

AD-A279 645

ATION PAGE

Form Approved
GPM No.

①

Public
and no
suppl
28503-30 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering
Send comments regarding this burden estimate or any other aspect of this collection of information, including
Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA
Management and Budget, Washington, DC 20503.

1. AG

3. REPORT TYPE AND DATES

4. TITLE AND

940325S1.11354, AVE: 94ddc500 4
DDC-I, DACS Sun SPARC/Solaris Native Ada Compiler System,
Version 4.6.2

5. FUNDING

6. AUTHORS:

National Institute of Standards and Technology
Gaithersburg, Maryland

7. PERFORMING ORGANIZATION NAME(S) AND

National Institute of Standards and Technology
Building 255, Room A266
Gaithersburg, Maryland 20899
USA8. PERFORMING
ORGANIZATION

9. SPONSORING/MONITORING AGENCY NAME(S) AND

Ada Joint Program Office
The Pentagon, Rm 3E118
Washington, DC 20301-308010. SPONSORING/MONITORING
AGENCY

11. SUPPLEMENTARY

DTIC

ELECTE

MAY 26 1994

12a. DISTRIBUTION/AVAILABILITY

Approved for Public Release; distribution unlimited

12b. DISTRIBUTION

13. (Maximum 200)

Host/Target: Sun SPARCclassic (under Solaris, Release 2.1)

94-15744



14. SUBJECT

Ada programming language, Ada Compiler Validation Summary Report, Ada
Compiler Val. Capability Val. Testing, Ada Val. Office, Ada Val. Facility
ANSI/MIL-STD-1815A, AJP0

15. NUMBER OF

17. SECURITY

CLASSIFICATION
UNCLASSIFIED

18. SECURITY

UNCLASSIFIED

19. SECURITY

CLASSIFICATION
UNCLASSIFIED

20. LIMITATION OF

UNCLASSIFIED

940325

Standard Form 298, (Rev. 2-89)
Prescribed by ANSI Std.

DTIC QUALITY INSPECTED 1

94 5 25 027

AVF Control Number: NIST94DDC500_4_1.11

DATE COMPLETED

BEFORE ON-SITE: 94-03-18

AFTER ON-SITE: 94-03-28

REVISIONS: 94-04-11

Ada COMPILER

VALIDATION SUMMARY REPORT:

Certificate Number: 940325S1.11354

DDC-I

DACS Sun SPARC/Solaris Native Ada Compiler System, Version 4.6.2

Sun SPARCclassic => Sun SPARCclassic

Prepared By:

Software Standards Validation Group

Computer Systems Laboratory

National Institute of Standards and Technology

Building 225, Room A266

Gaithersburg, Maryland 20899

U.S.A.

Accession For	
NTIS CRA&I	☒
DTIC TAB	☐
Unannounced	☐
Justification _____	
By _____	
Distribution /	
Availability Codes	
Dist	Avail and/or Special
A-1	

AVF Control Number: NIST94DDC500_4_1.11

Certificate Information

The following Ada implementation was tested and determined to pass ACVC 1.11. Testing was completed on March 25, 1994.

Compiler Name and Version: DACS Sun SPARC/Solaris Native Ada Compiler System, Version 4.6.2

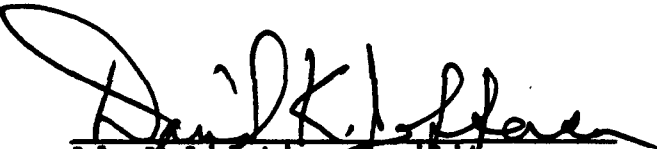
Host Computer System: Sun SPARCclassic running under Solaris, Release 2.1

Target Computer System: Sun SPARCclassic running under Solaris, Release 2.1

See section 3.1 for any additional information about the testing environment.

As a result of this validation effort, Validation Certificate 940325S1.11354 is awarded to DDC-I. This certificate expires 2 years after ANSI/MIL-STD-1815B is approved by ANSI.

This report has been reviewed and is approved.


Ada Validation Facility
Dr. David K. Jefferson
Chief, Information Systems
Engineering Division (ISED)


Ada Validation Facility
Mr. L. Arnold Johnson
Manager, Software Standards
Validation Group

Computer Systems Laboratory (CSL)
National Institute of Standards and Technology
Building 225, Room A266
Gaithersburg, Maryland 20899
U.S.A.


Ada Validation Organization
Director, Computer & Software
Engineering Division
Institute for Defense Analyses
Alexandria VA 22311


Ada Joint Program Office
David R. Basel
Deputy Director,
Ada Joint Program Office
Defense Information Systems Agency,
Center for Information Management
Washington DC 20301

U.S.A.

NIST94DDC500_4_1.11

DECLARATION OF CONFORMANCE

The following declaration of conformance was supplied by the customer.

Customer: DDC-I

Certificate Awardee: DDC-I

Ada Validation Facility: National Institute of Standards and
Technology
Computer Systems Laboratory (CSL)
Software Standards Validation Group
Building 225, Room A266
Gaithersburg, Maryland 20899
U.S.A.

ACVC Version: 1.11

Ada Implementation:

Compiler Name and Version: DACS Sun SPARC/Solaris Native Ada Compiler System,
Version 4.6.2

Host Computer System: Sun SPARCclassic running under Solaris, Release 2.1

Target Computer System: Sun SPARCclassic running under Solaris, Release 2.1

Declaration:

I the undersigned, declare that I have no knowledge of deliberate deviations from the
Ada Language Standard ANSI/MIL-STD-1815A ISO 8652-1987 in the implementation listed
above.

Jay H. Hoff
Customer Signature
Company DDC-I
Title

94-03-25
Date

Jay H. Hoff
Certificate Awardee Signature
Company DDC-I
Title

94-03-25
Date

TABLE OF CONTENTS

CHAPTER 1.....	1-1
INTRODUCTION.....	1-1
1.1 USE OF THIS VALIDATION SUMMARY REPORT.....	1-1
1.2 REFERENCES.....	1-2
1.3 ACVC TEST CLASSES.....	1-2
1.4 DEFINITION OF TERMS.....	1-3
CHAPTER 2.....	2-1
IMPLEMENTATION DEPENDENCIES.....	2-1
2.1 WITHDRAWN TESTS.....	2-1
2.2 INAPPLICABLE TESTS.....	2-1
2.3 TEST MODIFICATIONS.....	2-3
CHAPTER 3.....	3-1
PROCESSING INFORMATION.....	3-1
3.1 TESTING ENVIRONMENT.....	3-1
3.2 SUMMARY OF TEST RESULTS.....	3-1
3.3 TEST EXECUTION.....	3-2
APPENDIX A.....	A-1
MACRO PARAMETERS.....	A-1
APPENDIX B.....	B-1
COMPILATION SYSTEM OPTIONS.....	B-1
LINKER OPTIONS.....	B-2
APPENDIX C.....	C-1
APPENDIX F OF THE Ada STANDARD.....	C-1

CHAPTER 1

INTRODUCTION

The Ada implementation described above was tested according to the Ada Validation Procedures [Pro92] against the Ada Standard [Ada83] using the current Ada Compiler Validation Capability (ACVC). This Validation Summary Report (VSR) gives an account of the testing of this Ada implementation. For any technical terms used in this report, the reader is referred to [Pro92]. A detailed description of the ACVC may be found in the current ACVC User's Guide [UG89].

1.1 USE OF THIS VALIDATION SUMMARY REPORT

Consistent with the national laws of the originating country, the Ada Certification Body may make full and free public disclosure of this report. In the United States, this is provided in accordance with the "Freedom of Information Act" (5 U.S.C. #552). The results of this validation apply only to the computers, operating systems, and compiler versions identified in this report.

The organizations represented on the signature page of this report do not represent or warrant that all statements set forth in this report are accurate and complete, or that the subject implementation has no nonconformities to the Ada Standard other than those presented. Copies of this report are available to the public from the AVF which performed this validation or from:

National Technical Information Service
5285 Port Royal Road
Springfield, Virginia 22161
U.S.A.

Questions regarding this report or the validation test results should be directed to the AVF which performed this validation or to:

Ada Validation Organization
Computer and Software Engineering Division
Institute for Defense Analyses
1801 North Beauregard Street
Alexandria, Virginia 22311-1772
U.S.A.

1.2 REFERENCES

- [Ada83] Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815A, February 1983 and ISO 8652-1987.
- [Pro92] Ada Compiler Validation Procedures, Version 3.1, Ada Joint Program Office, August 1992.
- [UG89] Ada Compiler Validation Capability User's Guide, 21 June 1989.

1.3 ACVC TEST CLASSES

Compliance of Ada implementations is tested by means of the ACVC. The ACVC contains a collection of test programs structured into six test classes: A, B, C, D, E, and L. The first letter of a test name identifies the class to which it belongs. Class A, C, D, and E tests are executable. Class B and class L tests are expected to produce errors at compile time and link time, respectively.

The executable tests are written in a self-checking manner and produce a PASSED, FAILED, or NOT APPLICABLE message indicating the result when they are executed. Three Ada library units, the packages REPORT and SPPRT13, and the procedure CHECK_FILE are used for this purpose. The package REPORT also provides a set of identity functions used to defeat some compiler optimizations allowed by the Ada Standard that would circumvent a test objective. The package SPPRT13 is used by many tests for Chapter 13 of the Ada Standard. The procedure CHECK_FILE is used to check the contents of text files written by some of the Class C tests for Chapter 14 of the Ada Standard. The operation of REPORT and CHECK_FILE is checked by a set of executable tests. If these units are not operating correctly, validation testing is discontinued.

Class B tests check that a compiler detects illegal language usage. Class B tests are not executable. Each test in this class is compiled and the resulting compilation listing is examined to verify that all violations of the Ada Standard are detected. Some of the class B tests contain legal Ada code which must not be flagged illegal by the compiler. This behavior is also verified.

Class L tests check that an Ada implementation correctly detects violation of the Ada Standard involving multiple, separately compiled units. Errors are expected at link time, and execution is attempted.

In some tests of the ACVC, certain macro strings have to be replaced by implementation-specific values--for example, the

largest integer. A list of the values used for this implementation is provided in Appendix A. In addition to these anticipated test modifications, additional changes may be required to remove unforeseen conflicts between the tests and implementation-dependent characteristics. The modifications required for this implementation are described in section 2.3.

For each Ada implementation, a customized test suite is produced by the AVF. This customization consists of making the modifications described in the preceding paragraph, removing withdrawn tests (see section 2.1) and, possibly some inapplicable tests (see Section 3.2 and [UG89]).

In order to pass an ACVC an Ada implementation must process each test of the customized test suite according to the Ada Standard.

1.4 DEFINITION OF TERMS

Ada Compiler	The software and any needed hardware that have to be added to a given host and target computer system to allow transformation of Ada programs into executable form and execution thereof.
Ada Compiler Validation Capability (ACVC)	The means for testing compliance of Ada implementations, Validation consisting of the test suite, the support programs, the ACVC Capability User's Guide and the template for the validation summary (ACVC) report.
Ada Implementation	An Ada compiler with its host computer system and its target computer system.
Ada Joint Program Office (AJPO)	The part of the certification body which provides policy and guidance for the Ada certification Office system.
Ada Validation Facility (AVF)	The part of the certification body which carries out the procedures required to establish the compliance of an Ada implementation.
Ada Validation Organization (AVO)	The part of the certification body that provides technical guidance for operations of the Ada certification system.
Compliance of an Ada Implementation	The ability of the implementation to pass an ACVC version.

Computer System	A functional unit, consisting of one or more computers and associated software, that uses common storage for all or part of a program and also for all or part of the data necessary for the execution of the program; executes user-written or user-designated programs; performs user-designated data manipulation, including arithmetic operations and logic operations; and that can execute programs that modify themselves during execution. A computer system may be a stand-alone unit or may consist of several inter-connected units.
Conformity	Fulfillment by a product, process, or service of all requirements specified.
Customer	An individual or corporate entity who enters into an agreement with an AVF which specifies the terms and conditions for AVF services (of any kind) to be performed.
Declaration of Conformance	A formal statement from a customer assuring that conformity is realized or attainable on the Ada implementation for which validation status is realized.
Host Computer System	A computer system where Ada source programs are transformed into executable form.
Inapplicable Test	A test that contains one or more test objectives found to be irrelevant for the given Ada implementation.
ISO	International Organization for Standardization.
LRM	The Ada standard, or Language Reference Manual, published as ANSI/MIL-STD-1815A -1983 and ISO 8652-1987. Citations from the LRM take the form "<section>.<subsection>:<paragraph>."
Operating System	Software that controls the execution of programs and that provides services such as resource allocation, scheduling, input/output control, and data management. Usually, operating systems are predominantly software, but partial or complete hardware implementations are possible.
Target Computer System	A computer system where the executable form of Ada programs are executed.

**Validated Ada
Compiler**

The compiler of a validated Ada implementation.

**Validated Ada
Implementation**

An Ada implementation that has been validated successfully either by AVF testing or by registration [Pro92].

Validation

The process of checking the conformity of an Ada compiler to the Ada programming language and of issuing a certificate for this implementation.

Withdrawn Test

A test found to be incorrect and not used in conformity testing. A test may be incorrect because it has an invalid test objective, fails to meet its test objective, or contains erroneous or illegal use of the Ada programming language.

CHAPTER 2

IMPLEMENTATION DEPENDENCIES

2.1 WITHDRAWN TESTS

Some tests are withdrawn by the AVO from the ACVC because they do not conform to the Ada Standard. The following 104 tests had been withdrawn by the Ada Validation Organization (AVO) at the time of validation testing. The rationale for withdrawing each test is available from either the AVO or the AVF. The publication date for this list of withdrawn tests is 93-11-22.

B27005A	E28005C	B28006C	C32203A	C34006D	C35507K
C35507L	C35507N	C35507O	C35507P	C35508I	C35508J
C35508M	C35508N	C35702A	C35702B	C37310A	B41308B
C43004A	C45114A	C45346A	C45612A	C45612B	C45612C
C45651A	C46022A	B49008A	B49008B	A54B02A	C55B06A
A74006A	C74308A	B83022B	B83022H	B83025B	B83025D
B83026B	C83026A	C83041A	B85001L	C86001F	C94021A
C97116A	C98003B	BA2011A	CB7001A	CB7001B	CB7004A
CC1223A	BC1226A	CC1226B	BC3009B	BD1B02B	BD1B06A
AD1B08A	BD2A02A	CD2A21E	CD2A23E	CD2A32A	CD2A41A
CD2A41E	CD2A87A	CD2B15C	BD3006A	BD4008A	CD4022A
CD4022D	CD4024B	CD4024C	CD4024D	CD4031A	CD4051D
CD5111A	CD7004C	ED7005D	CD7005E	AD7006A	CD7006E
AD7201A	AD7201E	CD7204B	AD7206A	BD8002A	BD8004C
CD9005A	CD9005B	CDA201E	CE2107I	CE2117A	CE2117B
CE2119B	CE2205B	CE2405A	CE3111C	CE3116A	CE3118A
CE3411B	CE3412B	CE3607B	CE3607C	CE3607D	CE3812A
CE3814A	CE3902B				

2.2 INAPPLICABLE TESTS

A test is inapplicable if it contains test objectives which are irrelevant for a given Ada implementation. The inapplicability criteria for some tests are explained in documents issued by ISO and the AJPO known as Ada Commentaries and commonly referenced in the format AI-ddddd. For this implementation, the following tests were determined to be inapplicable for the reasons indicated; references to Ada Commentaries are included as appropriate.

The following 201 tests have floating-point type declarations requiring more digits than SYSTEM.MAX_DIGITS:

C24113L..Y (14 tests)

C35705L..Y (14 tests)

C35706L..Y (14 tests)	C35707L..Y (14 tests)
C35708L..Y (14 tests)	C35802L..Z (15 tests)
C45241L..Y (14 tests)	C45321L..Y (14 tests)
C45421L..Y (14 tests)	C45521L..Z (15 tests)
C45524L..Z (15 tests)	C45621L..Z (15 tests)
C45641L..Y (14 tests)	C46012L..Z (15 tests)

C24113I..K (3 tests) use a line length in the input file which exceeds 126 characters.

The following 20 tests check for the predefined type `LONG_INTEGER`; for this implementation, there is no such type:

C35404C	C45231C	C45304C	C45411C	C45412C
C45502C	C45503C	C45504C	C45504F	C45611C
C45613C	C45614C	C45631C	C45632C	B52004D
C55B07A	B55B09C	B86001W	C86006C	CD7101F

C35404D, C45231D, B86001X, C86006E, and CD7101G check for a predefined integer type with a name other than `INTEGER`, `LONG_INTEGER`, or `SHORT_INTEGER`; for this implementation, there is no such type.

C35713B, C45423B, B86001T, and C86006H check for the predefined type `SHORT_FLOAT`; for this implementation, there is no such type.

C35713D and B86001Z check for a predefined floating-point type with a name other than `FLOAT`, `LONG_FLOAT`, or `SHORT_FLOAT`; for this implementation, there is no such type.

C45531M..P and C45532M..P (8 tests) check fixed-point operations for types that require a `SYSTEM.MAX_MANTISSA` of 47 or greater; for this implementation, `MAX_MANTISSA` is less than 47.

C45624A..B (2 tests) check that the proper exception is raised if `MACHINE_OVERFLOW` is `FALSE` for floating point types and the results of various floating-point operations lie outside the range of the base type; for this implementation, `MACHINE_OVERFLOW` is `TRUE`.

C4A013B contains a static universal real expression that exceeds the range of this implementation's largest floating-point type; this expression is rejected by the compiler.

B86001Y uses the name of a predefined fixed-point type other than type `DURATION`; for this implementation, there is no such type.

C96005B uses values of type `DURATION`'s base type that are outside the range of type `DURATION`; for this implementation, the ranges are the same.

CA2009C and CA2009F check whether a generic unit can be

instantiated before its body (and any of its subunits) is compiled; this implementation creates a dependence on generic units as allowed by AI-00408 and AI-00506 such that the compilation of the generic unit bodies makes the instantiating units obsolete. (See section 2.3.)

CD1009C checks whether a length clause can specify a non-default size for a floating-point type; this implementation does not support such sizes.

CD2A84A, CD2A84E, CD2A84I..J (2 tests), and CD2A84O use length clauses to specify non-default sizes for access types; this implementation does not support such sizes.

The 18 tests listed in the following table check that USE_ERROR is raised if the given file operations are not supported for the given combination of mode and access method; this implementation supports these operations.

Test	File Operation	Mode	File Access Method
CE2102E	CREATE	OUT_FILE	SEQUENTIAL_IO
CE2102F	CREATE	INOUT_FILE	DIRECT_IO
CE2102J	CREATE	OUT_FILE	DIRECT_IO
CE2102N	OPEN	IN_FILE	SEQUENTIAL_IO
CE2102O	RESET	IN_FILE	SEQUENTIAL_IO
CE2102P	OPEN	OUT_FILE	SEQUENTIAL_IO
CE2102Q	RESET	OUT_FILE	SEQUENTIAL_IO
CE2102R	OPEN	INOUT_FILE	DIRECT_IO
CE2102S	RESET	INOUT_FILE	DIRECT_IO
CE2102T	OPEN	IN_FILE	DIRECT_IO
CE2102U	RESET	IN_FILE	DIRECT_IO
CE2102V	OPEN	OUT_FILE	DIRECT_IO
CE2102W	RESET	OUT_FILE	DIRECT_IO
CE3102F	RESET	Any Mode	TEXT_IO
CE3102G	DELETE	-----	TEXT_IO
CE3102I	CREATE	OUT_FILE	TEXT_IO
CE3102J	OPEN	IN_FILE	TEXT_IO
CE3102K	OPEN	OUT_FILE	TEXT_IO

The 3 tests listed in the following table check the given file operations for the given combination of mode and access method; this implementation does not support these operations.

Test	File Operation	Mode	File Access Method
CE2105A	CREATE	IN_FILE	SEQUENTIAL_IO
CE2105B	CREATE	IN_FILE	DIRECT_IO
CE3109A	CREATE	IN_FILE	TEXT_IO

CE2203A checks that WRITE raises USE_ERROR if the capacity of an

external sequential file is exceeded; this implementation cannot restrict file capacity.

CE2403A checks that WRITE raises USE_ERROR if the capacity of an external direct file is exceeded; this implementation cannot restrict file capacity.

CE3111B and CE3115A associate multiple internal text files with the same external file and attempt to read from one file what was written to the other, which is assumed to be immediately available; this implementation buffers output. (See section 2.3.)

CE3304A checks that SET_LINE_LENGTH and SET_PAGE_LENGTH raise USE_ERROR if they specify an inappropriate value for the external file; there are no inappropriate values for this implementation.

CE3413B checks that PAGE raises LAYOUT_ERROR when the value of the page number exceeds COUNT'LAST; for this implementation, the value of COUNT'LAST is greater than 150000, making the checking of this objective impractical.

2.3 TEST MODIFICATIONS

Modifications (see section 1.3) were required for 69 tests.

The following tests were split into two or more tests because this implementation did not report the violations of the Ada Standard in the way expected by the original tests.

B22003A	B26001A	B26002A	B26005A	B28003A	B29001A	B33301B
B35101A	B37106A	B37301B	B37302A	B38003A	B38003B	B38009A
B38009B	B55A01A	B61001C	B61001F	B61001H	B61001I	B61001M
B61001R	B61001W	B67001H	B83A07A	B83A07B	B83A07C	B83E01C
B83E01D	B83E01E	B85001D	B85008D	B91001A	B91002A	B91002B
B91002C	B91002D	B91002E	B91002F	B91002G	B91002H	B91002I
B91002J	B91002K	B91002L	B95030A	B95061A	B95061F	B95061G
B95077A	B97103E	B97104G	BA1001A	BA1101B	BC1109A	BC1109C
BC1109D	BC1202A	BC1202F	BC1202G	BE2210A	BE2413A	

CD2A83A was graded passed by Test Modification as directed by the AVO. This test uses a length clause to specify the collection size for an access type whose designated type is STRING; eight designated objects are allocated, with a combined length of 30 characters. Because of this implementation's heap-management strategy and alignment requirements, the collection size at line 22 had to be increased to 812.

CA2009C and CA2009F were graded inapplicable by Evaluation Modification as directed by the AVO. These tests contain instantiations of a generic unit prior to the compilation of that unit's body; as allowed by AI-00408 and AI-00506, the compilation

of the generic unit bodies makes the compilation unit that contains the instantiations obsolete.

BC3204C and BC3205D were graded passed by Processing Modification as directed by the AVO. These tests check that instantiations of generic units with unconstrained types as generic actual parameters are illegal if the generic bodies contain uses of the types that require a constraint. However, the generic bodies are compiled after the units that contain the instantiations, and this implementation creates a dependence of the instantiating units on the generic units as allowed by AI-00408 and AI-00506 such that the compilation of the generic bodies makes the instantiating units obsolete--no errors are detected. The processing of these tests was modified by re-compiling the obsolete units; all intended errors were then detected by the compiler.

CE3111B and CE3115A were graded inapplicable by Evaluation Modification as directed by the AVO. The tests assume that output from one internal file is unbuffered and may be immediately read by another file that shares the same external file. This implementation raises END_ERROR on the attempts to read at lines 87 and 101, respectively.

CHAPTER 3

PROCESSING INFORMATION

3.1 TESTING ENVIRONMENT

The Ada implementation tested in this validation effort is described adequately by the information given in the initial pages of this report.

For technical information about this Ada implementation, contact:

Forrest Holemon
410 North 44th Street, Suite 320
Phoenix, Arizona 85008 (U.S.A.)
Telephone: 602-275-7172
Telefax: 602-275-7502

For sales information about this Ada implementation, contact:

Mike Halpin
410 North 44th Street, Suite 320
Phoenix, Arizona 85008 (U.S.A.)
Telephone: 602-275-7172
Telefax: 602-275-7502

Testing of this Ada implementation was conducted at the customer's site by a validation team from the AVF.

3.2 SUMMARY OF TEST RESULTS

An Ada Implementation passes a given ACVC version if it processes each test of the customized test suite in accordance with the Ada Programming Language Standard, whether the test is applicable or inapplicable; otherwise, the Ada Implementation fails the ACVC [Pro92].

For all processed tests (inapplicable and applicable), a result was obtained that conforms to the Ada Programming Language Standard.

The list of items below gives the number of ACVC tests in various categories. All tests were processed, except those that were withdrawn because of test errors (item b; see section 2.1), those that require a floating-point precision that exceeds the implementation's maximum precision (item e; see section 2.2), and those that depend on the support of a file system--if none is supported (item d). All tests passed, except those that are listed in sections 2.1 and 2.2 (counted in items b and f, below).

a) Total Number of Applicable Tests	3783	
b) Total Number of Withdrawn Tests	104	
c) Processed Inapplicable Tests	283	
d) Non-Processed I/O Tests	0	
e) Non-Processed Floating-Point Precision Tests	0	
f) Total Number of Inapplicable Tests	283	(c+d+e)
g) Total Number of Tests for ACVC 1.11	4170	(a+b+f)

3.3 TEST EXECUTION

A magnetic tape containing the customized test suite (see section 1.3) was taken on-site by the validation team for processing. The contents of the magnetic tape were loaded directly onto the host computer.

After the test files were loaded onto the host computer, the full set of tests was processed by the Ada implementation.

The tests were compiled, linked, and executed on the host/target computer system. The results were captured on the host/target computer system.

Testing was performed using command scripts provided by the customer and reviewed by the validation team. See Appendix B for a complete listing of the processing options for this implementation. It also indicates the default options. The options invoked explicitly for validation testing during this test were:

-list

Test output, compiler and linker listings, and job logs were captured on magnetic tape and archived at the AVF. The listings examined on-site by the validation team were also archived.

5 ADA COMPILER

The Ada Compiler compiles all program units within the specified source file and inserts the generated objects into the current program library. Compiler options are provided to allow the user control of optimization, run-time checks, and compiler input and output options such as list files, configuration files, and the program library used.

The input to the compiler consists of the source file, the configuration file (which controls the format of the list file), and the compiler options. Section 5.1 provides a list of all compiler options, and Section 5.2 describes the source and configuration files.

Output consists of an object placed in the program library, diagnostic messages, and optional listings. The configuration file and the compiler options specify the format and contents of the list information. Output is described in Section 5.3.

If any diagnostic messages are produced during the compilation, they are output in the diagnostic file and on the current output file. The diagnostic file and the diagnostic message format are described in Section 5.3.2.

The compiler uses a program library during the compilation. The compilation unit may refer to units from the program library, and an internal representation of the compilation unit will be included in the program library as a result of a successful compilation. The program library is described in Chapter 3. Section 5.4 briefly describes how the Ada compiler uses the library.

5.1 Invocation Command

Invoke the Ada compiler with the following command:

```
ada {<option>} <source-or-unit-name> {<source-or-unit-name>}
```

5.1.1 Summary of Options

This section presents a summary of options supported by the compiler.

OPTIONS	DESCRIPTION	REFERENCE
-auto_inline	Small local subprograms are automatically inline expanded.	5.1.2
-body	Compile body unit from source saved in library.	5.1.3
-nocheck <keyword>{,<keyword>}	Suppress generation of run-time constraint checks.	5.1.4
-configuration <file-name>	Specifies the file used by the compiler.	5.1.5
-debug	Specifies that information for the DDC-I Symbolic Ada Debugger is to be generated.	5.1.6
-library <file-name>	Specifies the program library to be used.	5.1.7
-list	Creates a source list file.	5.1.8
-machine_code	Generates a machine code dump for the compilation.	5.1.9
-optimize <keyword>{,<keyword>}	Specifies compiler optimizations.	5.1.10
-profile	Specifies that code for profiling is to be generated.	5.1.11
-progress	Displays compiler progress.	5.1.12
-nosave_source	The source is not saved in the program library.	5.1.13
-specification	Compile specification unit from source saved in library.	5.1.14
-warnings	Suppress warnings from the compiler.	5.1.15
-xref	Creates a cross reference listing.	5.1.16

Example:

```
$ ada -list testprog
```

This example compiles the source file testprog and generates a list file with the name testprog.lis.

Example:

```
$ ada -lib my_library test
```

This example compiles the source file test into the library my_library.

Default values exist for options as indicated in the following sections. Option names may be abbreviated (characters omitted from the right) as long as no ambiguity arises.

5.1.2 AUTO_INLINE

Syntax:

-auto_line

This option specifies whether local subprograms should be inline expanded. The inline expansion only occurs if the subprogram contains no more than 3 object declarations (and no other declarations), no more than 5 statements and no exception handler and if the subprogram fulfills the requirements defined for pragma inline. A warning is issued when automatic inline expansion is performed.

5.1.3 BODY

Syntax:

-body

When using the option **-body** the Ada compiler will recompile the body of the unit specified as parameter to the Ada compiler (see section 5.1.17) into the current sublibrary. The source code saved in the program library at the previous compilation of the body is used as the source code to be compiled. If no source code is present or the body for the unit does not exist in the library, an error message is issued. This option is primarily for use by the Ada Recompiler (see chapter 7).

5.1.4 NOCHECK

Syntax:

-nocheck <keyword>{,<keyword>}

By default, all run-time checks listed below will be generated by the compiler. The following explicit checks can be suppressed:

ALL	Suppress all checks.
INDEX	Index check.
ACCESS	Check for access values being non NULL.
DISCRIMINANT	Checks for discriminated fields.
LENGTH	Array length check.

RANGE	Checks for values being in range.
OVERFLOW	Explicit overflow checks.
ELABORATION	Checks for subprograms being elaborated.
STORAGE	Checks for sufficient storage available.

Note that the `Division_check` suppression mentioned in ARM 11.7 is not implemented.

5.1.5 CONFIGURATION_FILE

Syntax:

-configuration <file-name>

This option specifies the configuration file to be used by the compiler. The configuration file allows the user to format compiler listings, set error limits, etc. If the option is omitted the configuration file designated by the name `config` is used by default. Section 5.2.2 contains a description of the configuration file.

5.1.6 DEBUG

Syntax:

-debug

Generate debug information for the compilation and store the information in the program library. This is necessary if the unit is to be debugged with the DDC-I Ada Symbolic Cross Debugger. Note that the program must also be linked with the `-debug` option, if the program is to be debugged with the DDC-I Ada Symbolic Cross Debugger. See Section 6.3.3.1. Please note that no extra information is included in the code or data generated.

5.1.7 LIBRARY

Syntax:

-library <library-name>

This option specifies the current sublibrary that will be used in the compilation and will receive the object when the compilation is complete. By specifying a current sublibrary, the current program library (current sublibrary and ancestors up to root) is also implicitly specified.

If this option is omitted, the sublibrary designated by the environment variable name **ADA_LIBRARY** is used as the current sublibrary. Section 5.4 describes how the Ada compiler uses the library.

5.1.8 LIST

Syntax:

-list

-list specifies that a source listing will be produced. The source listing is written on the list file, which has the name of the source file with the extension **.lis**. Section 5.3.1.1 contains a description of the source listing.

5.1.9 MACHINE_CODE

Syntax:

-machine_code

Dump a machine code list of the compiled code at standard output. The instructions are dumped symbolically, but addresses are not. Calls are described by a "patch", which consists of a unit number and an entry number. The unit number is a unique number defining the library unit and the entry number is the number of the subprogram within that unit.

5.1.10 OPTIMIZE

Syntax:

-optimize <keyword>{,<keyword>}

This option specifies which optimizations will be performed during code generation. Default is no optimizations.

Selection of optimizations can be done in two basic ways.

- 1) Selecting individual optimizations.
- 2) Selecting predefined classes of optimizations.

[NO]LOOP_REGISTERS[=<number-of-iterations>]

Controls the extent to which registers are allocated to variables in loops, particularly inner loops.

It is possible to specify an optimization level, where the level specifies the number of times the optimizer shall loop through the code. If no level is specified only one loop will be performed.

[NO]COMMON_SUBEXPRESSION_ELIMINATION[=<number-of-iterations>]

Specify to which extent common subexpression elimination should be performed.

It is possible to specify an optimization level, where the level specifies the number of times the optimizer shall loop through the code. If no level is specified only one loop will be performed.

[NO]COPY_PROPAGATION[=<number-of-iterations>]

Specify to which extent copy propagation should be performed.

It is possible to specify an optimization level, where the level specifies the number of times the optimizer shall loop through the code. If no level is specified only one loop will be performed.

[NO]CONSTANT_FOLDING

Controls whether arithmetic expressions which have become static due to other optimizations are calculated at compile time and folded into the code.

[NO]LOOP_UNROLLING

Performs unrolling of static loops into sequential code. The algorithm for deciding whether a loop is a candidate for unrolling is given in the Reference Manual.

[NO]LOOP_INVARIANT_CODE_MOTION

Controls the movement of invariant code outside of loops.

[NO]DEAD_CODE_REMOVAL[=<number-of-iterations>]

Controls whether dead code should be removed or not. Dead code can occur when conditions become static or when a variable is not used anymore. Please note that this optimization can be a very time consuming.

It is possible to specify an optimization level, where the level specifies the number of times the optimizer shall loop through the code. If no level is specified only one loop will be performed.

The following options select a predefined level of optimizations:

LOW - Selects a predefined set of optimizations equal to the following list:

LOOP_REGISTERS, COMMON_SUBEXPR,
COPY_PROPAGATION, CONSTANT_FOLDING,
NODEAD_CODE_REMOVAL, LOOP_UNROLLING,
LOOP_INVARIANT_CODE_MOTION

DACS Sun SPARC/SunOS and Sun SPARC/Solaris Native Ada Compiler System, User's Guide Ada Compiler

MEDIUM - Selects a predefined set of optimizations equal to the following list:

LOOP_REGISTERS=25, COMMON_SUBEXPR=25,
COPY_PROPAGATION=25, CONSTANT_FOLDING,
DEAD_CODE_REMOVAL, LOOP_UNROLLING,
LOOP_INVARIANT_CODE_MOTION

HIGH - Selects a predefined set of optimizations equal to the following list:

LOOP_REGISTERS=1000, COMMON_SUBEXPR=1000,
COPY_PROPAGATION=1000, CONSTANT_FOLDING,
DEAD_CODE_REMOVAL=1000, LOOP_UNROLLING,
LOOP_INVARIANT_CODE_MOTION

ALL - Equivalent to HIGH

Example:

```
$ ada -optimize all example_1
```

Both of these commands compile the program with all the optimizations active at their highest levels.

```
$ ada -opt low,loop_reg=1000,noloop_unroll) example_2
```

This command compile the program with low optimizations, but no loop_unrolling is wanted and registers should be used to the greatest extent possible in loops.

5.1.11 PROFILE

Syntax:

-profile

This option specifies that code for profiling shall be generated. This option in conjunction with the linker option profile enables profiling of an executable program.

5.1.12 PROGRESS

Syntax:

-progress

When this option is given, the compiler will output data about which pass the compiler is currently running.

5.1.13 NOSAVE_SOURCE

Syntax:

-nosave_source

When **-nosave_source** is specified, source code will not be retained in the program library, this save some space in the sublibrary. The default is to save a copy of the compiled source code in the program library. Hereby the user is always certain of what version of the source code compiled. The source code may be displayed from the sublibrary with the PLU Type command. Using **-nosave_source** will help keeping library sizes smaller, but it will not be possible to use the recompiler or the symbolic Ada Debugger.

5.1.14 SPECIFICATION

Syntax:

-specification

When using the option **-specification** the Ada compiler will recompile the specification of the unit specified as parameter to the Ada compiler (see section 5.1.17) into the current sublibrary. The source code saved in the program library at the previous compilation of the specification is used as the source code to be compiled. If no source code is present or the specification for the unit does not exist in the library, an error message is issued. This option is primarily for use by the Ada Recompile (see chapter 7).

5.1.15 WARNINGS

Syntax:

-warnings

Suppress warnings from the compiler in the diagnostics file. All diagnostics will always come on standard output, only the contents of the diagnostics file is affected by the warnings option. If a compilation only generates warnings and the warnings option is specifies no diagnostics file is created.

5.1.16 XREF

Syntax:

-xref

A cross-reference listing can be requested by the user by means of the option **-xref** in conjunction with option list. If the **-xref** option is given and no severe or fatal errors are found during the compilation, the cross-reference listing is written to the list file. The cross-reference listing is described in Section 5.3.1.3.

5.1.17 Source File Parameter

<source-or-unit-name> {<source-or-unit-name>}

This parameter specifies either the text file containing the Ada source text to be compiled or, when option **-body -specification** is used, the name of the unit to be compiled. When interpreted as a file name, the file type **.ada** is assumed by default. More than one file name can be specified, each **<source-file-name>** can be a file name with wildcards as defined by the shell.

The compilation starts with the leftmost file name in the file name list, and ends with the rightmost. If the list of file names includes a file name with wildcards, the files matching the wildcard name are compiled in alphabetical order. If any file name occurs several times in the list of file names, the file is compiled several times, i.e. one file is compiled as many times as its name occurs in the list of file names.

The allowed format of the source text is described in Section 5.2.1.

5.2 Compiler Input

Input to the compiler consists of the command line options, a list of source text files and, optionally, a configuration file.

5.2.1 Source Text

The user submits one file containing a source text in each compilation. The source text may consist of one or more compilation units (see ARM Section 10.1).

The format of the source text must be in ISO-FORMAT ASCII. This format requires that the source text is a sequence of ISO characters (ISO standard 646), where each line is terminated by either one of the following termination sequences (CR means carriage return, VT means vertical tabulation, LF means line feed, and FF means form feed):

- 1) A sequence of one or more CRs, where the sequence is neither immediately preceded nor immediately followed by any of the characters VT, LF, or FF.
- 2) Any of the characters VT, LF, or FF, immediately preceded and followed by a sequence of zero or more CRs.

In general, ISO control characters are not permitted in the source text with the following exceptions:

- 1) The horizontal tabulation (HT) character may be used as a separator between lexical units.
- 2) LF, VT, FF, and CR may be used to terminate lines, as described above.

The maximum number of characters in an input line is determined by the contents of the configuration file (see Section 5.1.4). The control characters CR, VT, LF, and FF are not considered a part of the line. Lines containing more than the maximum number of characters are truncated and an error message is issued.

5.2.2 Configuration File

Certain processing characteristics of the compiler, such as format of input and output, and error limit, may be modified by the user. These characteristics are passed to the compiler by means of a configuration file, which is a standard UNIX text file. The contents of the configuration file must be an Ada positional aggregate, written on one line, of the type `CONFIGURATION_RECORD`, which is described below.

The configuration file "config" is not accepted by the compiler in the following cases:

- 1) The syntax does not conform with the syntax for positional Ada aggregates.
- 2) A value is outside the ranges specified.
- 3) A value is not specified as a literal.
- 4) `LINES_PER_PAGE` is not greater than `TOP_MARGIN + BOTTOM_MARGIN`.
- 5) The aggregate occupies more than one line.

If the compiler is unable to accept the configuration file, an error message is written on the current output file and the compilation is terminated.

This is the record whose values must appear in aggregate form within the configuration file. The record declaration makes use of some other types (given below) for the sake of clarity.

```
type CONFIGURATION_RECORD is
  record
    IN_FORMAT      : INFORMATTING;
    OUT_FORMAT     : OUTFORMATTING;
    ERROR_LIMIT    : INTEGER;
  end record;

type INPUT_FORMATS is (ASCII);

type INFORMATTING is
  record
    INPUT_FORMAT      : INPUT_FORMATS;
    INPUT_LINELENGTH : INTEGER range 70..250;
  end record;

type OUTFORMATTING is
  record
    LINES_PER_PAGE : INTEGER range 30..100;
    TOP_MARGIN     : INTEGER range 4.. 90;
    BOTTOM_MARGIN  : INTEGER range 0.. 90;
```

DACS Sun SPARC/SunOS and Sun SPARC/Solaris Native Ada Compiler System, User's Guide
Ada Linker

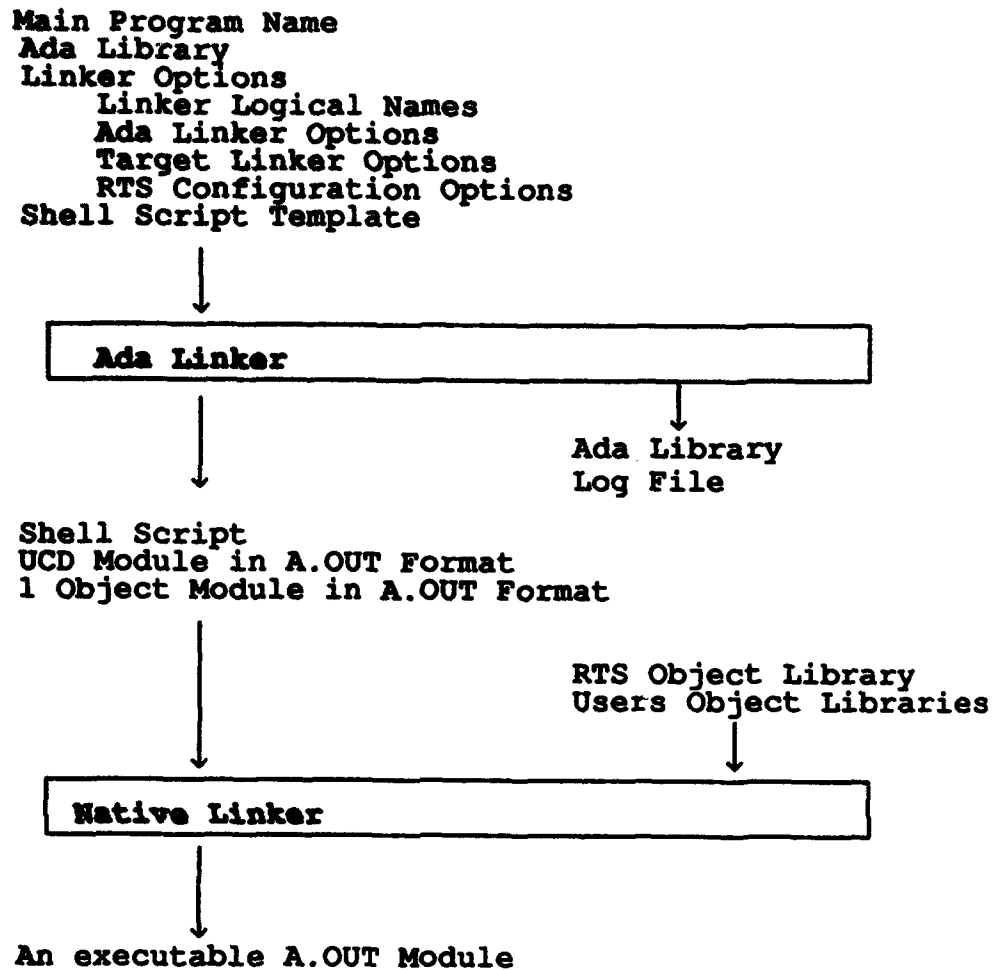


Figure 6-1
The Linking Process

6.3 Invocation Command

Enter the following command to invoke the linker:

al {<option>} <unit-name>

The options and parameters supported by the linker are described in the following sections.

6.3.1 Parameter

<unit-name>

This parameter is required and indicates the main program. The <unit_name> must be a library unit in the current program library, but not necessarily of the current sublibrary.

Note, that a main program must be a procedure without parameters, and that <unit-name> is the identifier of the procedure, not a file specification. The main procedure is not checked for parameters, but the execution of a program with a main procedure with parameters is undefined.

6.3.2 Summary of Options

This section briefly describes all options supported by the Ada linker.

OPTIONS	DESCRIPTION	REFERENCE
-debug	Specify that the executable file is to be used by the DDC-I Symbolic Ada Debugger.	6.3.3.1
-noexceptions	No spellings of user exceptions will be included in the executable file.	6.3.3.2
-executable <file-name>	Specify the name of the executable file.	6.3.3.3
-keep	Perform Ada link only, and keeps object files.	6.3.3.4
-library <library-name>	Specify the library to be used in the link.	6.3.3.5
-log <file-name>	Specify creation of a log file.	6.3.3.6
-main_stack_size <natural>	Default stack size for main program.	6.3.3.7
-period <duration>	Timer resolution.	6.3.3.8
-priority <positive>	Default task priority.	6.3.3.9
-profile	Enable profiling of the executable program.	6.3.3.10
-selective	Enables selective linking.	6.3.3.11
-statistics	Display statistics.	6.3.3.12
-target_options <string>	Specify a string which is passed to the template without interpretation.	6.3.3.13

DACS Sun SPARC/SunOS and Sun SPARC/Solaris Native Ada Compiler System, User's Guide Ada Linker

-task_stack_size <natural>	Default stack size for all tasks.	6.3.3.14
-tasks <natural>	Maximum number of tasks.	6.3.3.15
-template_file <template-name>	Specify template file for the target link.	6.3.3.16
-notimer	Disable timer setup in the executable program.	6.3.3.17
-time_slice <duration>	Task time slicing enabled and time slice value.	6.3.3.18
-traceback_mode <keyword>	Enable traceback when a program has an unhandled exception.	6.3.3.19
-usr_library <file-name>	Libraries or object modules to include in link.	6.3.3.20
-warnings	Specify that warnings are displayed.	6.3.3.21

All options must be entered in lowercase, and may be abbreviated to the minimal unique substring (e.g. "-d" is sufficient to denote "-debug").

6.3.3 Ada Link Options

This section describes in detail all Ada link options, including default settings.

6.3.3.1 DEBUG

Syntax:

-debug

This option specifies that information for the DDC-I Symbolic Ada Debugger is to be generated. Please note that no extra information is included in the code or data generated.

6.3.3.2 NOEXCEPTION TABLES

Syntax:

-noexceptions

This option specifies that no table containing the spellings of user-defined exceptions should be included in the executable file. Without spellings for user-defined exceptions a stack trace for an unhandled user-defined exception will appear with a reference to the unit number in which the exception is defined and an exception number within the unit.

6.3.3.3 EXECUTABLE FILENAME

Syntax:

-executable <file-name>

Specifies the name of the output module. Default is the name of the main program.

Examples:

```
$ a1 p
```

- Links the subprogram P and stores the executable program in the file p

```
$ a1 -exec my_exe_dir/main p
```

- Links the subprogram P and stores the executable program in the file main in the directory my_exe_dir.

6.3.3.4 KEEP

Syntax:

-keep

This option controls whether or not the native link phase is executed, i.e., whether or not control is passed to the flexible linker template file, and the intermediate link files are preserved. The option specifies to stop linking before the native link and keep the intermediate link files.

6.3.3.5 LIBRARY

Syntax:

-library <library-name>

This option defines the program library that contains the <unit-name>. If <library-name> is not specified, the default library specified by the environment variable ADA_LIBRARY will be selected.

6.3.3.6 LOG

Syntax:

-log <file-name>

The option specifies if a log file will be produced from the front end linker. As default, no log file is produced. The log file contains extensive information on the results of the link. The file includes:

- 1) An elaboration order list with an entry for each unit included, showing the order in which the units will be elaborated.
- 2) All options and their actual values.
- 3) The full name of the program library (the current sublibrary and its ancestor sublibraries).

6.3.3.7 MAIN STACK SIZE

Syntax:

-main_stack_size <natural> (Default is 4096)

The main stack size option specifies the main program stack size N in 32 bit words. The range of this parameter is limited by physical memory size. The range of main stack size is from 0 to 2,147,483,647

Configurable Data

The Ada linker generates the following data structures:

UCD\$MP_Stack_Size

N

UCD\$MP_Stack

Lowest addr
of MP stack

UCD\$MP_Stack_Start

Highest addr
of MP stack

Example:

```
$ al -main 1024 p
```

- Link the program p with a stack of 4096 bytes.

6.3.3.8 PERIOD

Syntax:

-period <decimal-number> (Default is 0.05)

The period option specifies the resolution of calls to the Run-Time System routine RTS\$TIMER. The number specifies the number of seconds between two successive calls to RTS\$TIMER. The number must be within the range duration'small to 2.0

Configurable Data

The Ada Linker generates the following 32 bit integer:

UCD\$Timer

absolute integer

6.3.3.9 PRIORITY

Syntax:

-priority <integer> (Default is 16)

The priority option specifies the default priority N for task execution. The main program will run at this priority, as well as tasks which have had no priority level defined via pragma PRIORITY. The range of priorities is from 1 to 32.

Configurable Data

The Ada Linker generates the following constant data:

UCD\$Priority

N

Example:

\$ al -priority 8 p

- Link the subprogram P which has the main program and tasks running at default priority 8.

6.3.3.10 PROFILE

Syntax:

-profile

This option specifies that the executable program shall allocate memory for profiling information and be linked with a profiling library. The option is used together with the DDC-I Ada Profiler. If problems with interrupted system calls then link with option **-notimer** (see Section 6.3.3.17). Furthermore, the profiler timer has a higher resolution than the Ada timer, which makes better profile information. If the program use task do not link with option **-notimer**.

6.3.3.11 SELECTIVE LINKING

Syntax:

-selective

Specifies that only those subprograms from each compilation which are referred to from other subprograms i.e. only those subprograms which actually are in the program are linked within the program, and thereby minimizing the size of the executable program.

6.3.3.12 STATISTICS

Syntax:

-statistics

Specifies that short statistics shall be displayed about how many compilation units included in the program and how many dependencies they have.

6.3.3.13 TARGET OPTIONS

Syntax:

-target_options <string>

This option allows the user to specify target options which is passed to the native linker (ld) without interpretation. It therefore allows the user to specify other options than those mentioned in this section.

Example:

```
$ al -target "-L/usr/local/lib" p
```

- Links the subprogram P and substitutes the macro template %target_options% by the string "-L/usr/local/lib".

This option allows any string to be propagated to the resulting command file via the macro template %TARGET_OPTIONS%.

6.3.3.14 TASK STACK SIZE

Syntax:

-task_stack_size <natural> (Default is 1024)

This option sets the default storage size N in 32 bit words for stacks of all tasks. This value can be overridden with a representation clause.

Configurable Data

The Ada Linker generates the following data structure:

UCD\$Task_Stack_Size

N

6.3.3.15 TASKS

Syntax:

-tasks <natural> (default is 128)

This option specifies the maximum number of tasks allowed by the RTS. If specified, N must be greater than or equal to zero.

Configurable Data

For the `tasks` option, the linker generates the following configurable data:

UCD\$Max_Tasks

N

UCD\$TCBs

N Task
Control
Blocks
(TCBs)

Example:

```
$ al -tasks 3 p
```

- Link the program `p`, which has at most 3 tasks, including the main program.

6.3.3.16 TEMPLATE

Syntax:

`-template_file <file-name>`

This option specifies a template file to use for the native link. The default is to use the file named `Ada_template.txt` placed in the same directory as the Ada linker. See section 6.5 for an explanation of the template file and the flexible linker.

6.3.3.17 NOTIMER

Syntax:

`-notimer`

This option disables timer setup in the executable program. Specifies that the Ada timer is not set up. This causes that delays waits forever, and that the option `-time_slice` will not function. This option is useful when debugging programs using `dbx`. The option is also useful when using the `-profile` option, because the profile timer has a higher resolution than the Ada timer, which gives a more detailed profiling information. The option can also be used to prevent interrupts at I/O operations with blocking and other kernel calls.

6.3.3.18 TIME SLICE

Syntax:

`-time_slice <decimal-number>`

The time slice option specifies the time slicing period for tasks.

If specified, it is a decimal number of seconds representing the default time slice to be used. If not specified, there will be no time slicing. The number must be in the range Duration'Small..2.0 and must be greater than or equal to the distance between two successive calls to RTS\$TIMER.

Time slicing only applies to tasks running at equal priority. Because the RTS is a preemptive priority scheduler, the highest priority task will always run before any lower priority task. Only when two or more tasks are running at the same priority is time slicing applied to each task.

Time slicing is not applicable unless tasking is being used. This means that the tasks option must be set to at least 2 for time slice to be effective.

Configurable Data

The Ada Linker generates the following data:

UCD\$Time_Slice

absolute integer

= 0 -> No time slicing

/= 0 -> The length of a time slice

The number of timer ticks (USD\$Time_Slice) constituting a time_slice is computed as $\lceil \text{TimeSlice} / \text{Period} \rceil$

Example:

```
$ al -time 0.125 p
```

- Specifies tasks of equal priority to be time sliced each eighth of a second.

6.3.3.19 TRACEBACK MODE

Syntax:

-traceback_mode (never | main | always) (default is main)

This option instructs the exception handler to produce a stack trace when a program terminates because of an unhandled exception. Disabling traceback (with never) exclude traceback tables from the executable program. If never is specified, the RTS variable UCD\$TRACE will be set to 0 and no trace will be produced if the program terminates with an unhandled exception. If main is specified a trace will be produced if the main program terminates with an unhandled exception. The RTS variable UCD\$TRACE will be set to 1. If the always is specified the RTS variable UCD\$TRACE will be set to 2 and a trace will be produced if either the main program or a task terminate with an unhandled exception.

Configurable Data

UCD\$Trace

absolute integer

= 0 -> Trace disabled

= 1 -> Trace enabled for main program

= 2 -> Trace enabled for main program and tasks

6.3.3.20 USER LIBRARY

Syntax:

-usr_library <file-name>

The user library option is intended for the specification of libraries or object files which should contain the users own object code.

The user library option is also intended to specify libraries of routines referenced from the Ada program via pragma INTERFACE.

6.3.3.21 WARNINGS

Syntax:

-warnings

This option specifies that warnings are displayed if detected by the linker, otherwise they are suppressed. Warnings can be generated when conflicts between target program attributes and the specified options are found and when a package has an inconsistent body.

6.4 Linker Output

This chapter describes the results of the linking process.

6.4.1 Executable File

Using the default options and the template provided with the system the linking process will result in an executable file which is ready for execution. This file is named after the main program:

<main_program_name>

6.4.2 Diagnostic Messages

Diagnostic messages from the Ada Linker are output to the current output file. The messages are output in the order they are generated by the linker.

The linker may issue two kinds of diagnostic messages: warnings and severe errors.

6.4.2.1 Warnings

A warning reports something which does not prevent a successful linking, but which might be an error. A warning is issued if there is something wrong with the body unit of a program unit which formally does not need a body unit, e.g. if the body unit is invalid or if there is no object code container for the body unit.

6.4.2.2 Severe Errors

A severe error message reports an error which prevents a successful linking. Any inconsistency detected by the linker will, for instance, cause a severe error message, e.g. if some required unit does not exist in the library or if some time stamps do not agree.

A unit not marked as invalid in the program library may be reported as being invalid by the linker if there is something wrong with the unit itself or with some of the units it depends on.

6.4.2.3 Return Codes

The linker set the return code to the following values:

Error code	Description
0	Success, warnings
1	Errors
2	Fatal Error

6.5 Flexible Linker

The DACS Ada Linker is referred to as a flexible linker because it has been designed to be very flexible in the way it interfaces to target tools such as assemblers, librarians, and linkers. The flexible linker can produce a target link in any manner that the user desires, i.e, the linker adapts to each user's needs in a simple and straightforward manner.

APPENDIX A

MACRO PARAMETERS

This appendix contains the macro parameters used for customizing the ACVC. The meaning and purpose of these parameters are explained in [UG89]. The parameter values are presented in two tables. The first table lists the values that are defined in terms of the maximum input-line length, which is the value for \$MAX_IN_LEN--also listed here. These values are expressed here as Ada string aggregates, where "V" represents the maximum input-line length.

Macro Parameter	Macro Value
\$MAX_IN_LEN	126 -- Value of V
\$BIG_ID1	(1..V-1 => 'A', V => '1')
\$BIG_ID2	(1..V-1 => 'A', V => '2')
\$BIG_ID3	(1..V/2 => 'A') & '3' & (1..V-1-V/2 => 'A')
\$BIG_ID4	(1..V/2 => 'A') & '4' & (1..V-1-V/2 => 'A')
\$BIG_INT_LIT	(1..V-3 => '0') & "298"
\$BIG_REAL_LIT	(1..V-5 => '0') & "690.0"
\$BIG_STRING1	"" & (1..V/2 => 'A') & ""
\$BIG_STRING2	"" & (1..V-1-V/2 => 'A') & '1' & ""
\$BLANKS	(1..V-20 => ' ')
\$MAX_LEN_INT_BASED_LITERAL	"2:" & (1..V-5 => '0') & "11:"
\$MAX_LEN_REAL_BASED_LITERAL	"16:" & (1..V-7 => '0') & "F.E:"
\$MAX_STRING_LITERAL	"" & (1..V-2 => 'A') & ""

The following table contains the values for the remaining macro parameters.

Macro Parameter	Macro Value
ACC_SIZE	: 32
ALIGNMENT	: 4
COUNT_LAST	: 2_147_483_647
DEFAULT_MEM_SIZE	: 2048*1024
DEFAULT_STOR_UNIT	: 8
DEFAULT_SYS_NAME	: DACS_SPARC
DELTA_DOC	: 2#1.0#E-31
ENTRY_ADDRESS	: SYSTEM."-(16#10709180#)
ENTRY_ADDRESS1	: SYSTEM."-(16#10709188#)
ENTRY_ADDRESS2	: SYSTEM."-(16#10709190#)
FIELD_LAST	: 35
FILE_TERMINATOR	: ' '
FIXED_NAME	: NO_SUCH_FIXED_TYPE
FLOAT_NAME	: NO_SUCH_FIXED_TYPE
FORM_STRING	: ""
FORM_STRING2	: "CANNOT RESTRICT_FILE_CAPACITY"
GREATER_THAN_DURATION	: 100000.0
GREATER_THAN_DURATION_BASE_LAST	: 200000.0
GREATER_THAN_FLOAT_BASE_LAST	: 16#1.0#E+32
GREATER_THAN_FLOAT_SAFE_LARGE	: 16#5.FFFF_F0#E+31
GREATER_THAN_SHORT_FLOAT_SAFE_LARGE	: 16#5.FFFF_F0#E+31
HIGH_PRIORITY	: 31
ILLEGAL_EXTERNAL_FILE_NAME1	: /NODIRECTORY/FILENAME1
ILLEGAL_EXTERNAL_FILE_NAME2	: /NODIRECTORY/FILENAME2
INAPPROPRIATE_LINE_LENGTH	: -1
INAPPROPRIATE_PAGE_LENGTH	: -1
INCLUDE_PRAGMA1	: PRAGMA INCLUDE ("A28006D1.ADA")
INCLUDE_PRAGMA2	: PRAGMA INCLUDE ("B28006E1.ADA")
INTEGER_FIRST	: -2147483648
INTEGER_LAST	: 2147483647
INTEGER_LAST_PLUS_1	: 2147483648
INTERFACE_LANGUAGE	: AS
LESS_THAN_DURATION	: -75000.0
LESS_THAN_DURATION_BASE_FIRST	: -131073.0
LINE_TERMINATOR	: ' '
LOW_PRIORITY	: 0
MACHINE_CODE_STATEMENT	: NULL;
MACHINE_CODE_TYPE	: NO_SUCH_TYPE
MANTISSA_DOC	: 31
MAX_DIGITS	: 15
MAX_INT	: 2147483647

MAX_INT_PLUS_1	: 2147483648
MIN_INT	: -2147483648
NAME	: NO_SUCH_TYPE_AVAILABLE
NAME_LIST	: DACS_SPARC
NAME_SPECIFICATION1	:
	/home/sparcla/ada/test/prevalid/work/X2120A
NAME_SPECIFICATION2	:
	/home/sparcla/ada/test/prevalid/work/X2120B
NAME_SPECIFICATION3	:
	/home/sparcla/ada/test/prevalid/work/X3119A
NEG_BASED_INT	: 16#F000000E#
NEW_MEM_SIZE	: 2097152
NEW_STOR_UNIT	: 8
NEW_SYS_NAME	: DACS_SPARC
PAGE_TERMINATOR	: ' '
RECORD_DEFINITION	: NEW_INTEGER
RECORD_NAME	: MACHINE_INSTRUCTION
TASK_SIZE	: 32
TASK_STORAGE_SIZE	: 1024
TICK	: 2#1.0#E-14
VARIABLE_ADDRESS	: SYSTEM."-(16#10709198#)
VARIABLE_ADDRESS1	: SYSTEM."-(16#107091A0#)
VARIABLE_ADDRESS2	: SYSTEM."-(16#107091A8#)
YOUR_PRAGMA	: INTERFACE_SPELLING

APPENDIX B

COMPILATION SYSTEM OPTIONS

The compiler options of this Ada implementation, as described in this Appendix, are provided by the customer. Unless specifically noted otherwise, references in this appendix are to compiler documentation and not to this report.

LINKER OPTIONS

The linker options of this Ada implementation, as described in this Appendix, are provided by the customer. Unless specifically noted otherwise, references in this appendix are to linker documentation and not to this report.

APPENDIX C

APPENDIX F OF THE Ada STANDARD

The only allowed implementation dependencies correspond to implementation-dependent pragmas, to certain machine-dependent conventions as mentioned in Chapter 13 of the Ada Standard, and to certain allowed restrictions on representation clauses. The implementation-dependent characteristics of this Ada implementation, as described in this Appendix, are provided by the customer. Unless specifically noted otherwise, references in this Appendix are to compiler documentation and not to this report. Implementation-specific portions of the package STANDARD, which are not a part of Appendix F, are:

package STANDARD is

type SHORT_INTEGER is range -32_768 .. 32_767;

type INTEGER is range -2_147_483_648 .. 2_147_483_647;

type FLOAT is digits 6
range -16#0.FFFF_FF#E32 .. 16#0.FFFF_FF#E32;

type LONG_FLOAT is digits 15
range -16#0.FFFF_FFFF_FFFF_F8#E256 .. 16#0.FFFF_FFFF_FFFF_F8#E256;

type DURATION is delta 2#1.0#E-14 range -131_072.0 .. 131_071.0;

end STANDARD;

APP. F - IMPLEMENTATION DEPENDENT CHARACTERISTICS

This appendix describes the implementation dependent characteristics of DACS Sun SPARC/SunOS and Sun SPARC/Solaris as required in Appendix F of the Ada Reference Manual (ANSI/MIL-STD-1815A).

F.1 Implementation Dependent Pragmas

This section describes all implementation defined pragmas.

F.1.1 Pragma `INTERFACE_SPELLING`

This pragma allows an Ada program to call a non-Ada program whose name contains characters that would be an invalid Ada subprogram identifier. This pragma must be used in conjunction with pragma `INTERFACE`, i.e., pragma `INTERFACE` must be specified for the non-Ada subprogram name prior to using pragma `INTERFACE_SPELLING`.

F.1.1.1 Format

The pragma has the format:

```
pragma INTERFACE_SPELLING (subprogram name, string literal);
```

where the subprogram name is that of one previously given in pragma `INTERFACE` and the string literal is the exact spelling of the interfaced subprogram in its native language. This pragma is only required when the subprogram name contains invalid characters for Ada identifiers.

F.1.1.2 Example

```
function ASSEMBLY_MODULE_NAME return INTEGER;  
  
pragma INTERFACE (AS, ASSEMBLY_MODULE_NAME);  
pragma INTERFACE_SPELLING (ASSEMBLY_MODULE_NAME,  
                           "Illegal$Ada_Name");
```

F.1.2 Pragma `EXTERNAL_NAME`

This pragma allows an Ada program to export the name of an Ada subprogram so that it can be

called from a non-Ada component.

F.1.2.1 Format

The pragma has the format:

```
pragma EXTERNAL_NAME (ada entity, string literal)
```

where ada entity should be the name of:

- A permanent object, i.e. an object placed in the permanent pool of the compilation unit - such objects originate in package specifications and bodies only.
- A constant object, i.e. an object placed in the constant pool of the compilation unit - please note that scalar constants embedded in the code, and composite constants are NOT always placed in the constant pool, because the constant is not considered constant by the compiler.
- A subprogram name, i.e. a name of a subprogram defined in this compilation unit - please notice that separate subprogram specifications cannot be used, the code for the subprogram MUST be present in the compilation unit code.

Objects or subprograms that are local to subprograms or block cannot have external names associated. The entity being made external ("public") MUST be defined in the compilation unit itself. Attempts to name entities from other compilation units will be rejected with a warning.

The string literal is the external name associated the ada entity. The external name should be unique. Specifying identical spellings for different ada entities will generate errors at compile and/or link time, and the responsibility for this is left to the user. Also the user should avoid spellings similar to the spellings generated by the compiler, e.g. M_<unit-no>_<section-no> and other internal identifications.

It is also the responsibility of the user to preserve/restore register %g2 when/if returning to Ada.

When an entity in an object the value associated with the symbol will be the relocatable address of the first byte assigned to the object.

F.1.2.2 Example

This example shows how to export the name of an Ada subprogram and an integer object to be used in a C routine. The C routine is called from the Ada program.

The Ada program:

```
package ext is
  var : integer := 0;
  pragma external_name(var, "_var_name");
  procedure outer_proc(a : integer);

  pragma external_name(inner_proc, "_inner_proc");
end;
```

DACS Sun SPARC/SunOS and Sun SPARC/Solaris Native Ada Compiler System, User's Guide Implementation Dependent Characteristics

```
package body ext is
  procedure inner_proc(a : integer) is
  begin
    var := var + a;
  end inner_proc;
end;

with ext; use ext;
with report; use report;
procedure extern_test is
  procedure call_inner_proc(a : integer);
  pragma interface(c, call_inner_proc);
begin
  test("EXTERN_TEST", "");
  var := 1;
  call_inner_proc(3);
  if var /= 7 then
    failed("inner_proc not call properly");
  end if;
  result;
end;
```

The C program:

```
extern inner_proc();
extern int var_name;
call_inner_proc(in)
int in;
{
  var_name = var_name + in;
  inner_proc(3);
}
```

F.2 Implementation Dependent Attributes

No implementation dependent attributes are defined.

F.3 Package SYSTEM

The package SYSTEM is described in ARM 13.4.

package SYSTEM is

```
type ADDRESS is new INTEGER;
type NAME is (DACS_SPARC);
```

```
SYSTEM_NAME      : constant NAME := DACS_SPARC;
STORAGE_UNIT     : constant      := 8;
MEMORY_SIZE      : constant      := 2048 * 1024;
MIN_INT          : constant      := -2_147_483_648;
MAX_INT          : constant      := 2_147_483_647;
MAX_DIGITS       : constant      := 15;
MAX_MANTISSA     : constant      := 31
```

DACS Sun SPARC/SunOS and Sun SPARC/Solaris Native Ada Compiler System, User's Guide Implementation Dependent Characteristics

```
FINE_DELTA      : constant      := 2#1.0#E-31;
TICK             : constant      := 2#1.0#E-14;

subtype PRIORITY is INTEGER range 1..31;
type INTERFACE_LANGUAGE is (C, AS); -- implementation dependent

-- Compiler system dependent types:

subtype Integer_16 is short_integer;
subtype Natural_16 is Integer_16 range 0..Integer_16'last;
subtype Positive_16 is Integer_16 range 1..Integer_16'last;

subtype Integer_32 is integer;
subtype Natural_32 is Integer_32 range 0..Integer_32'last;
subtype Positive_32 is Integer_32 range 0..Integer_32'last;

end SYSTEM;
```

F.4 Representation Clauses

The representation clauses that are accepted are described below. Note that representation specifications can be given on derived types as well.

F.4.1 Length Clause

Four kinds of length clauses are accepted.

Size specifications:

The size attribute for a type T is accepted in the following cases:

- If T is a discrete type then the specified size must be greater than or equal to the number of bits needed to represent a value of the type, and less than or equal to 32. Note that when the number of bits needed to hold any value of the type is calculated, the range is extended to include 0 if necessary, i.e. the range 3..4 cannot be represented in 1 bit, but needs 3 bits.
- If T is a fixed point type, then the specified size must be greater than or equal to the smallest number of bits needed to hold any value of the fixed point type, and less than 32 bits. Note that the Reference Manual permits a representation, where the lower bound and the upper bound is not representable in the type. Thus the type

```
type FIX is delta 1.0 range -1.0 .. 7.0;
```

is representable in 3 bits. As for discrete types, the number of bits needed for a fixed point type is calculated using the range of the fixed point type possibly extended to include 0.0.

- If T is a floating point type, an access type or a task type the specified size must be equal to the number of bits used to represent values of the type per default (floating points: 32 or 64, access types : 32 bits and task types : 32 bits).

- If T is a record type the specified size must be greater or equal to the minimal number of bits used to represent values of the type per default.
- If T is an array type the size of the array must be static, i.e. known at compile time and the specified size must be equal to the minimal number of bits used to represent values of the type per default.

The size given in the length clause will be used when allocating space for values of the type in all contexts e.g. as part of an array or record. For declared objects the size will be rounded to the nearest number of bytes before the object is allocated.

Collection size specifications:

Using the `STORAGE_SIZE` attribute on an access type will set an upper limit on the total size of objects allocated in the collection allocated for the access type. If further allocation is attempted, the exception `STORAGE_ERROR` is raised. The specified storage size must be less than or equal to `INTEGER'LAST`

Task storage size

When the `STORAGE_SIZE` attribute is given on a task type, the task stack area will be of the specified size. The specified storage size must be less than or equal to `INTEGER'LAST`.

Small specifications

Any value of the `SMALL` attribute less than the specified delta for the fixed point type can be given.

F.4.2 Enumeration Representation Clauses

Enumeration representation clauses may specify representations in the range of `SHORT_INTEGER'FIRST` .. `SHORT_INTEGER'LAST`. An enumeration representation clause may be combined with a length clause. If an enumeration representation clause has been given for a type the representational values are considered when the number of bits needed to hold any value of the type is evaluated. Thus the type

```
type ENUM is (A,B,C);  
for ENUM use (1,2,3);
```

needs 3 bits to represent any value of the type.

F.4.3 Record Representation Clauses

When component clauses are applied to a record type, the following should be noted:

- Components can start at any bit boundary. Placing e.g. non packed arrays on odd bit boundaries will cause costly implicit conversion to be generated, however.
- All values of the component type must be representable within the specified number of bits in the component clause.
- If the component type is either a discrete type or a fixed point type, then the component is packed into the specified number of bits (see however the restriction in the paragraph above).
- If the component type is not one of the types specified in the paragraph above, the default size calculated by the compiler must be given as the bit width, i.e. the component must be specified as

component at N range X..X + component_type'SIZE - 1

where N specifies the relative storage unit number (0,1,...) from the beginning of the record, and X is any bit number.

- The maximum bit width for components of describe or fixed point types is 32.

If the record type contains components which are not covered by a component clause, they are allocated consecutively after the component with the highest offset specified by a component clause. Holes created because of component clauses are not otherwise utilized by the compiler.

When the compiler determines the size of a record component the following is taken into account in the specified order:

- . a component clause
- . a length clause ('SIZE) on the component type
- . a possible pragma PACK on the record type
- . the default size of the component type

F.4.3.1 Alignment Clauses

Alignment clauses for records are supported with the following restrictions:

- The specified alignment boundary must be 1, 2, 4, 8, or 16.
- The specified alignment must not conflict with the alignment requirement for the record components, i.e. an alignment boundary of 4 is not accepted if the record has a component of an array type with size 100 bytes (such arrays should be aligned on a 16 byte boundary).

F.5 Names for Implementation Dependent Components

None defined by the compiler.

F.6 Address Clauses

Address clauses are supported for scalar and for composite objects whose size can be determined at compile time. Address clauses are not supported for subprograms, packages, tasks or task entries.

F.7 Unchecked Conversion

Unchecked conversion is only allowed between objects of the same "size". However, if scalar type has different sizes (packed and unpacked), unchecked conversion between such a type and another type is accepted if either the packed or the unpacked size fits the other type.

F.8 Input/Output Packages

The implementation supports all requirements of the Ada language and the POSIX standard described in document P1003.5 Draft 4.0/WG15-N45. It is an effective interface to the SunOS/Solaris file system, and in the case of text I/O, it is also an effective interface to the SunOS/Solaris standard input, standard output, and standard error streams.

This section describes the functional aspects of the interface to the SunOS/Solaris file system, including the methods of using the interface to take advantage of the file control facilities provided.

The Ada input-output concept as defined in Chapter 14 of the ARM does not constitute a complete functional specification of the input-output packages. Some aspects of the I/O system are not described at all, with others intentionally left open for implementation. This section describes those sections not covered in the ARM. Please notice that the POSIX standard puts restrictions on some of the aspects not described in Chapter 14 of the ARM.

The SunOS/Solaris operating system considers all files to be sequences of bytes. Files can either be accessed sequentially or randomly. Files are not structured into records, but an access routine can treat a file as a sequence of records if it arranges the record level input-output.

Note that for sequential or text files (Ada files not SunOS/Solaris external files) RESET on a file in mode OUT_FILE will empty the file. Also, a sequential or text file opened as an OUT_FILE will be emptied.

F.8.1 External Files

An external file is either a SunOS/Solaris disk file, a SunOS/Solaris FIFO (named pipe), a SunOS/Solaris pipe, or any device defined in the SunOS/Solaris directory. The use of devices such as a tape drive or communication line may require special access permissions or have restrictions. If an inappropriate operation is attempted on a device, the `USE_ERROR` exception is raised.

External files created within the SunOS/Solaris file system shall exist after the termination of the program that created it, and will be accessible from other Ada programs. However, pipes and temporary files will not exist after program termination.

Creation of a file with the same name as an existing external file will cause the existing file to be overwritten.

Creation of files with mode `IN_FILE` will cause `USE_ERROR` to be raised.

The name parameter to the input-output routines must be a valid SunOS/Solaris file name. If the name parameter is empty, then a temporary file is created in the `/usr/tmp` directory. Temporary files are automatically deleted when they are closed.

F.8.2 File Management

This section provides useful information for performing file management functions within an Ada program.

The only restrictions in performing Sequential and Direct I/O are:

- The maximum size of an object of `ELEMENT_TYPE` is 2_147_483_647 bits.
- If the size of an object of `ELEMENT_TYPE` is variable, the maximum size must be determinable at the point of instantiation from the value of the `SIZE` attribute.

The NAME parameter

The `NAME` parameter must be a valid SunOS/Solaris path name (unless it is the empty string). If any directory in the path name is inaccessible, a `USE_ERROR` or a `NAME_ERROR` is raised.

The SunOS/Solaris names `"stdin"`, `"stdout"`, and `"stderr"` can be used with `TEXT_IO.OPEN`. No physical opening of the external file is performed and the internal Ada file will be associated with the already open external file. These names have no significance for other I/O packages.

Temporary files (`NAME = null string`) are created using `tmpname(3)` and are deleted when `CLOSED`. Abnormal program termination may leave temporary files in existence. The name function will return the full name of a temporary file when it exists.

The FORM parameter

The Form parameter, as described below, is applicable to DIRECT_IO, SEQUENTIAL_IO and TEXT_IO operations. The value of the Form parameter for Ada I/O shall be a character string. The value of the character string shall be a series of fields separated by commas. Each field shall consist of optional separators, followed by a field name identifier, followed by optional separators, followed by "=>", followed by optional separators, followed by a field value, followed by optional separators. The allowed values for the field names and the corresponding field values are described below. All field names and field values are case-insensitive.

The following BNF describes the syntax of the FORM parameter:

form	::= [field [, field]*]
fields	::= rights append blocking terminal_input fifo posix_file_descriptor
rights	::= OWNER GROUP WORLD => access {,access_underscore}
access	::= READ WRITE EXECUTE NONE
access_underscore	::= _READ _WRITE _EXECUTE _NONE
append	::= APPEND => YES NO
blocking	::= BLOCKING => TASKS PROGRAM
terminal_input	::= TERMINAL_INPUT => LINES CHARACTERS
fifo	::= FIFO => YES NO
posix_file_descriptor	::= POSIX_FILE_DESCRIPTOR => 2

The FORM parameter is used to control the following :

- File ownership:

Access rights to a file is controlled by the following field names "OWNER", "GROUP" and "WORLD". The field values are "READ", "WRITE", "EXECUTE" and "NONE" or any combination of the previously listed values separated by underscores. The access rights field names are applicable to TEXT_IO, DIRECT_IO and SEQUENTIAL_IO. The default value is OWNER => READ_WRITE, GROUP => READ_WRITE and WORLD => READ_WRITE. The actual access rights on a created file will be the default value subtracted the value of the environment variable umask.

Example

To make a file readable and writable by the owner only, the Form parameter should look something like this:

"Owner =>read_write, World=> none, Group=>none"

If one or more of the field names are missing the default value is used. The permission field is evaluated in left-to-right order. An ambiguity may arise with a Form parameter of the following:

"Owner=>Read_Execute_None_Write_Read"

In this instance, using the left-to-right evaluation order, the "None" field will essentially reset the permissions to none and this example would have the access rights WRITE and READ.

- Appending to a file:

Appending to a file is achieved by using the field name "APPEND" and one of the two field values "YES" or "NO". The default value is "NO". "Append" is allowed with both TEXT_IO and SEQUENTIAL_IO. The effect of appending to a file is that all output to that file is written to the end of the named external file. This field may only be used with the "OPEN" operation, using the field name "APPEND" in connection with a "CREATE" operation shall raise USE_ERROR. Furthermore, a USE_ERROR is raised if the specified file is a terminal device or another device.

Example

To append to a file, one would write:

"Append => Yes"

- Blocking vs. non-blocking I/O:

The blocking field name is "Blocking" and the field values are "TASKS" and "PROGRAM". The default value is "PROGRAM". "Blocking=>Tasks" causes the calling task, but no others, to wait for the completion of an I/O operation. "Blocking=>program" causes the all tasks within the program to wait for the completion of the I/O operation. The blocking mechanism is applicable to TEXT_IO, DIRECT_IO and SEQUENTIAL_IO. UNIX does not allow the support of "BLOCKING=>TASKS" currently.

- How characters are read from the keyboard:

The field name is "TERMINAL_INPUT" and the field value is either "LINES" or

"CHARACTERS". The effect of the field value **"Terminal_input => Characters"** is that characters are read in a noncanonical fashion with **Minimum_count=1**, meaning one character at a time and **Time=0.0** corresponding to that the read operation is not satisfied until **Minimum_Count** characters are received. If the field value **"LINES"** is used the characters are read one line at a time in canonical mode. The default value is **Lines**. **"TERMINAL_INPUT"** has no effect if the specified file is not already open or if the file is not open on a terminal. It is permitted for the same terminal device to be opened for input in both modes as separate Ada file objects. In this case, no user input characters shall be read from the input device without an explicit input operation on one of the file objects. The **"TERMINAL_INPUT"** mechanism is only applicable to **TEXT_IO**.

- **Creation of FIFO files:**

The field name is **"Fifo"** and the field value is either **"YES"** or **"NO"**. **"FIFO => YES"** means that the file shall be a named FIFO file. The default value is **"No"**.

For use with **TEXT_I/O**, the **"Fifo"** field is only allowed with the **Create** operation. If used in connection with an open operation an **USE_ERROR** is raised.

For **SEQUENTIAL_IO**, the FIFO mechanism is applicable for both the **Create** and **Open** operation.

In connection with **SEQUENTIAL_IO**, an additional field name **"O_NDELAY"** is used. The field values allowed for **"O_NDELAY"** are **"YES"** and **"NO"**. Default is **"NO"**. The **"O_NDELAY"** field name is provided to allow waiting or immediate return. If, for example, the following form parameter is given:

"Fifo=>Yes, O_Ndelay=>Yes"

then waiting is performed until completion of the operation. The **"O_Ndelay"** field name only has meaning in connection with the FIFO facility and is otherwise ignored.

- **Access to Open POSIX files:**

The field name is **"POSIX_File_Descriptor"**. The field value is the character string **"2"** which denotes the **stderr** file. Any other field value will result in **USE_ERROR** being raised. The **Name** parameter provides the value which will be returned by subsequent usage of the **Name** function. The operation does not change the state of the file. During the period that the Ada file is open, the result of any file operations on the file descriptor are undefined. Note that this is a method to make **stderr** accessible from an Ada program.

File Access

The following guidelines should be observed when performing file I/O operations:

- At a given instant, any number of files in an Ada program can be associated with corresponding external files.

DACS Sun SPARC/SunOS and Sun SPARC/Solaris Native Ada Compiler System, User's Guide Implementation Dependent Characteristics

- When sharing files between programs, it is the responsibility of the programmer to determine the effects of sharing files.
- The RESET and OPEN operations to files with mode OUT_FILE will empty the contents of the file in SEQUENTIAL_IO and TEXT_IO.
- Files can be interchanged between SEQUENTIAL_IO and DIRECT_IO without any special operations if the files are of the same object type.

F.8.3 Buffering

The Ada I/O system provides buffering in addition to the buffering provided by SunOS/Solaris. The Ada TEXT_IO packages will flush all output to the operating system under the following circumstances:

1. The device is a terminal device and an end of line, end of page, or end of file has occurred.
2. The device is a terminal device and the same Ada program makes an Ada TEXT_IO input request or another file object representing the same device.

Please refer to Appendix E (Root Library Support) for the full specifications of all I/O packages.

F.9 Machine Code Insertions

Currently machine code insertions are not supported.