target object's representation are zero-filled.

If the target size is smaller than the source size, then the lowest addressable bits of the source object's representation are moved to the target object.

## Implementation Advice

**13.9(14)** **The Size of an array object should not include its bounds; hence, the bounds should not be part of the converted data.**

MAXAda follows this advice.

**13.9(15)** **The implementation should not generate unnecessary run-time checks to ensure that the representation of S is a representation of the target type. It should take advantage of the permission to return by reference when possible. Restrictions on unchecked conversions should be avoided unless required by the target environment.**

MAXAda follows this advice.

The implementation will not generate any unnecessary checks to determine if S is a valid representation of the target type.

The implementation will take advantage of the permission to return by reference when reasonable. No warnings or info messages will be issued alerting the user, however, if the implementation is unable to return by reference.

Restrictions on unchecked conversions are avoided by the implementation only when necessary to determine the size of the target subtype.

**13.9(16)** **The recommended level of support for unchecked conversions is:**

**13.9(17)** **Unchecked conversions should be supported and should be reversible in the cases where this clause defines the result. To enable meaningful use of unchecked conversion, a contiguous representation should be used for elementary subtypes, for statically constrained array subtypes whose component subtype is one of the subtypes described in this paragraph, and for record subtypes without discriminants whose component subtypes are described in this paragraph.**

Unchecked conversions will be supported and reversible in the cases where RM95 13.9 defines the result (with the exception that the implementation defines `S'Size` and `Target'Size` differently from the RM).

A contiguous representation will be used for elementary subtypes, for statically constrained array subtypes whose component subtype is one of the subtypes described in RM95 13.9(17), and for record subtypes without discriminants whose component subtypes are described in RM95 13.9(17).

The implementation will additionally support unchecked conversion for `S'Size /= Target'Size`. The smaller of the two sizes will be used, and the excess target space, if any, will be sign extended or zero filled as needed.

## RM 13.11 Storage Management

Static Semantics

**13.11(17)** **The manner of choosing a storage pool for an access type when Storage_Pool is not specified for the type**

For each access type with a `'Storage_Size` clause, a distinct object of type `System.Storage_Pools.Standard.Collection_Storage_Pool` is created with the given size and used.

All types without `'Storage_Size` clauses use the object `System.Storage_Pools.Standard.Objects.Predefined` of the type `System.Storage_Pools.Standard.Predefined_Storage_Pool`.

A variety of aspects of the memory used for these standard storage pools can be configured with following implementation-defined pragmas:

- `MEMORY_POOL` - (see "Pragma MEMORY_POOL" on page 6-23)
- `POOL_CACHE_MODE` - (see "Pragma POOL_CACHE_MODE" on page 6-25)
- `POOL_LOCK_STATE` - (see "Pragma POOL_LOCK_STATE" on page 6-25)
- `POOL_SIZE` - (see "Pragma POOL_SIZE" on page 6-26)

**13.11(17)** **Whether or not the implementation provides user-accessible names for the standard pool type(s)**

MAXAda does provide user-accessible names for the standard pool types. The storage pool type used for types with a `'Storage_Size` clause is `System.Storage_Pools.Standard.Collection_Storage_Pool`. The storage pool type used for types with neither a `'Storage_Pool` nor a `'Storage_Size` clause is `System.Storage_Pools.Standard.Predefined_Storage_Pool`.

There is a single object of type `Predefined_Storage_Pool` named `System.Storage_Pools.Standard.Objects.Predefined`. It is erroneous to create any other object of this type.

**13.11(18)** **The meaning of Storage_Size**

If neither 'Storage_Size nor 'Storage_Pool is specified for a particular access type, Storage_Size for that type is defined to return the value -1. The Storage_Size does not include the TCB (Task Control Block) for the task.

**13.11(20)** **The effect of calling Allocate and Deallocate for a standard storage pool directly (rather than implicitly via an allocator or an instance of Unchecked_Deallocation) is unspecified.**

The primitives Allocate and Deallocate operate on memory directly. They are unaware of the manner in which that memory will be used. As such, it is erroneous to attempt to allocate or deallocate a controlled object by directly calling these routines. Instead, an allocator or an instance of Ada.Unchecked_Deallocation should be used.

## Documentation Requirements

**13.11(22)** **Implementation-defined aspects of storage pools**

The set of values that a user-defined Allocate or Deallocate procedure needs to accept are:

1, 2, 4, 8, 16

**13.11(22)** **Information of how storage is allocated by the standard storage pools**

The System.Storage_Pools.Standard.Predefined_Storage_Pool, the storage pool used for types with neither 'Storage_Pool nor 'Storage_Size clauses, allocates memory via the **mmap(2)** system service. It allocates an amount of memory equal to that specified in pragma POOL_SIZE(COLLECTION, DEFAULT, size). If the value is the keyword UNLIMITED or if no such pragma exists, then 512K is allocated initially, although more may be allocated later, also via **mmap(2)**.

The System.Storage_Pools.Standard.Collection_Storage_Pool, the storage pool used for types with 'Storage_Size clauses, allocates memory based on the context in which the access type is declared. If in a context with a stack frame, memory will generally be allocated inside that stack frame. This is generally possible if the type is declared within one of the following constructs:

- subprogram
- task body
- protected operation body
- handled_sequence_of_statements in a package body

There are a couple circumstances where, even though a stack frame is present for a given construct, memory cannot be allocated from it:

- The size of the storage pool cannot be determined when the stack frame is created. This can occur if the type is declared inside a separate package body or inside the body of an instance whose corresponding generic body is not declared within the same compilation unit as the instance or is separate.

- The memory attributes (see "Memory Attributes" on page 6-20) of the collection differ from those specified for the stack.

In any case where memory cannot be allocated from a stack frame, it is allocated instead from the `System.Storage_Pools.Standard.Object.Predefined` storage pool.

Implementation Advice

**13.11(23)** **An implementation should document any cases in which it dynamically allocates heap storage for a purpose other than the evaluation of an allocator.**

MAXAda performs dynamic implicit heap allocations for the following operations:

- creation of a task, or object of a type with task parts
- creation of a protected object, or object of a type with protected parts
- creation of an object with controlled parts
- creation of a package body stub
- elaboration of a package instance whose corresponding generic is not declared within the same compilation unit as the instance or is separate
- elaboration of an instance of `Ada.Task_Attributes`
- elaboration of a shared instance whose generic environment (the memory space containing information required to differentiate a shared instance from other shared instances of the same generic) is larger than 51.2 Kb. See "Pragma SHARE_BODY" on page M-129.
- call to the function `Ada.Exceptions.Save_Occurrence` (but not the procedure)
- elaboration of a master, other than that associated with the `ENVIRONMENT` task, which contains any of the following declarations:
  - access type
  - separate body
  - instance whose corresponding generic is not declared within the same compilation unit as the instance or is separate
- any of the following operations performed at library-level (i.e. any operation not performed within a subprogram or task):
  - creation of an object of a dynamically constrained type
  - conversion of a value of a dynamically constrained type
  - string catenation producing a dynamically constrained result
  - non-string catenation
  - logical or `not` operator expression involving dynamically constrained arrays of booleans

- copy of a dynamically sized bit-aligned actual used for parameter passing
- copy of a dynamically sized atomic actual whose corresponding formal type is not atomic (see RM C.6(19))
- call of an `'Input` attribute whose prefix is a composite type
- call of an instance of `Ada.Unchecked_Conversion` with a dynamically constrained target type
- elaboration of a `'Storage_Size` representation clause

**13.11(24)** **A default (implementation-provided) storage pool for an access-to-constant type should not have overhead to support deallocation of individual objects.**

MAXAda does not follow this advice in this release.

**13.11(25)** **A storage pool for an anonymous access type should be created at the point of an allocator for the type, and be reclaimed when the designated object becomes inaccessible.**

MAXAda does not follow this advice in this release.

## RM 13.11.2 Unchecked Storage Deallocation

### Implementation Advice

**13.11.2(17)** **For a standard storage pool, Free should actually reclaim the storage.**

MAXAda follows this advice.

### Implementation Permissions

**13.11.3(8)** **An implementation need not support garbage collection, in which case, a pragma Controlled has no effect.**

Pragma `CONTROLLED` will be accepted, but will have no effect since this implementation does not perform garbage collection.

## RM 13.12 Pragma Restrictions

**13.12(7)** **The set of restrictions allowed in a pragma Restrictions**

MAXAda supports all restrictions defined in Section D.7 of the Ada 95 Reference Manual and the following implementation-defined restriction:

```
No_Stream_Attributes
```

See “Pragma RESTRICTIONS” on page M-128 for more details.

**13.12(9)** **The consequences of violating limitations on Restrictions pragmas**

An expression in a pragma `RESTRICTIONS` may contain only static, nonnegative values whose values are in the range of the type `Integer`. Any other values will result in a compilation error.

Because none of the restrictions defined in Section D.7 of the Ada 95 Reference Manual currently have an actual effect on the run-time, there are no further limitations.

## RM 13.13.2 Stream-Oriented Attributes

**13.13.2(9)** **The representations used by Read and Write attributes of elementary types in terms of stream elements**

A stream element is a value of the type `Ada.Streams.Stream_Element`. This is the smallest unit of data that is read from or written to a stream. For this implementation, a stream element is an 8-bit byte. Its size is the same as that of a storage element, defined in 13.3(8).

The implementation follows the advice of 13.13.2(9) for the Read and Write attributes:

- For a scalar type, the implementation will use the smallest number of stream elements that will represent all the values of the base range of the type. The normal, in-memory storage element representation will be used for the stream element representation of the value, with the stream elements ordered according to the `Bit_Order` aspect of the type. For this implementation, highest order first, lowest order last.

  If the `Size` of the type is smaller than the bits of the stream element representation, signed scalar values will be sign-extended. The extra highest order bits of a modular value will be zeroed.

- For access-object types, the value will be emitted as four stream elements, high order first, as for an object of type `System.Address`.

- For access-subprogram types, the value will be emitted as a sequence of 2 .. 3 values of access-object format:

  - subprogram entry address
  - protected object address | static link address
  - generic environment address, when necessary.

Implementation Advice

**13.13.2(17)** **If a stream element is the same size as a storage element, then the normal in-memory representation should be used by Read and Write for scalar objects. Otherwise, Read and Write should use the smallest number of stream elements needed to represent all values in the base range of the scalar type.**

MAXAda follows this advice.

# RM Annex A: Predefined Language Environment

Implementation Permissions

**A(4)** **The implementation may restrict the replacement of language-defined compilation units. The implementation may restrict the children of language-defined library units (other than Standard).**

MAXAda restricts the replacement of any of the following units or any children of the following units:

```
Ada.Asynchronous_Task_Control
Ada.Calendar
Ada.Dynamic_Priorities
Ada.Exceptions
Ada.Finalization
Ada.Interrupts
Ada.Real_Time
Ada.Task_Attributes
Ada.Task_Identification
Ada.Tags
Interfaces.Restricted_Fortran
System
System.Machine_Code
System.Storage_Elements
System.Storage_Pools
System.Storage_Pools.Standard
System.Storage_Pools.Standard.Objects
```

In addition, MAXAda restricts the replacement of any of the units within the following package or any of the units within its children:

```
Ada.RTS
```

## RM A.1 The Package Standard

Static Semantics

**A.1(3)** **The names and characteristics of the numeric subtypes declared in the visible part of package Standard**

```
subtype natural  is integer range 0 .. integer'last;
subtype positive is integer range 1 .. integer'last;
```

where, in this implementation, the type `Integer` is defined as:

```
type Integer is range -2**31 .. 2**31-1;
```

Implementation Advice

**A.1(52)** **If an implementation provides additional named predefined integer types, then the names should end with ``Integer'' as in ``Long_Integer''. If an implementation provides additional named predefined floating point types, then the names should end with ``Float'' as in ``Long_Float''.**

MAXAda follows this advice.

MAXAda supplies the following additional named predefined types:

`Long_Integer`

`Short_Integer` (for compatibility only; its use is not recommended)

`Tiny_Integer` (for compatibility only; its use is not recommended)

`Long_Float`

## RM A.3.2 The Package Characters.Handling

Implementation Advice

**A.3.2(49)** **If an implementation provides a localized definition of Character or Wide_Character, then the effects of the subprograms in Characters.Handling should reflect the localizations. See also 3.5.2.**

MAXAda does not provide localized definitions of `Character` or `Wide_Character`; thus the advice is not relevant.

## RM A.4.4 Bounded-Length String Handling

Implementation Advice

**A.4.4(106)** **Bounded string objects should not be implemented by implicit pointers and dynamic allocation.**

MAXAda does not follow this advice in this release.

## RM A.5.1 Elementary Functions

**A.5.1(1)** **The accuracy actually achieved by the elementary functions**

These functions use the underlying math library, **`libm.a`**. Function results are expressed in `float` or `long_float`, which equate to C `float` or `double`. Ada `float` has 6 digits of precision, where `long_float` has 15 digits of precision.

Implementation Requirements

**A.5.1(46)** **The sign of a zero result from some of the operators or functions in Numerics.Generic_Elementary_Functions, when Float_Type'Signed_Zeros is True**

The sign of a prescribed zero result in the aforementioned cases would be positive (+0.0).

## RM A.5.2 Random Number Generation

Static Semantics

**A.5.2(27)** **The value of Numerics.Float_Random.Max_Image_Width**

The following line appears in `Numerics.Float_Random`:

```
max_image_width : constant := 12 + 4 ; -- base 16
integer literal, 12 digits
```

**A.5.2(27)** **The value of Numerics.Discrete_Random.Max_Image_Width**

The following line appears in `Numerics.Discrete_Random`:

```
max_image_width : constant := 12 + 4 ; -- base 16
integer literal, 12 digits
```

**A.5.2(32)** **The algorithms for random number generation**

MAXAda uses the standard C functions, `erand48`, `nrand48`, and `jrand48` in its random number generation algorithms. These functions generate pseudo-random numbers using the well-known linear congruential algorithm and 48-bit integer arithmetic.

`erand48` returns non-negative double-precision floating-point values uniformly distributed over the interval [0.0, 1.0).

`nrand48` returns non-negative long integers uniformly distributed over the interval [0, 2**31).

`jrand48` returns signed long integers uniformly distributed over the interval [-2**31, 2**31).

**A.5.2(38)** **The string representation of a random number generator's state**

The string representation of a random number generator's state is a 12-digit integer literal in base 16, such as, `16#123456789abc#`. This 12-digit literal is a 48-bit number composed of three 16-bit entities which the generator uses.

Documentation Requirements

**A.5.2(44)** **No one algorithm for random number generation is best for all applications. To enable the user to determine the suitability of the random number generators for the intended application, the implementation shall describe the algorithm used and shall give its period, if known exactly, or a lower bound on the period, if the exact period is**

**unknown. Periods that are so long that the periodicity is unobservable in practice can be described in such terms, without giving a numerical bound.**

The base algorithm of `erand48`, `nrand48`, and `jrand48` for generating 48 bit random numbers is a linear congruential scheme with the formula:

```
X(n+1) = (a*X(n)+c)mod(m)
```

where the arithmetic is carried out using 48 bit arithmetic.

`nrand48` returns the high order 31 bits of `X(n+1)`

`jrand48` returns the high order 32 bits of `X(n+1)`

`erand48` returns all 48 bits of `X(n+1)` considered as a fraction with the binary point before the first bit.

`a` = 0x5deece66d, `c`=0x0b, and `m`=2^48.

*Knuth - Art of Computer Programming, Seminumerical Algorithms Vol II 3.2.1.1 pg 15 Theorem A* states that:

> The linear congrential sequence has a period of length `m` if and only if
>
> i. `c` is relatively prime to `m`;
>
> ii. `b` = `a-1` is a multiple of `p`, for every prime p dividing `m`;
>
> iii. `b` is a multiple of 4, if `m` is a multiple of 4.
>
> In our case:
>
> i. `c` = 11 which is a prime so it is relatively prime to 2^48
>
> ii. `b` = 0x5deece66c is divisible by 2, and 2 is the only prime dividing 2^48
>
> iii. `m` is divisible by 4 as is `b` = 0x5deece66c.
>
> Hence the period is 2^48.

**A.5.2(45)** **The minimum time interval between calls to the time-dependent Reset procedure that are guaranteed to initiate different random number sequences**

The time-dependent `Reset` procedure is based on a value from `Ada.Real_Time.Clock`.

The actual rate at which the clock ticks is dependent on the specific system type where the application runs.

Thus, minimum time interval between successive calls to the `Reset` procedure that are guaranteed to initiate different random number sequences is zero.

Implementation Advice

**A.5.2(46)** **Any storage associated with an object of type Generator should be reclaimed on exit from the scope of the object.**

MAXAda follows this advice.

**A.5.2(47)** **If the generator period is sufficiently long in relation to the number of distinct initiator values, then each possible value of Initiator passed to Reset should initiate a sequence of random numbers that does not, in a practical sense, overlap the sequence initiated by any other value. If this is not possible, then the mapping between initiator values and generator states should be a rapidly varying function of the initiator value.**

The number of possible initial values is 2^48 which is the same as the period so this is not applicable to MAXAda.

## RM A.5.3 Attributes of Floating Point Types

Static Semantics

**A.5.3(72)** **The values of the Model_Mantissa, Model_Emin, Model_Epsilon, Model, Safe_First, and Safe_Last attributes, if the Numerics Annex is not supported**

The Numerics Annex is not supported in this release of MAXAda.

| Attribute | IEEE_Float_32 | IEEE_Float_64 |
|---|---|---|
| `'Model_Mantissa` | 24<br>*(same as 'Machine_Mantissa)* | 53<br>*(same as 'Machine_Mantissa)* |
| `'Model_Emin` | -125<br>*(same as 'Machine_Emin)* | -1021<br>*(same as 'Machine_Emin)* |
| `'Model_Epsilon` | 2.0**(-23) | 2.0**(-52) |
| `'Model` | returns the same value as the parameter passed in | returns the same value as the parameter passed in |
| `'Safe_First` | -2.0**128*(1.0-2.0**(-24)) | -2.0**1024*(1.0-2.0**(-53)) |
| `'Safe_Last` | 2.0**128*(1.0-2.0**(-24)) | 2.0**1024*(1.0-2.0**(-53)) |

## RM A.7 External Files and File Objects

Static Semantics

**A.7(14)** **Any implementation-defined characteristics of the input-output packages**

The MAXAda implementation of the standard Ada I/O packages support form parameters of the following syntax and semantics for the `Open` and `Create` subprograms:

*form_parameters* `::= [` *form_specification* `{, ` *form_specification*`} ]`
*form_specification* `::=` *form_name* `=>` *form_value*

The following list defines the supported *form_name* and *form_values*:

`Append => True | False`

When specified to the `Open` subprogram:

- If the mode is `out_file` or `inout_file`, then if the `form_value` is `True`, the file will be opened in `append` mode, and if the `form_value` is `False`, the file will be truncated.
- If the mode is `append_file`, then this form parameter is ignored and the file is opened in `append` mode.
- If the mode is `in_file`, then this form parameter is irrelevent and ignored.

`Use_Error` is raised if specified to the `Create` subprogram.

```
Owner => read | write | execute | read_write | ...
Group => read | write | execute | read_write | ...
Other => read | write | execute | read_write | ...
```

The file being created will have the permissions as defined by *form_name* and *form_value*. Note that *form_value* may be any combination of `read`, `write`, or `execute`, separated by an underscore (e.g., `write_read_execute`).

`Use_Error` is raised if specified to the `Open` subprogram.

`File_Descriptor =>` *n*

This specifies that the high-level `file_type` be associated with an existing open file descriptor, as specified by *n*. *n* should be of a form consistent with `integer'` image.

`Use_Error` is raised if specified to the `Create` subprogram.

`Page_Terminators => True | False`

If `False`, then page terminators are not output to the external file. If `Ada.Characters.Latin_1.FF` is encountered while reading from the external file, it is interpreted as a character `Ada.Characters.Latin_1.FF` and not as a page terminator. `Use_Error` will be raised upon explicit calls to `Ada.Text_IO.New_Page` or to `Ada.Text_IO.Set_Line` when the current line number exceeds the specified argument.

If `True`, page termination on output will result in `Ada.Characters.Latin_1.FF` being written to the external file. Encountering `Ada.Characters.Latin_1.FF` on input is interpreted as a new page (e.g. `Ada.Text_IO.Get` would never see an `Ada.Characters.Latin_1.FF` returned to it). `True` is the default.

```
Terminal_Input => Lines | Characters
```

If `Lines`, terminal input shall be done in canonical mode. This is the default.

If `Characters`, terminal input shall be done in non-canonical mode, such that the minimum input count is 1 character, and the minimum input time is 0 seconds.

This form specification has no effect if the associated `file_type` is not used for terminal input.

```
Echo => True | False
```

If `True`, echoing of characters is done on input operations to the associated terminal device. This is the default.

If `False`, echoing of characters is not done on input operations to the associated terminal device for non-canonical processing. `Use_Error` is raised if the non-canonical processing has not been specified.

```
File_Structure => Regular | Fifo
```

If `Fifo`, then the file being created will be a named FIFO file. Otherwise, the file being created will be a regular file, which is the default.

`Use_Error` is raised if specified to the `Open` subprogram.

```
Blocking => Tasks | Program
```

If all the tasks in the running program have *task_weight* bound, then the *form_value* must be `Tasks`; otherwise, `Use_Error` is raised.

If all the tasks in the running program have *task_weight* multiplexed, then the *form_value* must be `Program`; otherwise, `Use_Error` is raised.

If the running program has tasks of both bound and multiplexed *task_weight*, then the *form_value* must be `Program`; otherwise, `Use_Error` is raised. This use of `Program` blocking behavior is intended to indicate that if a task blocks while performing I/O on the associated file, other tasks in the program may be blocked. The actual blocking behavior depends on the *task_weight* of a blocked task.

```
WCEM => n | h
```

If `n`, `wide_characters` are not allowed to be written or read. An attempt to write a character that is not in type `Character` will result in `Use_Error`. On a read, any encoded `wide_character` will be interpreted only as the constituent characters of the encoding.

If `h`, `wide_characters` are allowed to be written and read. Any character that is in type `Character` except `ESC` (decimal value 27) is written and read normally. Any other character is written or read in a hex-encoded format: an `ESC` character followed by four hexadecimal digits that represents the character's 2 digit row-octet followed by its 2 digit cell-octet. An attempt to read an `ESC` followed by anything other than 4 hexadecimal digits will result in `Data_Error`.

This form parameter has no effect on types other than `Ada.Text_IO.File_Type` and `Ada.Wide_Text_IO.File_Type`.

The default value for `Ada.Text_IO.File_Type` is 'n'. The default value for `Ada.Wide_Text_IO.File_Type` is 'h'.

Implementation-defined exception propagations in I/O packages are not known at this time.

## RM A.9 The Generic Package Storage_IO

Static Semantics

**A.9(10)** **The value of Buffer_Size in Storage_IO**

```
(Element_Type'Size + System.Storage_Unit - 1) / System.Storage_Unit
```

## RM A.10 Text Input-Output

Static Semantics

**A.10(5)** **external files for standard input, standard output, and standard error**

The following are the external files for standard input, standard output, and standard error:

- **`stdin`** - standard input
- **`stdout`** - standard output
- **`stderr`** - standard error

# RM A.10.7 Input-Output of Characters and Strings

Implementation Advice

**A.10.7(23)** **The Get_Immediate procedures should be implemented with unbuffered input. For a device such as a keyboard, input should be "available" if a key has already been typed, whereas for a disk file, input should always be available except at end of file. For a file associated with a keyboard-like device, any line-editing features of the underlying operating system should be disabled during the execution of Get_Immediate.**

MAXAda follows this advice.

# RM A.10.9 Input-Output for Real Types

Implementation Permissions

**A.10.9(36)** **The accuracy of the value produced by Put**

Values of type `float` have 6 digits of precision.

Values of type `long_float` have 15 digits of precision.

# RM A.13 Exceptions in Input-Output

Documentation Requirements

**A.13(15)** **The implementation shall document the conditions under which Name_Error, Use_Error and Device_Error are propagated.**

`Name_Error`

- When a null string is used to create a temporary file
- When `mkfifo` or `open` return `ENOTDIR` or `ENOENT` (i.e an invalid file-name is provided to an "open" call)

`Use_Error`

- When `mkfifo` or `open` return an error other than `ENOENT` or `ENOTDIR`
- If status for the file cannot be obtained
- If a file is opened for writing but the opener does not have write access
- If a file is opened for reading but the opener does not have read access
- If the file cannot be opened for any reason
- If the supplied file descriptor for an `open` is invalid
- If a file is already open but the file position is unknown

- A semaphore used to control `file_locking` fails
- An invalid file descriptor is used in an attempt to close a file
- If an attempt is made to "put" a wide character when the `file_encoding` mode prohibits it
- If a file name is reused in a form string
- If `create` and `append` are used in the same form string
- When one of owner, group or other is not used in association with `create` in a form string
- If a null file or `file_descriptor` is passed to `text_support.name`
- If the `file_structure` form parameter is used without the `create` parameter
- If the `file_descriptor` form parameter is used with the `create` parameter

`Device_Error`

- If an error occurs while reading a file (other than `EOF`)
- If an error occurs opening or reading from a tty device
- If an `fstat` operation performed by a `read`, `write` or `open` call returns an error status

# RM A.15 The Package Command_Line

**A.15(1)** **The meaning of Argument_Count, Argument, and Command_Name**

These functions are implemented as transformations of the standard C parameters, `argc` and `argv`:

| | |
|---|---|
| `argument_count` | integer value of `argc-1` |
| `argument` | string value of the argument `argv[n]` |
| `command_name` | string value of `argv[0]` |

# RM Annex B: Interface to Other Languages

## RM B.1 Interfacing Pragmas

MAXAda supports the `Ada`, `Assembler`, and `C` conventions as required by RM B for use in pragmas `IMPORT`, `EXPORT`, and `CONVENTION`. MAXAda generally follows the advice and recommendations of RM B for these conventions, as indicated in the following sections."

MAXAda also supports the `Unchecked_C` and `Restricted_Fortran` conventions, as well as the internal conventions `Intrinsic`, `Protected`, and `Entry`.

The `COBOL` and `Fortran` conventions are not supported in this implementation.

The implementation supports elaboration by a foreign language program in a slightly more versatile manner than that specified in Ada 95 Reference Manual B.1(39). See “a.partition” on page 4-62 and “Elaboration and Finalization Methods” on page 3-16 for more information.

### Legality Rules

**B.1(11)** **Implementation-defined convention names**

The allowable conventions are:

- `Ada`
- `Assembler`
- `C`
- `Unchecked_C`
- `Restricted_Fortran`
- `Intrinsic`
- `Entry` (internal use only)
- `Protected` (internal use only)

### Static Semantics

**B.1(36)** **The meaning of link names**

The link name passed to the system linker is identical to the `Link_Name` parameter as specified in a pragma `IMPORT` or `EXPORT`.

**B.1(36)** **The manner of choosing link names when neither the link name nor the addresses of an imported or exported entity is specified**

If a link name is not specified, then the link name is obtained according to the following rules for the specified conventions:

- `Ada`, `Assembler`, `Intrinsic`:

- If an *external_name* is specified, then the link name is obtained by prepending a "`A_`" prefix to the reversed *expanded_name* specified in the *external_name* string. For example, an *external_name* of "`my_package.my_subprogram`" will be transformed to a link name of "`A_my_subprogram.my_package`".
- If no *external_name* is specified, then the link name is obtained by prepending "`A_`" prefix to the reversed fully *expanded_name* of the entity with implementation-defined names inserted for unnamed constructs and overload resolution.

- `C` and `Unchecked_C`:
  - If the *external_name* is specified, it is used verbatim as the link name.
  - If no *external_name* is specified, the entity's simple Ada name is used as the link name converted to lowercase.
- `Restricted_Fortran`:

  If the *external_name* is specified then:
  - Any object where the *external_name* contains a '`/`' will be interpreted as a Fortran datapool element. The link name is obtained by appending two underscores after the datapool name and by prepending a '`$`' to the datapool element. ie. an *external_name* of "`/dp/aa`" will be transformed to a link name of "`dp__$aa`".
  - For all other entities the link name will be obtained by appending an underscore to the *external_name*.

  If the *external_name* is not specified then:
  - For all other entities the link name will be obtained by appending an underscore to the entity's simple Ada name.

**B.1(37) The effect of pragma Linker_Options**

Pragma `LINKER_OPTIONS` has one required parameter, a string within quotes containing the link options to be passed to the linker (**`a.link`**). Multiple link options within this string can be separated by spaces or tabs.

Link options specified within a compilation unit via this pragma will be added to the set of linker options for the resultant partition. The ordering of link options within a compilation unit will be preserved. But the ordering of link options between units is chosen arbitrarily. Link options specified by this pragma within mulitiple compilation units are arbitrarily combined and added to the set of link options for the resultant partition.

Any conflicts (such as those between **`-trace`**/**`-notrace`**) will be resolved as necessary.

Link options **-bound**, **-multiplexed**, **-skipobscurity**, **-nosoclosure**, and **-forgive** are not supported by this pragma in this release of MAXAda.

See "Pragma LINKER_OPTIONS" on page M-119 for more details about this pragma.

See "Link Options" on page 4-109 for more information about link options.

## Implementation Advice

**B.1(39)** **If an implementation supports pragma Export to a given language, then it should also allow the main subprogram to be written in that language. It should support some mechanism for invoking the elaboration of the Ada library units included in the system, and for invoking the finalization of the environment task. On typical systems, the recommended mechanism is to provide two subprograms whose link names are "adainit" and "adafinal". Adainit should contain the elaboration code for library units. Adafinal should contain the finalization code. These subprograms should have no effect the second and subsequent time they are called.**

MAXAda follows this advice with the following exceptions:

- The user is not forced to use the names `adainit` and `adafinal` for the subprograms. These names are user-configurable and are specified at the time of partition creation. The elaboration and finalization routines for these units may be called by the user. Of course, the user is free to choose the names `adainit` and `adafinal`.

  Optionally, the user may specify that these routines are automatically called. In this case, MAXAda will create elaboration and finalization routines with internal names not available to the user.

  See "a.partition" on page 4-62 and "Elaboration and Finalization Methods" on page 3-16 for more information.

- Calling `adainit` and `adafinal` more than once is defined differently than described by the RM. A second (or subsequent) call to `adainit` will not generally cause the elaboration library units to be elaborated. It will, however, be remembered that the second (or subsequent) call occurred. Calls to `adafinal` will not have an effect until it has been called an equal number of times as `adainit` was called. Calls to `adainit` after an effective call to `adafinal` will cause re-elaboration to occur. This results in nested elaboration/finalization behavior. For example, if the `adainit` and `adafinal` routines are called as described below, only those marked as `effective` will actually elaborate or finalize library units:

```
adainit         -- effective
   adainit
      adainit
      adafinal
      adainit
      adafinal
   adafinal
   adainit
      adainit
```

```
      adafinal
   adafinal
adafinal       -- effective
adainit        -- effective
   adainit
   adafinal
adafinal       -- effective
```

See "Elaboration and Finalization Methods" on page 3-16 for more information.

**B.1(40)** **Automatic elaboration of preelaborated packages should be provided when pragma Export is supported.**

MAXAda does not follow this advice.

**B.1(41)** **For each supported convention L other than Intrinsic, an implementation should support Import and Export pragmas for objects of L-compatible types and for subprograms, and pragma Convention for L-eligible types and for subprograms, presuming the other language has corresponding features. Pragma Convention need not be supported for scalar types.**

MAXAda supports the `IMPORT`, `EXPORT`, and `CONVENTION` pragmas for each of the supported conventions with the following restrictions applicable to all conventions:

- It is illegal to apply more than one of the interfacing pragmas to an entity.
- The `EXPORT` pragma cannot be applied to a *local_name* that is ambiguous.
- The `IMPORT`, `EXPORT` and `CONVENTION` pragma cannot be applied to formal parameters.
- The `IMPORT` and `EXPORT` pragma cannot be applied to components.

In addition, the following restrictions apply specifically to each of the conventions listed below:

- `Ada`
  - The `EXPORT` and `CONVENTION` pragmas can only be applied to subprograms that are declared at the library level.
- `Assembler`
  - The `EXPORT` and `CONVENTION` pragmas can only be applied to subprograms that are declared at the library level.
- `C`
  - The `EXPORT` and `CONVENTION` pragma are disallowed for subprograms containing unconstrained array formals and/or result types.
  - The `IMPORT`, `EXPORT`, and `CONVENTION` pragmas are disallowed for functions returning an array type.

- The `EXPORT` and `CONVENTION` pragma can only be applied to subprograms that are declared at the library level.
- The `IMPORT`, `EXPORT` and `CONVENTION` pragmas are disallowed for subprograms containing by-refence record formals of mode **in** or by-reference record return types.
- Private and incomplete types whose full type is not visible are not considered C-compatible.

In addition, only the types listed below are considered to be C-compatible:

- Scalar types with the exception of fixed-point types.
- `System.Address` and its derivatives.
- Array types with an unconstrained or a statically-constrained first subtype, if its component type is C-compatible.
- Non-tagged record types having components with statically-constrained subtypes, if each component type is C-compatible.
- Access-to-object type, if its designated type is C-compatible.
- Access-to-subprogram type, if its designated profile's parameter and result types are all C-compatible.
- A type derived from a C-compatible type.

• `Unchecked_C`

- No additional restrictions.

• `Restricted_Fortran`

- Private and incomplete types whose full type is not visible are not considered `Restricted_Fortran`-compatible.
- The `EXPORT` and `CONVENTION` pragma can only be applied to subprograms that are declared at the library level.

Only the types listed below are considered `Restricted_Fortran` compatible:

- `System.Address` and its derivatives.
- `Standard.Integer` and its derivatives.

# RM B.2 The Package Interfaces

**B.2(1)** **The contents of the visible part of package Interfaces and its language-defined descendants**

The following files contain the package `Interfaces` and its language-defined descendants. They can be found in **`/usr/ada/`***rel_name***`/predefined`** (where *rel_name* is the name of the MAXAda release).

- `Interfaces.a`
- `Interfaces.C.a`
- `Interfaces.C.Pointers.a`
- `Interfaces.C.Strings.a`

Implementation Permissions

**B.2(11)** **Implementation-defined children of package Interfaces. The contents of the visible part of package Interfaces**

The implementation-defined children of package `Interfaces` are:

- `Interfaces.Restricted_Fortran.a`
- `Interfaces.Unchecked_C.a`

The contents of the visible part of package `Interfaces` can be found in **`/usr/ada/`***rel_name***`/predefined/interfaces.a`** (where *rel_name* is the name of the MAXAda release).

Implementation Advice

**B.2(12)** **For each implementation-defined convention identifier, there should be a child package of package Interfaces with the corresponding name. This package should contain any declarations that would be useful for interfacing to the language (implementation) represented by the convention. Any declarations useful for interfacing to any language on the given hardware architecture should be provided directly in Interfaces.**

MAXAda provides the implementation-defined packages:

- `Interfaces.Restricted_Fortran`
- `Interfaces.Unchecked_C`

MAXAda does not provide a child package of package `Interfaces` for the convention identifier `Assembler`.

**B.2(13)** **An implementation supporting an interface to C, COBOL, or Fortran should provide the corresponding package or packages described in the following clauses.**

MAXAda follows this advice.

## RM B.3 Interfacing with C

Implementation Advice

**B.3(63)** **An implementation should support the following interface correspondences between Ada and C.**

**B.3(64)** **An Ada procedure corresponds to a void-returning C function.**

MAXAda follows this advice.

**B.3(65)** **An Ada function corresponds to a non-void C function.**

MAXAda follows this advice.

**B.3(66)** **An Ada in scalar parameter is passed as a scalar argument to a C function.**

MAXAda follows this advice.

An Ada **in** scalar parameter is passed by value. Some types may have to be promoted when they are passed to a C function.

Discrete types whose size is smaller than `Standard.Integer` are promoted to `Standard.Integer`. For example, `Interfaces.C.Short` is promoted to `Standard.Integer`.

**B.3(67)** **An Ada in parameter of an access-to-object type with designated type T is passed as a t* argument to a C function, where t is the C type corresponding to the Ada type T.**

MAXAda follows this advice.

**B.3(68)** **An Ada access T parameter, or an Ada out or in out parameter of an elementary type T, is passed as a t* argument to a C function, where t is the C type corresponding to the Ada type T. In the case of an elementary out or in out parameter, a pointer to a temporary copy is used to preserve by-copy semantics.**

MAXAda follows this advice.

**B.3(69)** **An Ada parameter of a record type T, of any mode, is passed as a t* argument to a C function, where t is the C struct corresponding to the Ada type T.**

MAXAda does not follow this advice.

MAXAda implements this in the following manner:

- An Ada parameter of a record type T, of mode **out** or **in out**, is passed as a t* argument to a C function, where t is the C struct corresponding to the Ada type T.

**B.3(70)** **An Ada parameter of an array type with component type T, of any mode, is passed as a t* argument to a C function, where t is the C type corresponding to the Ada type T.**

MAXAda follows this advice.

**B.3(71)** **An Ada parameter of an access-to-subprogram type is passed as a pointer to a C function whose prototype corresponds to the designated subprogram's specification.**

MAXAda follows this advice.

Notes

Additional conventions not listed above are as follows:

`C`

Parameter passing conventions:

- `System.Address` and its derivatives are passed by value.

Return conventions:

- Elementary types are returned by value.
- Composite types are returned by copying the contents of the composite object to the address passed in by the caller as the dummy first argument.

`Unchecked_C`

Same as the `C` convention for `C`-compatible entities. All other entities are handled as by the `Ada` convention. Exceptions to non `C`-compatible entities being passed as by the `Ada` convention are listed below:

- Records and arrays are passed by reference; however, no additional information (ie. dope vectors and constraint flags) is passed.

## RM B.4 Interfacing with COBOL

### Static Semantics

**B.4(50)** **The types Floating, Long_Floating, Binary, Long_Binary, Decimal_Element, and COBOL_Character; and the initializations of the variables Ada_To_COBOL and COBOL_To_Ada, in Interfaces.COBOL**

Interfaces.COBOL is not supported by MAXAda.

## RM B.5 Interfacing with Fortran

### Implementation Advice

**B.5(22)** **An Ada implementation should support the following interface correspondences between Ada and Fortran:**

**B.5(23)** **An Ada procedure corresponds to a Fortran subroutine.**

MAXAda does not support the Fortran convention in the current release.

**B.5(24)** **An Ada function corresponds to a Fortran function.**

MAXAda does not support the Fortran convention in the current release.

**B.5(25)** **An Ada parameter of an elementary, array, or record type T is passed as a Tf argument to a Fortran procedure, where Tf is the Fortran type corresponding to the Ada type T, and where the INTENT attribute of the corresponding dummy argument matches the Ada formal parameter mode; the Fortran**

**implementation's parameter passing conventions are used. For elementary types, a local copy is used if necessary to ensure by-copy semantics.**

MAXAda does not support the Fortran convention in the current release.

**B.5(26)** **An Ada parameter of an access-to-subprogram type is passed as a reference to a Fortran procedure whose interface corresponds to the designated subprogram's specification.**

MAXAda does not support the Fortran convention in the current release.

## Notes

Additional conventions not listed above are as follows:

`Restricted_Fortran`

- All `Restricted_Fortran` compatible entities are passed by value.

# RM Annex C: Systems Programming

## RM C.1 Access to Machine Operations

**C.1(1)** **Support for access to machine instructions**

Support for access to machine instructions is provided through the `System.Machine_Code` package in the predefined environment.

### Implementation Advice

**C.1(3)** **The machine code or intrinsics support should allow access to all operations normally available to assembly language programmers for the target environment, including privileged instructions, if any.**

A list of these instructions may be found in the `System.Machine_Code` package. This list can also be found in the "Pentium Instruction Set" on page M-49.

**C.1(4)** **The interfacing pragmas (see Annex B) should support interface to assembler; the default assembler should be associated with the convention identifier Assembler.**

MAXAda follows this advice.

**C.1(5)** **If an entity is exported to assembly language, then the implementation should allocate it at an addressable location, and should ensure that it is retained by the linking process, even if not otherwise referenced from the Ada code. The implementation should assume that any call to a machine code or assembler subprogram is allowed to read or update every object that is specified as exported.**

MAXAda follows this advice.

### Documentation Requirements

**C.1(6)** **The implementation shall document the overhead associated with calling machine-code or intrinsic subprograms, as compared to a fully-inlined call, and to a regular out-of-line call.**

This information has not yet been documented.

**C.1(7)** **The implementation shall document the types of the package System.Machine_Code usable for machine code insertions, and the attributes to be used in machine code insertions for references to Ada entities.**

**C.1(8)** **The implementation shall document the subprogram calling conventions associated with the convention identifiers available for use with the interfacing pragmas (Ada and Assembler, at a minimum), including register saving, exception propagation, parameter passing, and function value returning.**

This information has not yet been documented.

**C.1(9)** **Implementation-defined aspects of access to machine operations**

This information has not yet been documented.

Implementation Advice

**C.1(10)** **The implementation should ensure that little or no overhead is associated with calling intrinsic and machine-code subprograms.**

MAXAda follows this advice.

**C.1(11)** **It is recommended that intrinsic subprograms be provided for convenient access to any machine operations that provide special capabilities or efficiency and that are not otherwise available through the language constructs. Examples of such instructions include:**

**C.1(12)** **Atomic read-modify-write operations -- e.g., test and set, compare and swap, decrement and test, enqueue/dequeue.**

MAXAda provides the `indivisible_operations` package in the **`vendorlib`** environment which contains the following subprograms:

```
test_and_set
fetch_and_store
fetch_and_add
fetch_and_increment
increment
decrement
```

**C.1(13)** **Standard numeric functions -- e.g., sin, log.**

MAXAda does not supply any intrinsic subprograms for convenient access to standard numeric functions.

**C.1(14)** **String manipulation operations -- e.g., translate and test.**

MAXAda does not supply any intrinsic subprograms for convenient access to string manipulation operations.

**C.1(15)** **Vector operations -- e.g., compare vector against thresholds.**

MAXAda does not supply any intrinsic subprograms for convenient access to vector operations.

**C.1(16)** **Direct operations on I/O ports.**

MAXAda does not supply any intrinsic subprograms for convenient access to direct operations on I/O ports.

## RM C.3 The Package Interrupts

Dynamic Semantics

**C.3(2)** **Implementation-defined aspects of interrupts**

MAXAda supports two forms of interrupts: software and hardware. Software interrupts are operating system signals (see **`sigaction(2)`**). Hardware interrupts are machine-generated interrupts from devices such as real-time clocks, edge triggered devices, etc. Hardware interrupts are identified by names in the `Ada.Interrupts.Names` package.

Consult the user-defined package `Ada.Interrupts.Services` and Chapter 7 - Interrupt Handling for important information on using interrupts.

## Documentation Requirements

**C.3(12)** **The implementation shall document the following items:**

**C.3(13)** **For each interrupt, which interrupts are blocked from delivery when a handler attached to that interrupt executes (either as a result of an interrupt delivery or of an ordinary call on a procedure of the corresponding protected object).**

See Chapter 7 - Interrupt Handling.

**C.3(14)** **Any interrupts that cannot be blocked, and the effect of attaching handlers to such interrupts, if this is permitted.**

All interrupts that can be attached can be blocked. See Chapter 7 - Interrupt Handling.

**C.3(15)** **Which run-time stack an interrupt handler uses when it executes as a result of an interrupt delivery; if this is configurable, what is the mechanism to do so; how to specify how much space to reserve on that stack.**

The run-time stack for the COURIER or INTR_COURIER task associated with the attached interrupt. The size of the stack for such tasks can be adjusted by the user via application of pragma POOL_SIZE. See “Pragma POOL_SIZE” on page 6-26.

**C.3(16)** **Any implementation- or hardware-specific activity that happens before a user-defined interrupt handler gets control (e.g., reading device registers, acknowledging devices).**

See Chapter 7 - Interrupt Handling.

**C.3(17)** **Any timing or other limitations imposed on the execution of interrupt handlers.**

No limitations are imposed for software interrupt handlers.

Severe limitations are imposed for hardware interrupt handlers operating in restricted mode. See Chapter 7 - Interrupt Handling.

**C.3(18)** **The state (blocked/unblocked) of the non-reserved interrupts when the program starts; if some interrupts are unblocked, what is the mechanism a program can use to protect itself before it can attach the corresponding handlers.**

For software interrupts, all non-reserved signals are blocked for all tasks (including the `ENVIRONMENT` task).

For hardware interrupts, no interrupts are blocked, however, no hardware interrupts are ever delivered to a program unless the program has an attachment to them. Hardware interrupts can be restricted to specific CPUs in some machine configurations. See the "Real-Time Performance" chapter in the *RedHawk Linux User's Guide* (0898004).

**C.3(19)** **Whether the interrupted task is allowed to resume execution before the interrupt handler returns.**

Tasks which are preempted by an interrupt may resume execution before the interrupt handler returns if sufficient system resources are available. Tasks which are interrupted (in the sense that an interrupt causes them to execute code they would not otherwise execute) will notify the appropriate implementation-defined COURIER task and then resume execution based on system resources and priority. See Chapter 7 - Interrupt Handling for more information.

**C.3(20)** **The treatment of interrupt occurrences that are generated while the interrupt is blocked; i.e., whether one or more occurrences are held for later delivery, or all are lost.**

See Chapter 7 - Interrupt Handling for more information.

**C.3(21)** **Whether predefined or implementation-defined exceptions are raised as a result of the occurrence of any interrupt, and the mapping between the machine interrupts (or traps) and the predefined exceptions.**

No exceptions are raised upon occurence of a non-reserved interrupt. The reserved interrupts, `SIGFPE` and `SIGSEGV`, cause the exceptions `Constraint_Error` and `Storage_Error` to be raised, respectively.

**C.3(22)** **On a multi-processor, the rules governing the delivery of an interrupt to a particular processor.**

For software interrupts, the processor affected is the processor which is running the task to be interrupted.

For hardware interrupts, the system configuration defines which processor is interrupted. See the "Real-Time Performance" chapter in the *RedHawk Linux User's Guide* (0898004).

## Implementation Advice

**C.3(28)** **If the Ceiling_Locking policy is not in effect, the implementation should provide means for the application to specify which interrupts are to be blocked during protected actions, if the underlying system allows for a finer-grain control of interrupt blocking.**

The `CEILING_LOCKING` policy is the only locking policy currently supported.

## RM C.3.1 Protected Procedure Handlers

Implementation Advice

**C.3.1(20)** **Whenever possible, the implementation should allow interrupt handlers to be called directly by the hardware.**

There are no facilities for user programs to directly handle hardware interrupts under RedHawk Linux. The INTR_COURIER tasks block in `ioctl` calls waiting for the corresponding kernel device driver to notify them when an interrupt occurs.

**C.3.1(21)** **Whenever practical, the implementation should detect violations of any implementation-defined restrictions before run time.**

MAXAda generally does not follow this advice.

## RM C.3.2 The Package Interrupts

Documentation Requirements

**C.3.2(24)** **If the Ceiling_Locking policy (see D.3) is in effect the implementation shall document the default ceiling priority assigned to a protected object that contains either the Attach_Handler or Interrupt_Handler pragmas, but not the Interrupt_Priority pragma. This default need not be the same for all interrupts.**

The ceiling priority for protected objects with `ATTACH_HANDLER` or `INTERRUPT_HANDLER` pragmas but not `INTERRUPT_PRIORITY` pragma is `System.Interrupt_Priority'First`.

Implementation Advice

**C.3.2(25)** **If implementation-defined forms of interrupt handler procedures are supported, such as protected procedures with parameters, then for each such form of a handler, a type analogous to Parameterless_Handler should be specified in a child package of Interrupts, with the same operations as in the predefined package Interrupts.**

MAXAda does not provide implementation-defined forms of protected procedure handlers.

## RM C.4 Preelaboration Requirements

Documentation Requirements

**C.4(12)** **The implementation shall document any circumstances under which the elaboration of a preelaborated package causes code to be executed at run time.**

Preelaboration is not fully implemented in this release of MAXAda. Specific documentation on when code is generated for Preelaborated packages is not currently available. Generally, application of pragma Preelaborate does not affect whether code is generated for the elaboration of such packages. However, MAXAda adheres

to all legality rules for this pragma and elaborates all "preelaborated" packages before any other packages.

**C.4(13)** **Implementation-defined aspects of preelaboration**

Preelaboration is not fully implemented in this release of MAXAda.

Implementation Advice

**C.4(14)** **It is recommended that preelaborated packages be implemented in such a way that there should be little or no code executed at run time for the elaboration of entities not already covered by the Implementation Requirements.**

MAXAda does not follow this advice.

## RM C.5 Pragma Discard_Names

Static Semantics

**C.5(7)** **The semantics of pragma Discard_Names**

In the current release of MAXAda, pragma DISCARD_NAMES does not reduce the storage of entities to which it is applied. Therefore, the semantics of the various attributes and functions (`'Wide_Image`, `'Wide_Value`, `Text_IO.Enumeration_IO`, `Tags.Expanded_Name`, and `Exceptions.Exception_Name`) are the same as if the pragma had not been applied.

Implementation Advice

**C.5(8)** **If the pragma applies to an entity, then the implementation should reduce the amount of storage used for storing names associated with that entity.**

MAXAda does not follow this advice. In the current release of MAXAda, pragma DISCARD_NAMES does not reduce the storage of entities to which it is applied.

## RM C.7.1 The Package Task_Identification

Dynamic Semantics

**C.7.1(7)** **The result of the Task_Identification.Image attribute**

For an actual parameter which is a non-null `Task_ID` obtained from:

- a task object declared by a single_task_declaration:

  `Image` returns the defining_identifier that appears in the corresponding single_task_declaration.

- a stand-alone variable or constant of a task subtype:

`Image` returns the defining_identifier from the corresponding object_declaration.

- any other expression of a task subtype (after any implicit dereference):

  `Image` returns the defining_identifier from the corresponding task_type_declaration.

A `Task_ID` is obtained from a task using the `Ada.Task_Identification.Current_Task` function, or by applying the `'Identity` attribute to the task object.

### Bounded (Run-Time) Errors

**C.7.1(17)** **The value of Current_Task when in a protected entry or interrupt handler**

`Current_Task` returns the task ID of whatever task is actually executing the protected entry at the time of the call (this is not necessarily the task which made the entry call).

`Current_Task` returns `Null_Task_ID` when called from a protected procedure interrupt handler (during the execution of the interrupt).

### Documentation Requirements

**C.7.1(19)** **The effect of calling Current_Task from an entry body or interrupt handler**

Calling `Current_Task` from an entry body returns the task ID associated with the task actually executing the entry body (this is not necessarily the task which made the associated entry call).

Calling `Current_Task` from a protected procedure interrupt handler results in the value `Null_Task_ID` being returned (during the execution of the interrupt).

## RM C.7.2 The Package Task_Attributes

### Documentation Requirements

**C.7.2(18)** **The implementation shall document the limit on the number of attributes per task, if any, and the limit on the total storage for attribute values per task, if such a limit exists.**

There are no limits on the number of attributes per task or the total storage for attributes values per task.

**C.7.2(19)** **In addition, if these limits can be configured, the implementation shall document how to configure them (Implementation-defined aspects of Task_Attributes)**

There are no such limits.

Implementation Advice

**C.7.2(30)** **Some implementations are targeted to domains in which memory use at run time must be completely deterministic. For such implementations, it is recommended that the storage for task attributes will be pre-allocated statically and not from the heap. This can be accomplished by either placing restrictions on the number and the size of the task's attributes, or by using the pre-allocated storage for the first N attribute objects, and the heap for the others. In the latter case, N should be documented.**

MAXAda currently uses dynamic heap allocation for the implementation of task attributes.

# RM Annex D: Real-Time Systems

Metrics

**D(2)** **Values of all Metrics**

The metrics as required by Annex D are not available at the time of this release of MAXAda.

## RM D.1 Task Priorities

Static Semantics

**D.1(11)** **The declarations of Any_Priority and Priority**

The following declarations appear in package `System`:

```
subtype any_priority is integer      range   0..98 ;
subtype priority     is any_priority range   0..100 ;
```

Dynamic Semantics

**D.1(15)** **Implementation-defined execution resources**

## RM D.2.1 The Task Dispatching Model

Dynamic Semantics

**D.2.1(3)** **Whether, on a multiprocessor, a task that is waiting for access to a protected object keeps its processor busy**

This depends on the locking policy in use. The only available locking policy in this release is `CEILING_LOCKING`. A task with this locking policy that is waiting for access to a protected object keeps its processor busy.

Implementation Permissions

**D.2.1(9)** **The affect of implementation defined execution resources on task dispatching**

## RM D.2.2 The Standard Task Dispatching Policy

Legality Rules

**D.2.2(3)** **Implementation-defined policy_identifiers allowed in a pragma Task_Dispatching_Policy**

The following *policy*_identifiers are valid for pragma task_dispatching_policy:

- DEFAULT
- FIFO_WITHIN_PRIORITIES
- ROUND_ROBIN_PRIORITIES
- ROUND_ROBIN_ADJUSTABLE_PRIORITIES

See "Pragma TASK_DISPATCHING_POLICY" on page 6-2 for a detailed description of these policy identifiers.

### Documentation Requirements

**D.2.2(14)** **Priority inversion is the duration for which a task remains at the head of the highest priority ready queue while the processor executes a lower priority task. The implementation shall document:**

**D.2.2(15)** **The maximum priority inversion a user task can experience due to activity of the implementation (on behalf of lower priority tasks), and**

**D.2.2(16)** **Implementation-defined aspects of priority inversion**

Documentation on priority inversion durations is not yet availble.

### Implementation Permissions

**D.2.2(18)** **Implementation defined task dispatching**

## RM D.3 Priority Ceiling Locking

### Legality Rules

**D.3(4)** **Implementation-defined policy_identifiers allowed in a pragma Locking_Policy**

The following *policy*_identifiers are valid for pragma LOCKING_POLICY:

- CEILING_LOCKING
- SLEEPY_CEILING_LOCKING - (not yet implemented)
- SLEEPY_INHERITANCE_LOCKING - (not yet implemented)

See "Pragma LOCKING_POLICY" on page 6-3 for a detailed description of these policy identifiers.

### Dynamic Semantics

**D.3(10)** **Default ceiling priorities**

The default ceiling priority is Interrupt_Priority'First.

### Implementation Permissions

**D.3(16)** **The ceiling of any protected object used internally by the implementation**

This is not applicable to MAXAda.

### Implementation Advice

**D.3(17)** **The implementation should use names that end with ``_Locking'' for implementation-defined locking policies.**

The following *policy*_identifiers are valid for pragma `LOCKING_POLICY`:

- `CEILING_LOCKING`
- `SLEEPY_CEILING_LOCKING` - (not yet implemented)
- `SLEEPY_INHERITANCE_LOCKING` - (not yet implemented)

## RM D.4 Entry Queuing Policies

**D.4(1)** **Implementation-defined queuing policies**

MAXAda supports only those queuing policies as defined by the Ada 95 Reference Manual. They are:

- `FIFO_QUEUING`
- `PRIORITY_QUEUING`

These are defined in the Ada 95 Reference Manual, Section D.4. The default is `FIFO_QUEUING.`

There are no other implementation-defined queuing policies.

### Implementation Advice

**D.4(16)** **The implementation should use names that end with ``_Queuing'' for implementation-defined queuing policies.**

MAXAda follows this advice.

## RM D.6 Preemptive Abort

### Documentation Requirements

**D.6(3)** **On a multiprocessor, any conditions that cause the completion of an aborted construct to be delayed later than what is specified for a single processor**

Documentation on preemptive abort is not yet available.

Implementation Advice

**D.6(9)** **Even though the abort_statement is included in the list of potentially blocking operations (see 9.5.1), it is recommended that this statement be implemented in a way that never requires the task executing the abort_ statement to block.**

**D.6(10)** **On a multi-processor, the delay associated with aborting a task on another processor should be bounded; the implementation should use periodic polling, if necessary, to achieve this.**

## RM D.7 Tasking Restrictions

Static Semantics

**D.7(8)** **Any operations that implicitly require heap storage allocation**

MAXAda performs dynamic implicit heap allocations for the following operations:

- creation of a task, or object of a type with task parts
- creation of a protected object, or object of a type with protected parts
- creation of an object with controlled parts
- creation of a package body stub
- elaboration of a package instance whose corresponding generic is not declared within the same compilation unit as the instance or is separate
- elaboration of an instance of `Ada.Task_Attributes`
- elaboration of a shared instance whose generic environment (the memory space containing information required to differentiate a shared instance from other shared instances of the same generic) is larger than 51.2 Kb. See “Pragma SHARE_BODY” on page M-129.
- call to the function `Ada.Exceptions.Save_Occurrence` (but not the procedure)
- elaboration of a master, other than that associated with the `ENVIRONMENT` task, which contains any of the following declarations:
  - access type
  - separate body
  - instance whose corresponding generic is not declared within the same compilation unit as the instance or is separate
- any of the following operations performed at library-level (i.e. any operation not performed within a subprogram or task):
  - creation of an object of a dynamically constrained type
  - conversion of a value of a dynamically constrained type

- string catenation producing a dynamically constrained result
- non-string catenation
- logical or "not" operator expression involving dynamically constrained arrays of booleans
- copy of a dynamically sized bit-aligned actual used for parameter passing
- copy of a dynamically sized atomic actual whose corresponding formal type is not atomic (see RM C.6(19))
- call of an `'Input` attribute whose prefix is a composite type
- call of an instance of `Ada.Unchecked_Conversion` with a dynamically constrained target type
- elaboration of a `'Storage_Size` representation clause

### Dynamic Semantics

**D.7(20)** **Implementation-defined aspects of pragma Restrictions**

The effects of the use of pragma RESTRICTIONS are described in the section titled “Pragma RESTRICTIONS” on page M-128.

### Implementation Advice

**D.7(21)** **When feasible, the implementation should take advantage of the specified restrictions to produce a more efficient implementation.**

The specified restrictions have no effect upon the run-time in this release. A future release will optimize the run-time based upon which restrictions are present.

## RM D.8 Monotonic Time

### Static Semantics

**D.8(17)** **Implementation-defined aspects of package Real_Time**

Most of the implementation-defined aspects are dependent on the speed of the specific system.

The following table provides a representative example of a Pentium system:

| | |
|---|---|
| `Time_Unit` | 0.7 ns |
| `Tick` | 0.7 ns |
| `Time_Span_First`<br>`Time_First` | -6.6E+09 sec |
| `Time_Span_Last`<br>`Time_Last` | 6.6E+09 sec |

| | |
|---|---|
| Clock Jump | 63 ns |
| Epoch | 4/25/2003@00:00:00 |
| Years Representable | 1793 .. 2213 |

The above implementation-defined items are declared in package `Ada.Real_Time`.

## Documentation Requirements

**D.8(33)** **The implementation shall document the values of Time_First, Time_Last, Time_Span_First, Time_Span_Last, Time_Span_Unit, and Tick.**

All of the following values are internally represented using a record which emulates a signed 64-bit number of machine clock ticks:

```
Time_First := (low => 0, high => -2**31-1)
Time_Last := (low => 2**32-1, high => 2**31-1)
Time_Span_First := (low => 0, high => -2**31-1)
Time_Span_Last := (low => 2**32-1, high => 2**31-1)
Time_Span_Zero := (low => 0, high => 0)
Time_Span_Unit := (low => 1, high => 0)
Tick := (low => 1, high => 0)
```

**D.8(34)** **The implementation shall document the properties of the underlying time base used for the clock and for type Time, such as the range of values supported and any relevant aspects of the underlying hardware or operating system facilities used.**

| System Type | Time Source |
|---|---|
| iHawk series | Pentium Time Stamp Counter (TSC) register |

**D.8(35)** **The implementation shall document whether or not there is any synchronization with external time references, and if such synchronization exists, the sources of synchronization information, the frequency of synchronization, and the synchronization method applied.**

The clock base is zeroed upon system power up and is not synchronized subsequently.

**D.8(36)** **The implementation shall document any aspects of the the external environment that could interfere with the clock behavior as defined in this clause.**

There are no known aspects of the external environment that could interfere with clock behavior.

Implementation Advice

**D.8(47)** **When appropriate, implementations should provide configuration mechanisms to change the value of Tick.**

It is not appropriate to change the value of `Tick`; as such, no mechanism is provided to do that.

**D.8(48)** **It is recommended that Calendar.Clock and Real_Time.Clock be implemented as transformations of the same time base.**

`Calendar.Clock` is implemented as a set of transformations of the underlying time base for `Ada.Real_Time.clock`.

**D.8(49)** **It is recommended that the "best" time base which exists in the underlying system be available to the application through Clock. "Best" may mean highest accuracy or largest range.**

The time base (as shown in the table under **D.8(34)**) is the "best" time base which exists in the underlying hardware. It has both the highest accuracy and the largest range of any time base available on the system.

## RM D.9 Delay Accuracy

Documentation Requirements

**D.9(7)** **The implementation shall document the minimum value of the delay expression of a delay_relative_statement that causes the task to actually be blocked.**

**D.9(8)** **Implementation-defined aspects of delay_statements**

## RM D.12 Other Optimizations and Determinism Rules

Documentation Requirements

**D.12(5)** **The upper bound on the duration of interrupt blocking caused by the implementation**

# RM Annex G: Numerics

Implementation Advice

**G(7)** **If Fortran (respectively, C) is widely supported in the target environment, implementations supporting the Numerics Annex should provide the child package Interfaces.Fortran (respectively, Interfaces.C) specified in Annex B and should support a convention_identifier of Fortran (respectively, C) in the interfacing pragmas (see Annex B), thus allowing Ada programs to interface with programs written in that language.**

`Interfaces.C` is supplied by MAXAda. Accordingly, the convention_identifier `C` is supported by the interfacing pragmas.

## RM G.1 Complex Arithmetic

**G.1(1)** **The accuracy actually achieved by the complex elementary functions and by other complex arithmetic operations**

MAXAda does not provide complex arithmetic packages in this release.

## RM G.1.1 Complex Types

Implementation Requirements

**G.1.1(53)** **The sign of a zero result (or a component thereof) from any operator or function in Numerics.Generic_Complex_Types, when Real'Signed_Zeros is True**

MAXAda does not provide complex arithmetic packages in this release.

Implementation Advice

**G.1.1(56)** **Because the usual mathematical meaning of multiplication of a complex operand and a real operand is that of the scaling of both components of the former by the latter, an implementation should not perform this operation by first promoting the real operand to complex type and then performing a full complex multiplication. In systems that, in the future, support an Ada binding to IEC 559:1989, the latter technique will not generate the required result when one of the components of the complex operand is infinite. (Explicit multiplication of the infinite component by the zero component obtained during promotion yields a NaN that propagates into the final result.) Analogous advice applies in the case of multiplication of a complex operand and a pure-imaginary operand, and in the case of division of a complex operand by a real or pure-imaginary operand.**

MAXAda does not provide complex arithmetic packages in this release.

**G.1.1(57)** **Likewise, because the usual mathematical meaning of addition of a complex operand and a real operand is that the imaginary operand remains unchanged, an implementation should not perform this operation by first promoting the real operand to com-**

**plex type and then performing a full complex addition. In implementations in which the Signed_Zeros attribute of the component type is True (and which therefore conform to IEC 559:1989 in regard to the handling of the sign of zero in predefined arithmetic operations), the latter technique will not generate the required result when the imaginary component of the complex operand is a negatively signed zero. (Explicit addition of the negative zero to the zero obtained during promotion yields a positive zero.) Analogous advice applies in the case of addition of a complex operand and a pure-imaginary operand, and in the case of subtraction of a complex operand and a real or pure-imaginary operand.**

MAXAda does not provide complex arithmetic packages in this release.

**G.1.1(58)** **Implementations in which Real'Signed_Zeros is True should attempt to provide a rational treatment of the signs of zero results and result components. As one example, the result of the Argument function should have the sign of the imaginary component of the parameter X when the point represented by that parameter lies on the positive real axis; as another, the sign of the imaginary component of the Compose_From_Polar function should be the same as (resp., the opposite of) that of the Argument parameter when that parameter has a value of zero and the Modulus parameter has a nonnegative (resp., negative) value.**

MAXAda does not provide complex arithmetic packages in this release.

## RM G.1.2 Complex Elementary Functions

### Implementation Requirements

**G.1.2(45)** **The sign of a zero result (or a component thereof) from any operator or function in Numerics.Generic_Complex_Elementary_Functions, when Complex_Types.Real'Signed_Zeros is True**

MAXAda does not provide complex arithmetic packages in this release.

### Implementation Advice

**G.1.2(49)** **Implementations in which Complex_Types.Real'Signed_Zeros is True should attempt to provide a rational treatment of the signs of zero results and result components. For example, many of the complex elementary functions have components that are odd functions of one of the parameter components; in these cases, the result component should have the sign of the parameter component at the origin. Other complex elementary functions have zero components whose sign is opposite that of a parameter component at the origin, or is always positive or always negative.**

MAXAda does not provide complex arithmetic packages in this release.

## RM G.2 Numeric Performance Requirements

### Implementation Permissions

**G.2(2)** **Whether the strict mode or the relaxed mode is the default**

MAXAda uses the relaxed mode as the default.

## RM G.2.1 Model of Floating Point Arithmetic

Implementation Requirements

**G.2.1(10)** **The result interval in certain cases of fixed-to-float conversion**

MAXAda does not support smalls that are not a power of 2 (`T'Machine_Radix`) so there are no implementation-defined result intervals.

**G.2.1(13)** **The result of a floating point arithmetic operation in overflow situations, when the Machine_Overflows attribute of the result type is False**

This is not applicable to MAXAda since `Machine_Overflows` is always `True`.

Implementation Permissions

**G.2.1(16)** **The result interval for division (or exponentiation by a negative exponent), when the floating point hardware implements division as multiplication by a reciprocal**

This is not applicable to MAXAda. Floating point division is based on exponent subtraction and division of significands.

## RM G.2.3 Model of Fixed Point Arithmetic

Implementation Requirements

**G.2.3(5)** **The definition of close result set, which determines the accuracy of certain fixed point multiplications and divisions**

**G.2.3(22)** **Conditions on a universal_real operand of a fixed point multiplication or division for which the result shall be in the perfect result set**

**G.2.3(27)** **The result of a fixed point arithmetic operation in overflow situations, when the Machine_Overflows attribute of the result type is False**

This is not applicable to MAXAda since `machine_overflows` is always `True`.

## RM G.2.4 Accuracy Requirements for the Elementary Functions

**G.2.4(4)** **The result of an elementary function reference in overflow situations, when the Machine_Overflows attribute of the result type is False**

This is not applicable to MAXAda since `machine_overflows` is always `True`.

**G.2.4(10)** **The value of the angle threshold, within which certain elementary functions, complex arithmetic operations, and complex elementary functions yield results conforming to a maximum relative error bound**

**The accuracy of certain elementary functions for parameters beyond the angle threshold**

Implementation Advice

**G.2.4(19)** **The versions of the forward trigonometric functions without a Cycle parameter should not be implemented by calling the corresponding version with a Cycle parameter of 2.0*Numerics.Pi, since this will not provide the required accuracy in some portions of the domain. For the same reason, the version of Log without a Base parameter should not be implemented by calling the corresponding version with a Base parameter of Numerics.e.**

MAXAda follows this advice.

## RM G.2.6 Accuracy Requirements for Complex Arithmetic

**G.2.6(5)** **The result of a complex arithmetic operation or complex elementary function reference in overflow situations, when the Machine_Overflows attribute of the corresponding real type is False**

This is not applicable to MAXAda since `Machine_Overflows` is always `True`.

**G.2.6(8)** **The accuracy of certain complex arithmetic operations and certain complex elementary functions for parameters (or components thereof) beyond the angle threshold**

MAXAda does not provide complex arithmetic packages in this release.

Implementation Advice

**G.2.6(15)** **The version of the Compose_From_Polar function without a Cycle parameter should not be implemented by calling the corresponding version with a Cycle parameter of 2.0*Numerics.Pi, since this will not provide the required accuracy in some portions of the domain.**

MAXAda does not provide complex arithmetic packages in this release.

# RM Annex J: Obsolescent Features

## RM J.7.1 Interrupt Entries

### Documentation Requirements

**J.7.1(12)** **The implementation shall document to which interrupts a task entry may be attached.**

**J.7.1(13)** **The implementation shall document whether the invocation of an interrupt entry has the effect of an ordinary entry call, conditional call, or a timed call, and whether the effect varies in the presence of pending interrupts.**

### Implementation Permissions

**J.7.1(17)** **The implementation is allowed to impose restrictions on the specifications and bodies of tasks that have interrupt entries.**

A *requeue_statement* is not allowed within the *handled_sequence_of_statements* of an *accept_statement* if the corresponding entry is an interrupt entry.

# RM Annex K: Language-Defined Attributes

The implementation-defined attributes of MAXAda are discussed in “4.1.4(12) Implementation-defined attributes” on page M-13.

# RM Annex L: Pragmas

The following lists all implementation-dependent and **implementation-defined** pragmas.


| | |
|---|---|
| Pragma ALL_CALLS_REMOTE - (not yet supported) | page M-106 |
| **Pragma ASSIGNMENT** | **page M-106** |
| Pragma ASYNCHRONOUS - (not yet supported) | page M-106 |
| Pragma ATOMIC | page M-106 |
| Pragma ATOMIC_COMPONENTS | page M-107 |
| Pragma ATTACH_HANDLER | page M-107 |
| Pragma CONTROLLED | page M-107 |
| Pragma CONVENTION | page M-108 |
| **Pragma DATA_RECORD - (obsolete)** | **page M-109** |
| **Pragma DEBUG** | **page M-109** |
| **Pragma DEPRECATED_FEATURE** | **page M-110** |
| Pragma DISCARD_NAMES | page M-110 |
| **Pragma DONT_ELABORATE** | **page M-110** |
| Pragma ELABORATE | page M-111 |
| Pragma ELABORATE_ALL | page M-111 |
| Pragma ELABORATE_BODY | page M-111 |
| Pragma EXPORT | page M-111 |
| **Pragma EXTERNAL_NAME - (obsolete)** | **page M-112** |
| **Pragma FAST_INTERRUPT_TASK** | **page M-113** |
| **Pragma GROUP_CPU_BIAS** | **page M-113** |
| **Pragma GROUP_PRIORITY** | **page M-113** |
| **Pragma GROUP_SERVERS** | **page M-114** |
| **Pragma IMPLICIT_CODE** | **page M-114** |
| Pragma IMPORT | page M-114 |
| Pragma INLINE | page M-115 |
| Pragma INSPECTION_POINT - (not yet supported) | page M-116 |
| **Pragma INTERESTING** | **page M-117** |
| **Pragma INTERFACE - (obsolete)** | **page M-117** |
| **Pragma INTERFACE_NAME - (obsolete)** | **page M-117** |
| **Pragma INTERFACE_OBJECT - (obsolete)** | **page M-118** |
| **Pragma INTERFACE_SHARED - (obsolete)** | **page M-118** |
| Pragma INTERRUPT_HANDLER | page M-118 |
| Pragma INTERRUPT_PRIORITY | page M-118 |
| **Pragma LINK_OPTION - (obsolete)** | **page M-119** |
| Pragma LINKER_OPTIONS | page M-119 |
| Pragma LIST | page M-119 |
| Pragma LOCKING_POLICY | page M-120 |
| **Pragma MAP_FILE** | **page M-120** |

**Pragma MEMORY_POOL** **page M-120**
Pragma NORMALIZE_SCALARS - (not yet supported) page M-121
**Pragma OPT_FLAGS** **page M-121**
**Pragma OPT_LEVEL** **page M-122**
Pragma OPTIMIZE page M-122
Pragma PACK page M-123
Pragma PAGE page M-123
**Pragma PASSIVE_TASK - (obsolete)** **page M-123**
**Pragma POOL_CACHE_MODE** **page M-124**
**Pragma POOL_LOCK_STATE** **page M-124**
**Pragma POOL_PAD** **page M-124**
**Pragma POOL_SIZE** **page M-124**
Pragma PREELABORATE page M-125
Pragma PRIORITY page M-125
**Pragma PROTECTED_PRIORITY** **page M-125**
Pragma PURE page M-127
Pragma QUEUING_POLICY page M-127
Pragma REMOTE_CALL_INTERFACE - (not yet supported) page M-127
Pragma REMOTE_TYPES - (not yet supported) page M-127
Pragma RESTRICTIONS page M-128
**Pragma RETURN_CONVENTION** **page M-128**
Pragma REVIEWABLE - (not yet supported) page M-129
**Pragma RUNTIME_DIAGNOSTICS** **page M-129**
**Pragma SERVER_CACHE_SIZE** **page M-129**
**Pragma SHARE_BODY** **page M-129**
**Pragma SHARE_MODE** **page M-130**
**Pragma SHARED - (obsolete)** **page M-131**
**Pragma SHARED_PACKAGE** **page M-131**
Pragma SHARED_PASSIVE - (not yet supported) page M-131
**Pragma SPECIAL_FEATURE** **page M-131**
Pragma STORAGE_SIZE page M-132
Pragma SUPPRESS page M-132
**Pragma SUPPRESS_ALL** **page M-133**
**Pragma TASK_CPU_BIAS** **page M-133**
Pragma TASK_DISPATCHING_POLICY page M-133
**Pragma TASK_HANDLER** **page M-134**
**Pragma TASK_PRIORITY** **page M-134**
**Pragma TASK_QUANTUM** **page M-134**
**Pragma TASK_WEIGHT** **page M-135**
**Pragma TDESC** **page M-135**
**Pragma TRAMPOLINE** **page M-135**
Pragma VOLATILE page M-135
Pragma VOLATILE_COMPONENTS page M-136

## Pragma ALL_CALLS_REMOTE - (not yet supported)

Pragma `ALL_CALLS_REMOTE` is not supported in this release.

See "RM Annex L: Pragmas" on page M-104 for a list of all implementation-dependent and implementation-defined pragmas.

## Pragma ASSIGNMENT

**NOTE**

Pragma `ASSIGNMENT` is reserved for internal MAXAda use only; it is not intended for use in user-defined code.

See "RM Annex L: Pragmas" on page M-104 for a list of all implementation-dependent and implementation-defined pragmas.

## Pragma ASYNCHRONOUS - (not yet supported)

Pragma `ASYNCHRONOUS` is not supported in this release.

See "RM Annex L: Pragmas" on page M-104 for a list of all implementation-dependent and implementation-defined pragmas.

## Pragma ATOMIC

Pragma `ATOMIC` is implemented as described in Section C.6 of the Ada 95 Reference Manual.

Its syntax is:

```
pragma ATOMIC(local_name);
```

This pragma accepts a single variable name which must be of a type which can be atomic for the pragma to apply. All reads and updates of an atomic object are indivisible. An atomic object is also defined to be volatile (see "Pragma VOLATILE" on page M-135).

Pragma `ATOMIC` should be used on any variable that may be modified concurrently by different threads of a program (e.g. semaphores).

See "RM Annex L: Pragmas" on page M-104 for a list of all implementation-dependent and implementation-defined pragmas.

# Pragma ATOMIC_COMPONENTS

Pragma ATOMIC_COMPONENTS is implemented as described in Section C.6 of the Ada 95 Reference Manual.

Its syntax is:

```
pragma ATOMIC_COMPONENTS(array_local_name);
```

This pragma accepts an array name, the components of which must be of a type which can be atomic for the pragma to apply. All reads and updates of an atomic object are indivisible. An atomic object is also defined to be volatile (see "Pragma VOLATILE" on page M-135).

Pragma ATOMIC_COMPONENTS should be used on variables that may be modified concurrently by different threads of a program (e.g. semaphores).

See "RM Annex L: Pragmas" on page M-104 for a list of all implementation-dependent and implementation-defined pragmas.

# Pragma ATTACH_HANDLER

Pragma ATTACH_HANDLER is implemented as described in Section C.3.1 of the Ada 95 Reference Manual.

Its syntax is:

```
pragma ATTACH_HANDLER(handler_name,expresion);
```

See "RM Annex L: Pragmas" on page M-104 for a list of all implementation-dependent and implementation-defined pragmas.

# Pragma CONTROLLED

Pragma CONTROLLED is recognized by the implementation but does not have an effect in this release.

Its syntax is:

```
pragma CONTROLLED(first_subtype_local_name);
```

Pragma CONTROLLED is used to prevent any automatic reclamation of storage for the objects created by allocators of a given access type.

This pragma accepts a single argument which shall be the defining identifier of a non-derived access type declaration.

Pragma CONTROLLED has no effect in this release of MAXAda as garbage collection is not supported. (Ada 95 Reference Manual 13.11.3(8))

See Section 13.11.3 of the Ada 95 Reference Manual for more information about this pragma.

See "RM Annex L: Pragmas" on page M-104 for a list of all implementation-dependent and implementation-defined pragmas.

## Pragma CONVENTION

Pragma `CONVENTION` is implemented as described in Section B.1 of the Ada 95 Reference Manual.

Its syntax is:

```
pragma CONVENTION([Convention=>]convention_identifier,
                  [Entity=>]local_name);
```

This pragma is used to specify that an Ada entity should use the conventions of another language. This pragma is referred to in the Ada 95 Reference Manual as an *interfacing pragma*.

An interfacing pragma defines the convention of the entity denoted by `local_name`. The convention represents the calling convention or representation convention of the entity.

The *convention*`_identifier` is the name of a *convention*. The convention names represent the calling conventions of foreign languages, language implementations, or specific run-time models.

The allowable conventions are:

- `Ada`
- `Assembler`
- `C`
- `Unchecked_C`
- `Restricted_Fortran`
- `Entry` (internal use only)
- `Intrinsic` (internal use only)
- `Protected` (internal use only)

See "RM B.1 Interfacing Pragmas" on page M-74 for details on the implementation of this pragma with respect to the Ada 95 Reference Manual.

See "RM Annex L: Pragmas" on page M-104 for a list of all implementation-dependent and implementation-defined pragmas.

# Pragma DATA_RECORD - (obsolete)

The implementation-defined pragma `DATA_RECORD` is obsolete. It will be removed in a future release and should not be used. Use "Pragma DEBUG" on page M-109 and possibly "Pragma INTERESTING" on page M-117 instead.

In this release, if pragma `DATA_RECORD` is used, pragma `DEBUG` will be activated instead.

See "RM Annex L: Pragmas" on page M-104 for a list of all implementation-dependent and implementation-defined pragmas.

# Pragma DEBUG

The implementation-defined pragma `DEBUG` specifies the debug level for a compilation unit from within the Ada source code.

Its syntax is:

```
pragma DEBUG([unit_name,] debug_level);
```

where *unit_name*, if specified, is the name of the compilation unit for which the debug level is being specified, and where *debug_level* is the debug level which should be used for that compilation unit. The possible values for *debug_level* are `NONE`, `LINES`, and `FULL`.

The single-parameter form of this pragma is allowed only immediately within a library unit or as a configuration pragma. When specified within a library unit, it applies only to that library unit. When specified as a configuration pragma, it applies to all units within the same compilation, if any, or to all units in the environment, if none. The two-parameter form of this pragma is allowed only immediately following the unit which is specified as the *unit_name* argument. It applies only to the unit which is specified.

If applied to a specification, the debug level does ***not*** apply to the body or any separate bodies of the unit. If applied to a body, the debug level does ***not*** apply to any separate bodies of the unit. If the debug level is desired for any such units, it must be specified for them, too.

The pragma is meaningless when applied to a generic unit. If so applied, it will not be applied to any instantiations of that generic. The debug level applied to an instantiation is the debug level of the unit which contains it, or if the instantiation is library-level, is determined in the same way as for any other library-level unit.

See "Real-Time Debugging" on page 3-38 for more information.

See "RM Annex L: Pragmas" on page M-104 for a list of all implementation-dependent and implementation-defined pragmas.

# Pragma DEPRECATED_FEATURE

**NOTE**

Pragma `DEPRECATED_FEATURE` is reserved for internal MAX-Ada use only; it is not intended for use in user-defined code.

This pragma marks packages that have been deprecated and may be significantly changed or completely removed in future releases of MAXAda. Whenever a compilation unit requires another unit (via a `with` clause) that has been marked with this pragma, a compiler alert (diagnostic) is issued. The alert consists of a standard MAXAda diagnostic header followed by the exact text of the string that is the single required argument of the pragma.

See "RM Annex L: Pragmas" on page M-104 for a list of all implementation-dependent and implementation-defined pragmas.

# Pragma DISCARD_NAMES

Pragma `DISCARD_NAMES` is recognized by the implementation but does not have an effect in this release.

Its syntax is:

```
pragma DISCARD_NAMES[([On=>]local_name)];
```

See Section C.5 of the Ada 95 Reference Manual for more information about this pragma.

See "RM Annex L: Pragmas" on page M-104 for a list of all implementation-dependent and implementation-defined pragmas.

# Pragma DONT_ELABORATE

The implementation-defined pragma `DONT_ELABORATE` prevents dynamic elaboration of any library units to which it applies.

Its syntax is:

```
pragma DONT_ELABORATE[(library_unit_name)];
```

See "RM Annex L: Pragmas" on page M-104 for a list of all implementation-dependent and implementation-defined pragmas.

## Pragma ELABORATE

Pragma ELABORATE is implemented as described in Section 10.2.1 of the Ada 95 Reference Manual.

Its syntax is:

```
pragma ELABORATE(library_unit_name{,library_unit_name});
```

See "RM Annex L: Pragmas" on page M-104 for a list of all implementation-dependent and implementation-defined pragmas.

## Pragma ELABORATE_ALL

Pragma ELABORATE_ALL is implemented as described in Section 10.2.1 of the Ada 95 Reference Manual.

Its syntax is:

```
pragma ELABORATE_ALL(library_unit_name{,library_unit_name});
```

See "RM Annex L: Pragmas" on page M-104 for a list of all implementation-dependent and implementation-defined pragmas.

## Pragma ELABORATE_BODY

Pragma ELABORATE_BODY is implemented as described in Section 10.2.1 of the Ada 95 Reference Manual.

Its syntax is:

```
pragma ELABORATE_BODY[(library_unit_name)];
```

See "RM Annex L: Pragmas" on page M-104 for a list of all implementation-dependent and implementation-defined pragmas.

## Pragma EXPORT

Pragma EXPORT is implemented as described in Section B.1 of the Ada 95 Reference Manual.

Its syntax is:

```
pragma EXPORT([Convention=>]convention_identifier,
              [Entity=>]local_name
              [,[External_Name=>]string_expression]
              [,[Link_Name=>]string_expression]);
```

This pragma is used to export an Ada entity to a foreign language, thus allowing an Ada subprogram to be called from a foreign language, or an Ada object to be accessed from a foreign language. This pragma is referred to in the Ada 95 Reference Manual as an *interfacing pragma*.

An interfacing pragma defines the convention of the entity denoted by `local_name`. The convention represents the calling convention or representation convention of the entity.

The *convention*`_identifier` is the name of a *convention*. The convention names represent the calling conventions of foreign languages, language implementations, or specific run-time models.

The allowable conventions are:

- `Ada`
- `Assembler`
- `C`
- `Unchecked_C`
- `Restricted_Fortran`
- `Entry` (internal use only)
- `Intrinsic` (internal use only)
- `Protected` (internal use only)

Pragma `EXPORT` optionally specifies an entity's external name, link name, or both.

An `External_Name` is a string value for the name used by the foreign language program for referring to an entity that an Ada program exports.

A `Link_Name` is a string value for the name of the exported entity, based on the conventions of the foreign language's compiler in interfacing with the system's linker tool.

The meaning of link names is implementation defined. If neither a link name nor the Address attribute of an imported entity is specified, then a link name is chosen in an implementation-defined manner, based on the external name if one is specified. If both the external name and the link name are specified, the external name is ignored.

See "RM B.1 Interfacing Pragmas" on page M-74 for details on the implementation of this pragma with respect to the Ada 95 Reference Manual.

See "RM Annex L: Pragmas" on page M-104 for a list of all implementation-dependent and implementation-defined pragmas.

## Pragma EXTERNAL_NAME - (obsolete)

The implementation-defined pragma `EXTERNAL_NAME` is obsolete. It will be removed in a future release and should not be used. Use "Pragma IMPORT" on page M-114 or "Pragma EXPORT" on page M-111 instead.

In this release, if pragma EXTERNAL_NAME is used with pragma INTERFACE (see "Pragma INTERFACE - (obsolete)" on page M-117), pragma IMPORT will be activated instead. Otherwise, pragma EXPORT will be activated instead.

See "RM Annex L: Pragmas" on page M-104 for a list of all implementation-dependent and implementation-defined pragmas.

## Pragma FAST_INTERRUPT_TASK

The implementation-defined pragma FAST_INTERRUPT_TASK provides extremely fast interrupt handling.

Its syntax is:

```
pragma FAST_INTERRUPT_TASK;
```

This pragma has no effect in this release.

See "RM Annex L: Pragmas" on page M-104 for a list of all implementation-dependent and implementation-defined pragmas.

## Pragma GROUP_CPU_BIAS

The implementation-defined pragma GROUP_CPU_BIAS specifies the CPU bias for all the servers in a given group.

Its syntax is:

```
pragma GROUP_CPU_BIAS(cpu_bias, group_spec) ;
```

See "Pragma GROUP_CPU_BIAS" on page 6-19 for a complete description.

See "RM Annex L: Pragmas" on page M-104 for a list of all implementation-dependent and implementation-defined pragmas.

## Pragma GROUP_PRIORITY

The implementation-defined pragma GROUP_PRIORITY specifies the operating system scheduling priority of all the servers in a given group.

Its syntax is:

```
pragma GROUP_PRIORITY(scheduling_priority, group_spec) ;
```

See "Pragma GROUP_PRIORITY" on page 6-18 for a complete description.

See "RM Annex L: Pragmas" on page M-104 for a list of all implementation-dependent and implementation-defined pragmas.

# Pragma GROUP_SERVERS

The implementation-defined pragma `GROUP_SERVERS` controls the number of servers for a particular group, including the `PREDEFINED` group.

Its syntax is:

```
pragma GROUP_SERVERS(group_size, group_spec) ;
```

See "Pragma GROUP_SERVERS" on page 6-19 for a complete description.

See "RM Annex L: Pragmas" on page M-104 for a list of all implementation-dependent and implementation-defined pragmas.

# Pragma IMPLICIT_CODE

The implementation-defined pragma `IMPLICIT_CODE` provides a way to eliminate the stack frame and the copying of parameters when using the `Machine_Code` package.

Its syntax is:

```
pragma IMPLICIT_CODE(flag) ;
```

This pragma takes a single argument (`ON` or `OFF`). When `OFF`, it does not generate code for the argument copies, nor does it generate any return code upon exiting. It can be used as an optimization for writing `Machine_Code` routines to eliminate the generation of the implicit code.

See "RM Annex L: Pragmas" on page M-104 for a list of all implementation-dependent and implementation-defined pragmas.

# Pragma IMPORT

Pragma `IMPORT` is implemented as described in Section B.1 of the Ada 95 Reference Manual.

Its syntax is:

```
pragma IMPORT([Convention=>]convention_identifier,
              [Entity=>]local_name
              [,[External_Name=>]string_expression]
              [,[Link_Name=>]string_expression]);
```

This pragma is used to import an entity defined in a foreign language into an Ada program, thus allowing a foreign-language subprogram to be called from Ada, or a foreign-language variable to be accessed from Ada. This pragma is referred to in the Ada 95 Reference Manual as an *interfacing pragma*.

An interfacing pragma defines the convention of the entity denoted by `local_name`. The convention represents the calling convention or representation convention of the entity.

The *convention*`_identifier` is the name of a *convention*. The convention names represent the calling conventions of foreign languages, language implementations, or specific run-time models.

The allowable conventions are:

- `Ada`
- `Assembler`
- `C`
- `Unchecked_C`
- `Restricted_Fortran`
- `Entry` (internal use only)
- `Intrinsic` (internal use only)
- `Protected` (internal use only)

Pragma `IMPORT` optionally specifies an entity's external name, link name, or both.

An `External_Name` is a string value for the name used by the foreign language program for the entitiy that an Ada program imports.

A `Link_Name` is a string value for the name of the imported entity, based on the conventions of the foreign language's compiler in interfacing with the system's linker tool.

The meaning of link names is implementation defined. If neither a link name nor the Address attribute of an imported entity is specified, then a link name is chosen in an implementation-defined manner, based on the external name if one is specified. If both the external name and the link name are specified, the external name is ignored.

See "RM B.1 Interfacing Pragmas" on page M-74 for details on the implementation of this pragma with respect to the Ada 95 Reference Manual.

See "RM Annex L: Pragmas" on page M-104 for a list of all implementation-dependent and implementation-defined pragmas.

## Pragma INLINE

Pragma `INLINE` is implemented as described in Section 6.3.2 of the Ada 95 Reference Manual.

Its syntax is:

```
pragma INLINE(name{,name});
```

However, there are a number of restrictions on inline subprogram expansion. The compiler will issue a warning, not perform the inline expansion, and output code for a subprogram call if any of these restrictions are violated or exceeded.

The restrictions and limitations on inline subprogram expansion include:

- The body of the subprogram must be compiled before it can be expanded inline. The **a.build** utility, when the **-IO** option is specified with a value other than `0`, attempts to compile bodies that define inline subprograms before bodies that use inline subprograms, however, if two bodies contain mutual inline dependencies, **a.build** chooses, in an arbitrary manner, which to compile first.
- There are a number of Ada constructs that prevent inline expansion if they appear in the declarations of a subprogram marked with pragma `INLINE`. These constructs include tasks, most generic instantiations, and (inner) subprograms that perform up-level addressing.
- Direct or indirect recursive calls are never inline-expanded.
- The actual parameters to the inline expanded subprogram must not contain task objects, must not contain dependent arrays, and must have complete type declarations.
- Subprograms marked with pragma `INTERFACE` are never inline-expanded.

The uncontrolled use of inline expansion can adversely affect the performance of the MAXAda compiler itself. Inline expansion can be controlled by using MAXAda configuration management described in "Qualifier Keywords (`-Q` options)" on page 4-105.

Subprograms that contain machine-code insertion statements are always inline expanded if they are marked with pragma `INLINE`, regardless of any configuration limits.

**WARNING**

Inline expansion of machine-code procedures is supported, but the user should exercise caution. It is not recommended practice to inline-expand machine-code procedures, as the compiler does not track register uses and definitions made by machine-code procedures.

See "RM Annex L: Pragmas" on page M-104 for a list of all implementation-dependent and implementation-defined pragmas.

## Pragma INSPECTION_POINT - (not yet supported)

Pragma `INSPECTION_POINT` is not supported in this release.

See "RM Annex L: Pragmas" on page M-104 for a list of all implementation-dependent and implementation-defined pragmas.

# Pragma INTERESTING

The implementation-defined pragma `INTERESTING` specifies the degree of interest of a named entity.

Its syntax is:

**`pragma INTERESTING(`***`static`***`_expression[,`***`simple`***`_name]);`**

The specified *static*`_expression` must be a static integer value. The *simple*`_name` is an optional argument denoting an entity visible at the place of the pragma and declared within the same declarative part as the pragma. If omitted, the pragma must be in a declarative part and then applies to that declarative part.

This pragma indicates in the debug information the degree of interest of a named entity. This information is only useful if full debug information is enabled (see "Pragma DEBUG" on page M-109 or "Debug Level (`-g[level]`)" on page 4-100).

This information is useful in conjunction with the `Real_Time_Data_Monitoring` package. A minimum interest "threshold" may be specified to restrict the set of objects or components to be monitored using the `interest_threshold` parameter (see "rtdm" on page 9-12).

This information is also useful in conjunction with the NightView debugger. A minimum interest threshold may be specified via the **`interest`** command to restrict the set of routines to be displayed in various circumstances.

In addition, the `-Qinteresting` compile option may be used to indicate the default degree of interest for every entity in the compilation. See "Qualifier Keywords (`-Q` options)" on page 4-105 for more information.

See "RM Annex L: Pragmas" on page M-104 for a list of all implementation-dependent and implementation-defined pragmas.

# Pragma INTERFACE - (obsolete)

The implementation-defined pragma `INTERFACE` is obsolete. It will be removed in a future release and should not be used. Use "Pragma IMPORT" on page M-114 instead.

In this release, if pragma `INTERFACE` is used, pragma `IMPORT` will be activated instead.

See "RM Annex L: Pragmas" on page M-104 for a list of all implementation-dependent and implementation-defined pragmas.

# Pragma INTERFACE_NAME - (obsolete)

The implementation-defined pragma `INTERFACE_NAME` is obsolete. It will be removed in a future release and should not be used. Use "Pragma IMPORT" on page M-114 instead.

In this release, if pragma INTERFACE_NAME is used, pragma IMPORT will be activated instead.

See "RM Annex L: Pragmas" on page M-104 for a list of all implementation-dependent and implementation-defined pragmas.

## Pragma INTERFACE_OBJECT - (obsolete)

The implementation-defined pragma INTERFACE_OBJECT is obsolete. It will be removed in a future release and should not be used. Use "Pragma IMPORT" on page M-114 instead.

In this release, if pragma INTERFACE_OBJECT is used, pragma IMPORT will be activated instead.

See "RM Annex L: Pragmas" on page M-104 for a list of all implementation-dependent and implementation-defined pragmas.

## Pragma INTERFACE_SHARED - (obsolete)

The implementation-defined pragma INTERFACE_SHARED is obsolete. It will be removed in a future release and should not be used. Use "Pragma IMPORT" on page M-114 with "Pragma VOLATILE" on page M-135 instead.

In this release, if pragma INTERFACE_SHARED is used, pragma IMPORT and pragma VOLATILE will be activated instead.

See "RM Annex L: Pragmas" on page M-104 for a list of all implementation-dependent and implementation-defined pragmas.

## Pragma INTERRUPT_HANDLER

Pragma INTERRUPT_HANDLER is implemented as described in Section C.3.1 of the Ada 95 Reference Manual.

Its syntax is:

```
pragma INTERRUPT_HANDLER(handler_name);
```

See "RM Annex L: Pragmas" on page M-104 for a list of all implementation-dependent and implementation-defined pragmas.

## Pragma INTERRUPT_PRIORITY

Pragma INTERRUPT_PRIORITY is implemented as described in Section D.1 of the Ada 95 Reference Manual.

Its syntax is:

```
pragma INTERRUPT_PRIORITY[(expression)];
```

See “RM Annex L: Pragmas” on page M-104 for a list of all implementation-dependent and implementation-defined pragmas.

## Pragma LINK_OPTION - (obsolete)

The implementation-defined pragma `LINK_OPTION` is obsolete. It will be removed in a future release and should not be used. Use “Pragma LINKER_OPTIONS” on page M-119 instead.

In this release, if pragma `LINK_OPTION` is used, pragma `LINKER_OPTIONS` will be activated instead.

See “RM Annex L: Pragmas” on page M-104 for a list of all implementation-dependent and implementation-defined pragmas.

## Pragma LINKER_OPTIONS

Pragma `LINKER_OPTIONS` is implemented as described in Section B.1 of the Ada 95 Reference Manual.

Its syntax is:

```
pragma LINKER_OPTIONS(string_expression);
```

See also “B.1(37) The effect of pragma Linker_Options” on page M-75 for implementation-defined aspects of this pragma.

See “RM Annex L: Pragmas” on page M-104 for a list of all implementation-dependent and implementation-defined pragmas.

## Pragma LIST

Pragma `LIST` is implemented as described in Section 2.8 of the Ada 95 Reference Manual.

Its syntax is:

```
pragma LIST(identifier);
```

See “RM Annex L: Pragmas” on page M-104 for a list of all implementation-dependent and implementation-defined pragmas.

# Pragma LOCKING_POLICY

Pragma LOCKING_POLICY is implemented as described in Section D.3 of the Ada 95 Reference Manual.

Its syntax is:

```
pragma LOCKING_POLICY(policy_identifier);
```

This pragma sets the protected object locking policy. See "Pragma LOCKING_POLICY" on page 6-3 for a complete description.

See also "D.3(4) Implementation-defined policy_identifiers allowed in a pragma Locking_Policy" on page M-92.

See "RM Annex L: Pragmas" on page M-104 for a list of all implementation-dependent and implementation-defined pragmas.

# Pragma MAP_FILE

The implementation-defined pragma MAP_FILE causes a map file to be emitted at link time.

Its syntax is:

```
pragma MAP_FILE(file_name) ;
```

See "Pragma MAP_FILE" on page 6-2 for a complete description.

See "RM Annex L: Pragmas" on page M-104 for a list of all implementation-dependent and implementation-defined pragmas.

# Pragma MEMORY_POOL

The implementation-defined pragma MEMORY_POOL changes physical memory pool attributes from their default values for a memory pool.

Its syntax is:

```
pragma MEMORY_POOL(pool_spec, memory_spec) ;
```

See "Pragma MEMORY_POOL" on page 6-23 for a complete description.

See "RM Annex L: Pragmas" on page M-104 for a list of all implementation-dependent and implementation-defined pragmas.

# Pragma NORMALIZE_SCALARS - (not yet supported)

Pragma `NORMALIZE_SCALARS` is not supported in this release.

See "RM Annex L: Pragmas" on page M-104 for a list of all implementation-dependent and implementation-defined pragmas.

# Pragma OPT_FLAGS

The implementation-defined pragma `OPT_FLAGS` provides a method for overriding the optimization parameters defined by a MAXAda environment's configuration.

Its syntax is:

**`pragma OPT_FLAGS(`***`string_`***`expression);`**

The `OPT_FLAGS` pragma takes a single string literal as an argument. This string, enclosed in quotes, should contain all of the optimizer flags to be overridden for the compilation, along with the value to be observed. The literal string argument must take the form:

"*flag = value, flag = value, flag = value ...*"

Nine flags are recognized by the MAXAda compiler. Many of these flags are described in detail in "Qualifier Keywords (`-Q` options)" on page 4-105 of this manual and are configurable not only via the pragma, but also as parameters to the **`a.options`** tool. The optimizer flags are:

```
objects
loops
unroll_limit_const
unroll_limit_var
unroll_limit
growth_limit
optimize_for_space
opt_class
noreorder
```

By specifying a configuration value for an optimizer parameter using this pragma, the given value is observed by the MAXAda compiler when the enclosing unit is compiled (regardless of the value specified for the optimizer parameter(s) in the environment's configuration).

For example, the line:

```
pragma OPT_FLAGS("growth_limit=200, unroll_limit=5");
```

optimizes a total of 200 objects and uses a loop unrolling limit of 5 for the compilation unit whose declarative part contains the preceding pragma. These values override any values given by a local or system configuration record for the compilation.

Compilation units that omit any flags from the pragma or that omit the pragma altogether observe the optimizer flag values specified by corresponding **-Q** options applied to the unit or the environment.

See "Compile Options" on page 4-99 for more information.

See "RM Annex L: Pragmas" on page M-104 for a list of all implementation-dependent and implementation-defined pragmas.

## Pragma OPT_LEVEL

The implementation-defined pragma OPT_LEVEL controls the level of optimization performed by the compiler.

Its syntax is:

**pragma OPT_LEVEL([***unit_name***,]** *level***);**

where *unit_name*, if specified, is the name of the compilation unit for which the optimization level is being specified, and where the *level* is one of: MINIMAL, GLOBAL, or MAXIMAL.

The single-parameter form of this pragma is allowed only immediately within a library unit or as a configuration pragma. When specified within a library unit, it applies only to that library unit. When specified as a configuration pragma, it applies to all units within the same compilation, if any, or to all units in the environment, if none. The two-parameter form of this pragma is allowed only immediately following the unit which is specified as the *unit_name* argument. It applies only to the unit which is specified.

If applied to a specification, the optimization level does ***not*** apply to the body or any separate bodies of the unit. If applied to a body, the optimization level does ***not*** apply to any separate bodies of the unit. If the optimization level is desired for any such units, it must be specified for them, too.

The pragma is meaningless when applied to a generic unit. If so applied, it will not be applied to any instantiations of that generic. The optimization level applied to an instantiation is the optimization level of the unit which contains it, or if the instantiation is library-level, is determined in the same way as for any other library-level unit.

See "RM Annex L: Pragmas" on page M-104 for a list of all implementation-dependent and implementation-defined pragmas.

## Pragma OPTIMIZE

Pragma OPTIMIZE is recognized by the implementation but does not have an effect in this release.

Its syntax is:

**pragma OPTIMIZE(identifier);**

See the **-O** compile option for code optimization levels (see page 4-99) or the implementation-defined pragma OPT_LEVEL.

See Section 2.8 of the Ada 95 Reference Manual for more information about this pragma.

See "RM Annex L: Pragmas" on page M-104 for a list of all implementation-dependent and implementation-defined pragmas.

## Pragma PACK

Pragma PACK is implemented as described in Section 13.2 of the Ada 95 Reference Manual.

Its syntax is:

**pragma PACK(***first_subtype_***local_name);**

This pragma causes the compiler to choose a non-aligned representation for elements of composite types. Application of the pragma causes objects to be packed to the bit level.

See "RM Annex L: Pragmas" on page M-104 for a list of all implementation-dependent and implementation-defined pragmas.

## Pragma PAGE

Pragma PAGE is implemented as described in Section 2.8 of the Ada 95 Reference Manual.

Its syntax is:

**pragma PAGE;**

See "RM Annex L: Pragmas" on page M-104 for a list of all implementation-dependent and implementation-defined pragmas.

## Pragma PASSIVE_TASK - (obsolete)

The implementation-defined pragma PASSIVE_TASK is obsolete. It will be removed in a future release and should not be used. Use protected objects instead.

See "RM Annex L: Pragmas" on page M-104 for a list of all implementation-dependent and implementation-defined pragmas.

## Pragma POOL_CACHE_MODE

The implementation-defined pragma `POOL_CACHE_MODE` defines the cache mode for a memory pool.

Its syntax is:

```
pragma POOL_CACHE_MODE(pool_spec, cache_mode) ;
```

See “Pragma POOL_CACHE_MODE” on page 6-25 for a complete description.

See “RM Annex L: Pragmas” on page M-104 for a list of all implementation-dependent and implementation-defined pragmas.

## Pragma POOL_LOCK_STATE

The implementation-defined pragma `POOL_LOCK_STATE` defines the lock state of a memory pool.

Its syntax is:

```
pragma POOL_LOCK_STATE(pool_spec, lock_state);
```

See “Pragma POOL_LOCK_STATE” on page 6-25 for a complete description.

See “RM Annex L: Pragmas” on page M-104 for a list of all implementation-dependent and implementation-defined pragmas.

## Pragma POOL_PAD

The implementation-defined pragma `POOL_PAD` sets the pad for a STACK memory pool.

Its syntax is:

```
pragma POOL_PAD(paddable_spec, size);
```

See “Pragma POOL_PAD” on page 6-28 for a complete description.

See “RM Annex L: Pragmas” on page M-104 for a list of all implementation-dependent and implementation-defined pragmas.

## Pragma POOL_SIZE

The implementation-defined pragma `POOL_SIZE` sets the size for a STACK or COLLECTION memory pool.

Its syntax is:

```
pragma POOL_SIZE(sizeable_spec, size_spec) ;
```

See “Pragma POOL_SIZE” on page 6-26 for a complete description.

See “RM Annex L: Pragmas” on page M-104 for a list of all implementation-dependent and implementation-defined pragmas.

## Pragma PREELABORATE

Pragma PREELABORATE is implemented as described in Section 10.2.1 of the Ada 95 Reference Manual.

Its syntax is:

```
pragma PREELABORATE[(library_unit_name)];
```

See “RM Annex L: Pragmas” on page M-104 for a list of all implementation-dependent and implementation-defined pragmas.

## Pragma PRIORITY

Pragma PRIORITY is implemented as described in Section D.1 of the Ada 95 Reference Manual.

Its syntax is:

```
pragma PRIORITY(expression);
```

Priorities range from 0 through 287, with 287 being the most urgent.

See “Pragma TASK_PRIORITY” on page M-134 for a related pragma.

See “RM Annex L: Pragmas” on page M-104 for a list of all implementation-dependent and implementation-defined pragmas.

## Pragma PROTECTED_PRIORITY

The implementation-defined pragma PROTECTED_PRIORITY sets the scheduling priority for a given protected object.

Its syntax is:

```
pragma PROTECTED_PRIORITY(scheduling_priority
                          [,protected_object_specifier ]);
```

See “Pragma PROTECTED_PRIORITY” on page 6-28 for a complete description of protected priorities.

See “RM Annex L: Pragmas” on page M-104 for a list of all implementation-dependent and implementation-defined pragmas.

## Pragma PURE

Pragma PURE is implemented as described in Section 10.2.1 of the Ada 95 Reference Manual.

Its syntax is:

```
pragma PURE[(library_unit_name)];
```

See “RM Annex L: Pragmas” on page M-104 for a list of all implementation-dependent and implementation-defined pragmas.

## Pragma QUEUING_POLICY

Pragma QUEUING_POLICY is implemented as described in Section D.4 of the Ada 95 Reference Manual.

Its syntax is:

```
pragma QUEUING_POLICY(policy_identifier);
```

The implementation-defined pragma QUEUING_POLICY sets the entry queuing policy.

See “Pragma QUEUING_POLICY” on page 6-2 for a complete description.

See “RM Annex L: Pragmas” on page M-104 for a list of all implementation-dependent and implementation-defined pragmas.

## Pragma REMOTE_CALL_INTERFACE - (not yet supported)

Pragma REMOTE_CALL_INTERFACE is not supported in this release.

See “RM Annex L: Pragmas” on page M-104 for a list of all implementation-dependent and implementation-defined pragmas.

## Pragma REMOTE_TYPES - (not yet supported)

Pragma REMOTE_TYPES is not supported in this release.

See “RM Annex L: Pragmas” on page M-104 for a list of all implementation-dependent and implementation-defined pragmas.

# Pragma RESTRICTIONS

In this release, pragma RESTRICTIONS is supported only for the tasking restrictions defined in Section D.7 of the Ada 95 Reference Manual and for the implemenation-defined restriction No_Stream_Attributes.

Its syntax is:

```
pragma RESTRICTIONS(restriction{,restriction});
```

The dynamic restrictions Max_Storage_At_Blocking, Max_Asynchronous_Select_Nesting, and Max_Tasks have no effect in this release. A future release will enforce the limits set by these restrictions.

The presence of the restriction No_Stream_Attributes indicates that the 'Read, 'Write, 'Input, and 'Output attributes can never be referenced. This allows the implementation to omit routines to implement stream attributes for tagged types in any units to which this restriction applies. This results in a space savings. To achieve best results with this restriction, it should be applied to all units in a partition. A stand-alone configuration pragma (see "Configuration Pragmas" on page 3-9) can be used to guarantee this.

The presence of any restrictions defined in Section D.7 of the Ada 95 Reference Manual has no effect upon the run-time in this release. A future release will optimize the compiled code and the run-time based upon which restrictions are present. That is, the Implementation Advice at D.7(22) is ignored in this release.

See Section 13.12 of the Ada 95 Reference Manual for more information about this pragma.

See "RM Annex L: Pragmas" on page M-104 for a list of all implementation-dependent and implementation-defined pragmas.

# Pragma RETURN_CONVENTION

The implementation-defined pragma RETURN_CONVENTION is used to specify that a composite type should be returned BY_REGISTER rather than BY_COPY.

Its syntax is:

```
pragma RETURN_CONVENTION(convention, identifier);
```

where:

```
convention ::= BY_REGISTER | BY_COPY
identifier ::= subtype_mark | function_identifier
```

The *subtype_mark* (or result type of the specified function) must resolve to denote a return-by-copy type as per Ada 95 Reference Manual 6.5(17).

The current implementation further restricts the application of RETURN_CONVENTION to record types (or functions returning record types) whose sizes are 8 bytes or less. Further, the BY_REGISTER convention is the only convention currently allowed in the pragma.

See “RM Annex L: Pragmas” on page M-104 for a list of all implementation-dependent and implementation-defined pragmas.

## Pragma REVIEWABLE - (not yet supported)

Pragma `REVIEWABLE` is not supported in this release.

See “RM Annex L: Pragmas” on page M-104 for a list of all implementation-dependent and implementation-defined pragmas.

## Pragma RUNTIME_DIAGNOSTICS

The implementation-defined pragma `RUNTIME_DIAGNOSTICS` controls whether or not the run-time emits warning diagnostics.

Its syntax is:

```
pragma RUNTIME_DIAGNOSTICS(boolean);
```

See “Pragma RUNTIME_DIAGNOSTICS” on page 6-1 for a complete description.

See “RM Annex L: Pragmas” on page M-104 for a list of all implementation-dependent and implementation-defined pragmas.

## Pragma SERVER_CACHE_SIZE

The implementation-defined pragma `SERVER_CACHE_SIZE` sets the size of the server cache.

Its syntax is:

```
pragma SERVER_CACHE_SIZE(cache_size);
```

See “Pragma SERVER_CACHE_SIZE” on page 6-4 for a complete description.

See “RM Annex L: Pragmas” on page M-104 for a list of all implementation-dependent and implementation-defined pragmas.

## Pragma SHARE_BODY

The implementation-defined pragma `SHARE_BODY` indicates whether or not an instantiation is to be shared.

Its syntax is:

```
pragma SHARE_BODY(generic_name, boolean_literal)
```

The pragma may reference the generic unit or the instantiated unit. When it references a generic unit, it sets sharing on/off for all instantiations of the generic, unless overridden by specific SHARE_BODY pragmas for individual instantiations. When it references an instantiated unit, sharing is on/off only for that unit. For this release, the default is to not share any generics.

Pragma SHARE_BODY is allowed only 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.

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.

See "RM Annex L: Pragmas" on page M-104 for a list of all implementation-dependent and implementation-defined pragmas.

## Pragma SHARE_MODE

The implementation-defined pragma SHARE_MODE sets the *share_mode* for a compilation unit from within the Ada source code. The format of the pragma is:

```
pragma SHARE_MODE([unit_name,] share_mode);
```

where *unit_name*, if specified, is the name of the compilation unit for which the *share_mode* is being specified, and where the *share_mode* is one of: SHARED, NON_SHARED, or BOTH.

The single-parameter form of this pragma is allowed only immediately within a library unit or as a configuration pragma. When specified within a library unit, it applies only to that library unit. When specified as a configuration pragma, it applies to all units within the same compilation, if any, or to all units in the environment, if none. The two-parameter form of this pragma is allowed only immediately following the unit which is specified as the *unit_name* argument. It applies only to the unit which is specified.

If applied to a specification, the share mode does ***not*** apply to the body or any separate bodies of the unit. If applied to a body, the share mode does ***not*** apply to any separate bodies of the unit. If the share mode is desired for any such units, it must be specified for them, too.

The pragma is meaningless when applied to a generic unit. If so applied, it will not be applied to any instantiations of that generic. The share mode applied to an instantiation is the share mode of the unit which contains it, or if the instantiation is library-level, is determined in the same way as for any other library-level unit.

See also "Shared Objects" on page 3-13 for more information.

See "RM Annex L: Pragmas" on page M-104 for a list of all implementation-dependent and implementation-defined pragmas.

## Pragma SHARED - (obsolete)

The implementation-defined pragma `SHARED` is obsolete. It will be removed in a future release and should not be used. Use "Pragma ATOMIC" on page M-106 instead.

In this release, if pragma `LINK_OPTION` is used, pragma `VOLATILE` will be activated instead. See "Pragma VOLATILE" on page M-135 for more information.

See "RM Annex L: Pragmas" on page M-104 for a list of all implementation-dependent and implementation-defined pragmas.

## Pragma SHARED_PACKAGE

The implementation-defined pragma `SHARED_PACKAGE` provides for the sharing and communication of data declared within the specification of library-level packages.

Its syntax is:

```
pragma SHARED_PACKAGE[("params")];
```

Pragma `SHARED_PACKAGE` accepts as an optional argument, "*params*", that, if specified, must be a string constant containing a comma-separated list of system shared-segment configuration parameters.

See "Pragma SHARED_PACKAGE" on page 8-1 for details.

See also "4.1.4(12) Implementation-defined attributes" on page M-13 for more information.

See "RM Annex L: Pragmas" on page M-104 for a list of all implementation-dependent and implementation-defined pragmas.

## Pragma SHARED_PASSIVE - (not yet supported)

Pragma `SHARED_PASSIVE` is not supported in this release.

See "RM Annex L: Pragmas" on page M-104 for a list of all implementation-dependent and implementation-defined pragmas.

## Pragma SPECIAL_FEATURE

**NOTE**

Pragma `SPECIAL_FEATURE` is reserved for internal MAXAda use only; it is not intended for use in user-defined code.

The implementation-defined pragma SPECIAL_FEATURE forces the compiler to assume that the specified *feature* is used by the unit associated with this pragma.

Its syntax is:

```
pragma SPECIAL_FEATURE(feature);
```

where *feature* can be one of the following:

| | | | |
|---|---|---|---|
| hardware_interrupts | software_interrupts | cpu_biases | priorities |
| rescheduling_variables | page_locking | quanta | data_monitoring |
| shmbind | trace_dump | needs_xlib | needs_xt |
| needs_motif | tasks | protected_objects | |

See “RM Annex L: Pragmas” on page M-104 for a list of all implementation-dependent and implementation-defined pragmas.

## Pragma STORAGE_SIZE

Pragma STORAGE_SIZE is implemented as described in Section 13.3 of the Ada 95 Reference Manual.

Its syntax is:

```
pragma STORAGE_SIZE(expression);
```

See “RM Annex L: Pragmas” on page M-104 for a list of all implementation-dependent and implementation-defined pragmas.

## Pragma SUPPRESS

Pragma SUPPRESS is implemented as described in Section 11.5 of the Ada 95 Reference Manual.

Its syntax is:

```
pragma SUPPRESS(identifier[,[On=>]name]);
```

The double parameter form of the pragma, with a name of an object, type, or subtype is recognized, but has no effect.

See “RM Annex L: Pragmas” on page M-104 for a list of all implementation-dependent and implementation-defined pragmas.

## Pragma SUPPRESS_ALL

The implementation-defined pragma `SUPPRESS_ALL` gives permission to the implementation to suppress all run-time checks.

Its syntax is:

```
pragma SUPPRESS_ALL;
```

Pragma `SUPPRESS_ALL` does not have any parameters. It may appear immediately within a declarative part or immediately within a package specification or as a configuration pragma. Its effects are equivalent to a list of `SUPPRESS` pragmas, each naming a different check.

See "RM Annex L: Pragmas" on page M-104 for a list of all implementation-dependent and implementation-defined pragmas.

## Pragma TASK_CPU_BIAS

The implementation-defined pragma `TASK_CPU_BIAS` sets the CPU assignments for a given bound task.

Its syntax is:

```
pragma TASK_CPU_BIAS(cpu_bias[,task_specifier]);
```

See "Pragma TASK_CPU_BIAS" on page 6-12 for a complete description of task CPU assignments.

See "RM Annex L: Pragmas" on page M-104 for a list of all implementation-dependent and implementation-defined pragmas.

## Pragma TASK_DISPATCHING_POLICY

Pragma `TASK_DISPATCHING_POLICY` is implemented as described in Section D.2.2 of the Ada 95 Reference Manual.

Its syntax is:

```
pragma TASK_DISPATCHING_POLICY(policy_identifier);
```

This pragma sets the task dispatching policy.

See "Pragma TASK_DISPATCHING_POLICY" on page 6-2 for a complete description.

See "RM Annex L: Pragmas" on page M-104 for a list of all implementation-dependent and implementation-defined pragmas.

## Pragma TASK_HANDLER

The implementation-defined pragma `TASK_HANDLER` calls the specified procedure when the task to which it is applied completes because of an unhandled exception.

This pragma is especially useful when applied to the **ENVIRONMENT** task. It will be called for any unhandled exception that would cause completion of the environment task, and thus of the application.

It is also especially useful when applied to the **DEFAULT** task. It will be called for any unhandled exception that would cause completion of any task which otherwise happens silently without any notification to the user.

Its syntax is:

```
pragma TASK_HANDLER(handler_name[, task_specifier]);
```

See "Pragma TASK_HANDLER" on page 6-15 for a complete description.

See "RM Annex L: Pragmas" on page M-104 for a list of all implementation-dependent and implementation-defined pragmas.

## Pragma TASK_PRIORITY

The implementation-defined pragma `TASK_PRIORITY` sets the scheduling priority for a given task *within* the server group and for entry queuing. It also sets the operating system priority for bound tasks.

Its syntax is:

```
pragma TASK_PRIORITY(scheduling_priority[, task_specifier]);
```

See "Pragma TASK_PRIORITY" on page 6-11 for a complete description of task priorities.

See "RM Annex L: Pragmas" on page M-104 for a list of all implementation-dependent and implementation-defined pragmas.

## Pragma TASK_QUANTUM

The implementation-defined pragma `TASK_QUANTUM` sets the task time-slice duration for a given task.

Its syntax is:

```
pragma TASK_QUANTUM(quantum[, task_specifier]) ;
```

See "Pragma TASK_QUANTUM" on page 6-14 for a complete description of task time slicing.

See “RM Annex L: Pragmas” on page M-104 for a list of all implementation-dependent and implementation-defined pragmas.

## Pragma TASK_WEIGHT

The implementation-defined pragma `TASK_WEIGHT` specifies the weight of a task.

Its syntax is:

```
pragma TASK_WEIGHT(weight[,task_specifier]);
```

See “Pragma TASK_WEIGHT” on page 6-9 for a complete description.

See “RM Annex L: Pragmas” on page M-104 for a list of all implementation-dependent and implementation-defined pragmas.

## Pragma TDESC

**NOTE**

Pragma `TDESC` is reserved for internal MAXAda use only; it is not intended for use in user-defined code.

See “RM Annex L: Pragmas” on page M-104 for a list of all implementation-dependent and implementation-defined pragmas.

## Pragma TRAMPOLINE

**NOTE**

Pragma `TRAMPOLINE` is reserved for internal MAXAda use only; it is not intended for use in user-defined code.

See “RM Annex L: Pragmas” on page M-104 for a list of all implementation-dependent and implementation-defined pragmas.

## Pragma VOLATILE

Pragma `VOLATILE` is implemented as described in Section C.6 of the Ada 95 Reference Manual.

Its syntax is:

```
pragma VOLATILE(local_name);
```

This pragma accepts a single variable name which must be of a type which can be volatile for the pragma to apply. All accesses to this variable results in memory references. Pragma VOLATILE should be used on any variable that may be accessed concurrently by different threads of a program, e.g., a variable shared between tasks.

See "RM Annex L: Pragmas" on page M-104 for a list of all implementation-dependent and implementation-defined pragmas.

## Pragma VOLATILE_COMPONENTS

Pragma VOLATILE_COMPONENTS is implemented as described in Section C.6 of the Ada 95 Reference Manual.

Its syntax is:

```
pragma VOLATILE_COMPONENTS(array_local_name);
```

This pragma accepts an array name, the components of which must be of a type which can be volatile for the pragma to apply. All accesses to these variables result in memory references. Pragma VOLATILE_COMPONENTS should be used on variables that may be accessed concurrently by different threads of a program (e.g., a variable shared between tasks).

See "RM Annex L: Pragmas" on page M-104 for a list of all implementation-dependent and implementation-defined pragmas.

# Glossary

This glossary defines terms used in the documentation. Terms in *italics* are defined here.

**active partition**

An active partition is the simplest form of *partition* and it describes how to build an executable program.

**Ada binding**

Ada bindings provide a pure Ada interface to libraries of routines and services which have been originally developed in another programming language.

**ADMIN ghost task**

A *ghost task* that exists only in programs that contain tasking (other than the *ENVIRONMENT task*). If it exists, it is a *bound task* that is responsible for the creation of all named *server groups* and for the creation of the ENVIRONMENT task. It also detects the termination of all other tasks and performs cleanup operations on those tasks, including deallocation of memory associated with those tasks.

**alert**

A diagnostic message that conveys information to the user about packages, *pragmas*, or options that are considered to be obsolete in this *release*.

**ambiguous unit**

Upon introducing a *unit* having the same name as a previously introduced unit, MAXAda labels both units as ambiguous.

**archive**

An archive is a collection of routines and data that is associated with an application during the link and execution phases. Archives contain statically-built (i.e. non-shared) objects within them.

**ARMS**

The Ada Real-time Multiprocessor System, also known as the *run-time system*.

**attachment index**

Denotes a particular static attachment based on its textual order within a particular handler. (e.g. the first `attach_handler` pragma in a particular handler is "`1`", the second is "`2`", etc. The first `attach_handler` in a different handler is "`1`", etc...)

**AXI**

The optional Ada X Interface. It provides an Ada binding to the full Xlib, Xt, and Motif libraries.

**bound task**

A task that is served by an anonymous *server group* containing exactly one server. This server group exists only to execute the single task for which it was created, dedicated for its exclusive use. See *task weight*.

**cache mode**

A system cache memory attribute that is either COPYBACK or NCACHE.

**clone**

An operating system process which shares almost all of its attributes with its parent, including: the address space, file descriptors, signal actions, etc. See **`clone(2)`** for more information.

**collection (memory)**

A memory region (heap) used for designated objects of user-defined access types, dynamically sized objects, internal run-time structures, etc.

**companion ghost task**

A *ghost task* that is associated with some user-defined real task or a user-defined entry of a real task.

**compilation unit**

See *unit*.

**compile options**

Stored as part of the *environment* or as part of an individual *unit*'s information, these options do not need to be specified on the command line for each compilation. Rather, they are "remembered" when the MAXAda compilation tools are used.

**concurrent program**

Any Ada program that elaborates the body of a shared package and whose span of execution, from elaboration of such a package to termination, overlaps that of another such program.

**configuration pragma**

Configuration pragmas are syntactical entities that are not part of a *unit*. Configuration pragmas can appear either at the beginning of a source file containing library units or independently in a source file with no units. See also *pragma*.

**consistency**

The compilation of a unit is consistent if its source file has not been modified since it has been compiled and all of the units on which it depends are still consistently compiled. In addition, the unit can only remain consistent if it and the units on which it depends have not become *ambiguous* or *obscured*. In addition, a unit can only remain consistent if the compile options for that unit and the units on which it depends have not changed.

Each *unit* is considered consistent up to a particular state. This means that it is valid *up to that state of compilation*. Any recompilation of the unit can start from that state. It does not need to go through the earlier stages of recompilation.

**COURIER ghost task**

A *bound task* associated with the *hardware interrupt* entry. It does not execute at interrupt level. It may be involved in forwarding the hardware interrupt from an *INTR_COURIER task* to the real task.

**CPU bias**

A mask in which the relative bit number identifies a CPU number (*LSB* corresponds to CPU #0). It is used to assign a *server* to a CPU.

**data (memory)**

Statically sized memory segment used for the allocation of library-level variables, such as those in library-level packages.

**data monitoring**

The real-time display and modification of static Ada variables via the `real_time_data_monitoring` package. This is usually performed during program debugging.

**debug level**

The setting for the amount of debug information to be generated for a compilation unit. Debug level can be established via the **`-g`** option to several utilities or pragma `DEBUG` in the source. Values include: `none`, `lines`, and `full`.

**DEFAULT pseudo group**

The non-executing pseudo *group* that provides default group-attribute values for any groups that omit any group *configuration pragmas*.

**DEFAULT pseudo task**

The non-executing pseudo task that provides default task-attribute values for any tasks that omit any task *configuration pragmas*.

**deprecated**

A MAXAda *environment* shipped with this *release* for compatibility purposes with previous versions only. It will be removed in a future release of MAXAda.

**deprecated feature**

A MAXAda feature considered to be obsolete in this *release* but which is still supported for backward compatibility.

**distributed application**

A program that requires the use of CPUs on more than one CPU board.

**DWARF**

The debug format that MAXAda supports. The debug information is stored in the executable *program*.

**dynamic-linking phase**

This phase occurs only for programs that use *shared objects*. The initial phase of execution when actual associations and memory allocations occur. The dynamic linker tries to locate each essential shared-object and bind its physical pages into the application program.

**effective options**

The resultant set of compile options based on the hierarchical relationship between the *environment-wide compile options*, *permanent unit options*, and *temporary unit options*.

**elaboration**

**ELF**

The executable and object format that MAXAda supports.

**environment**

An entity that is associated with an operating system directory and that contains Ada units. MAXAda uses the concept of environments as its basic structure of organization. The environments **`vendorlib`**, **`publiclib`**, and **`predefined`** are supplied as part of MAXAda.

**Environment Search Path**

MAXAda uses the concept of an Environment Search Path to allow users to specify that units from *environments* other than the current environment should be made available in the current environment. This Environment Search Path relates only to

each particular environment and each environment has its own Environment Search Path.

**ENVIRONMENT task**

The task the *run-time* creates at start-up to perform library-level package elaboration and execute the *main subprogram*.

**environment-wide compile options**

Environment-wide compile options apply to all units within that *environment*. All compilations within this environment then observe these environment-wide options unless overridden.

**executive**

See *ARMS*.

**fast interrupt task**

A task that executes directly at interrupt level to accept *hardware interrupts* instead of relying on an *INTR_COURIER ghost task* to do so.

**fatal error**

An error of such severity that meaningful recovery is impossible and compilation stops.

**foreign environments**

Foreign environments are all *environments* other than the local (or current) environment.

**foreign units**

Foreign units are those *units* that exist in other *environments* which are on the *Environment Search Path*.

**frozen environment**

An *environment* that is made unalterable by the **`a.freeze`** utility. Since frozen environments are unalterable, accesses to these environments are much faster than accesses to environments that are not frozen.

**general error**

An error that is semantic in nature but does not fall within a specific Ada 95 Reference Manual reference.

**ghost task**

A task artificially created by the *run-time* executive for various internal purposes (overhead).

**global memory**

Physical memory available to all CPUs via a *system-wide bus*. See *local memory* and *remote memory.*

**group**

See *server group*.

**HAPSE**

Harris Ada Programming Support Environment. The predecessor to the *MAXAda* product. Based on the Ada 83 Reference Manual.

**hardness**

An attribute of physical *local memory*, either SOFT or HARD, that controls whether physical *global memory* is used if insufficient physical local memory is available.

**hardware interrupt**

An *interrupt* generated by a hardware device. For example, real-time clock, edge-triggered interrupt, and all system and VME interrupts. See *software interrupt*.

**heap**

See *collection (memory)*.

**immediate binding**

During dynamic linking, all shared objects that the application requires are allocated and linked into the application's address space before any of them are ever needed.

**independent configuration pragmas**

*Configuration pragmas* that appear independently in a source file with no *units*.

**internal error**

An error due to faults within the compiler.

**interoptimization**

MAXAda provides a method of optimization that controls the compilation order such that inlined subprogram calls will be performed whenever possible.

**interrupt**

An event external to the currently executing process. The two types are *hardware interrupt* and *software interrupt*.

**interrupt handler**

A subprogram that is called when an *interrupt* occurs.

**INTR_COURIER ghost task**

The real *ghost task* associated with a *hardware interrupt* entry. It has a *bound* weight and executes at *interrupt* level. It receives the hardware interrupt and forwards it to the real task with which it is associated in such a way that the entry call seems to be coming from the *SHADOW task*. A *COURIER task* may also be involved in forwarding the hardware interrupt to the real task.

**lazy binding**

By default, the dynamic linker does not link any *shared objects* into the application's address space until they are needed. If, during execution of the application, an as-yet unrelocated reference occurs, control passes to the dynamic linker which then relocates the reference.

**lexical error**

An error in the formation of literals, identifiers, and delimiters.

**library unit**

A library unit is a separately compiled program unit, and is always a package, subprogram, or generic unit. Library units may have other (logically nested) library units as children, and may have other program units physically nested within them.

**link method**

The link method specifies the manner in which a *unit* is included in the linking process. It can instruct the linker to use the object of a unit directly (`object` method), utilize the unit found in an *archive* (`archive` method), or include the unit contained within a *shared object* (`shared_object` method). These methods are used in conjunction with the *link rule*.

**link options**

Each MAXAda *partition* has a set of link options associated with it. These options are persistent and remain in effect for the life of the partition. They are normally set and modified using the **`-oset`**, **`-oappend`**, **`-oprepend`**, and **`-oclear`** options to **`a.partition`**.

**link rule**

The order the linker follows to determine the *link method* for each required *unit*.

**local environment**

The local environment, or the current environment, is the current working directory.

**local-locking**

The case when an application has a requirement for pages to be locked into local memory.

**local memory**

Physical memory available to CPUs via a local bus physically located on the same CPU as the local memory. See *global memory* and *remote memory.*

**lock state**

An attribute of *memory pools* that determines if memory pages are physically locked in memory and thus cannot be swapped out by the operating system.

**logical address**

A virtual memory address in an executing program's address space. See *machine address*.

**LSB**

Least-significant bit.

**machine address**

A physical memory address. See *logical address*.

**machine-code insertion**

A MAXAda feature that allows the inclusion of assembly language instructions in an Ada program. This is accomplished via the `machine_code` package.

**main subprogram**

A non-generic subprogram without parameters that is either a procedure or a function returning an Ada `STANDARD.INTEGER` (the predefined type). It is specified to **`a.partition`**.

**map file**

A file containing ASCII descriptions of attributes of the run-time configuration of the generated program, including: the layout of *memory pools*; dynamic memory management parameters; and task attributes, such as, *CPU bias*, *quantum*, *priority*, *weight*, and stack usage.

**MAPSE**

The Minimum Ada Programming Support Environment.

**MAXAda**

MAXAda is a high-performance system intended for the large-scale development of Ada application, real-time, and systems software. MAXAda supports the Ada language specification as defined in the Ada 95 Reference Manual

**MAXAda installation**

Any complete MAXAda directory structure that contains a version of MAXAda.

**MAXAda release**

Any released version of MAXAda or any MAXAda release containing a valid configuration of patches intended for that release.

**MCI**

See *machine code insertion*.

**memory pool**

A physical region of *global memory* or *local memory.*

**MSB**

Most-significant bit.

**multiplexed task**

A task that shares the resources of a single pool and is served by a named *server group*, which may contain one or more servers. See *task weight*.

**native unit**

A native unit is a *unit* which has been introduced into an *environment* by using the **`a.intro`** function.

**naturalized unit**

A naturalized unit is the compiled form of a foreign unit in the local environment created by the compilation system. A naturalized unit retains the options from its original environment.

**NightSim**

An optional, graphical, non-intrusive tool for scheduling and monitoring real-time single and multi-process applications running on one or more CPUs. It allows interactive control of the high-resolution Frequency-Based Scheduler (FBS) and interactive or deferred performance monitoring.

**NightTrace**

An optional, graphical debugging and performance-analysis tool that works with single and multi-process programs running on one or more CPUs.

**NightView**

An optional symbolic debugger that supports debugging of Ada, C, and Fortran programs running on one or more CPUs.

**NUMA**

Non-Uniform Memory Access. An architecture classification with a local/global/remote memory subclass that underlies Series 6000 computers.

**obscurities**

Obscurities occur when the natural behavior of MAXAda and the *Environment Search Path* mechanism prevent an intended file from being used for a particular compilation.

**obsolescent**

A MAXAda *environment* containing packages whose functionality is largely redundant with other features defined in the Ada 95 Reference Manual. Use of these features is not recommended in newly written programs.

**operating system quantum**

A *quantum* associated with a *server group* that determines the length of time that its servers execute on a CPU before being preempted.

**operating system scheduling priority**

A *scheduling priority* associated with a *server group* that determines how the real-time kernel selects *groups* for execution on CPUs.

**opportunism**

Make opportunistic use of *unit* bodies to improve code optimization (beyond inlining).

**optimization level**

The setting for the amount of compile-time optimization to occur for a *compilation unit*. Optimization level can be established via the **-O** option to several utilities or pragma `OPT_LEVEL` in the source. Option values include: `1`, `2`, and `3`. Corresponding pragma values include: `MINIMAL`, `GLOBAL`, and `MAXIMAL`.

**panic**

An error due to faults within the compiler.

**partition**

A partition is an executable, *archive*, or *shared object* that can be invoked outside of MAXAda. The user can explicitly assign *units* to partitions. The units included in a partition are those of the explicitly assigned units, as well as other units needed by those explicitly assigned. MAXAda manages these units and their dependencies, as well as link options and configuration information for each partition within the context of an *environment*.

**permanent unit options**

This set of options is associated with a unit and override its *environment-wide compile options*. Each unit has its own set of permanent unit options. They may be specified and later modified via the **`a.options`** utility.

**persistent options**

Unlike most other compilation systems, MAXAda uses persistent options that need not be specified on the command line to the compilation system. These options can be either associated with an *environment* or a particular *unit* and are "remembered" by the compilation system.

**position independent code (PIC)**

Position independent code refers to the fact that the generated code does not rely on labels, data, or routines being in known locations. This type of code allows for *shared objects* to be dynamically linked to an executable.

**pragma**

A pragma is a compiler directive. There are language-defined pragmas that give instructions for optimization, listing control, etc. An implementation may support additional (implementation-dependent) pragmas. See also *configuration pragmas*.

**predefined**

The Ada Predefined Language Environment, as specified in Annex A of the Ada 95 Reference Manual. It contains standard, system, I/O packages, etc.

**PREDEFINED group**

The predefined *group* the *run-time* creates at start-up that usually includes and executes the *ENVIRONMENT task* and the *DEFAULT pseudo task*.

**priority**

See *scheduling priority.*

**process**

The full-weight operating system entity that is spawned when the executable image is initiated.

**program**

The *ENVIRONMENT task* and the entire set of Ada tasks that are included in the Ada program as defined by its dependencies.

**protected procedure handler**

A protected procedure with a parameterless profile and declared as a handler with an `attach_handler` or an `interrupt_handler` pragma.

**publiclib**

*Environment* that contains packages not maintained or guaranteed by Concurrent.

**quantum**

The length of time an entity spends executing on an execution resource before being preempted.

**queuing policy**

The entry queuing policy, either FIFO_QUEUING or PRIORITY_QUEUING.

**release**

Any released version of MAXAda or any MAXAda release containing a valid configuration of patches intended for that release.

**relocation**

The dynamic linker's final address resolution of *shared object* symbol references in internal symbol tables.

**remote memory**

Physical memory on another remote CPU board than the CPU accessing it.

**rtdm**

A MAXAda *environment* containing a package which provides a flexible interface to the key features of data monitoring.

**run-time system**

See *ARMS*.

**scheduling priority**

Used by the real-time kernels and the *run-time* executive to schedule tasks for execution within a *group*.

**semantic error**

An error in the semantic usage of language constructs.

**server**

The basic execution entity in the tasking mode. A server is an anonymous entity that executes on a CPU and is utilized by Ada tasks. Servers are identified by entities called *server groups*.

**server cache**

A set of *servers* that are currently unneeded by the application, but which can be placed back into service when they become necessary.

**server group**

A collection of one or more *servers*. Server groups are considered the execution resources that are available to Ada tasks. Server groups can be either named or anonymous, depending on their usage. See *task weight*.

**SHADOW ghost task**

A *ghost task* associated with a *software interrupt* or *hardware interrupt* task entry. It is not a physical task in any real sense. It merely acts as the virtual caller of the real task's entry. It does not, however, physically execute on any *server* or CPU.

**share mode**

The setting for a *compilation unit* or library that determines whether *shared objects* will be used. Share mode can be established via the **`-sm`** option to several utilities or pragma `SHARE_MODE` in the source. Values include: `shared`, `non_shared`, and `both`.

**shared object**

A shared collection of routines and data associated with a user's application during the link and execution phases of program generation. Shared objects are dynamically built (i.e. shared) objects that contain *position independent code*.

**shared package**

A package with all variables declared in its specification allocated in shared memory.

**software interrupt**

An operating system signal. See *interrupt* and *hardware interrupt*.

**stack (memory)**

A memory region used for subprogram and task data. Stacks dynamically grow and shrink during execution.

**syntax error**

An error in the form of grammatical constructs.

**system bus**

A single data path to *global memory* that all CPUs on Series 6000 systems share.

**task monitoring**

The real-time display and modification of user-defined tasks, *ghost tasks*, *server groups*, and the display of heap and virtual memory and system information via the **`a.monitor`** utility. This is usually performed during program debugging.

**task quantum**

A *quantum* associated with a task. It determines how long the task executes on a *server group* before being preempted by the *run-time* executive.

**task scheduling priority**

A *scheduling priority* associated with a task. It determines how the *run-time* executive schedules tasks for execution on *server groups*.

**task weight**

A configuration attribute that is *bound*, *multiplexed*, or *passive*. It determines whether a task is to have a specific server (execution resource) dedicated for its exclusive use, to share *servers* from a *server group*, or to borrow another task's server when executing.

**temporary unit options**

This set of options is temporarily associated with a unit and override its *permanent unit options*. The temporary unit options allow the user to "try out" options under consideration. These options can then be discarded or, if desired, can be added to the permanent unit options.

**text (memory)**

Statically sized memory segment used for the allocation of machine instructions, literals, and some constant data.

**TIMER ghost task**

A *ghost task* that exists only in programs that contain *multiplexed tasks* (other than the *ENVIRONMENT task*). If it exists, it is a *bound task* that is responsible for all timing operations associated with multiplexed tasks. The TIMER task acts as an "alarm clock" that triggers rescheduling events when certain times have been reached because of these operations.

**tracing**

A means of debugging and analyzing the performance of Ada applications, including multi-tasking applications via the **`a.trace`** and possibly the *NightTrace* tools. It involves the logging and display of predefined and user-defined trace events, data values (arguments), and timings with minimal impact on the application.

**unit**

Shorthand for *compilation units* as defined in the Ada 95 Reference Manual, units are the basic building blocks of the MAXAda *environments*. It is through units that MAXAda performs most all its library management and compilation activities.

**vendorlib**

*Environment* that contains mathematical functions; real-time, system service, and operating system bindings; and miscellaneous packages.

**warning**

An error message about a problem that is not sufficiently serious to prevent code generation or that indicates questionable use of a construct.

**weight**

See *task weight*.

# Index

## Symbols


### 4-19
.pprc file **4-78**, 4-80, 4-81
/tmp directory 4-34


## A


a.build 1-2, 2-12, 2-15, **4-3**, 4-6, 11-12
  automatic compilation 3-20
  example 2-4, 2-13, 2-14, 2-17, 4-17
  -IO option 3-25
  -noimport option 2-15
a.cat 1-1, **4-7**
  example 2-5
  -h option 2-5
a.chmod 1-1, **4-8**
a.compile 1-2, 3-21, **4-9**
a.demangle **4-11**
a.deps 1-2, **4-13**
a.edit 1-1, 2-12, **4-15**
  example 2-12
a.error 1-2, 3-26, **4-16**, 4-51
  example 4-18
a.expel 1-2, 3-10, **4-21**
a.fetch 1-2, 3-3, 3-5, **4-22**
a.freeze 1-1, **4-25**
a.help 1-2, **4-26**
a.hide 1-2, 3-10, **4-27**
a.install 1-2, **4-28**
a.intro 1-2, 2-2, 3-5, 4-27, **4-30**
  example 2-3, 2-12, 2-16, 4-17
a.invalid 1-2, **4-32**
a.link 1-2, **4-33**
a.ls 1-1, **4-35**
  example 2-5, 2-14
  -l option 2-5
  -v option 2-5
a.lssrc **4-42**
a.man 1-2, **4-44**
a.map 1-2, **4-47**, 6-2, 6-9, 6-22
a.mkenv 1-1, 2-1, 2-2, 3-1, **4-53**
  example 2-2, 2-12, 4-17
a.monitor 1-2, 3-38, **4-55**, 7-10, **12-4**, B-2
a.monitor
  C-3
a.nfs 1-1, **4-56**
a.options 1-1, 2-7, 3-20, 3-21, 3-31, **4-58**, -121
  modifying default options 4-20
a.partition 1-2, 3-5, 3-12, 3-13, 3-14, 3-16, 3-17, 3-18, **4-62**, 11-19
  -elab option 3-12, 3-13, 3-16, 3-17
  example 2-3, 2-6, 4-17
  -final option 3-12, 3-13, 3-16
  -list option 2-6
  -List option example 2-6
  -sl option 3-14
a.path 1-1, 3-3, 3-5, **4-74**, 11-12
  example 2-7, 2-13, 2-14
a.pclookup 1-2, **4-76**
a.pp 1-2, **4-77**, 4-82
a.release 1-1, **4-83**
a.resolve 1-2, 3-10, **4-85**
  example 2-17
a.restore 1-1, **4-86**
a.rmenv 1-1, **4-87**
a.rmsrc 1-2, 3-11, **4-88**
a.script 1-1, 3-6, **4-89**
a.syntax 1-2, **4-92**
a.tags 1-2, **4-94**
a.touch 1-2, **4-97**
a.trace 1-2, **4-98**, 4-111, 11-23, 11-24, 11-25, 11-26
  viewing trace events 11-24
Access
  Alignment **M-39**
Access type 3-25, 6-22, 8-4, 12-1, 12-2
Active partitions 2-3, 3-12, Glossary-1
AD (Ada) scheduling class 5-9
Ada
  packages 1-6, 1-7, 3-14, 6-19, 8-3, 8-4, 8-5, 9-10, 9-11, **10-1**, 10-1, 10-2, 10-4, 10-5, 10-6, 10-8, 10-10, M-13, M-35, M-36, M-48, -114
  pragmas 3-25, 3-31, 5-4, 8-1, 8-5, 12-2, A-1, A-3, B-2
Ada (AD) scheduling class 5-9
Ada 83 Reference Manual Glossary-6

Ada 95 Reference Manual 1-1, 2-7, 3-1, 3-3, 3-12, 3-20, 6-7, M-1, -106, -107, -108, -111, -114, -115, -118, -119, -120, -123, -125, -127, -128, -132, -133, -135, -136, Glossary-9, Glossary-10, Glossary-11, Glossary-15
Ada bindings 1-7, Glossary-1
Ada Executive 11-19
ADA.DYNAMIC_PRIORITIES package B-2
ADDR M-14
Address **M-35**
  logical M-35, Glossary-8
  machine M-35, Glossary-8
Address space 3-13
ADMIN ghost task **5-5**, 6-5, 6-28, Glossary-1
Alert errors **3-32**, -110, Glossary-1
Alignment **M-36**
  attribute **M-36**
  clause M-44
  minimal **M-37**
  optimal **M-37**
ALL_CALLS_REMOTE pragma -106
Ambiguous units 2-15, 3-10, 4-27, Glossary-1
ANSI/ISO/IEC-8652
  1995 5-1, 5-2, M-1
Application 5-1
Application configuration A-2
Archives 3-12, Glossary-1
ARMS **5-1**, Glossary-1
Array
  Alignment **M-39**
ASSIGNMENT pragma -106
ASYNCHRONOUS pragma -106
at clause M-44
at mod M-44
ATOMIC pragma -106
ATOMIC_COMPONENTS pragma -107
ATTACH_HANDLER pragma -107
Attachment index Glossary-1
Attribute
  'ADDR M-14
  'Address **M-35**
  'Alignment **M-36**
  'Component_Size **M-42**
  'External_Tag **M-43**
  'HAS_DISCRIMINANTS M-15
  'HAS_TAG M-15
  'INTERNAL_TAG M-15
  'KEY M-13
  'LOCK 8-3, 8-5, 10-3, M-13, **M-13**
  'PART_HAS_TAG M-15
  'REF M-14, M-48
  'SHM_ID M-13
  'Size
    Object **M-39**
    Subtype **M-40**
  'STORAGE_SIZE 5-12, 6-22, **6-27**, A-2
  'TAGGED M-15
  'UNLOCK 8-3, 8-5, 10-3, M-13, **M-13**

## B

Back-end 3-31
Binary semaphores 8-3, 8-5, **10-2**
BINARY_SEMAPHORES package **10-2**, 10-2, 10-3, 10-5
Binding 1-7
  immediate 3-13, 3-14, Glossary-6
  lazy 3-13, 3-14, Glossary-7
  NightTrace 11-8
  POSIX 1003.5 1-7
  sockets 1-7
Bound task **5-3**, A-2, C-1, -70
Busy wait 10-1, 10-10

## C

Cache 8-3
  COPYBACK mode 6-25
  mode Glossary-2
  NCACHE mode 6-25
  server 6-4, Glossary-13
CAP_SYS_ADMIN 12-4
Capability
  CAP_SYS_ADMIN 12-4
`CEILING_LOCKING locking policy` 5-9, 6-3
chmod 8-3
Class-wide
  Alignment **M-39**
Client-server services 10-11, A-3
Collection memory **5-11**, 6-22, Glossary-2
Comment 4-19
Communication
  inter-process **8-1**, 10-11
Companion ghost task Glossary-2
Compilation
  automatic 3-20
  separate 3-20
Compilation states 2-18, 3-20
  a.build -state 4-4
  a.compile -state 4-10
  a.ls -C state 4-35
  categorized (a.ls -n) 4-36
  compiled 3-22
  drafted 3-22

listing (a.ls -l) 4-36
parsed 3-22
uncompiled 3-22
Compilation unit Glossary-2
Compilation Utilities
a.build **4-3**
a.partition **4-62**
Compile options 2-10, **4-99**, Glossary-2
clearing 4-60
deleting 2-10, 4-60
effective 2-10, 3-20, 3-22, 4-59, Glossary-4
environment-wide 2-7, 3-6, 3-21, 4-59, Glossary-4, Glossary-5
listing 2-8, 4-59
modifying 2-9, 4-60
propagating temporary to permanent 2-10, 4-61
setting 2-8, 4-60
unit 3-11
permanent 2-8, 3-21, 4-59, Glossary-4, Glossary-11
temporary 2-9, 3-21, 4-59, Glossary-4, Glossary-14
Compiler
error message 3-26
error message processing 3-26
Component
Storage place **M-46**
Component_Size **M-42**
Components
implementation-defined **M-44**
Composite
Alignment **M-39**
type -123
Concurrent access A-2
Concurrent program 8-4, Glossary-2
Configuration
application A-2
errors A-1
kernel A-1
stack size 6-27
system A-1, B-1
Configuration Pragmas 3-9
Independent 3-9
Configuration pragmas 3-7, Glossary-2
independent Glossary-6
Consistency 3-23, Glossary-3
Context C-2
Control block
task C-2, C-3
Controlled
Alignment **M-39**
CONTROLLED pragma -107
CONVENTION pragma -108
COPYBACK cache mode 6-25
Core Utilities
a.build **4-3**
a.intro **4-30**
a.mkenv **4-53**
a.partition **4-62**
COURIER ghost task 6-5, 6-10, Glossary-3
cpp 4-78
cprs 3-40
CPU bias **5-4**, 6-12, 6-23, A-4, Glossary-3
cpu_bias 5-4
Cross referencing **4-94**
crossref a-monitor 3-38
Cyclic scheduler 10-6
CYCLIC_SCHEDULER package 10-6, B-2

**D**

Data memory **5-11**, 6-22, Glossary-3
Data monitoring 12-1, Glossary-3
DATA_RECORD pragma **-109**
Dead-code elimination 4-106
Debug level **4-100**, **-109**, Glossary-3
DEBUG pragma 12-1, **-109**
Debug Utilities
a.man **4-44**
a.map **4-47**
a.monitor **4-55**
a.pclookup **4-76**
a.trace **4-98**
Debugging
NightView 3-38, **C-1**, Glossary-10
tools 3-38
DEFAULT pseudo group **6-8**, Glossary-3
DEFAULT pseudo task **6-4**, 6-6, 6-8, 6-22, Glossary-3
DEFAULT_HANDLER package 3-14
Defaults 4-81
Dependency
analysis 4-5
loop 4-6
deprecated 1-6, 9-1, 9-13, Glossary-4
Deprecated feature Glossary-4
DEPRECATED_FEATURE pragma -110
Digits of precision M-11
Directory
/tmp 4-34
DISCARD_NAMES pragma -110
Discrete
Alignment **M-38**
DISPLAY environment variable 11-12
Distributed application Glossary-4
DONT_ELABORATE pragma -110

DWARF Glossary-4
Dynamic linker 3-13
Dynamic linking 3-13, Glossary-4

## E

Edge-triggered interrupts 7-1
EDITOR environment variable 2-12, 4-51
EDITOR envrionment variable 4-13, 4-30, 4-47, 4-92, 4-99
Effective compile options 2-10, 3-20, 3-22, 4-59, Glossary-4
ELABORATE pragma -111
ELABORATE_ALL pragma -111
ELABORATE_BODY pragma -111
Elaboration 6-4, 6-24, 8-4, **8-5**, 10-3, 10-5, A-1, M-13, Glossary-4
  archives 3-12
  shared objects 3-13
ELF Glossary-4
emacs **4-94**, **4-95**
Enumeration
  Alignment **M-38**
Enumeration type 12-2
Environment Search Path 2-7, 2-13, 2-14, 3-2, 3-10, 4-69, Glossary-4
  a.path 3-3
  adding environments to 2-13
  viewing 2-14
ENVIRONMENT task 5-1, 5-2, 5-5, 5-8, 5-9, 5-11, 5-12, Glossary-5
Environment task **6-4**, 6-6, 6-8, 6-16, 6-24, -134
Environment variable
  TMPDIR 4-34
Environment variables
  DISPLAY 11-12
  EDITOR 2-12, 4-13, 4-30, 4-47, 4-51, 4-92, 4-99
  LD_BIND_NOW 3-14
  PATH 2-1
  TMPDIR 4-34
Environment/State Utilities
  a.chmod **4-8**
  a.freeze **4-25**
  a.mkenv **4-53**
  a.options **4-58**
  a.path **4-74**
  a.release **4-83**
  a.restore **4-86**
  a.rmenv **4-87**
Environments 3-1, Glossary-4
  creating 2-1, 2-12
  environment-wide compile options 3-6, 3-21
  foreign 3-2, Glossary-5
  freezing Glossary-5
  local 3-2, Glossary-7
  relocating 3-4
  restoring 3-4
  supplied 1-6, 3-3
    deprecated 1-6, 9-1, 9-13, Glossary-4
    general 1-7, 9-1, 9-16, 11-8
    obsolescent 1-6, 9-1, 9-13, Glossary-10
    posix_1003.1 1-7, 9-1, 9-14
    posix_1003.5 1-7, 9-1, 9-15
    predefined 1-6, 3-3, 9-1, 9-6, Glossary-4, Glossary-11
    publiclib 1-6, 9-1, 9-11, Glossary-4, Glossary-12
    rtdm 1-6, 9-1, 9-12, Glossary-12
    sockets 1-7, 9-1, 9-16
    vendorlib 1-6, 3-14, 9-1, 9-8, 10-1, 11-3, Glossary-4, Glossary-15
Environment-wide compile options 2-7, 3-6, 3-21, 4-59, Glossary-4, Glossary-5
Environment-wide link options 3-34
Errors
  alert **3-32**, -110, Glossary-1
  configuration A-1
  fatal **3-33**, Glossary-5
  general **3-30**
  internal **3-33**, Glossary-6
  lexical **3-27**, 4-18, 4-19, Glossary-7
  messages **3-26**, 3-31
  panics **3-33**, Glossary-10
  processing 3-26, 3-31
  redirecting to a file 4-17
  run-time A-4
  semantic **3-29**, Glossary-13
  syntax **3-28**, Glossary-14
  user A-2
  warnings **3-32**
Exceptions C-3
  addresses 9-9
  and optimization 3-25
  misaligned access **3-25**
  originating_instruction 9-9
  PROGRAM_ERROR 8-3, 8-5, M-13
  propagation_map 9-9
  SEMAPHORE_ERROR **10-3**, 10-5
  STORAGE_ERROR A-2
  TASKING_ERROR **A-1**, A-4
  unhandled 6-15, -134
  USE_ERROR -70
Executive
  run-time **5-1**, 5-2, 5-3, 5-7, 6-11, 6-18, 10-11, A-2,

A-3, A-4, Glossary-12
Exit status 3-19
EXPORT pragma -111
Expressions 4-80
Extensibility 5-11
EXTERNAL_NAME pragma -112
External_Tag **M-43**

## F

Fast interrupt task Glossary-5
FAST_INTERRUPT_TASK pragma -113
Fatal errors **3-33**, Glossary-5
Fetched units 3-3, 3-10, 4-61
FIFO_WITHIN_PRIORITIES 5-3, 5-8, 5-9, 6-3
File
.pprc **4-78**, 4-80, 4-81
ipc.h 9-10
map 6-2, Glossary-8
shm.h 9-10
Finalization
archives 3-12
shared objects 3-13
Fixed point
Alignment **M-38**
Fixed-point type 12-2
FLOAT type M-11
Floating point
Alignment **M-38**
Floating-point type 12-2
Foreign environments 3-2, Glossary-5
Foreign units 3-10, Glossary-5
ftok 8-2, M-13
Function
UNCHECKED_CONVERSION 5-13

## G

general 1-7, 9-1, 9-16, 11-8
General errors **3-30**, Glossary-5
GENERAL passive task Glossary-5
Generic
debugging C-4
Ghost Task 12-9, 12-12
Ghost task 4-50, **5-5**, 6-4, Glossary-5
ADMIN **5-5**, 6-5, 6-28, Glossary-1
companion Glossary-2
COURIER 6-5, 6-10
INTR_COURIER 6-5, 6-10, Glossary-7
SHADOW 6-5, 6-10, **7-4**, **7-6**, Glossary-13
TIMER **5-5**, 6-5, Glossary-14
Global memory 6-23, **6-24**, Glossary-6
GLOBAL optimization 3-25
graphic_character M-5
Group Glossary-6
DEFAULT 6-8, Glossary-3
PREDEFINED **6-8**, 6-19, **6-19**
server 5-1, **6-8**
GROUP_CPU_BIAS pragma 5-4, 6-13, **6-19**, 6-19, B-2, -113
GROUP_PRIORITY pragma **6-18**, -113
GROUP_SERVERS pragma **6-19**, -114
growth_limit qualifier keyword **4-107**, -121

## H

Handler
interrupt Glossary-7
protected procedure Glossary-12
HAPSE Glossary-6
Hardness of memory Glossary-6
Hardware interrupt 7-6, Glossary-6
HAS_DISCRIMINANTS M-15
HAS_TAG M-15
Heap 5-11, Glossary-6
Hung processes A-3

## I

Immediate binding 3-13, 3-14, Glossary-6
Implementation-defined Characteristics M-1
Implementation-defined components **M-44**
IMPLICIT_CODE pragma **-114**
Implicitly-included libraries 4-72
IMPORT pragma -114
Incrementally updateable partition 4-110
Independent configuration pragmas 3-9, Glossary-6
Index
attachment Glossary-1
INDIVISIBLE_OPERATIONS package **10-8**, 10-10
Informational messages 3-31
INLINE pragma 3-25, **-115**
inline_line_count qualifier keyword **4-105**
inline_nesting_depth qualifier keyword **4-105**
inline_statement_limit qualifier keyword **4-106**
inlines_per_compilation qualifier keyword **4-106**
Insertion
machine code -116, Glossary-8
machine-code **M-48**, -116
INSPECTION_POINT pragma -116

Instantiation C-4
Instruction set
    Pentium M-49
Integer
    Alignment **M-38**
Integer type 12-2
Interest levels 9-13
INTERESTING pragma 9-13, -117
interesting qualifier keyword **4-107**
INTERFACE pragma -117
INTERFACE_NAME pragma -117
INTERFACE_OBJECT pragma -118
INTERFACE_SHARED pragma -118
Internal errors **3-33**, Glossary-6
Internal Utilities
    a.compile **4-9**
    a.deps **4-13**
    a.error **4-16**
    a.install **4-28**
    a.link **4-33**
    a.pp **4-77**
INTERNAL_TAG M-15
Interoptimization 3-24, 4-3, Glossary-6
Inter-process communication **8-1**, 10-11
Interrupt Glossary-6
    handler Glossary-7
    hardware 7-6, Glossary-6
    software Glossary-13
    task Glossary-5
INTERRUPT_HANDLER pragma -118
INTERRUPT_PRIORITY pragma B-2, -118
INTR_COURIER ghost task 6-5, 6-10, Glossary-7
IPC **8-1**, 10-11
IPC flags 8-3
ipc.h file 9-10
ipcrm 8-3, 8-4, 8-5
ipcs 8-3, 8-4

## K

Kernel configuration A-1
KEY M-13

## L

Lazy binding 3-13, 3-14, Glossary-7
LD_BIND_NOW environment variable 3-14
Level
    debug Glossary-3
    optimization Glossary-10
Lexical
    errors 4-18, 4-19
Lexical errors **3-27**, Glossary-7
Libraries
    implicitly-included 4-72
Library
    supplied 1-6
Library unit Glossary-7
Limits
    shell 6-27
Link method 4-67, Glossary-7
Link options 3-34, 4-65, **4-109**, Glossary-7
    -bound 4-111
    environment-wide 3-34
    in source code 3-35
    -incr 4-110
    incrementally updateable partition 4-110
    -multiplexed 4-111
    -nosoclosure 4-111
    obscurity checks 4-112
    share path 4-110
    shared object transitive closure 4-111
    -skipobscurity 4-112
    -sl 4-110
    -sp 4-110
    specifying 3-34
    task weight 4-111
    -trace 4-110
    tracing 4-110
Link rule 4-67, Glossary-7
LINK_OPTION pragma -119
Linker
    dynamic 3-13
LINKER_OPTIONS pragma -119
Linking
    dynamic Glossary-4
    static Glossary-14
list C-4
LIST pragma -119
Listing
    partitions 2-6
    units 2-5
listing effective options 2-10
Local environments 3-2, Glossary-7
Local memory 6-23, **6-24**, 8-3, Glossary-8
Local units 3-9
LOCK M-13, **M-13**
Lock
    memory pages 8-3, A-4
    spin 10-1
    state **6-25**, Glossary-8
Locking policy
    `CEILING_LOCKING` 5-9, 6-3
    default 5-9, 6-3

protected object 6-3
LOCKING_POLICY pragma 5-9, **6-3**, -120
Logical address M-35, Glossary-8
LONG_FLOAT type M-11
Loops in dependencies 4-6
loops qualifier keyword **4-106**, -121
LSB Glossary-8

## M

Machine address M-35, Glossary-8
MACHINE_CODE package -114
Machine-code insertion **M-48**, -116, Glossary-8
Pentium M-48
Main subprogram 2-3, 4-36, 4-64, 5-1, Glossary-8
exit status 3-19
requirements 3-19
Map file 6-2, Glossary-8
MAP_FILE pragma **6-2**, -120
MAPSE Glossary-8
MAX_PRIORITY 6-18
MAXAda Glossary-9
MAXAda installation Glossary-9
MAXAda release Glossary-9
MAXIMAL optimization 3-25
MCI **M-48**, -116, Glossary-8, Glossary-9
memadvise 6-25
memcntl A-4
Memory
attributes 6-24
collection **5-11**, 6-22, Glossary-2
data **5-11**, 6-22, Glossary-3
global 6-23, **6-24**, Glossary-6
heap 5-11
local 6-23, **6-24**
management 5-11
page locking 8-3, A-4
pool **6-23**
remote Glossary-12
segment 3-13
shared **8-1**
stack **5-12**, 6-22, Glossary-13
text **5-11**, 6-21, Glossary-14
Memory pool **6-21**, 6-25, 6-26, A-4, Glossary-9
lock state **6-25**, Glossary-8
pad 6-28
size 6-26
MEMORY_POOL pragma 5-4, **6-23**, A-1, B-2, -120
Minimal alignment **M-37**
Misaligned access **3-25**
mlock 6-26
mmap A-4
Monitoring 3-38
data 12-1, Glossary-3
task 12-3, Glossary-14
mpadvise A-5
MSB Glossary-9
Multiple process communication 8-1
Multiplexed task C-1, -70
Multithreading A-2
Mutual exclusion 10-1, 10-2, 10-4, 10-10

## N

Nationalities 3-9
Native units 3-9, Glossary-9
Naturalization 2-15, 3-3
inhibiting 2-15
Naturalized units 3-3, 3-9, Glossary-9
NCACHE cache mode 6-25
NIGHT_TRACE_BINDINGS package 11-8
NightBench 1-7
NightProbe 1-8
NightSim 1-8, Glossary-9
NightTrace 1-7, 4-111, 11-8, 11-12, 11-23, Glossary-10
binding 11-8
configuration file
creating 11-25
modifying 11-26
display utility 11-12
ntrace 11-12, 11-25
ntraceud 11-12, 11-21
user daemon 11-15, 11-17, 11-19, 11-21
viewing trace events 11-25
NightView 1-8, 3-38, 11-13, **C-1**, Glossary-10
NightView debugger command
handle C-3
info exception C-3
list C-4
print C-5
select-context C-3
set-language C-5
no_bsem parameter 8-5, M-14
Non-tasking
run-time 5-7, 5-8
noreorder qualifier keyword -121
NORMALIZE_SCALARS pragma -121
ntrace 4-111, 11-12, 11-25
ntraceud 11-12, 11-21
NUMA Glossary-10
nview 3-38

## O


Objects
  protected 5-9, 6-3, -120
  shared 3-13, Glossary-13
objects qualifier keyword 3-31, **4-106**, -121
Obscurities Glossary-10
Obscurity checks 4-112
obsolescent 1-6, 9-1, 9-13, Glossary-10
Operating system quantum 6-14, Glossary-10
Operating system scheduling priority 6-11, Glossary-10
Opportunism **4-100**, Glossary-10
opt_class qualifier keyword **4-106**, -121
OPT_FLAGS pragma 3-25, 3-31, **-121**
OPT_LEVEL pragma **-122**
Optimal alignment **M-37**
Optimization 3-31
  levels **4-101**, -122, Glossary-10
optimization_size_limit qualifier keyword **4-106**
OPTIMIZE pragma -122
optimize_for_space qualifier keyword **4-106**, -121
Options
  compile **4-99**, Glossary-2
    See also Compile options
  effective (compile) 3-20, 3-22, Glossary-4
  hierarchical relationship 3-20, 3-22, Glossary-4
  link **4-109**
  negating 3-22
  permanent unit (compile) 2-8, 3-21, 4-59, Glossary-4, Glossary-11
  persistent 3-20, Glossary-11
  -Q 4-104, **4-105**
  temporary unit (compile) 2-9, 3-21, 4-59, Glossary-4, Glossary-14
OS scheduling classes 5-7, 5-8
  Ada (AD) 5-9


## P


P_CPUBIAS A-1
P_TSHAR A-1
PACK pragma 3-25, -123
Package
  ADA.DYNAMIC_PRIORITIES B-2
  BINARY_SEMAPHORES **10-2**, 10-2, 10-3, 10-5
  CYCLIC_SCHEDULER 10-6, B-2
  DEFAULT_HANDLER 3-14
  INDIVISIBLE_OPERATIONS **10-8**, 10-10
  MACHINE_CODE -114
  NIGHT_TRACE_BINDINGS 11-8
  REAL_TIME_DATA_MONITORING 3-40, 9-12, 12-2, B-2, -117
  RESCHEDULING_CONTROL B-2
  RTC_CONTROL 10-6
  RUNTIME_CONFIGURATION 6-1, 6-4, 6-8, 6-9, 6-13, 6-18, 6-19, 6-20, A-4, B-2
  SHARED_MEMORY_SUPPORT 8-4, **9-10**
  SPIN_LOCKS **10-1**, B-2
  SYNC_PACKAGE 10-2
  SYSTEM_INFORMATION 9-11
  TASK_SYNCHRONIZATION 10-6, B-2
  TASKING_SEMAPHORES 10-4
  USER_TRACE 11-3
  user_trace 10-8
  user_trace.raw 10-8
  USERDMA_SUPPORT 10-11
Packages
  shared Glossary-13
PAGE pragma -123
Panics **3-33**, Glossary-10
Parallel
  compilation 4-5
  dependency analysis 4-5
PART_HAS_TAG M-15
Partition Glossary-11
Partitions
  active 2-3, 3-12, Glossary-1
  archives 3-12
  building 2-4
  defining 2-3
  elaboration - archives 3-12
  elaboration - shared objects 3-13
  finalization - archives 3-12
  finalization - shared objects 3-13
  incrementally updateable 4-110
  listing 2-6
  obscurity checks 4-112
  shared objects 3-13, Glossary-13
  types 3-12
Passive task C-1
  GENERAL Glossary-5
  SERVER Glossary-13
PASSIVE_TASK pragma -123
PATH environment variable 2-1
Pentium
  machine-code insertions M-48
Performance 5-2
Permanent unit compile options 2-8, 3-21, 4-59, Glossary-4, Glossary-11
Persistent options Glossary-11
plock 6-26
Pool
  lock state **6-25**, Glossary-8
  memory **6-21**, **6-23**, 6-25, 6-26, Glossary-9
  pad 6-28

size 6-26
POOL_CACHE_MODE pragma **6-25**, -124
POOL_LOCK_STATE pragma **6-25**, B-2, -124
POOL_PAD pragma **6-28**, -124
POOL_SIZE pragma **6-26**, A-2, -124
Position independent code (PIC) 3-14, Glossary-11
POSIX 1-7
posix_1003.1 1-7, 9-1, 9-14
posix_1003.5 1-7, 9-1, 9-15
PowerPC
instruction set M-49
Pragma 4-81
ALL_CALLS_REMOTE -106
ASSIGNMENT -106
ASYNCHRONOUS -106
ATOMIC -106
ATOMIC_COMPONENTS -107
ATTACH_HANDLER -107
CONTROLLED -107
CONVENTION -108
DATA_RECORD **-109**
DEBUG 12-1, **-109**
DEPRECATED_FEATURE -110
DISCARD_NAMES -110
DONT_ELABORATE -110
ELABORATE -111
ELABORATE_ALL -111
ELABORATE_BODY -111
EXPORT -111
EXTERNAL_NAME -112
FAST_INTERRUPT_TASK -113
GROUP_CPU_BIAS 5-4, 6-13, **6-19**, 6-19, B-2, -113
GROUP_PRIORITY **6-18**, -113
GROUP_SERVERS **6-19**, -114
IMPLICIT_CODE **-114**
IMPORT -114
INLINE 3-25, **-115**
INSPECTION_POINT -116
INTERESTING 9-13, -117
INTERFACE -117
INTERFACE_NAME -117
INTERFACE_OBJECT -118
INTERFACE_SHARED -118
INTERRUPT_HANDLER -118
INTERRUPT_PRIORITY B-2, -118
LINK_OPTION -119
LINKER_OPTIONS -119
LIST -119
LOCKING_POLICY 5-9, **6-3**, -120
MAP_FILE **6-2**, -120
MEMORY_POOL 5-4, **6-23**, A-1, B-2, -120
NORMALIZE_SCALARS -121
OPT_FLAGS 3-25, 3-31, **-121**
OPT_LEVEL **-122**
OPTIMIZE -122
PACK 3-25, -123
PAGE -123
PASSIVE_TASK -123
POOL_CACHE_MODE **6-25**, -124
POOL_LOCK_STATE **6-25**, B-2, -124
POOL_PAD **6-28**, -124
POOL_SIZE **6-26**, A-2, -124
PREELABORATE -125
PRIORITY B-2, -125
PROTECTED_PRIORITY 6-28, -125
PURE -127
QUEUING_POLICY **6-2**, -127
REMOTE_CALL_INTERFACE -127
REMOTE_TYPES -127
RESTRICTIONS -128
RETURN_CONVENTION -128
REVIEWABLE -129
RUNTIME_DIAGNOSTICS **6-1**, A-4, -129
SERVER_CACHE_SIZE **6-4**, -129
SHARE_BODY **-129**
SHARE_MODE **-130**
SHARED -131
SHARED_PACKAGE **8-1**, **8-2**, 8-4, 8-5, 12-2, M-13, -131
SHARED_PASSIVE -131
SPECIAL_FEATURE **-131**
STORAGE_SIZE -132
SUPPRESS 3-25, 5-13, -132
SUPPRESS_ALL **-133**
TASK_CPU_BIAS 5-4, **6-12**, 6-13, 6-19, A-1, B-2, -133
TASK_DISPATCHING_POLICY 5-9, -133
TASK_HANDLER **6-15**, -134
TASK_PRIORITY **6-11**, A-1, B-2, -134
TASK_QUANTUM 5-8, **6-14**, -134
TASK_WEIGHT **6-9**, -135
TDESC -135
TRAMPOLINE -135
VOLATILE A-3, -135
VOLATILE_COMPONENTS -136
Pragmas Glossary-11
Precision M-11
predefined 1-6, 3-3, 9-1, 9-6, Glossary-4, Glossary-11
PREDEFINED group **6-8**, 6-19, **6-19**
Predefined Language Environment 2-7
Predefined trace events 11-1, 11-19, 11-21, 11-24
predefined trace events 11-2
PREELABORATE pragma -125
print C-5
Priorities 5-9
Priority **6-11**, 10-6, M-13, -134
inheritance 10-2, 10-11

operating system 6-11, Glossary-10
scheduling Glossary-12
task scheduling 6-11, Glossary-14
PRIORITY pragma B-2, -125
PRIORITY_OF_ENVIRONMENT 5-9
Privilege
P_CPUBIAS A-1
P_TSHAR A-1
Process Glossary-11
communication **8-1**, 10-11
hung A-3
Program 5-1, Glossary-12
concurrent 8-4
PROGRAM_ERROR exception 8-3, M-13
Programming
caveats 3-25
hints 3-25
Protected
Alignment **M-39**
Protected objects 5-9, 6-3, -120
protected procedure handler Glossary-12
PROTECTED_PRIORITY pragma 6-28, -125
Pseudo group
DEFAULT 6-8, Glossary-3
Pseudo task
DEFAULT **6-4**, 6-6, 6-8, 6-22, Glossary-3
publiclib 1-6, 9-1, 9-11, Glossary-4, Glossary-12
PURE pragma -127


## Q


-Q options 4-104, **4-105**
growth_limit **4-107**, -121
inline_line_count **4-105**
inline_nesting_depth **4-105**
inline_statement_limit **4-106**
inlines_per_compilation **4-106**
interesting **4-107**, 9-13
loops **4-106**, -121
noreorder -121
objects 3-31, **4-106**, -121
opt_class **4-106**, -121
optimization_size_limit **4-106**
optimize_for_space **4-106**, -121
unroll_limit -121
unroll_limit_const -121
unroll_limit_var -121
Qualifier keyword. See -Q options
Quantum 5-3, Glossary-12
operating system 6-14, Glossary-10
task 6-14, Glossary-14
Queuing policy 6-2, Glossary-12
QUEUING_POLICY pragma **6-2**, -127


## R


REAL_TIME_DATA_MONITORING package 3-40, 9-12, 12-2, B-2, -117
Real-time
Ada tasking 5-2, 5-7
clocks 7-1
data monitoring 12-1, Glossary-3
debugging 3-38
extensions 10-1
task monitoring 12-3, Glossary-14
vendorlib packages 10-1
Record
Alignment **M-39**
representation clauses **M-44**
REF M-14
release Glossary-12
Relevance 3-23
Relocation 3-13, Glossary-12
Remote memory Glossary-12
REMOTE_CALL_INTERFACE pragma -127
REMOTE_TYPES pragma -127
RESCHEDULING_CONTROL package B-2
RESTRICTIONS pragma -128
RETURN_CONVENTION pragma -128
REVIEWABLE pragma -129
ROUND_ROBIN_ADJUSTABLE_PRIORITIES 5-8
ROUND_ROBIN_PRIORITIES 5-8
RTC_CONTROL package 10-6
rtdm 1-6, 9-1, 9-12, Glossary-12
Run-time errors A-4
Run-time executive **5-1**, 5-2, 5-3, 5-7, 6-11, 6-18, 10-11, A-2, A-3, A-4, Glossary-12
RUNTIME_CONFIGURATION package 6-1, 6-4, 6-8, 6-9, 6-13, 6-18, 6-19, 6-20, A-4, B-2
RUNTIME_DIAGNOSTICS pragma **6-1**, A-4, -129


## S


s# 4-96
Scheduling
classes 5-7, 5-8
priority Glossary-12
task 5-3
Scheduling classes 5-7, 5-8
Ada (AD) 5-9
Scheduling priority
operating system 6-11, Glossary-10

task 6-11, Glossary-14
select-context C-3
Semantic errors **3-29**, Glossary-13
SEMAPHORE_ERROR exception **10-3**, 10-5
Semaphores
binary 8-3, 8-5, **10-2**
Server 5-1, Glossary-13
Server cache 6-4, Glossary-13
Server group 5-1, **6-8**, Glossary-13
SERVER passive task Glossary-13
server_block 10-11, **A-3**
SERVER_CACHE_SIZE pragma **6-4**, -129
server_wake1 10-11, **A-3**
server_wakevec 10-11, **A-3**
set-language C-5
SHADOW ghost task 6-5, 6-10, **7-4**, **7-6**, Glossary-13
Share mode 3-14, **4-101**, Glossary-13
Share path 3-14, 4-110
SHARE_BODY pragma **-129**
SHARE_MODE pragma **-130**
Shared
memory **8-1**
Shared memory segment **8-3**, 8-5, M-13, M-14
Shared objects 3-13, Glossary-13
issues to consider 3-15
share mode 3-14, **4-101**, Glossary-13
share path 3-14
transitive closure 4-111
Shared packages 8-4, 8-5, Glossary-13
SHARED pragma -131
SHARED_MEMORY_SUPPORT package 8-4, **9-10**
SHARED_PACKAGE pragma **8-1**, **8-2**, 8-4, 8-5, 12-2, M-13, -131
bind parameter 8-3
ipc parameter 8-2
key parameter 8-2
mode parameter 8-3
no_bsem parameter 8-3
SHM_HARD parameter 8-3
SHM_LOCAL parameter 8-3
SHM_LOCK parameter 8-3
SHM_RDONLY parameter 8-2
SHARED_PASSIVE pragma -131
Shell environment variable
TMPDIR 4-34
Shell limits 6-27
SHM flags 8-3
shm.h file 9-10
SHM_COPYBACK parameter 8-3
SHM_ID M-13
SHM_RDONLY parameter 8-5, **M-14**
shmat 8-1, 9-10
shmbind 8-5
shmctl 8-4, 9-10
shmdt 9-10
shmget 8-1, **8-2**, **8-3**, **M-13**
SIGADA signal 7-3
SIGBUG signal 4-112, 4-114
SIGFPE signal 7-3
SIGILL signal 4-112, 4-114
Signals 7-1
SIGADA 7-3
SIGBUS 4-112, 4-114
SIGFPE 7-3
SIGILL 4-112, 4-114
SIGSEGV 4-112, 4-114, 7-3, 8-3
SIGSEGV signal 4-112, 4-114, 7-3, 8-3
sinfo 9-11
Size
Object **M-39**
Subtype **M-40**
Sleepy wait 10-2, 10-4
Sockets 1-7
sockets 1-7, 9-1, 9-16
Soft links 3-14, 4-110
Software interrupt Glossary-13
Source File Utilities
a.intro **4-30**
a.rmsrc **4-88**
a.syntax **4-92**
a.tags **4-94**
SPECIAL_FEATURE pragma **-131**
SPIN_LOCKS package **10-1**, B-2
Stack memory **5-12**, 6-22, Glossary-13
Static linking Glossary-14
stderr 3-26, 4-30, 4-47, 4-92, 4-99
stdin 4-11, 4-13, 4-30, 4-47, 4-48, 4-76, 4-78, 4-88, 4-92, 4-94
stdout 4-9, 4-11, 4-13, 4-16, 4-17, 4-30, 4-47, 4-78, 4-88, 4-92, 4-94, 4-99
Storage place
component **M-46**
STORAGE_ERROR exception A-2
STORAGE_SIZE pragma -132
stub# 4-96
Subprogram
main 2-3, 5-1, Glossary-8
TEST_AND_SET 10-8
SUPPRESS pragma 3-25, 5-13, -132
SUPPRESS_ALL pragma **-133**
SYNC_PACKAGE package 10-2
Syntax errors **3-28**, Glossary-14
System bus Glossary-14
System configuration A-1, B-1
`System.Priority` 5-8
SYSTEM_INFORMATION package 9-11

## T


TAGGED M-15
Tagged
  Alignment **M-39**
Task
  ADMIN **5-5**, 6-5, 6-28
  Alignment **M-39**
  attributes 6-9
  bound **5-3**, A-2, C-1, -70
  control block C-2, C-3
  COURIER 6-5, 6-10
  CPU binding 5-4
  DEFAULT **6-4**, 6-6, 6-8, 6-22, Glossary-3
  dispatching policy 5-3, 5-8, 5-9, 6-3, -133
  ENVIRONMENT 5-1, 5-2, 5-5, 5-8, 5-9, 5-11, 5-12
  environment **6-4**, 6-6, 6-8, 6-16, 6-24, -134
  fast interrupt Glossary-5
  GENERAL Glossary-5
  ghost **5-5**, 6-4, Glossary-1, Glossary-5, Glossary-7
  interrupt entries 7-1
  INTR_COURIER 6-5, 6-10
  monitoring 12-3, Glossary-14
  multiplexed C-1, -70
  multithreading A-2
  passive C-1
  priority 6-11
  quantum 6-14, Glossary-14
  scheduling 5-3
  scheduling priority 6-11, Glossary-14
  SERVER Glossary-13
  SHADOW 6-5, 6-10, **7-4**, **7-6**, Glossary-13
  time slicing 5-3, 6-14
  TIMER **5-5**, 6-5, Glossary-14
  type 6-5, 6-7, 6-22, 12-2
  weight 4-111, 5-1, 5-3, 6-9, Glossary-14
Task dispatching policy 5-3, 5-8, 5-9, 6-3, -133
  `FIFO_WITHIN_PRIORITIES` 5-3, 5-8, 5-9, 6-3
  `ROUND_ROBIN_ADJUSTABLE_PRIORITIES` 5-8
  `ROUND_ROBIN_PRIORITIES` 5-8
Task weight 4-111
TASK_CPU_BIAS pragma 5-4, **6-12**, 6-13, 6-19, A-1, B-2, -133
TASK_DISPATCHING_POLICY pragma 5-9, -133
TASK_HANDLER pragma **6-15**, -134
TASK_PRIORITY pragma **6-11**, A-1, B-2, -134
TASK_QUANTUM pragma 5-8, **6-14**, -134
TASK_SYNCHRONIZATION package 10-6, B-2
TASK_WEIGHT pragma **6-9**, -135
Tasking
  model 5-1
  real-time Ada 5-2
  run-time 5-7, 5-8
  semaphores 10-4
TASKING_ERROR exception A-1, A-4
TASKING_SEMAPHORES package 10-4
TCB C-2, C-3
TDESC pragma -135
Temporary unit compile options 2-9, 3-21, 4-59, Glossary-4, Glossary-14
TEST_AND_SET subprogram 10-8
Text memory **5-11**, 6-21, Glossary-14
TIMER ghost task **5-5**, 6-5, Glossary-14
TMPDIR environment variable 4-34
Trace events 11-1
  predefined 11-1, 11-2, 11-19, 11-21, 11-24
  user-defined 11-2, 11-19, 11-21, 11-24
  viewing 11-23, 11-24, 11-25
Trace points 11-1
Tracing 4-110, Glossary-15
  log files 11-22
  trace events 11-1
  trace points 11-1
  user table 11-23, 11-25
TRAMPOLINE pragma -135
Transitive closure 4-111
Troubleshooting A-1
Type
  access 3-25, 6-22, 8-4, 12-1, 12-2
  access Alignment **M-39**
  array Alignment **M-39**
  class-wide Alignment **M-39**
  composite -123
  composite Alignment **M-39**
  controlled Alignment **M-39**
  discrete Alignment **M-38**
  enumeration 12-2
  enumeration Alignment **M-38**
  fixed point Alignment **M-38**
  fixed-point 12-2
  FLOAT M-11
  floating point Alignment **M-38**
  floating-point 12-2
  integer 12-2
  integer Alignment **M-38**
  LONG_FLOAT M-11
  protected Alignment **M-39**
  record Alignment **M-39**
  tagged Alignment **M-39**
  task 6-5, 6-7, 6-22, 12-2
  task Alignment **M-39**
  universal_real M-11

## U


UNCHECKED_CONVERSION function 5-13
Understand for Ada 1-8
Unhandled exceptions. See Exceptions - unhandled.
Unit compile options 3-11
Unit Utilities
  a.cat **4-7**
  a.demangle **4-11**
  a.edit **4-15**
  a.expel **4-21**
  a.fetch **4-22**
  a.hide **4-27**
  a.invalid **4-32**
  a.ls **4-35**
  a.lssrc **4-42**
  a.resolve **4-85**
  a.touch **4-97**
Units Glossary-15
  ambiguous 2-15, 3-10, 4-27, Glossary-1
  compile options 3-11
  configuration pragmas 3-7, Glossary-2
  consistency 3-23, Glossary-3
  fetched 3-3, 3-10, 4-61
  foreign 3-10, Glossary-5
  introducing 2-2
  library Glossary-7
  listing 2-5
  local 3-9
  modifying 2-12
  nationalities 3-9
  native 3-9, Glossary-9
  naturalized 3-3, 3-9, Glossary-9
  viewing source 2-5
universal_real type M-11
UNLOCK M-13, **M-13**
unroll_limit qualifier keyword -121
unroll_limit_const qualifier keyword -121
unroll_limit_var qualifier keyword -121
USE_ERROR exception -70
User errors A-2
User table 11-23, 11-25
USER_TRACE package 11-3
user_trace package 10-8
user_trace.raw package 10-8
User-defined trace events 11-2, 11-19, 11-21, 11-24
USERDMA_SUPPORT package 10-11
usermap 10-11
Utilities
  a.build **4-3**, 4-6, 11-12
  a.cat **4-7**
  a.chmod **4-8**
  a.compile **4-9**
  a.demangle **4-11**
  a.deps **4-13**
  a.edit **4-15**
  a.error 3-26, **4-16**, 4-51
  a.expel **4-21**
  a.fetch 3-5, **4-22**
  a.freeze **4-25**
  a.help **4-26**
  a.hide **4-27**
  a.install **4-28**
  a.intro 3-5, 4-27, **4-30**
  a.invalid **4-32**
  a.link **4-33**
  a.ls **4-35**
  a.lssrc **4-42**
  a.man **4-44**
  a.map **4-47**, 6-2, 6-9, 6-22
  a.mkenv **4-53**
  a.monitor 3-38, **4-55**, 7-10, **12-4**, B-2, C-3
  a.nfs **4-56**
  a.options 2-7, 3-31, **4-58**, -121
  a.partition 3-5, **4-62**, 11-19
  a.path 3-5, **4-74**, 11-12
  a.pclookup **4-76**
  a.pp **4-77**, 4-82
  a.release **4-83**
  a.resolve **4-85**
  a.restore **4-86**
  a.rmenv **4-87**
  a.rmsrc **4-88**
  a.script 3-6, **4-89**
  a.syntax **4-92**
  a.tags **4-94**
  a.touch **4-97**
  a.trace **4-98**, 4-111, 11-23, 11-24, 11-25, 11-26
  NightSim Glossary-9
  NightTrace 4-111, Glossary-10
  NightTrace display 11-12
  NightView **C-1**, Glossary-10
  nview 3-38
Utility
  a.demangle **4-11**


## V


vendorlib 1-6, 3-14, 9-1, 9-8, 10-1, 11-3, Glossary-4, Glossary-15
vi 4-16, 4-18, 4-19, 4-20, **4-94**, 4-95, 4-96
VOLATILE pragma A-3, -135
VOLATILE_COMPONENTS pragma -136

## W

Wait
    busy  10-1, 10-10
    sleepy  10-2, 10-4
Warnings  **3-32**
Weight
    task  5-1, 5-3, 6-9, Glossary-14

## X

X server  11-12

**Spine for 2.0” Binder**

**Product Name: 0.5" from top of spine, Helvetica, 36 pt, Bold**

**Volume Number (if any): Helvetica, 24 pt, Bold**

**Volume Name (if any): Helvetica, 18 pt, Bold**

**Manual Title(s): Helvetica, 10 pt, Bold, centered vertically within space above bar, double space between each title**

**Bar: 1" x 1/8" beginning 1/4" in from either side**

**Part Number: Helvetica, 6 pt, centered, 1/8" up**