

AD-A239 811



FORMATION PAGE

Form Approved
OPM No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Service, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Information and Regulatory Affairs, Office of Management and Budget, Washington, DC 20503.

1. AGENCY USE ONLY (Leave Blank)		2. REPORT DATE		3. REPORT TYPE AND DATES COVERED Final: 13 Feb 1991 to 01 June 1993	
4. TITLE AND SUBTITLE Alsys GmbH & Co. KG, AlsCOMP_055, Version 1.82, VAX 8530 (Host) to KWS EB68020 (Target), 91200111.11128				5. FUNDING NUMBERS	
6. AUTHOR(S) IABG-AVF Ottobrunn, Federal Republic of Germany					
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) IABG-AVF, Industrieanlagen-Betriebsgesellschaft Dept. SZT/ Einsteinstrasse 20 D-8012 Ottobrunn FEDERAL REPUBLIC OF GERMANY				8. PERFORMING ORGANIZATION REPORT NUMBER IABG-VSR 095	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Ada Joint Program Office United States Department of Defense Pentagon, Rm 3E114 Washington, D.C. 20301-3081				10. SPONSORING/MONITORING AGENCY REPORT NUMBER	
11. SUPPLEMENTARY NOTES					
12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution unlimited.				12b. DISTRIBUTION CODE	
13. ABSTRACT (Maximum 200 words) Alsys GmbH & Co. KG, AlsCOMP_055, Version 1.82, VAX 8530 under VMS, Version 5.3 (Host) to KWS EB68020 under OS-9/68020, Version 2.3 (Target), AVCV 1.11. DTIC ELECTE S B D AUG 26 1991 91-08766 					
14. SUBJECT TERMS Ada programming language, Ada Compiler Val. Summary Report, Ada Compiler Val. Capability, Val. Testing, Ada Val. Office, Ada Val. Facility, ANSI/MIL-STD-1815A, AJPO.				15. NUMBER OF PAGES	
				16. PRICE CODE	
17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED		18. SECURITY CLASSIFICATION UNCLASSIFIED		19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	
20. LIMITATION OF ABSTRACT					

Certificate Information

The following Ada implementation was tested and determined to pass ACVC 1.11. Testing was completed on 1 February, 1991.

Compiler Name and Version: AlsysCOMP_055, Version 1.82

Host Computer System: VAX 8530 under VMS, Version 5.3

Target Computer System: KWS EB68020 under OS-9/68020, Version 2.3

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

As a result of this validation effort, Validation Certificate 910201I1.11128 is awarded to Alsys. This certificate expires on 1 March, 1993.

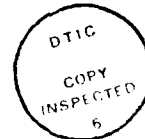
This report has been reviewed and is approved.

Michael Tonndorf

IABG, Abt. ITE
Michael Tonndorf
Einsteinstr. 20
W-8012 Ottobrunn
Germany

for [Signature]

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



John P. Solomond

Ada Joint Program Office
Dr. John Solomond, Director
Department of Defense
Washington DC 20301

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

AVF Control Number: IABG-VSR 095
13 February 1991

Ada COMPILER
VALIDATION SUMMARY REPORT:
Certificate Number: #91020111.11128
Alsys GmbH & Co. KG
AlsyCOMP_055, Version 1.82
VAX 8530 => KWS EB68020

== based on TEMPLATE Version 90-08-15 ==

Prepared By:
IABG mbH, Abt. ITE
Einsteinstr. 20
W-8012 Ottobrunn
Germany

Certificate Information

The following Ada implementation was tested and determined to pass ACVC 1.11. Testing was completed on 1 February, 1991.

Compiler Name and Version: AlsysCOMP_055, Version 1.82

Host Computer System: VAX 8530 under VMS, Version 5.3

Target Computer System: KWS EB68020 under OS-9/68020, Version 2.3

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

As a result of this validation effort, Validation Certificate 910201I1.11128 is awarded to Alsys. This certificate expires on 1 March, 1993.

This report has been reviewed and is approved.



IABG, Abt. ITE
Michael Tonndorf
Einsteinstr. 20
W-8012 Ottobrunn
Germany



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

Ada Joint Program Office
Dr. John Solomond, Director
Department of Defense
Washington DC 20301

DECLARATION OF CONFORMANCE

The following declaration of conformance was supplied by the customer.

Declaration of Conformance

Customer: Alsys GmbH & Co. KG
Certificate Awardee: Alsys GmbH & Co. KG
Ada Validation Facility: IABG mbH Abt. ITE
ACVC Version: 1.11
Ada Implementation:
Ada Compiler Name and Version: AlsyCOMP_055, Version 1.82
Host Computer System: VAX 8530 under VMS, Version 5.3-1
Target Computer System: KWS EB68020 under OS-9/68020, Version 2.3

Declaration:

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



Customer Signature
Dr. Georg Winterstein

Jan. 31 1991

Date

TABLE OF CONTENTS

CHAPTER 1	INTRODUCTION	
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	IMPLEMENTATION DEPENDENCIES	
2.1	WITHDRAWN TESTS	2-1
2.2	INAPPLICABLE TESTS	2-1
2.3	TEST MODIFICATIONS	2-4
CHAPTER 3	PROCESSING INFORMATION	
3.1	TESTING ENVIRONMENT	3-1
3.2	SUMMARY OF TEST RESULTS	3-1
3.3	TEST EXECUTION	3-2
APPENDIX A	MACRO PARAMETERS	
APPENDIX B	COMPILATION SYSTEM OPTIONS	
APPENDIX C	APPENDIX F OF THE Ada STANDARD	

CHAPTER 1

INTRODUCTION

The Ada implementation described above was tested according to the Ada Validation Procedures [Pro90] 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 [Pro90]. 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 VA 22161

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

Ada Validation Organization
Institute for Defense Analyses
1801 North Beauregard Street
Alexandria VA 22311

1.2 REFERENCES

- [Ada83] Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815A, February 1983 and ISO 8652-1987.
- [Pro90] Ada Compiler Validation Procedures, Version 2.1, Ada Joint Program Office, August 1990.
- [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 SPRT13, 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 SPRT13 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 2.2 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, consisting of the test suite, the support programs, the ACVC user's guide and the template for the validation summary report.
Ada Implementation	An Ada compiler with its host computer system and its target computer 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.
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 [Pro90].
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

The following tests have been withdrawn by the AVO. 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 January 09, 1991.

E28005C	B28006C	C34006D	C35702A	B41308B	C43004A
C45114A	C45346A	C45612B	C45651A	C46022A	B49008A
A74006A	C74308A	B83022B	B83022H	B83025B	B83025D
B83026B	B85001L	C83026A	C83041A	C97116A	C98003B
BA2011A	CB7001A	CB7001B	CB7004A	CC1223A	BC1226A
CC1226B	BC3009B	AD1B08A	BD1B02B	BD1B06A	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. Reasons for a test's inapplicability may be supported by documents issued by ISO and the AJPO known as Approved 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 Approved Ada Commentaries are included as appropriate.

IMPLEMENTATION DEPENDENCIES

The following 159 tests have floating-point type declarations requiring more digits than `SYSTEM.MAX_DIGITS`:

C241130..Y (11 tests) (*)	C357050..Y (11 tests)
C357060..Y (11 tests)	C357070..Y (11 tests)
C357080..Y (11 tests)	C358020..Z (12 tests)
C452410..Y (11 tests)	C453210..Y (11 tests)
C454210..Y (11 tests)	C455210..Z (12 tests)
C455240..Z (12 tests)	C456210..Z (12 tests)
C456410..Y (11 tests)	C460120..Z (12 tests)

(*) C24113W..Y (3 tests) contain lines of length greater than 255 characters which are not supported by this implementation.

The following 21 tests check for the predefined type `LONG_INTEGER`:

C35404C	C45231C	C45304C	C45411C	C45412C
C45502C	C45503C	C45504C	C45504F	C45611C
C45612C	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`.

C35713D and B86001Z check for a predefined floating-point type with a name other than `FLOAT`, `LONG_FLOAT`, or `SHORT_FLOAT`.

C41401A checks that `CONSTRAINT_ERROR` is raised upon the evaluation of various attribute prefixes; this implementation derives the attribute values from the subtype of the prefix at compilation time, and thus does not evaluate the prefix or raise the exception. (See Section 2.3.)

C45531M..P (4 tests) and C45532M..P (4 tests) check fixed-point operations for types that require a `SYSTEM.MAX_MANTISSA` of 47 or greater.

C45624A checks that the proper exception is raised if `MACHINE_OVERFLOW` is `FALSE` for floating point types with digits 5. For this implementation, `MACHINE_OVERFLOW` is `TRUE`.

C45624B checks that the proper exception is raised if `MACHINE_OVERFLOW` is `FALSE` for floating point types with digits 6. For this implementation, `MACHINE_OVERFLOW` is `TRUE`.

C86001F recompiles package `SYSTEM`, making package `TEXT_IO`, and hence package `REPORT`, obsolete. For this implementation, the package `TEXT_IO` is dependent upon package `SYSTEM`.

B86001Y checks for a predefined fixed-point type other than `DURATION`.

C96005B checks for values of type `DURATION'BASE` that are outside the range of `DURATION`. There are no such values for this implementation.

IMPLEMENTATION DEPENDENCIES

CD1009C uses a representation clause specifying a non-default size for a floating-point type.

CD2A84A, CD2A84E, CD2A84I..J (2 tests), and CD2A84O use representation clauses specifying non-default sizes for access types.

BD8001A, BD8003A, BD8004A..B (2 tests), and AD8011A use machine code insertions.

The tests listed in the following table are not applicable because the given file operations are supported for the given combination of mode and file access method.

Test	File Operation	Mode	File Access Method
CE2102D	CREATE	IN_FILE	SEQUENTIAL_IO
CE2102E	CREATE	OUT_FILE	SEQUENTIAL_IO
CE2102F	CREATE	INOUT_FILE	DIRECT_IO
CE2102I	CREATE	IN_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
CE3102E	CREATE	IN_FILE	TEXT_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

CE2107C..D (2 tests), CE2107H, and CE2107L apply function NAME to temporary sequential, direct, and text files in an attempt to associate multiple internal files with the same external file; USE_ERROR is raised because temporary files have no name.

CE2108B, CE2108D, and CE3112B use the names of temporary sequential, direct, and text files that were created in other tests in order to check that the temporary files are not accessible after the completion of those tests; for this implementation, temporary files have no name.

CE2203A checks that WRITE raises USE_ERROR if the capacity of the external file is exceeded for SEQUENTIAL_IO. This implementation does not restrict file capacity.

EE2401D contains instantiations of package DIRECT_IO with unconstrained array types. This implementation raises USE_ERROR upon creation of such a file.

IMPLEMENTATION DEPENDENCIES

CE2403A checks that WRITE raises USE_ERROR if the capacity of the external file is exceeded for DIRECT_IO. This implementation does not 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 2.3).

CE3202A assumes that the NAME operation is supported for STANDARD_INPUT and STANDARD_OUTPUT. For this implementation the underlying operating system does not support the NAME operation for STANDARD_INPUT and STANDARD_OUTPUT. Thus the calls of the NAME operation for the standard files in this test raise USE_ERROR (see 2.3).

CE3304A checks that USE_ERROR is raised if a call to SET_LINE_LENGTH or SET_PAGE_LENGTH specifies a value that is inappropriate for the external file. This implementation does not have inappropriate values for either line length or page length.

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 24 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	B24009A	B29001A	B38003A	B38009A	B38009B
B91001H	BC2001D	BC2001E	BC3204B	BC3205B	BC3205D

C34007P and C34007S were graded passed by Evaluation Modification as directed by the AVO. These tests include a check that the evaluation of the selector "all" raises CONSTRAINT_ERROR when the value of the object is null. This implementation determines the result of the equality tests at lines 207 and 223, respectively, based on the subtype of the object; thus, the selector is not evaluated and no exception is raised, as allowed by LRM 11.6(7). The tests were graded passed given that their only output from Report.Failed was the message "NO EXCEPTION FOR NULL.ALL - 2".

C41401A was graded inapplicable by Evaluation Modification as directed by the AVO. This test checks that the evaluation of attribute prefixes that denote variables of an access type raises CONSTRAINT_ERROR when the value of the variable is null and the attribute is appropriate for an array or task type. This implementation derives the array attribute values from the subtype; thus, the prefix is not evaluated and no exception is raised, as allowed by LRM 11.6(7), for the checks at lines 77, 87, 97, 108, 121, 131, 141, 152, 165, & 175.

IMPLEMENTATION DEPENDENCIES

C83030C and C86007A were graded passed by Test Modification as directed by the AVO. These tests were modified by inserting "PRAGMA ELABORATE (REPORT);" before the package declarations at lines 13 and 11, respectively. Without the pragma, the packages may be elaborated prior to package Report's body, and thus the packages' calls to function REPORT.IDENT_INT at lines 14 and 13, respectively, will raise PROGRAM_ERROR.

BC3204C..D and BC3205C..D (4 tests) were graded passed by Evaluation Modification as directed by the AVO. These tests are expected to produce compilation errors, but this implementation compiles the units without error; all errors are detected at link time. This behavior is allowed by AI-00256, as the units are illegal only with respect to units that they do not depend on.

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 attempt to read the file at lines 87 & 101, respectively.

CE3202A was graded inapplicable by Evaluation Modification as directed by the AVO. This test applies function NAME to the standard input file, which in this implementation has no name; USE_ERROR is raised but not handled, so the test is aborted. The AVO ruled that this behavior is acceptable pending any resolution of the issue by the ARG.

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 a point of contact in Germany for technical and sales information about this Ada implementation system, see:

Alsys GmbH & Co. KG
Am Rüppurrer Schloß 7
W-7500 Karlsruhe 51
Germany
Tel. +49 721 883025

For a point of contact outside Germany for technical and sales information about this Ada implementation system, see:

Alsys Inc.
67 South Bedford Str.
Burlington MA
01803-5152
USA
Tel. +617 270 0030

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 [Pro90].

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

a) Total Number of Applicable Tests	3838	
b) Total Number of Withdrawn Tests	84	
c) Processed Inapplicable Tests	89	
d) Non-Processed I/O Tests	0	
e) Non-Processed Floating-Point Precision Tests	159	
f) Total Number of Inapplicable Tests	248	(c+d+e)
g) Total Number of Tests for ACVC 1.11	4170	(a+b+f)

All I/O tests of the test suite were processed because this implementation supports a file system. The above number of floating-point tests were not processed because they used floating-point precision exceeding that supported by the implementation. When this compiler was tested, the tests listed in section 2.1 had been withdrawn because of test errors.

3.3 TEST EXECUTION

Version 1.11 of the ACVC comprises 4170 tests. When this compiler was tested, the tests listed in section 2.1 had been withdrawn because of test errors. The AVF determined that 248 tests were inapplicable to this implementation. All inapplicable tests were processed during validation testing except for 159 executable tests that use floating-point precision exceeding that supported by the implementation. In addition, the modified tests mentioned in section 2.3 were also processed.

A Magnetic Tape Reel containing the customized test suite (see section 1.3) was taken on-site by the validation team for processing. The contents of the 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 and linked on the host computer system, as appropriate. The executable images were transferred to the target computer system via a V24 connection and run. The results were captured on the host 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:

Compiler options :

/LIST tells the compiler to produce full listings in the specified directory, used for all class B tests, all inapplicable executable tests and all class E tests.

/LOG tells the Compiler to write additional messages onto the specified file.

Linker options :

/EXEC together with a parameter specifies the name of the executable program module to be produced.

/LOG tells the implicitly invoked Completer to write additional messages onto the specified file.

/MEMORY=1000 specifies the memory size (K bytes) for both the stack and heap when running the executable program module on the target.

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

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 below.

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 following macro parameters are defined in term of the value V of `$MAX_IN_LEN` which is the maximum input line length permitted for the tested implementation. For these parameters, Ada string expressions are given rather than the macro values themselves.

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

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

Macro Parameter	Macro Value
\$MAX_IN_LEN	255
\$ACC_SIZE	32
\$ALIGNMENT	4
\$COUNT_LAST	2147483647
\$DEFAULT_MEM_SIZE	2147483648
\$DEFAULT_STOR_UNIT	8
\$DEFAULT_SYS_NAME	MOTOROLA_68020_OS9
\$DELTA_DOC	2#1.0#E-31
\$ENTRY_ADDRESS	SYSTEM.INTERRUPT_ADDRESS(1)
\$ENTRY_ADDRESS1	SYSTEM.INTERRUPT_ADDRESS(2)
\$ENTRY_ADDRESS2	SYSTEM.INTERRUPT_ADDRESS(3)
\$FIELD_LAST	512
\$FILE_TERMINATOR	' '
\$FIXED_NAME	NO_SUCH_FIXED_TYPE
\$FLOAT_NAME	NO_SUCH_FLOAT_TYPE
\$FORM_STRING	""
\$FORM_STRING2	"CANNOT_RESTRICT_FILE_CAPACITY"
\$GREATER_THAN_DURATION	0.0

\$GREATER_THAN_DURATION_BASE_LAST	200_000.0
\$GREATER_THAN_FLOAT_BASE_LAST	16#1.0#E+256
\$GREATER_THAN_FLOAT_SAFE_LARGE	16#0.8#E+256
\$GREATER_THAN_SHORT_FLOAT_SAFE_LARGE	16#0.8#E+32
\$HIGH_PRIORITY	15
\$ILLEGAL_EXTERNAL_FILE_NAME1	illegal!file1
\$ILLEGAL_EXTERNAL_FILE_NAME2	illegal*file2
\$INAPPROPRIATE_LINE_LENGTH	-1
\$INAPPROPRIATE_PAGE_LENGTH	-1
\$INCLUDE_PRAGMA1	PRAGMA INCLUDE ("A28006D1.TST")
\$INCLUDE_PRAGMA2	PRAGMA INCLUDE ("B28006F1.TST")
\$INTEGER_FIRST	-2147483648
\$INTEGER_LAST	2147483647
\$INTEGER_LAST_PLUS_1	2147483648
\$INTERFACE_LANGUAGE	ASSEMBLER
\$LESS_THAN_DURATION	-0.0
\$LESS_THAN_DURATION_BASE_FIRST	-200_000.0
\$LINE_TERMINATOR	ASCII.CR
\$LOW_PRIORITY	0

\$MACHINE_CODE_STATEMENT	NULL;
\$MACHINE_CODE_TYPE	NO_SUCH_TYPE
\$MANTISSA_DOC	31
\$MAX_DIGITS	18
\$MAX_INT	2147483647
\$MAX_INT_PLUS_1	2147483648
\$MIN_INT	-2147483648
\$NAME	NO_SUCH_TYPE
\$NAME_LIST	MOTOROLA_68020_OS9
\$NAME_SPECIFICATION1	/H0/vmo182/acvc11/X2120A
\$NAME_SPECIFICATION2	/H0/vmo182/acvc11/X2120B
\$NAME_SPECIFICATION3	/H0/vmo182/acvc11/X3119A
\$NEG_BASED_INT	16#FFFFFFFFE#
\$NEW_MEM_SIZE	2147483648
\$NEW_STOR_UNIT	8
\$NEW_SYS_NAME	MOTOROLA_68020_OS9
\$PAGE_TERMINATOR	' '
\$RECORD_DEFINITION	NEW INTEGER
\$RECORD_NAME	NO_SUCH_MACHINE_CODE_TYPE

\$TASK_SIZE	32
\$TASK_STORAGE_SIZE	10240
\$TICK	0.01
\$VARIABLE_ADDRESS	GET_VARIABLE_ADDRESS
\$VARIABLE_ADDRESS1	GET_VARIABLE_ADDRESS1
\$VARIABLE_ADDRESS2	GET_VARIABLE_ADDRESS2
\$YOUR_PRAGMA	RESIDENT

APPENDIX B

COMPILATION AND LINKER SYSTEM OPTIONS

The compiler and 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 compiler documentation and not to this report.

4 Compiling

After a program library has been created, one or more compilation units can be compiled in the context of this library. The compilation units can be placed on different source files or they can all be on the same file. One unit, a parameterless procedure, acts as the main program. If all units needed by the main program and the main program itself have been compiled successfully, they can be linked. The resulting code can then be executed.

§4.1 and Chapter 5 describe in detail how to call the Compiler and the Linker. In §4.2 the Completer, which is called to generate code for instances of generic units, is described.

Chapter 6 explains the information which is given if the execution of a program is abandoned due to an unhandled exception.

The information the Compiler produces and outputs in the Compiler listing is explained in §4.4.

Finally, the log of a sample session is given in Chapter 7.

4.1 Compiling Ada Units

To start the SYSTEAM Ada Compiler, use the SAS COMPILE command.

SAS COMPILE	Command Description
Format	
\$ SAS COMPILE file-spec[...]	
Command Qualifiers /[NO]ANALYZE_DEPENDENCY /LIBRARY=directory-spec /[NO]LOG[=file-spec] /[NO]RECOMPILE	Defaults /NOANALYZE_DEPENDENCY /LIBRARY=[.ADALIB] /NOLOG /NORECOMPILE
Positional Qualifiers /[NO]CHECK /[NO]COPY_SOURCE /[NO]INLINE /[NO]LIST[=file-spec] /[NO]MACHINE_CODE /[NO]OPTIMIZE	Defaults /CHECK /COPY_SOURCE /INLINE /NOLIST /NOMACHINE_CODE /OPTIMIZE

Command Parameters

`file-spec`

Specifies the file(s) to be compiled. The default directory is []. The default file type is ADA. The maximum length of lines in `file-spec` is 255. The maximum number of source lines in `file-spec` is 65534. Wild cards are allowed.

Note: If you specify a wild card the order of the compilation is alphabetical, which is not always successful. Thus wild cards should be used together with `/ANALYZE_DEPENDENCY`. With this qualifier the sources can be processed in any order.

Description

The source file may contain a sequence of compilation units (cf. LRM(§10.1)). All compilation units in the source file are compiled individually. When a compilation unit is compiled successfully, the program library is updated and the Compiler continues with the compilation of the next unit on the source file. If the compilation unit contained errors, they are reported (see §4.4). In this case, no update operation is performed on the program library and all subsequent compilation units in the compilation are only analyzed without generating code.

The Compiler delivers the status code WARNING on termination (cf. VAX/VMS, *DCL Dictionary*, command EXIT) if one of the compilation units contained errors. A message corresponding to this code has not been defined; hence %NONAME-W-NOMSG is printed upon notification of a batch job terminated with this status.

Command Qualifiers

`/ANALYZE_DEPENDENCY`

`/NOANALYZE_DEPENDENCY (D)`

Specifies that the Compiler only performs syntactical analysis and the analysis of the dependencies on other units. The units in `file-spec` are entered into the library if they are syntactically correct. The actual compilation is done later with the SAS AUTOCOMPILE command.

Note: An already existing unit with the same name as the new one is replaced and all dependent units become obsolete, unless the source file of both are identical. In this case the library is *not* updated because the dependencies are already known.

By default, the normal, full compilation is done.

/LIBRARY=dir-spec

Specifies the program library the command works on. The SAS COMPILE command needs write access to the library. The default is [.ADALIB].

/LOG[=file-spec]

/NOLOG (D)

Controls whether the Compiler writes additional messages onto the specified file. The default file name is SYS\$OUTPUT. The default file type is LOG.

By default, no additional messages are written.

/RECOMPILE

/NORECOMPILE (D)

Indicates that a recompilation of a previously analyzed source is to be performed. This qualifier should not be used unless the command was produced by the SYSTEAM Ada Recompiler. See the SAS RECOMPILE command.

Positional Qualifiers

/CHECK (D)

/NOCHECK

Controls whether all run-time checks are suppressed. If you specify /NOCHECK this is equivalent to the use of PRAGMA suppress for all kinds of checks.

By default, no run-time checks are suppressed, except in cases where PRAGMA suppress_all appears in the source.

/COPY_SOURCE (D)

/NOCOPY_SOURCE

Controls whether a copy of the source file is kept in the library. The copy in the program library is used for later access by the Debugger or tools like the Recompiler. The name of the copy is generated by the Compiler and need normally not be known by the user. The Recompiler and the Debugger know this name. You can use the SAS DIRECTORY/FULL command to see the file name of the copy. If a specified file contains several compilation units a copy containing only the source text of one compilation unit is stored in the library for each compilation unit. Thus the Recompiler can recompile a single unit.

If /NOCOPY_SOURCE is specified, the Compiler only stores the name of the source file in the program library. In this case the Recompiler and the Debugger are able to use the original file if it still exists.

`/COPY_SOURCE` cannot be specified together with `/ANALYZE_DEPENDENCY`.

`/INLINE (D)`

`/NOINLINE`

Controls whether inline expansion is performed as requested by `PRAGMA inline`. If you specify `/NOINLINE` these pragmas are ignored.

By default, inline expansion is performed.

`/LIST[=file-spec]`

`/NOLIST (D)`

Controls whether a listing file is created. One listing file is created for each source file compiled. If `/LIST` is placed as a command qualifier a listing file is created for all sources. If `/LIST` is placed as a parameter qualifier a listing file is created only for the corresponding source file.

The default directory for listing files is the current default directory. The default file name is the name of the source file being compiled unless `/RE-COMPILE` is specified. In this case the name of the original source file, which is stored in the library, is taken as default. The default file type is `LIS`. No wildcard characters are allowed in the file specification.

By default, the `COMPILE` command does not create a listing file.

`/MACHINE_CODE`

`/NOMACHINE_CODE (D)`

Controls whether machine code is appended at the listing file. `/MACHINE_CODE` has no effect if `/NOLIST` or `/ANALYZE_DEPENDENCY` is specified.

By default, no machine code is appended at the listing file.

`/OPTIMIZE (D)`

`/NOOPTIMIZE`

Controls whether full optimization is applied in generating code. There is no way to specify that only certain optimizations are to be performed.

By default, full optimization is done.

End of Command Description

5 Linking

An Ada program is a collection of units used by a main program which controls the execution. The main program must be a parameterless library procedure; any parameterless library procedure within a program library can be used as a main program.

The Linker generates an executable program module on the host without using the target, and a debug information file, debug file for short, which is used when debugging the Ada program.

To link a program, call the SAS LINK command.

SAS LINK

Command Description

Format

\$ SAS LINK unit

Command Qualifiers

```

/[NO]CHECK
/[NO]COMPLETE
/[NO]C_STARTUP
/[NO]DEBUG[=file-spec]
/EXECUTABLE=file-spec
/EXTERNAL=(file-spec[...])
/[NO]INLINE
/LIBRARY=directory-spec
/[NO]LIST[=file-spec]
/[NO]LOG[=file-spec]
/[NO]MACHINE_CODE
/MEMORY=integer
/[NO]OPTIMIZE
/[NO]SELECTIVE
/[NO]STB[=file-spec]

```

Defaults

```

/CHECK
/COMPLETE
/NOC_STARTUP
/DEBUG= ... see Text
see Text
/EXTERNAL=()
/INLINE
/LIBRARY=[.ADALIB]
/NOLIST
/NOLOG
/NOMACHINE_CODE
/MEMORY=256
/OPTIMIZE
/SELECTIVE
/NOSTB

```

Command Parameters

unit

Specifies the library unit which is the main program. This must be a parameterless library procedure.

Description

The SAS LINK command invokes the SYSTEAM Ada Linker.

The Linker builds an executable program module. The default file name of the executable program module is the file name of the source file which contained the specified library unit. The default file type is LOD. The default directory is [].

The name of the program module which is stored in the file is the filename after removing the file type, if any, shortened to 16 characters if necessary. The name of the program module can be used for starting the program on the target.

Command Qualifiers

/CHECK (D)

/NOCHECK

This qualifier is passed to the implicitly invoked Completer. See the same qualifier with the SAS COMPLETE command.

/COMPLETE (D)

/NOCOMPLETE

Controls whether the Completer of the SYSTEAM Ada System is invoked before the linking is performed. Only specify /NOCOMPLETE if you are sure that there are no instantiations or implicit package bodies to be compiled, e.g. if you repeat the SAS LINK command with different linker options.

/C_STARTUP

/NOC_STARTUP (D)

Controls whether special startup code required by C programs (cf. *OS-9/68000 C Compiler User's Manual, Chapter 9*) is linked to the program module. Specify /C_STARTUP if subprograms written in C are called in the Ada program which is being linked (see also §15.1.5). In this case a C I/O library must be specified with the /EXTERNAL qualifier because the C I/O is initialized by the startup code.

By default, no C startup code is linked to the program module.

/DEBUG[=file-spec] (D)

/NODEBUG

Controls whether a debug information file for the SYSTEAM Ada Debugger is to be generated. If the program is to run under the control of the Debugger it must be linked with the /DEBUG qualifier.

By default, debug information is generated. The default directory and file name of this file is that of the executable program module; the default file type is DBG.

/EXECUTABLE=file-spec

Specifies the name of the executable program module and thus the name of the program module which is stored in the specified file. The name of

the program module is the name of the file (after removing the file type, if any).

The default file name of the executable program module is the file name of the source file which contained the specified library unit. The default file type is LOD. The default directory is [].

/EXTERNAL=(file-spec[...])

Specifies files with external object code to be linked to the Ada program (i.e. object code of those program units which are written in Assembler or in C). If several files are given, they must be separated by commas. The default directory the files are located is []. If no file type is specified none is assumed.

/INLINE (D)

/NOINLINE

This qualifier is passed to the implicitly invoked Completer. See the same qualifier with the SAS COMPLETE command.

/LIBRARY=directory-spec

Specifies the program library the command works on. The SAS LINK command needs write access to the library unless /NOCOMPLETE is specified. If /NOCOMPLETE is specified the SAS LINK command needs only read access. The default library is [.ADALIB].

/LIST[=file-spec]

/NOLIST (D)

This qualifier is passed to the implicitly invoked Completer. See the same qualifier with the SAS COMPLETE command. By default the Completer does not create a listing file.

/LOG[=file-spec]

/NOLOG (D)

Controls whether the implicitly invoked Completer writes additional messages onto the specified file. The default file name is SYS\$OUTPUT. The default file type is LOG.

By default, no additional messages are written.

/MACHINE_CODE

/NOMACHINE_CODE (D)

This qualifier is passed to the implicitly invoked Completer. See the same qualifier with the SAS COMPLETE command. If /LIST and /MACHINE_CODE is specified the Linker of the SYSTEAM Ada System generates a listing with the machine code of the program starter in the file LINK.LIS. The program starter is a routine which contains the calls of the necessary elaboration routines and a call for the Ada subprogram which is the main program.

By default, no machine code is generated.

/MEMORY=number

The integer value specifies the memory size (in *numberK* bytes) for both the stack and heap when running the executable program module on the target. The default is 256K Bytes.

/OPTIMIZE (D)

/NOOPTIMIZE

This qualifier is passed to the implicitly invoked Completer. See the same qualifier with the SAS COMPLETE command.

/SELECTIVE (D)

/NOSELECTIVE

Controls selective linking. Selective linking means that only the code of those subprograms which can actually be called is included in the executable program module. By default, selective linking is performed. With the **/NOSELECTIVE** qualifier the code of all subprograms of all packages in the execution closure of the main procedure is linked into the executable program module.

/STB[=file-spec]

/NOSTB (D)

Controls whether an STB file is to be generated. This file is needed when using the OS-9 debugger instead of the Debugger of the SYSTEAM Ada System. The file is transferred to the target using the SAS SEND command. The default directory and file name of the STB file is that of the executable program module; the default file type is STB. By default, no STB file is generated.

End of Command Description

The following steps are performed during linking.

1. First the Completer is called, unless suppressed by the **/NOCOMPLETE** qualifier, to complete the bodies of all instances which are used by the main program and all packages which are used by the main program and which do not require a body.
2. Then the Pre-Linker, a component of the Linker, is executed. The Pre-Linker determines the compilation units that have to be linked together and a valid elaboration order, and generates a code sequence to perform the elaboration.
3. Finally, all object files including those specified with the **/EXTERNAL** qualifier are linked.

If object files generated by the OS-9 C Compiler or contained in a C library are to be linked to the program module, special startup code must be included too because C programs require a certain execution environment in order to run correctly. This

environment is not set up by default if the linked program includes calls of C routines (subprograms for which a PRAGMA interface (C) is given), but the `/C_STARTUP` qualifier serves for this purpose. So if `/C_STARTUP` is specified, an object module is included into the program module containing C global variables and initialization code, in particular:

- global variables `environ`, containing a pointer to the current environment, and `errno`, containing an appropriate error code after an unsuccessful UNIX system call has been performed;
- code to initialize variables for the stack-checking in C routines (see option `-s` of the C Compiler);
- code to handle uninitialized traps;
- a call of the routine `_iobinit`, which is defined in the C I/O libraries, to initialize the C I/O facility, and
- a call of the Ada main program using the entry point `main`. There is no return to the starter module after this call.

The code section defined in the starter module is executed first.

If `/C_STARTUP` is specified, at least a C I/O Library must be specified with the `/EXTERNAL` qualifier because of the call of `_iobinit` from inside the starter module. If such a library is missing the linker will report error messages because of unresolved references.

5.1 Inclusion of External Object Code

The Linker is able to read only those object files which were written by a tool of the SYSTEAM Ada System; files which have a format that does not conform to the internal object code format used by the SYSTEAM Ada System cannot be read. This restriction must be obeyed when additional code is linked to the program by use of the `/EXTERNAL` qualifier.

If an object file is transferred from the target to the host by use of the SAS RECEIVE command (cf. §3.1), the resulting file on the host has the appropriate format and no further action is necessary.

If an object file is copied to the host by another tool (that is not part of the SYSTEAM Ada System), the file must be converted into S-Record format before copying. On the host, this S-Record file is converted into the binary format appropriate for the Linker by the SAS EXBIN command.

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 contained in the following Predefined Language Environment (chapter 13 of the compiler user manual).

13 Predefined Language Environment

The predefined language environment comprises the package standard, the language-defined library units and the implementation-defined library units.

13.1 The Package STANDARD

The specification of the package standard is outlined here; it contains all predefined identifiers of the implementation.

PACKAGE standard IS

TYPE boolean IS (false, true);

-- The predefined relational operators for this type are as follows:

```
-- FUNCTION "=" (left, right : boolean) RETURN boolean;
-- FUNCTION "/=" (left, right : boolean) RETURN boolean;
-- FUNCTION "<" (left, right : boolean) RETURN boolean;
-- FUNCTION "<=" (left, right : boolean) RETURN boolean;
-- FUNCTION ">" (left, right : boolean) RETURN boolean;
-- FUNCTION ">=" (left, right : boolean) RETURN boolean;
```

-- The predefined logical operators and the predefined logical
negation operator are as follows:

```
-- FUNCTION "AND" (left, right : boolean) RETURN boolean;
-- FUNCTION "OR" (left, right : boolean) RETURN boolean;
-- FUNCTION "XOR" (left, right : boolean) RETURN boolean;

-- FUNCTION "NOT" (right : boolean) RETURN boolean;
```

-- The universal type universal_integer is predefined.

TYPE integer IS RANGE - 2_147_483_648 .. 2_147_483_647;

-- The predefined operators for this type are as follows:

```
-- FUNCTION "=" (left, right : integer) RETURN boolean;
-- FUNCTION "/=" (left, right : integer) RETURN boolean;
```

```

-- FUNCTION "<" (left, right : integer) RETURN boolean;
-- FUNCTION "<=" (left, right : integer) RETURN boolean;
-- FUNCTION ">" (left, right : integer) RETURN boolean;
-- FUNCTION ">=" (left, right : integer) RETURN boolean;

-- FUNCTION "+" (right : integer) RETURN integer;
-- FUNCTION "-" (right : integer) RETURN integer;
-- FUNCTION "ABS" (right : integer) RETURN integer;

-- FUNCTION "+" (left, right : integer) RETURN integer;
-- FUNCTION "-" (left, right : integer) RETURN integer;
-- FUNCTION "*" (left, right : integer) RETURN integer;
-- FUNCTION "/" (left, right : integer) RETURN integer;
-- FUNCTION "REM" (left, right : integer) RETURN integer;
-- FUNCTION "MOD" (left, right : integer) RETURN integer;

-- FUNCTION "***" (left : integer; right : integer) RETURN integer;

-- An implementation may provide additional predefined integer types.
-- It is recommended that the names of such additional types end
-- with INTEGER as in SHORT_INTEGER or LONG_INTEGER. The
-- specification of each operator for the type universal_integer, or
-- for any additional predefined integer type, is obtained by
-- replacing INTEGER by the name of the type in the specification
-- of the corresponding operator of the type INTEGER, except for the
-- right operand of the exponentiating operator.

TYPE short_integer IS RANGE - 32_768 .. 32_767;

-- The universal type universal_real is predefined.

TYPE float IS DIGITS 15 RANGE
    - 16#0.FFFF_FFFF_FFFF_F8#E256 ..
      16#0.FFFF_FFFF_FFFF_F8#E256;

-- The predefined operators for this type are as follows:

-- FUNCTION "=" (left, right : float) RETURN boolean;
-- FUNCTION "/=" (left, right : float) RETURN boolean;
-- FUNCTION "<" (left, right : float) RETURN boolean;
-- FUNCTION "<=" (left, right : float) RETURN boolean;
-- FUNCTION ">" (left, right : float) RETURN boolean;
-- FUNCTION ">=" (left, right : float) RETURN boolean;

-- FUNCTION "+" (right : float) RETURN float;
-- FUNCTION "-" (right : float) RETURN float;
-- FUNCTION "ABS" (right : float) RETURN float;

```

```

-- FUNCTION "+" (left, right : float) RETURN float;
-- FUNCTION "-" (left, right : float) RETURN float;
-- FUNCTION "*" (left, right : float) RETURN float;
-- FUNCTION "/" (left, right : float) RETURN float;

-- FUNCTION "***" (left : float; right : integer) RETURN float;

-- An implementation may provide additional predefined floating
-- point types. It is recommended that the names of such additional
-- types end with FLOAT as in SHORT_FLOAT or LONG_FLOAT.
-- The specification of each operator for the type universal_real,
-- or for any additional predefined floating point type, is obtained
-- by replacing FLOAT by the name of the type in the specification of
-- the corresponding operator of the type FLOAT.

TYPE short_float IS DIGITS 6 RANGE
    - 16#0.FFFF_FF#E32 .. 16#0.FFFF_FF#E32;

TYPE long_float IS DIGITS 18 RANGE
    - 16#0.FFFF_FFFF_FFFF_FFFF#E4096 ..
      16#0.FFFF_FFFF_FFFF_FFFF#E4096;

-- In addition, the following operators are predefined for universal
-- types:

-- FUNCTION "*" (left : UNIVERSAL_INTEGER; right : UNIVERSAL_REAL)
-- RETURN UNIVERSAL_REAL;
-- FUNCTION "*" (left : UNIVERSAL_REAL; right : UNIVERSAL_INTEGER)
-- RETURN UNIVERSAL_REAL;
-- FUNCTION "/" (left : UNIVERSAL_REAL; right : UNIVERSAL_INTEGER)
-- RETURN UNIVERSAL_REAL;

-- The type universal_fixed is predefined.
-- The only operators declared for this type are

-- FUNCTION "*" (left : ANY_FIXED_POINT_TYPE;
-- right : ANY_FIXED_POINT_TYPE) RETURN UNIVERSAL_FIXED;
-- FUNCTION "/" (left : ANY_FIXED_POINT_TYPE;
-- right : ANY_FIXED_POINT_TYPE) RETURN UNIVERSAL_FIXED;

-- The following characters form the standard ASCII character set.
-- Character literals corresponding to control characters are not
-- identifiers.

TYPE character IS
    (nul, soh, stx, etx, eot, enq, ack, bel,

```

```

bs,   ht,   lf,   vt,   ff,   cr,   so,   si,
dle,  dc1,  dc2,  dc3,  dc4,  nak,  syn,  etb,
can,  em,   sub,  esc,  fs,   gs,   rs,   us,
'!',  '"',  '#',  '$',  '%',  '&',  '...',
'(',  ')',  '*',  '+',  '-',  '.',  '/',
'0',  '1',  '2',  '3',  '4',  '5',  '6',  '7',
'8',  '9',  ':',  ';',  '<',  '=',  '>',  '?',
'@',  'A',  'B',  'C',  'D',  'E',  'F',  'G',
'H',  'I',  'J',  'K',  'L',  'M',  'N',  'O',
'P',  'Q',  'R',  'S',  'T',  'U',  'V',  'W',
'X',  'Y',  'Z',  '[',  '\',  ']',  '^',  '_',
'`',  'a',  'b',  'c',  'd',  'e',  'f',  'g',
'h',  'i',  'j',  'k',  'l',  'm',  'n',  'o',
'p',  'q',  'r',  's',  't',  'u',  'v',  'w',
'x',  'y',  'z',  '{',  '|',  '}',  '~',  del);

```

```

FOR character USE -- 128 ascii CHARACTER SET WITHOUT HOLES
(0, 1, 2, 3, 4, 5, ..., 125, 126, 127);

```

```

-- The predefined operators for the type CHARACTER are the same as
-- for any enumeration type.

```

```

PACKAGE ascii IS

```

```

-- Control characters:

```

```

nul : CONSTANT character := nul;   soh : CONSTANT character := soh;
stx : CONSTANT character := stx;   etx : CONSTANT character := etx;
eot : CONSTANT character := eot;   enq : CONSTANT character := enq;
ack : CONSTANT character := ack;   bel : CONSTANT character := bel;
bs  : CONSTANT character := bs;    ht  : CONSTANT character := ht;
lf  : CONSTANT character := lf;    vt  : CONSTANT character := vt;
ff  : CONSTANT character := ff;    cr  : CONSTANT character := cr;
so  : CONSTANT character := so;    si  : CONSTANT character := si;
dle : CONSTANT character := dle;   dc1 : CONSTANT character := dc1;
dc2 : CONSTANT character := dc2;   dc3 : CONSTANT character := dc3;
dc4 : CONSTANT character := dc4;   nak : CONSTANT character := nak;
syn : CONSTANT character := syn;   etb : CONSTANT character := etb;
can : CONSTANT character := can;   em  : CONSTANT character := em;
sub : CONSTANT character := sub;   esc : CONSTANT character := esc;
fs  : CONSTANT character := fs;    gs  : CONSTANT character := gs;
rs  : CONSTANT character := rs;    us  : CONSTANT character := us;
del : CONSTANT character := del;

```

```

-- Other characters:

```

```

exclam    : CONSTANT character := '!';
quotation : CONSTANT character := '"';
sharp     : CONSTANT character := '#';

```

```

dollar      : CONSTANT character := '$';
percent     : CONSTANT character := '%';
ampersand   : CONSTANT character := '&';
colon       : CONSTANT character := ':';
semicolon   : CONSTANT character := ';';
query       : CONSTANT character := '?';
at_sign     : CONSTANT character := '@';
l_bracket   : CONSTANT character := '[';
back_slash  : CONSTANT character := '\';
r_bracket   : CONSTANT character := ']';
circumflex  : CONSTANT character := '^';
underline   : CONSTANT character := '_';
grave       : CONSTANT character := '`';
l_brace     : CONSTANT character := '{';
bar         : CONSTANT character := '|';
r_brace     : CONSTANT character := '}';
tilde       : CONSTANT character := '~';

```

```
lc_a : CONSTANT character := 'a';
```

```
...
```

```
lc_z : CONSTANT character := 'z';
```

```
END ascii;
```

```
-- Predefined subtypes:
```

```
SUBTYPE natural IS integer RANGE 0 .. integer'last;
```

```
SUBTYPE positive IS integer RANGE 1 .. integer'last;
```

```
-- Predefined string type:
```

```
TYPE string IS ARRAY(positive RANGE <>) OF character;
```

```
PRAGMA pack(string);
```

```
-- The predefined operators for this type are as follows:
```

```
-- FUNCTION "=" (left, right : string) RETURN boolean;
```

```
-- FUNCTION "/=" (left, right : string) RETURN boolean;
```

```
-- FUNCTION "<" (left, right : string) RETURN boolean;
```

```
-- FUNCTION "<=" (left, right : string) RETURN boolean;
```

```
-- FUNCTION ">" (left, right : string) RETURN boolean;
```

```
-- FUNCTION ">=" (left, right : string) RETURN boolean;
```

```
-- FUNCTION "&" (left : string; right : string) RETURN string;
```

```
-- FUNCTION "&" (left : character; right : string) RETURN string;
```

```
-- FUNCTION "&" (left : string; right : character) RETURN string;
```

```
-- FUNCTION "&" (left : character; right : character) RETURN string;

TYPE duration IS DELTA 2#1.0#E-14 RANGE
    - 131_072.0 .. 131_071.999_938_964_843_75;

-- The predefined operators for the type DURATION are the same
-- as for any fixed point type.

-- the predefined exceptions:

constraint_error : EXCEPTION;
numeric_error    : EXCEPTION;
program_error    : EXCEPTION;
storage_error    : EXCEPTION;
tasking_error    : EXCEPTION;

END standard;
```

13.2 Language-Defined Library Units

The following language-defined library units are included in the master library:

- The package system
- The package calendar
- The generic procedure unchecked_deallocation
- The generic function unchecked_conversion
- The package io_exceptions
- The generic package sequential_io
- The generic package direct_io
- The package text_io
- The package low_level_io

13.3 Implementation-Defined Library Units

The master library also contains the implementation-defined library units

- collection_manager,
- timing,
- non_blocking_os_9_io, and
- text_io_extension.

13.3.1 The Package COLLECTION_MANAGER

In addition to unchecked storage deallocation (cf. LRM §13.10.1), this implementation provides the generic package `collection_manager`, which has advantages over unchecked deallocation in some applications; e.g. it makes it possible to clear a collection with a single reset operation. See §15.10 for further information on the use of the collection manager and unchecked deallocation.

The package specification is:

```
GENERIC
  TYPE elem IS LIMITED PRIVATE;
  TYPE acc  IS ACCESS elem;
PACKAGE collection_manager IS

  TYPE status IS LIMITED PRIVATE;

  PROCEDURE mark (s : OUT status);

    -- Marks the heap of type ACC and
    -- delivers the actual status of this heap.

  PROCEDURE release (s : IN status);

    -- Restore the status s on the collection of ACC.
    -- RELEASE without previous MARK raises CONSTRAINT_ERROR

  PROCEDURE reset;

    -- Deallocate all objects on the heap of ACC

  PRIVATE
    -- private declarations

END collection_manager;
```

A call of the procedure `release` with an actual parameter `s` causes the storage occupied by those objects of type `acc` which were allocated after the call of `mark` that delivered `s` as result, to be reclaimed. A call of `reset` causes the storage occupied by all objects of type `acc` which have been allocated so far to be reclaimed and cancels the effect of all previous calls of `mark`.

See §15.2.1 for information on static and dynamic collections and the attribute `STORAGE_SIZE`.

13.3.2 The Package `TIMING`

The package `timing` provides a facility for CPU-time measurement. The package specification is:

`PACKAGE timing IS`

`FUNCTION cpu_time RETURN natural;`

`timing_error : EXCEPTION;`

`END timing;`

A call of the function `cpu_time` returns the CPU-time consumed by the running process in milliseconds. The value `natural'last` will be reached after 24 days, 20 hours, 31 minutes, 23 seconds and 647 milliseconds.

The exception `timing_error` will be raised if a `constraint_error` or `numeric_error` occurs within `cpu_time`.

13.3.3 The Package `NON_BLOCKING_OS_9_IO`

OS-9 only offers synchronous I/O operations. As every synchronous I/O operation includes the necessary *wait* operation, OS-9 suspends the process which performs the I/O operation until the I/O operation has completed. Due to the fact that an Ada program - even if it consists of several Ada tasks - runs as one OS-9 process, the whole Ada program is suspended until the completion of an I/O operation. That is why an I/O implementation which directly uses the OS-9 I/O operations is said to be *blocking*. This holds for I/O operations called directly from the user program using `PRAGMA interface (OS9, ...)` and also for the I/O operations of the predefined I/O packages (cf. Chapter 17).

Blocking I/O is no problem as long as there is only one Ada task, the main program, but if there are other Ada tasks ready for execution one of those should be executed after suspending the task performing the I/O operation. As soon as the I/O operation is completed, this task should become eligible for execution again and a reschedule may

be performed dependent on the tasks' priorities (see Chapter 14). An implementation of the I/O that behaves as described above - i.e. so that a call of an I/O operation only suspends the calling task but not all other tasks of the Ada program - is said to be *non-blocking*.

The package `non_blocking_OS_9_IO` provides a facility to perform non-blocking I/O on devices which are not covered by the predefined I/O packages. To perform non-blocking I/O with the predefined I/O packages, use the FORM string `NON_BLOCKING` when opening the file, as described in §17.2.1.1.

This package offers - in Ada - the OS-9 I/O interface for arbitrary device drivers (which have to be implemented by the user if they are not offered by OS-9) with the sole difference that calls of these I/O operations are not blocking. This is achieved by creating a separate OS-9 process which performs I/O on behalf of those Ada tasks which want to do so.

The package specification is:

WITH system;

PACKAGE non_blocking_OS_9_IO IS

```
SUBTYPE access_mode IS integer RANGE 0 .. 255;
SUBTYPE error_code IS integer RANGE 0 .. 2**31-1;
e_ok : CONSTANT error_code := 0;           -- normal exit
```

```
SUBTYPE file_attributes IS integer RANGE 0 .. 255;
SUBTYPE path_name IS system.address;
    -- Address of a string terminated by ASCII.NUL.
SUBTYPE path_number IS integer RANGE 0 .. 31;
SUBTYPE buffer IS system.address;
SUBTYPE longword IS integer RANGE -2**31 .. 2**31-1;
```

```
PROCEDURE close (path : IN path_number;
                 error : OUT error_code);
```

```
PROCEDURE create (mode      : IN access_mode;
                  attr      : IN file_attributes;
                  initial_size : IN longword;
                  name      : IN path_name;
                  max_nr_bytes : IN longword := 512;
                  path      : OUT path_number;
                  error      : OUT error_code);
```

```
PROCEDURE open (mode      : IN access_mode;
                name      : IN path_name;
```

```

        max_nr_bytes : IN longword := 512;
        path         : OUT path_number;
        error        : OUT error_code);

PROCEDURE read (path         : IN path_number;
               max_nr_bytes : IN longword;
               input_buffer  : IN buffer;
               nr_bytes      : OUT longword;
               error         : OUT error_code);

PROCEDURE readln (path         : IN path_number;
                 max_nr_bytes : IN longword;
                 input_buffer  : IN buffer;
                 nr_bytes      : OUT longword;
                 error         : OUT error_code);

PROCEDURE write (path         : IN path_number;
                max_nr_bytes : IN longword;
                output_buffer : IN buffer;
                nr_bytes      : OUT longword;
                error         : OUT error_code);

PROCEDURE writln (path         : IN path_number;
                 max_nr_bytes : IN longword;
                 output_buffer : IN buffer;
                 nr_bytes      : OUT longword;
                 error         : OUT error_code);

END non_blocking_OS_9_IO;
```

The subprograms defined in this package implement the OS-9 operations `I$Create`, `I$Open`, `I$Read`, `I$ReadLn`, `I$Write`, `I$WritLn` and `I$Close`. The read and write operations are non-blocking, the create/open and close operations remain blocking. The meaning of the parameters of the operations corresponds to that of the related OS-9 operations.

The path parameter used in a read, readln, write and writln operation must be the result of a prior call to an open or create operation defined in the package `non_blocking_OS_9_IO`; otherwise the operations will return an error value of 1. The only exceptions are the path numbers 0, 1, and 2, which correspond to the standard input, output and error file of OS-9. For these files the non-blocking read and write operations may be called directly without the necessity of a prior call of an open operation.

In the following we give a short explanation of the implementation of the non-blocking I/O as well as some important constraints on its use.

In order for the Ada program not to be suspended, it must not perform the OS-9 I/O call; thus the I/O call *must* be executed in another process. For this purpose, with every non-blocking open (or create) operation a new OS-9 process (called I/O process in the following) is created which performs the I/O operations for that file on behalf of the tasks in the Ada program. The name of this process is `ada_nbio`. The synchronization with this process is made using an event (cf. *OS-9/68000 Operating System Technical Manual, Chapter 15*) for the start of the operation and a signal when the I/O operation has completed. The signal numbers reserved for these purposes lie in the range from `16#FFE0#` to `16#FFFF#`; the event names are strings of 11 characters starting with the fixed part `ADAI0` followed by 4 characters denoting the OS-9 process identification of the I/O process and 2 further characters containing the associated signal number decremented by `16#FFE0#`, both represented as a hexadecimal value.

Communication between the two processes is done using an OS-9 data module (cf. `F$DatMod`). In this module the kind of the desired operation as well as the IN and OUT parameters and the necessary I/O buffers are stored. Note that it is necessary to use an extra I/O buffer in this module because the I/O process cannot access the memory of the main program if it is running on an OS-9 system with memory protection.

The I/O process waits for the corresponding event, decodes the demanded operation, executes it, and sends a signal to the Ada program after completion. The task which initiates the I/O operation sets the event and suspends itself. The tasking system resumes the calling task when the corresponding signal is sent by the I/O process.

This implementation only supports one task performing I/O on each file object. As access to the data module is not synchronized, an Ada program with more than one task performing I/O on the same file will have unpredictable effects and thus is regarded to be erroneous.

For non-blocking I/O on the three standard files, three I/O processes are created when the package `non_blocking_OS_9_IO` is elaborated. No open operation is demanded for these files.

An I/O process created by an open or create operation on a file is terminated when the close operation is called. The I/O processes created for the standard files are terminated when the Ada program terminates.

If a file opened to do non-blocking I/O is not closed by the non-blocking I/O close operation, the created I/O process will not be terminated on termination of the Ada program. In addition the data module associated with each open file will remain in memory. This can be seen with the `mdir` command. Such a data module has the name

`ADADMpppps`

where `pppp` is the process number of the main program and `ss` is the path index of the respective file. For each remaining module *two* links exist (as shown by the `mdir` command) which must be explicitly removed by issuing twice the command `unlink ADADMpppps` for the respective data module. The same happens if the Ada program

is aborted by typing CONTROL C or CONTROL E (Keyboard interrupt resp. abort) and there are files still open. The three I/O processes associated with the standard files and their corresponding data modules will remain in any case when the Ada program is aborted in this way. So if an Ada program terminates abnormally, it is useful to check with the OS-9 command `procs` (cf. *Using Professional OS-9, Chapter 8*) if there are remaining I/O processes (which are named `ada_nbio`) and, if so, to kill them using the OS-9 command `kill <proc_id>`, where `<proc_id>` is the process identification shown by the `procs` command. Note that if the Ada program is abandoned because of an exception, the I/O processes for the standard files, but not those for explicitly opened files, are terminated by the Ada system.

The size of a data module used for the communication with the I/O process is dominated by the size of the I/O buffer in it. The size of a module is the sum of 52 bytes (all other information besides the buffer), the size of the buffer, and the size of the module header which is added by OS-9. The size of the buffer depends on the kind of the I/O to be performed:

- `buffer_size = 512`, for `text_io`,
- `buffer_size = recordsize`, for direct I/O and sequential I/O for constrained types,
- `buffer_size = 4096`, if the maximum record size would be greater than 4k or is not defined.

In the direct use of package `non_blocking_OS_9_IO` the buffer size can be specified and has the default of 512 bytes. If a read or write operation of `non_blocking_OS_9_IO` is performed with an actual element size greater than the buffer size, the I/O is split into several steps.

13.3.4 The Package `TEXT_IO_EXTENSION`

In OS-9 there are three standard I/O paths:

- the standard input path
- the standard output path
- the standard error path

(cf. *Using Professional OS-9, Page 5-8*).

The standard input (resp. output) file of `text_io` is associated with the OS-9 standard input (resp. output) path (cf. §17.3.2). The package `text_io_extension` may be used to send output using the standard error path.

```
PACKAGE text_io_extension IS
```

```
    FUNCTION standard_error RETURN text_io.file_type;
```

```
END text_io_extension;
```

The function `standard_error` delivers a file object which is associated with the standard error path of OS-9. At the beginning of program execution this file is always open and has the current mode `out_file`.

15 Appendix F

This chapter, together with the Chapters 16 and 17, is the Appendix F required in LRM, in which all implementation-dependent characteristics of an Ada implementation are described.

15.1 Implementation-Dependent Pragmas

The form, allowed places, and effect of every implementation-dependent pragma is stated in this section.

15.1.1 Predefined Language Pragmas

The form and allowed places of the following pragmas are defined by the language; their effect is (at least partly) implementation-dependent and stated here.

CONTROLLED
has no effect.

ELABORATE
is fully implemented. The SYSTEAM Ada System assumes a PRAGMA elaborate, i.e. stores a unit in the library as if a PRAGMA elaborate for a unit *u* was given, if the compiled unit contains an instantiation of *u* (or for a generic program unit in *u*) and if it is clear that *u* *must* have been elaborated before the compiled unit. In this case an appropriate information message is given. By this means it is avoided that an elaboration order is chosen which would lead to a PROGRAM_ERROR when elaborating the instantiation.

INLINE
Inline expansion of subprograms is supported with the following restrictions: the subprogram must not contain declarations of other subprograms, tasks, generic units or body stubs. If the subprogram is called recursively only the outer call of this subprogram will be expanded.

INTERFACE

is supported for Assembler, OS9 and C. `PRAGMA interface(assembler,...)` provides an interface with the internal calling conventions of the SYSTEAM Ada System. §15.1.3 describes how to use this pragma.

`PRAGMA interface(OS9,...)` is provided to support the call of OS-9 kernel functions. §15.1.4 describes how to use this pragma.

`PRAGMA interface(C,...)` is provided to support the C procedure calling conventions. §15.1.5 describes how to use this pragma.

`PRAGMA interface` should always be used in connection with the `PRAGMA external_name` (see §15.1.2), otherwise the Compiler will generate an internal name that leads to an unsolved reference during linking. These generated names are prefixed with an underline; therefore the user should not use names beginning with an underline.

LIST

is fully implemented. Note that a listing is only generated when the `/LIST` qualifier is specified with the `SAS COMPILE` (or `SAS COMPLETE` or `SAS LINK`) command.

MEMORY_SIZE

has no effect.

OPTIMIZE

has no effect; but see also the `/OPTIMIZE` qualifier with the `SAS COMPILE` command, §4.1

PACK

see §16.1.

PAGE

is fully implemented. Note that form feed characters in the source do not cause a new page in the listing. They are - as well the other format effectors (horizontal tabulation, vertical tabulation, carriage return, and line feed) - replaced by a ~ character in the listing.

PRIORITY

There are two implementation-defined aspects of this pragma: First, the range of the subtype priority, and second, the effect on scheduling (Chapter 14) of not giving this pragma for a task or main program. The range of subtype priority is 0 .. 15, as declared in the predefined library package system (see §15.3); and the effect on scheduling of leaving the priority of a task or main program undefined by not giving PRAGMA priority for it is the same as if the PRAGMA priority 0 had been given (i.e. the task has the lowest priority).

SHARED

is fully supported.

STORAGE_UNIT

has no effect.

SUPPRESS

has no effect, but see §15.1.2 for the implementation-defined PRAGMA suppress-all.

SYSTEM_NAME

has no effect.

15.1.2 Implementation-Defined Pragmas**BYTE_PACK**

see §16.1.

EXTERNAL_NAME (<string>, <ada_name>)

<ada_name> specifies the name of a subprogram or of an object declared in a library package, <string> must be a string literal. It defines the external name of the specified item. The Compiler uses a symbol with this name in the call instruction for the subprogram. The subprogram declaration of <ada_name> must precede this pragma. If several subprograms with the same name satisfy this requirement the pragma refers to that subprogram which is declared last.

Upper and lower cases are distinguished within <string>, i.e. <string> must be given exactly as it is to be used by external routines. This pragma will be used in connection with the pragmas `interface (OS9)`, `interface (assembler)` or `interface (C)` (see §15.1.1).

RESIDENT (<ada_name>)

this pragma causes the value of the object to be held in memory and prevents assignments of a value to the object <ada_name> from being eliminated by the optimizer (see §4.1) of the SYSTEAM Ada Compiler. The following code sequence demonstrates the intended usage of the pragma:

```
...
x : integer;
a : SYSTEM.address;
...
BEGIN
  x := 5;
  a := x'ADDRESS;
  do_something (a); -- let do_something be a non-local
                   -- procedure
                   -- a.ALL will be read in the body
                   -- of do_something
  x := 6;
  ...
```

If this code sequence is compiled by the SYSTEAM Ada Compiler without the `/NOOPTIMIZE` qualifier, the statement `x := 5;` will be eliminated because from the point of view of the optimizer the value of `x` is not used before the next assignment to `x`. Therefore

```
PRAGMA resident (x);
```

should be inserted after the declaration of `x`.

This pragma can be applied to all those kinds of objects for which the address clause is supported (cf. §16.5).

It will often be used in connection with the `PRAGMA interface (OS9, ...)` (see §15.1.4).

SUPPRESS_ALL

causes all the runtime checks described in the LRM §11.7 to be suppressed; this pragma is only allowed at the start of a compilation before the first compilation unit; it applies to the whole compilation.

15.1.3 Pragma Interface (Assembler,...)

This section describes the internal calling conventions of the SYSTEAM Ada System, which are identical to those assumed for subprograms for which a PRAGMA interface (*assembler,...*) is given. Thus the actual meaning of this pragma is simply that the body need and must not be provided in Ada, but in object form using the /EXTERNAL qualifier with the SAS LINK command.

The internal calling conventions are explained in four steps:

- Parameter passing mechanism
- Ordering of parameters
- Type mapping
- Saving registers

Parameter passing mechanism:

The parameters of a call to a subprogram are placed by the caller in an area called *parameter block*. This area is aligned on a longword boundary and contains parameter values (for parameter of scalar types), descriptors (for parameter of composite types) and alignment gaps.

For a function subprogram an extra field is assigned at the beginning of the parameter block containing the function result upon return. Thus the return value of a function is treated like an anonymous parameter of mode OUT. No special treatment is required for a function result except for return values of an unconstrained array type (see below).

A subprogram is called using the BSR instruction. The address pointing to the beginning of the parameter block is pushed onto the stack before calling the subprogram.

In general, the ordering of the parameter values within the parameter block does not agree with the order specified in the Ada subprogram specification. When determining the position of a parameter within the parameter block the calling mechanism and the size and alignment requirements of the parameter type are considered. The size and alignment requirements and the passing mechanism are described in the following:

Scalar parameters or parameters of access types are passed by value, i.e. the values of the actual parameters of modes IN or IN OUT are copied into the parameter block before the call. Then, after the subprogram has returned, values of the actual parameters of modes IN OUT and OUT are copied out of the parameter block into the associated actual parameters. The parameters are aligned within the parameter block according to their size: A parameter with a size of 8, 16 or 32 bits (or a multiple of 8 bits greater than 32) has an alignment of 1, 2 or 4 (which means that the object is aligned to a byte, word or longword boundary within the parameter block). If the size of the parameter is not a multiple of 8 bits (which may be achieved by attaching

a size specification to the parameter's type in case of an integer, enumeration or fixed point type) it will be byte aligned. Parameters of access types are always aligned to a longword boundary.

For parameters of composite types, descriptors are placed in the parameter block instead of the complete object values. A descriptor contains the address of the actual parameter object and, possibly, further information dependent on the specific parameter type. The following composite parameter types are distinguished:

- A parameter of a constrained array type is passed by reference for all parameter modes.
- For a parameter of an unconstrained array type, the descriptor consists of the address of the actual array parameter followed by the bounds for each index range in the array (i.e. FIRST(1), LAST(1), FIRST(2), LAST(2), ...). The space allocated for the bound elements in the descriptor depends on the type of the index constraint.
- For functions whose return value is an unconstrained array type a descriptor for the array is passed in the parameter block as for parameters of mode OUT. The fields for its address and all array index bounds are filled up by the function before it returns. In contrast to the procedure for an OUT parameter, the function allocates the array in its own stack space. The function then returns without releasing its stack space. After the function has returned, the calling routine copies the array into its own memory space and then deallocates the stack memory of the function.
- A constrained record parameter is passed by reference for all parameter modes.
- For an unconstrained record parameter of mode IN, the parameter is passed by reference using the address pointing to the record.

If the parameter has mode OUT or IN OUT, the value of the CONSTRAINED attribute applied to the actual parameter is passed as an additional boolean IN parameter (which occupies one byte in the parameter block and is aligned to a byte boundary). The boolean IN parameter and the address are treated like two consecutive parameters in a subprogram specification, i.e. the positions of the two parameters within the parameter block are determined independently of each other.

For all kinds of composite parameter types the pointer pointing to the actual parameter object is represented by a 32 bit address, which is always aligned to a longword boundary.

Ordering of parameters:

The ordering of the parameters in the parameter block is determined as follows:

The parameters are processed in the order they are defined in the Ada subprogram specification. For a function the return value is treated as an anonymous parameter of mode OUT at the start of the parameter list. Because of the size and alignment

requirements of a parameter it is not always possible to place parameters in such a way that two consecutive parameters are densely located in the parameter block. In such a situation a gap, i.e. a piece of memory space which is not associated with a parameter, exists between two adjacent parameters. Consequently, the size of the parameter block will be larger than the sum of the sizes used for all parameters. In order to minimize the size of the gaps in a parameter block an attempt is made to fill each gap with a parameter that occurs later in the parameter list. If during the allocation of space within the parameter block a parameter is encountered whose size and alignment fit the characteristics of an available gap, then this gap is allocated for the parameter instead of appending it at the end of the parameter block. As each parameter will be aligned to a byte, word or longword boundary the size of any gap may be one, two or three bytes. Every gap of size three bytes can be treated as two gaps, one of size one byte with an alignment of 1 and one of size two bytes with an alignment of 2. So, if a parameter of size two is to be allocated, a two byte gap, if available, is filled up. A parameter of size one will fill a one byte gap. If none exists but a two byte gap is available, this is used as two one byte gaps. By this first fit algorithm all parameters are processed in the order they occur in the Ada program.

A called subprogram accesses each parameter for reading or writing using the parameter block address incremented by an offset from the start of the parameter block suitable for the parameter. So the value of a parameter of a scalar type or an access type is read (or written) directly from (into) the parameter block. For a parameter of a composite type the actual parameter value is accessed via the descriptor stored in the parameter block which contains a pointer to the actual object. When standard entry code sequences are used within the Assembler subprogram (see below), the parameter block address is accessible at address 8(A5).

Type mapping:

To access individual components of array or record types, knowledge about the type mapping for array and record types is required. An array is stored as a sequential concatenation of all its components. Normally, pad bits are used to fill each component to a byte, word, longword or a multiple thereof depending on the size and alignment requirements of the components' subtype. This padding may be influenced using one of the PRAGMAs `pack` or `byte_pack` (cf. §16.1). The offset of an individual array component is then obtained by multiplying the padded size of one array component by the number of components stored in the array before it. This number may be determined from the number of elements for each dimension using the fact that the array elements are stored row by row. (For unconstrained arrays the number of elements for each dimension can be found in the descriptor stored in the parameter block.)

A record object is implemented as a concatenation of its components. Initially, locations are reserved for those components that have a component clause applied to them. Then locations for all other components are reserved. Any gaps large enough to hold components without component clauses are filled, so in general the record components are rearranged. Components in record variants are overlaid. The ordering mechanism

of the components within a record is in principle the same as that for ordering the parameters in the parameter block.

A record may hold implementation-dependent components (cf. §16.4). For a record component whose size depends on discriminants, a generated component holds the offset of the record component within the record object. If a record type includes variant parts there may be a generated component (cf. §16.4) holding the size of the record object. This size component is allocated as the first component within the record object if this location is not reserved by a component clause. Since the mapping of record types is rather complex record component clauses should be introduced for each record component if an object of that type is to be passed to a non Ada subprogram to be sure to access the components correctly.

Saving registers:

The last aspect of the calling conventions discussed here is that of saving registers. The calling subprogram assumes that the values of the registers a0, a1, d0-d2, fp0-fp7 will be destroyed by the called subprogram and saves them of its own accord. If the called subprogram wants to modify further registers it has to ensure that the old values are restored upon return from the subprogram.

Finally we give the appropriate code sequences for the subprogram entry and for the return, which both obey the rules stated above.

A subprogram for which PRAGMA interface(assembler,...) is specified is - in effect - called with the subprogram calling sequence

```

pea    <address of parameter block> | only for functions or
                                           | procedures with parameters
bsr    <subprogram address>
```

Thus the appropriate entry code sequence is

```

link   a5, #<frame-size>+4
clr.l  -4(a5)
        | The field at address -4(a5) is reserved
        | for use by the Ada runtime system
```

The return code sequence is then simply

```
rts
```

for procedures without parameters and

```
rtd    #4
```

for functions and procedures with parameters.

15.1.4 Pragma Interface(OS9,...)

The SYSTEAM Ada System supports PRAGMA interface(OS9,...).

With the help of this pragma the direct call of OS-9 kernel functions is supported.

The pragma ensures the OS-9 standard, in particular:

- Saving of registers
- Calling mechanism.

The name of the routine which implements the subprogram <ada_name> should be specified using the pragma `external_name` (see § 7.1.2), otherwise the Compiler will generate an internal name that leads to an unsolved reference during linking.

The functions of the OS-9 kernel use registers for the transport of parameters. Therefore, the package `system` provides some types to specify parameters.

```
SUBTYPE os9_wordlong IS integer RANGE - 2 ** 31 .. 2 ** 31 - 1;
  -- signed 4-bytes
```

```
TYPE os9_parameter
  IS RECORD
    d0, d1, d2, d3, d4, d5 : os9_wordlong;
    a0, a1, a2              : address;
    error                  : boolean;
  END RECORD;
```

The components `d0 .. d5` and `a0 .. a3` indicate the use of the corresponding registers of the target machine. Before any call of an OS-9 function the values of those components are copied into the corresponding registers. After the call the values of the registers are copied into the corresponding components of the parameter block. To indicate whether

the result of a call is valid the OS-9 functions will set the condition code register and the pragma will set a corresponding boolean value into the component error.

The SYSTEAM Ada Compiler does not check the correct use of the registers. If it is violated the call will be erroneous.

The following example will show the intended usage of the pragma interface (os9). The given procedure serves to open a file with a constant name. It is called in the body of the main program.

WITH system;

PROCEDURE os9_call IS

 read_mode : CONSTANT system.os9_wordlong := 2 ** 0;

 file_name : CONSTANT string := "/HO/TEST/f1" & ascii.nul;

 PRAGMA resident (file_name);

 -- The file "f1" must exist in the directory "/HO/TEST".

 param_os9 : system.os9_parameter;

 path : system.os9_wordlong;

 use_error : EXCEPTION;

 PROCEDURE os9_i_open (pb : IN OUT system.os9_parameter);

 PRAGMA interface (os9, os9_i_open);

 PRAGMA external_name ("I\$Open", os9_i_open);

BEGIN

 param_os9.d0 := read_mode;

 param_os9.a0 := file_name'address;

 os9_i_open (param_os9);

 IF param_os9.error THEN

 RAISE use_error;

 END IF;

 path := param_os9.d0;

END os9_call;

15.1.5 Pragma Interface(C,...)

The SYSTEAM Ada System supports PRAGMA interface(C,...).

With this pragma and by obeying some rules (described below), subprograms can be called which follow the C procedure calling standard. As the user must know something about the internal calling conventions of the SYSTEAM Ada System, we recommend reading §15.1.3 before reading this section and before using PRAGMA interface(C,...).

For each Ada subprogram for which

```
PRAGMA interface (C, <ada_name>);
```

is specified, a routine implementing the body of the subprogram <ada_name> must be provided, written in any language that obeys the C calling conventions (cf. *OS-9/68000 C Compiler User's Manual, Chapter 4*), in particular:

- Saving registers
- Calling mechanism
- C stack frame format.

Functions contained in the C Standard Library may be called too.

The following parameter and result types are supported:

Type	Ada Type
int	standard.integer
double	standard.float
pointer	system.address

The calling mechanism for all parameter types is call by value. The type address may serve to implement all kinds of call by reference: The user may build all kinds of objects and pass their addresses to the C subprogram.

The name of the routine which implements the subprogram <ada_name> should be specified using the pragma external_name (see §15.1.2), otherwise the Compiler will generate an internal name that leads to an unsolved reference during linking. These generated names are prefixed with an underline; therefore the user should not define names beginning with an underline.

As described in [OS-9/68000 C Compiler User's Manual, Chapter 3], C programs require a certain execution environment in order to run correctly. The C Compiler System of OS-9 establishes this environment by running a short code section, the C startup routine, as the first section of a C program. It declares the global variables `errno` and `environ` and contains code to initialize variables used to implement stack-checking in C. Further, it processes the command argument string to make it accessible for the C main procedure and initializes the C I/O facility (by a call of the function `_lobinit`).

This environment must also be established if C functions are called from inside an Ada program. This is not done automatically by the SYSTEAM Ada System when an Ada program contains a subprogram call for which a PRAGMA interface (C) is given, but it is done if the qualifier `/C_STARTUP` is specified in the SAS LINK command (cf. Chapter 5). In this case the Linker of the SYSTEAM Ada System includes appropriate C startup code into the program module. So the `/C_STARTUP` qualifier must be specified whenever the Ada program calls C subprograms or C library functions which use one of the above-mentioned C global variables, perform I/O or contain stack-checking code (C subprograms which were compiled without option `-s`). If the qualifier `/C_STARTUP` is missing in the LINK command but one of the C global variables or the C stack-checking functions are used, the Linker of the SYSTEAM Ada System will report error messages because of unresolved references.

The C startup routine delivered with the SYSTEAM Ada System is a slightly modified version of that of the C Compiler System. The initialization of the variables used for stack-checking and also the stack-checking functions are adapted to be appropriate for the memory management of the SYSTEAM Ada System. These modifications do not affect the functionality of the C functions.

Linking with the qualifier `/C_STARTUP` always requires a C I/O library to be specified with the `/EXTERNAL` qualifier because of the call of `_lobinit` from inside the startup module. It may be necessary to specify further C libraries (for example the C Math Library) and files containing the object code of those program units which are written in Assembler or in C. Any external object file specified, in particular the C Libraries, must reside on the host. It is recommended to copy the C libraries from the target to the host when installing the SYSTEAM Ada System and to place them into the installation directory using the file names which are usually used on OS-9. However, it is up to your system manager to do this or to choose a different directory or different file names (cf. Chapter 18).

The OS-9 system library must not be specified explicitly with `/EXTERNAL` because it is used by default during linking. It must be copied to the host during the installation in any case.

Care must be taken that the C libraries specified in the SAS LINK command are appropriate for the code generated by the C Compilation System. See [OS-9/68000 Compiler User's Manual, Page 1-12] for the dependencies between C Compiler options and the libraries to be chosen.

There are three functions defined in the C Standard Library which are not supported by the SYSTEAM Ada System. These are the functions `freemem`, `stacksiz` and `ibrk`. The functions `freemem` and `stacksiz` will always return a value of -1, the function `ibrk` the address `16#FFFFFFFF#`. This is the same value the C Compiler System returns if the `ibrk` area and the stack have grown together (cf. *OS-9/6800 C Compiler User's Manual, Page 4-128*). Any other function of the C Standard Library may be used in the Ada environment without any restriction.

If the C standard I/O functions are used to write to a file and the file is not closed on program termination (for example, the standard output and standard error file require no close operation) some information written out may be missing, because the C standard I/O is, by default, buffered and the buffer is not flushed in this case. In the C Compiler System, flushing the I/O buffers and closing all open files is done automatically by the `exit` function which is called on program termination. The SYSTEAM Ada System, however, uses the OS-9 system call `F$EXIT` to exit, causing only close but no flush operations to be done automatically. To ensure that all information is written to the file, close the files written by C standard I/O operations correctly by calling the `fclose` function, or clear the buffer by calling the C function `fflush` if the output was done to the standard output or standard error file.

C defines a global variable, called `errno`, which contains an appropriate error code if a call of a C library function fails. This C variable can be made available in the Ada program by use of the function `symbolic_address` defined in `PACKAGE system`. Declare an integer variable in the Ada program and apply an address clause to it using the address of the C variable `errno` (see the following example). Then, if a call of a C function returns an error value, the error code can be obtained by examining this variable.

The following example shows the intended usage of the `PRAGMA interface (C)` to call a library function. The given procedure serves to open a file with a fixed name. It may be called in the body of the main program.

```
WITH system;

PROCEDURE unix_call IS

    errno      : integer;
    FOR errno  USE AT system.symbolic_address ("errno");
    read_mode  : CONSTANT integer := 8#0#;

    file_name  : CONSTANT string := "/h0/test/f1" & ascii.nul;
    PRAGMA resident (file_name);

    ret_code   : integer;

    use_error  : EXCEPTION;
```

```
FUNCTION unix_open (name : system.address;
                   mode : integer) RETURN integer;
PRAGMA interface (C, unix_open);
PRAGMA external_name ("open", unix_open);

BEGIN
  ret_code := unix_open (file_name'address, read_mode);
  IF ret_code = -1 THEN
    RAISE use_error;
    -- the error code is available in the variable "errno"
  END IF;
END unix_call;

PROCEDURE mainprog is
BEGIN
  unix_call;
END;
```

When this example has been compiled, it can be linked as follows:

```
$ SAS LINK mainprog /C_STARTUP /EXTERNAL=<sas>:CLIB020.L
```

where it is assumed that the C I/O library CLIB020.L is transferred to the host (cf. §18.3.3) and resides in the directory <sas>. Here <sas> stands for the directory in which the SYSTEAM Ada System is located on your computer. The specification of this directory can be obtained by using the SAS SHOW LIBRARY command. It displays the directory in the line starting with "created by SAS at:".

15.2 Implementation-Dependent Attributes

The name, type and implementation-dependent aspects of every implementation-dependent attribute is stated in this section.

15.2.1 Language-Defined Attributes

The name and type of all the language-defined attributes are as given in LRM. We note here only the implementation-dependent aspects.

ADDRESS

If this attribute is applied to an object for which storage is allocated, it yields the address of the first storage unit that is occupied by the object.

If it is applied to a subprogram or to a task, it yields the address of the entry point of the subprogram or task body.

If it is applied to a task entry for which an address clause is given, it yields the address given in the address clause.

For any other entity this attribute is not supported and will return the value `system.address_zero`.

IMAGE

The image of a character other than a graphic character (cf. LRM §3.5.5(11)) is the string obtained by replacing each italic character in the indication of the character literal (given in the LRM Annex C(13)) by the corresponding upper-case character. For example, `character'image(nul) = "NUL"`.

MACHINE_OVERFLOW

Yields `true` for each real type or subtype.

MACHINE_ROUND

Yields `true` for each real type or subtype.

STORAGE_SIZE

The value delivered by this attribute applied to an access type is as follows:

If a length specification (`STORAGE_SIZE`, see §16.2) has been given for that type (static collection), the attribute delivers that specified value.

In case of a dynamic collection, i.e. when no length specification by `STORAGE_SIZE` is given for the access type, the attribute delivers the number of storage units currently allocated for the collection. Note that dynamic collections are extended if needed.

If the collection manager (cf. §13.3.1) is used for a dynamic collection the attribute delivers the number of storage units currently allocated for the collection. Note

that in this case the number of storage units currently allocated may be decreased by release operations.

The value delivered by this attribute applied to a task type or task object is as follows:

If a length specification (`STORAGE_SIZE`, see §16.2) has been given for the task type, the attribute delivers that specified value; otherwise, the default value is returned.

15.2.2 Implementation-Defined Attributes

There are no implementation-defined attributes.

15.3 Specification of the Package SYSTEM

The package system as required in the LRM §13.7 is reprinted here with all implementation-dependent characteristics and extensions filled in.

`PACKAGE system IS`

`TYPE designated_by_address IS LIMITED PRIVATE;`

`TYPE address IS ACCESS designated_by_address;`
`FOR address's storage_size USE 0;`

`address_zero : CONSTANT address := NULL;`

`FUNCTION "+" (left : address; right : integer) RETURN address;`
`FUNCTION "+" (left : integer; right : address) RETURN address;`
`FUNCTION "-" (left : address; right : integer) RETURN address;`
`FUNCTION "-" (left : address; right : address) RETURN integer;`

`SUBTYPE external_address IS STRING;`

`-- External addresses use hexadecimal notation with characters`
`-- '0'..'9', 'a'..'f' and 'A'..'F'. For instance:`
`-- "7FFFFFFF"`
`-- "80000000"`
`-- "8" represents the same address as "00000008"`

```

FUNCTION convert_address (addr : external_address) RETURN address;
-- convert_address raises CONSTRAINT_ERROR if the external address
-- addr is the empty string, contains characters other than
-- '0'..'9', 'a'..'f', 'A'..'F' or if the resulting address value
-- cannot be represented with 32 bits.

```

```

FUNCTION convert_address (addr : address) RETURN external_address;
-- The resulting external address consists of exactly 8 characters
-- '0'..'9', 'A'..'F'.

```

```

FUNCTION symbolic_address (symbol : string) RETURN address;
-- Returns the memory address of the object specified by <symbol>;
-- the memory allocated for <symbol> must lie inside the data area,
-- i.e. <symbol> must be declared to be a variable
-- (cf. OS-9 Assembler/Linker/Debugger User's Manual, Chapter 9)

```

```

TYPE name IS (motorola_68020_os9);
system_name : CONSTANT name := motorola_68020_os9;

```

```

storage_unit : CONSTANT := 8;
memory_size  : CONSTANT := 2 ** 31;
min_int      : CONSTANT := - 2 ** 31;
max_int      : CONSTANT := 2 ** 31 - 1;
max_digits   : CONSTANT := 18;
max_mantissa : CONSTANT := 31;
fine_delta   : CONSTANT := 2.0 ** (-31);
tick         : CONSTANT := 0.01;

```

```

SUBTYPE priority IS integer RANGE 0 .. 15;

```

```

TYPE interrupt_number IS RANGE 0 .. 16#FFDD#;

```

```

FUNCTION interrupt_address (interrupt : interrupt_number)
  RETURN address;
-- converts an interrupt_number (signal number) to an address;

```

```

non_ada_error : EXCEPTION;

```

```

-- non_ada_error is raised, if some event occurs which does not
-- correspond to any situation covered by Ada, e.g.:
--   illegal instruction encountered
--   error during address translation
--   illegal address

```

```

TYPE exception_id IS NEW address;

no_exception_id      : CONSTANT exception_id := NULL;

-- Coding of the predefined exceptions:

constraint_error_id : CONSTANT exception_id := ... ;
numeric_error_id   : CONSTANT exception_id := ... ;
program_error_id   : CONSTANT exception_id := ... ;
storage_error_id   : CONSTANT exception_id := ... ;
tasking_error_id   : CONSTANT exception_id := ... ;

non_ada_error_id    : CONSTANT exception_id := ... ;

status_error_id     : CONSTANT exception_id := ... ;
mode_error_id       : CONSTANT exception_id := ... ;
name_error_id       : CONSTANT exception_id := ... ;
use_error_id        : CONSTANT exception_id := ... ;
device_error_id     : CONSTANT exception_id := ... ;
end_error_id        : CONSTANT exception_id := ... ;
data_error_id       : CONSTANT exception_id := ... ;
layout_error_id     : CONSTANT exception_id := ... ;

time_error_id       : CONSTANT exception_id := ... ;

SUBTYPE os9_wordlong IS integer RANGE -2**31 .. 2**31-1;
TYPE os9_parameter
  IS RECORD
    d0, d1, d2, d3, d4, d5 : os9_wordlong;
    a0, a1, a2             : address;
    error                  : boolean;
  END RECORD;

no_error_code      : CONSTANT := 0;

TYPE exception_information
  IS RECORD
    excp_id           : exception_id;
    -- Identification of the exception. The codings of
    -- the predefined exceptions are given above.
    code_addr         : address;
    -- Code address where the exception occurred. Depending
    -- on the kind of the exception it may be be address of
    -- the instruction which caused the exception, or it

```

```
-- may be the address of the instruction which would
-- have been executed if the exception had not occurred.
error_code      : integer;
END RECORD;

PROCEDURE get_exception_information
    (excp_info : OUT exception_information);
-- The subprogram get_exception_information must only be called
-- from within an exception handler BEFORE ANY OTHER EXCEPTION
-- IS RAISED. It then returns the information record about the
-- actually handled exception.
-- Otherwise, its result is undefined.

PROCEDURE raise_exception_id
    (excp_id : exception_id);

PROCEDURE raise_exception_info
    (excp_info : exception_information);

-- The subprogram raise_exception_id raises the exception
-- given as parameter. It corresponds to the RAISE statement.

-- The subprogram raise_exception_info raises the exception
-- described by the information record supplied as parameter.
-- In addition to the subprogram raise_exception_id it allows to
-- explicitly define all components of the
-- exception information record.

-- IT IS INTENDED THAT BOTH SUBPROGRAMS ARE USED ONLY WHEN
-- INTERFACING WITH THE OPERATING SYSTEM.

TYPE exit_code IS NEW integer;

error      : CONSTANT exit_code := 10;
success    : CONSTANT exit_code := 0;

PROCEDURE set_exit_code (val : exit_code);
-- Specifies the exit code which is returned to the
-- operating system if the Ada program terminates normally.
-- The default exit code is 'success'. If the program is
-- abandoned because of an exception, the exit code is
-- 'error'.
```

PRIVATE

```
-- private declarations  
  
END system;
```

15.4 Restrictions on Representation Clauses

See Chapter 16 of this manual.

15.5 Conventions for Implementation-Generated Names

There are implementation generated components but these have no names. (cf. §16.4 of this manual).

15.6 Expressions in Address Clauses

See §16.5 of this manual.

15.7 Restrictions on Unchecked Conversions

The implementation supports unchecked type conversions for all kinds of source and target types with the restriction that the target type must not be an unconstrained array type. The result value of the unchecked conversion is unpredictable, if

```
target_type'SIZE > source_type'SIZE
```

15.8 Characteristics of the Input-Output Packages

The implementation-dependent characteristics of the input-output packages as defined in the LRM Chapter 14 are reported in Chapter 17 of this manual.

15.9 Requirements for a Main Program

A main program must be a parameterless library procedure. This procedure may be a generic instantiation; the generic procedure need not be a library unit.

15.10 Unchecked Storage Deallocation

The generic procedure `unchecked_deallocation` is provided; the effect of calling an instance of this procedure is as described in the LRM §13.10.1.

The implementation also provides an implementation-defined package `collection_manager`, which has advantages over unchecked deallocation in some applications (cf. §13.3.1).

Unchecked deallocation and operations of the `collection_manager` can be combined as follows:

- `collection_manager.reset` can be applied to a collection on which unchecked deallocation has also been used. The effect is that storage of all objects of the collection is reclaimed.
- After the first `unchecked_deallocation` (`release`) on a collection, all following calls of `release` (`unchecked deallocation`) until the next `reset` have no effect, i.e. storage is not reclaimed.
- after a `reset` a collection can be managed by `mark` and `release` (resp. `unchecked_deallocation`) with the normal effect even if it was managed by `unchecked_deallocation` (resp. `mark` and `release`) before the `reset`.

15.11 Machine Code Insertions

A package `machine_code` is not provided and machine code insertions are not supported.

15.12 Numeric Error

The predefined exception `numeric_error` is never raised implicitly by any predefined operation; instead the predefined exception `constraint_error` is raised.

16 Appendix F: Representation Clauses

In this chapter we follow the section numbering of Chapter 13 of LRM and provide notes for the use of the features described in each section.

16.1 Pragmas

PACK

As stipulated in the LRM §13.1, this pragma may be given for a record or array type. It causes the Compiler to select a representation for this type such that gaps between the storage areas allocated to consecutive components are minimized. For components whose type is an array or record type the PRAGMA PACK has no effect on the mapping of the component type. For all other component types the Compiler will choose a representation for the component type that needs minimal storage space (packing down to the bit level). Thus the components of a packed data structure will in general not start at storage unit boundaries.

BYTE_PACK

This is an implementation-defined pragma which takes the same argument as the predefined language PRAGMA PACK and is allowed at the same positions. For components whose type is an array or record type the PRAGMA BYTE_PACK has no effect on the mapping of the component type. For all other component types the Compiler will try to choose a more compact representation for the component type. But in contrast to PRAGMA PACK all components of a packed data structure will start at storage unit boundaries and the size of the components will be a multiple of `system.storage_unit`. Thus, the PRAGMA BYTE_PACK does not effect packing down to the bit level (for this see PRAGMA PACK).

16.2 Length Clauses

SIZE

For all integer, fixed point and enumeration types the value must be ≤ 32 ;
for `short_float` types the value must be $= 32$ (this is the amount of storage which is associated with these types anyway);
for `float` types the value must be $= 64$ (this is the amount of storage which is associated with these types anyway).
for `long_float` types the value must be $= 96$ (this is the amount of storage which is associated with these types anyway).
for access types the value must be $= 32$ (this is the amount of storage which is associated with these types anyway).
If any of the above restrictions are violated, the Compiler responds with a `RE-STRICITION` error message in the Compiler listing.

STORAGE_SIZE

Collection size: If no length clause is given, the storage space needed to contain objects designated by values of the access type and by values of other types derived from it is extended dynamically at runtime as needed. If, on the other hand, a length clause is given, the number of storage units stipulated in the length clause is reserved, and no dynamic extension at runtime occurs.

Storage for tasks: The memory space reserved for a task is 10K bytes if no length clause is given (cf. Chapter 14). If the task is to be allotted either more or less space, a length clause must be given for its task type, and then all tasks of this type will be allotted the amount of space stipulated in the length clause (the activation of a small task requires about 1.4K bytes). Whether a length clause is given or not, the space allotted is not extended dynamically at runtime.

SMALL

There is no implementation-dependent restriction. Any specification for `SMALL` that is allowed by LRM can be given. In particular those values for `SMALL` are also supported which are not a power of two.

16.3 Enumeration Representation Clauses

The integer codes specified for the enumeration type have to lie inside the range of the largest integer type which is supported; this is the type `integer` defined in package `standard`.

16.4 Record Representation Clauses

Record representation clauses are supported. The value of the expression given in an alignment clause must be 0, 1, 2 or 4. If this restriction is violated, the Compiler responds with a **RESTRICTION** error message in the Compiler listing. If the value is 0 the objects of the corresponding record type will not be aligned, if it is 1, 2 or 4 the starting address of an object will be a multiple of the specified alignment.

The number of bits specified by the range of a component clause must not be greater than the amount of storage occupied by this component. (Gaps between components can be forced by leaving some bits unused but not by specifying a bigger range than needed.) Violation of this restriction will produce a **RESTRICTION** error message.

There are implementation-dependent components of record types generated in the following cases :

- If the record type includes variant parts and if it has either more than one discriminant or else the only discriminant may hold more than 256 different values, the generated component holds the size of the record object.
- If the record type includes array or record components whose sizes depend on discriminants, the generated components hold the offsets of these record components (relative to the corresponding generated component) in the record object.

But there are no implementation-generated names (cf. LRM §13.4(8)) denoting these components. So the mapping of these components cannot be influenced by a representation clause.

16.5 Address Clauses

Address clauses are supported for objects declared by an object declaration and for single task entries. If an address clause is given for a subprogram, package or a task unit, the Compiler responds with a **RESTRICTION** error message in the Compiler listing.

If an address clause is given for an object, the storage occupied by the object starts at the given address. Address clauses for single entries are described in §16.5.1.

16.5.1 Interrupts

Under OS-9 it is not possible to handle hardware interrupts directly within the Ada program; all hardware interrupts are handled by the operating system. In OS-9, asynchronous events are dealt with by signals (cf. `F$Send`). In the remainder of this section the terms *signal* and *interrupt* should be regarded as synonyms.

An address clause for an entry associates the entry with a signal. When a signal occurs, the signal processing intercept routine, which is provided by the Ada runtime system, initiates the entry call.

By this mechanism, an interrupt acts as an entry call to that task; such an entry is called an *interrupt entry*. An interrupt causes the `ACCEPT` statement corresponding to the entry to be executed.

The interrupt is mapped to an *ordinary* entry call. The entry may also be called by an Ada entry call statement. However, it is assumed that when an interrupt occurs there is no entry call waiting in the entry queue. Otherwise, the program is erroneous and behaves in the following way:

- If an entry call stemming from an interrupt is already queued, this previous entry call is lost.
- The entry call stemming from the interrupt is inserted into the front of the entry queue, so that it is handled before any entry call stemming from an Ada entry call statement.

16.5.1.1 Association between Entry and Interrupt

The association between an entry and an interrupt is achieved via an interrupt number (type `system.interrupt_number`), the range of interrupt numbers being `0 .. 16#FFDD#` (this means that theoretically 65502 single entries can act as interrupt entries). The meaning of the interrupt (signal) numbers is as defined in `F$Send`. A single parameterless entry of a task can be associated with an interrupt by an address clause (the Compiler does not check these conventions). Since an address value must be given in the address clause, the interrupt number has to be converted into type `system.address`. The function `system.interrupt_address` is provided for this purpose.

The following example associates the entry `ir` with signal `S$Intrpt` (`S$Intrpt = 3`, i.e. the keyboard interrupt).

```
...  
TASK handler IS  
  
    ENTRY ir;  
  
    FOR ir USE AT system.interrupt_address(3);  
END;  
...
```

The task body contains ordinary accept statements for the entries.

16.5.1.2 Important Implementation Information

There are some important facts which the user of interrupt entries should know about the implementation. First of all, there are some signals which are used by the Ada Runtime System to implement exception handling and delay statements. These signals are 16#FFDE# .. 16#FFFF#. Programs using these signal numbers (e.g. in OS-9 system calls) are erroneous.

In the absence of address clauses for entries, the intercept routine in the Ada Runtime System handles only the signals mentioned above, and all other signals will lead to program abortion as specified in the OS-9 documentation.

A signal handler for a specific signal is established when a task which has an interrupt entry for this signal is activated. The signal handler is deactivated when the task has been completed. Several tasks with interrupt entries for the same signal may exist in parallel; in this case the signal handler is activated when the first of these tasks is created, and deactivated when the last of these tasks has been completed.

16.6 Change of Representation

The implementation places no additional restrictions on changes of representation.

17 Appendix F: Input-Output

In this chapter we follow the section numbering of Chapter 14 of LRM and provide notes for the use of the features described in each section.

17.1 External Files and File Objects

An external file is identified by a string that denotes a OS-9 file name.

The form string specified for external files is described in §17.2.1.1.

17.2 Sequential and Direct Files

Sequential and direct files are represented by OS-9 random block files with fixed-length or variable-length records. Each element of the file is stored in one record.

In case of a fixed record length each file element has the same size, which may be specified by a form parameter (see §17.2.1.1); if none is specified, it is determined to be $(\text{element_type}'\text{SIZE} + \text{system.storage_unit} - 1) / \text{system.storage_unit}$.

In contrast, if a variable record length is chosen, the size of each file element may be different. Each file element is written with its actual length. When reading a file element its size is determined as follows:

- If an object of the `element_type` has a size component (see §16.4) the element size is determined by first reading the corresponding size component from the file.
- If `element_type` is constrained, the size is the minimal number of bytes needed to hold a constrained object of that type.
- In all other cases, the size of the current file element is determined by the size of the variable given for reading.

17.2.1 File Management

Since there is a lot to say about this section, we shall introduce subsection numbers which do not exist in LRM.

17.2.1.1 The NAME and FORM Parameters

The name parameter must be a OS-9 file name. The function name will return a path name string which is the complete file name of the file opened or created.

The syntax of the form parameter string is defined by:

```

form_parameter ::= [ form_specification { , form_specification } ]

form_specification ::= keyword [ => value ]

keyword ::= identifier

value ::= identifier | numeric_literal

```

For identifier and numeric_literal see LRM Appendix E. Only an integer literal is allowed as numeric_literal (see LRM §2.4). In an identifier or numeric_literal, upper and lower case letters are not distinguished.

In the following, the form specifications which are allowed for all files are described.

```
ALLOCATION => numeric_literal
```

This value specifies the number of bytes which are allocated initially; it is only used in a create operation and ignored in an open operation. The default value for the initial file size is 0.

```
RECORD_SIZE => numeric_literal
```

This value specifies the record size in bytes. This form specification is only allowed for files with fixed record format. If the value is specified for an existing file, it must agree with the value of the external file.

By default, $(\text{element_type'SIZE} + \text{system.storage_unit} - 1) / \text{system.storage_unit}$ will be chosen as record size, if the evaluation of this expression does not raise an exception. In this case, the attempt to write or read a record will raise `use_error`.

If a fixed record format is used, all objects written to a file which are shorter than the record size are filled up. The content of this extended record area is undefined. An

attempt to write an element which is larger than the specified record size will result in the exception `use_error` being raised. This can only happen if the record size is specified explicitly.

NON_BLOCKING

This form string specified in a create or open operation for a file causes the input and output operations on that file to be non-blocking. Non-blocking I/O means that the call of an input or output operation only suspends the calling task until the I/O operation is completed, but not all other tasks of the Ada program.

I/O on the standard input and standard output files is always non-blocking. By default, I/O on files other than the standard files is blocking, which causes the whole Ada program to be suspended until an I/O operation has completed even if it consists of several Ada tasks.

Non-blocking I/O is implemented using the implementation-defined library package `non_blocking_OS_9_IO` (cf. §13.3.3), which may also be used directly to perform non-blocking I/O on devices which are not covered by the predefined I/O packages. As described in §13.3.3, non-blocking I/O is realized by separate OS-9 processes which perform the I/O operations. These are created when a file is created or opened and are terminated when the file is closed. If the Ada program terminates without performing the close operation on a file for which non-blocking I/O was specified (this may occur if the Ada program abandons because of an exception before the close operation is executed or if the Ada program is aborted by typing in CONTROL C or CONTROL E), an OS-9 process called `ada_nbio` will remain. In this case it is useful to check with the OS-9 command `procs` (cf. *Using Professional OS-9, Chapter 8*) if there are remaining I/O processes and, if so, to kill them using the OS-9 command `kill <proc_id>`, where `<proc_id>` is the process identification shown by the `procs` command. I/O processes for the standard files will remain in the system when the program is aborted by CONTROL C or CONTROL E (but not if the program abandons because of an exception).

Besides the I/O processes, data modules may also remain in memory. See §13.3.3 for instructions on how to detect and remove such data modules.

17.2.1.2 Sequential Files

A sequential file is represented a by random block file that is interpreted to be formatted with either fixed-length or variable-length records (this may be specified by the form parameter).

If a fixed record format is used, all objects written to a file which are shorter than the maximum record size are filled up. The content of this extended record area is undefined.

`RECORD_FORMAT => VARIABLE | FIXED`

This form specification is used to specify the record format. If the format is specified for an existing file it must agree with the format of the external file.

The default is variable record size. This means that each file element is written with its actual length. A read operation transfers exactly one file element with its actual length.

Fixed record size means that every record is written with the size specified as record size.

17.2.1.3 Direct Files

The implementation dependent type count defined in the package specification of `direct_io` has an upper bound of :

`COUNT'LAST = 2_147_483_647 (= INTEGER'LAST)`

A direct file is represented by a random block file that is interpreted to be formatted with records of fixed length. If not explicitly specified, the record size is equal to $(\text{element_type'SIZE} + \text{system.storage_unit} - 1) / \text{system.storage_unit}$.

17.3 Text Input-Output

Text files are represented as random block files or sequential character files depending on whether the file name denotes a disk file or a terminal device. Each line consists of a sequence of characters terminated by a line terminator, i.e. an ASCII.CR character.

A page terminator is represented as a line consisting of a single ASCII.FF. A page terminator is always preceded by a line terminator.

A file terminator is not represented explicitly in the external file; the end of the file is taken as a file terminator. A page terminator is assumed to precede the end of the file if there is not an explicit one as the last record of the file.

17.3.1 File Management

In the following, the form specifications which are only allowed for text files or have a special meaning for text files are described.

CHARACTER_IO

The predefined package `text_io` was designed for sequential text files; moreover, this implementation always uses sequential files with a record structure, even for terminal devices. It therefore offers no language-defined facilities for modifying data previously written to the terminal (e.g. changing characters in a text which is already on the terminal screen) or for outputting characters to the terminal without following them by a line terminator. It also has no language-defined provision for input of single characters from the terminal (as opposed to lines, which must end with a line terminator, so that in order to input one character the user must type in that character and then a line terminator) or for suppressing the echo on the terminal of characters typed in at the keyboard.

For these reasons, in addition to the input/output facilities with record structured external files, another form of input/output is provided for text files: It is possible to transfer single characters from/to a terminal device. This form of input/output is specified by the keyword `CHARACTER_IO` in the form string. If `CHARACTER_IO` is specified, no other form specification is allowed and the file name must denote a terminal device.

For an infile, the external file (associated with a terminal) is considered to contain a single line. Arbitrary characters (including all control characters) may be read; a character read is not echoed to the terminal.

For an outfile, arbitrary characters (including all control characters and escape sequences) may be written on the external file (terminal). A line terminator is represented as ASCII.CR followed by ASCII.LF, a page terminator is represented as ASCII.FF and a file terminator is not represented on the external file.

17.3.2 Default Input and Output Files

The Ada standard input and output files are associated with the corresponding standard files in OS-9.

I/O on the standard input and standard output files is always non-blocking (cf. §13.3.3 and §17.2.1.1). This may result in remaining OS-9 processes and data modules when an Ada program aborts (for details, see §17.2.11, FORM string NON_BLOCKING, which is the default for the standard input and standard output files).

17.3.3 Implementation-Defined Types

The implementation-dependent types `count` and `field` defined in the package specification of `text_io` have the following upper bounds :

```
COUNT'LAST = 2_147_483_647 (= INTEGER'LAST)
```

```
FIELD'LAST = 512
```

17.4 Exceptions in Input-Output

For each of `name_error`, `use_error`, `device_error` and `data_error` we list the conditions under which that exception can be raised. The conditions under which the other exceptions declared in the package `io_exceptions` can be raised are as described in LRM §14.4.

NAME_ERROR

- in an open operation, if the specified file does not exist;
- if the `name` parameter in a call of the `create` or `open` procedure is not a legal OS-9 file specification string; for example, if it contains illegal characters, is too long or is syntactically incorrect; and also if it contains wild cards, even if that would specify a unique file.

USE_ERROR

- whenever an error occurred during an operation of the underlying OS-9 system. This may happen if an internal error was detected, an operation is not possible for reasons depending on the file or device characteristics, a size restriction is violated, a capacity limit is exceeded or for similar reasons;
- if the function `name` is applied to a temporary file or to the standard input or output file;
- if an attempt is made to write or read to/from a file with fixed record format a record which is larger than the record size determined when the file was opened (cf. §17.2.1.1); in general it is only guaranteed that a file which is created by an Ada program may be reopened and read successfully by another program if the file types and the form strings are the same;
- in a `create` or `open` operation for a file with fixed record format (direct file or sequential file with `form` parameter `RECORD_FORMAT => FIXED`) if no record size is specified and the evaluation of the size of the element type will raise an exception (For example, if `direct_io` or `sequential_io` is instantiated with an unconstrained array type);
- if a given `form` parameter string does not have the correct syntax or if a condition on an individual form specification described in §§17.2-3 is not fulfilled.

DEVICE_ERROR

is never raised. Instead of this exception the exception `use_error` is raised whenever an error occurred during an operation of the underlying OS-9 system.

DATA_ERROR

the conditions under which `data_error` is raised by `text_io` are laid down in LRM.

In general, the exception `data_error` is not usually raised by the procedure `read` of `sequential_io` and `direct_io` if the element read is not a legal value of the element type because there is no information about the file type or form strings specified when the file was created.

An illegal value may appear if the package `sequential_io` or `direct_io` was instantiated with a different `element_type` or if a different form parameter string was specified when creating the file. It may also appear if reading a file element is done with a constrained object and the constraint of the file element does not agree with the constraint of the object.

If the element on the file is not a legal value of the element type the effect of reading is undefined. An access to the object that holds the element after reading may cause a `constrained_error`, `storage_error` or `non_ada_error`.

17.5 Low Level Input-Output

We give here the specification of the package `low_level_io`:

```
PACKAGE low_level_io IS

  TYPE device_type IS (null_device);

  TYPE data_type IS
    RECORD
      NULL;
    END RECORD;

  PROCEDURE send_control    (device : device_type;
                             data    : IN OUT data_type);

  PROCEDURE receive_control (device : device_type;
                             data    : IN OUT data_type);

END low_level_io;
```

Note that the enumeration type `device_type` has only one enumeration value, `null_device`; thus the procedures `send_control` and `receive_control` can be called, but `send_control` will have no effect on any physical device and the value of the actual parameter `data` after a call of `receive_control` will have no physical significance.