IENTATION PAGE

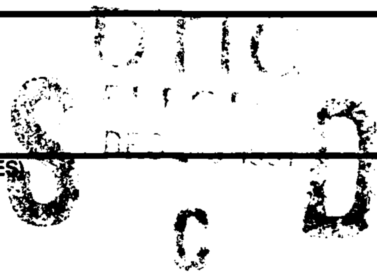
Form Approved
OPM No. 0704-0188

AD-A243 294

...erage 1 hour per response, including the time for reviewing instructions, searching existing data sources gathering and maintaining the data ...rding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington ..., 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Information and Regulatory Affairs, Office of

. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	Final: 19 Oct 1990 to 01 Jun 1993

4. TITLE AND SUBTITLE	5. FUNDING NUMBERS
Alsys, AlsyCOMP_004, Version 5.3, Apollo DN4000, under Domai..os SR10.2 (Host & Target), 901022A1.11044	
6. AUTHOR(S)	
AFNOR , Paris, FRANCE	



7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)	8. PERFORMING ORGANIZATION REPORT NUMBER
AFNOR Tour Europe, Cedex 7 7-92080 Paris La Defense France	AVF-VSR-AFNMOR-90-06
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSORING/MONITORING AGENCY REPORT NUMBER
Ada Joint Program Office United States Department of Defense Washington, D.C. 20301-3081	

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT	12b. DISTRIBUTION CODE
Approved for public release; distribution unlimited.	

13. ABSTRACT *(Maximum 200 words)*

Alsys, AlsyCOMP_004, Version 5.3, Apollo DN4000, Paris la Defense, under Domain/os SR10.2 (Host & Target), ACVC 1.11.

91-17735

14. SUBJECT TERMS	15. NUMBER OF PAGES
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.	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFED	UNCLASSIFIED	

NSN 7540-01-280-550

Standard Form 298, (Rev. 2-89)
Prescribed by ANSI Std. 239-128

Certificate Information

The following Ada implementation was tested and determined to pass ACVC 1.11. Testing was completed on 22 October 1990.

Compiler Name and Version: AlsyCOMP_004 Version 5.3

Host Computer System: Apollo DN4000 under Domain/OS SR10.2

Target Computer System: Apollo DN4000 under Domain/OS SR10.2

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

As a result of this validation effort, Validation Certificate 901022A1.11044 is awarded to Alsys. This certificate expires on 1992-06-01.

This report has been reviewed and is approved.

A Philippe

AFNOR
Philippe Alphonse
Tour Europe
Cedex 7
F-92049 Paris la Défense

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

A-1

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

AVF Control Number: AVF-VSR-AFNOR-90-06

Ada COMPILER
VALIDATION SUMMARY REPORT:
Certificate Number: 901022A1.11044
Alsys
AlsyCOMP_004 Version 5.3
Apollo DN4000

Prepared By:
AFNOR
Tour Europe
Cedex 7
F-92049 Paris la Défense

DECLARATION OF CONFORMANCE

Customer:	Alsys
Certificate Awardee:	Alsys
Ada Validation Facility:	AFNOR
ACVC Version:	1.11
Ada Implementation	
Ada Compiler Name and Version:	AlsyCOMP_004 Version 5.3
Host Computer System:	Apollo DN4000 under Domain/OS SR10.2
Target Computer System:	Apollo DN4000 under Domain/OS SR10.2

Declaration:

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

Etienne Morel
Managing Director
Alsys

90-10-19
Date

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 2-3

CHAPTER 3 PROCESSING INFORMATION

3.1 TESTING ENVIRONMENT 3-1
3.2 SUMMARY OF TEST RESULTS 3-2
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 SPPRT13, and the procedure CHECK_FILE are used for this purpose. The package REPORT also provides a set of identity functions used to defeat some compiler optimizations allowed by the Ada Standard that would circumvent a test objective. The package SPPRT13 is used by many tests for Chapter 13 of the Ada Standard. The procedure CHECK_FILE is used to check the contents of text files written by some of the Class C tests for Chapter 14 of the Ada Standard. The operation of REPORT and CHECK_FILE is checked by a set of executable tests. If these units are not operating correctly, validation testing is discontinued.

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

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

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

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

Term	Definition
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 90-10-12.

E28005C	B28006C	C34006D	B41308B	C43004A	C45114A
C45346A	C45612B	C45651A	C46022A	B49008A	A74006A
B83022B	B83022H	B83025B	B83025D	B83026B	B85001L
C83026A	C83041A	C97116A	C98003B	BA2011A	CB7001A
CB7001B	CB7004A	CC1223A	BC1226A	CC1226B	BC3009B
AD1B08A	BD2A02A	CD2A21E	CD2A23E	CD2A32A	CD2A41A
CD2A41E	CD2A87A	CD2B15C	BD3006A	CD4022A	CD4022D
CD4024B	CD4024C	CD4024D	CD4031A	CD4051D	CD5111A
CD7004C	ED7005D	CD7005E	AD7006A	CD7006E	AD7201A
AD7201E	CD7204B	BD8002A	BD8004C	CD9005A	CD9005B
CDA201E	CE2107I	CE2119B	CE2205B	CE2405A	CE3111C
CE3118A	CE3411B	CE3412B	CE3812A	CE3814A	CE3902B
BD1B02B	BD1B06A	C74308A	BD4008A	CE2117A	CE2117B
CE3607A	CE3607B	CE3607C			

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 Ada Commentaries are included as appropriate.

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

C24113L..Y C35705L..Y C35706L..Y C35707L..Y C35708L..Y C35802L..Z
C45241L..Y C45321L..Y C45421L..Y C45521L..Z C45524L..Z C45621L..Z
C45641L..Y C46012L..Z

The following 21 tests check for 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				

IMPLEMENTATION DEPENDENCIES

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

C35702A C35713B C45423B B86001T C86006H check for predefined type SHORT_FLOAT.

C45531M..P C45532M..P check fixed-point operations for types that require a SYSTEM.MAX_MANTISSA of 47 or greater.

C45536A C46013B C46031B C46033B C46034B contain 'SMALL representation clauses which are not powers of two.

C45624A and C45624B check that the proper exception is raised if MACHINE_OVERFLOWS is FALSE for floating point types; for this implementation, MACHINE_OVERFLOWS 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.

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

CD2A53A checks operations of a fixed-point type for which a length clause specifies a power-of-ten type'small. [See 2.3.]

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

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

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

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

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.

CE3202A expects that function NAME can be applied to the standard input and output files; in this implementation these files have no names, and USE_ERROR is raised. [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.

CE2401H, E2401D and EE2401G use instantiations of DIRECT_IO with unconstrained array and record types; this implementation raises USE_ERROR on the attempt to create a file.

2.3 TEST MODIFICATIONS

Modifications (see section 1.3) were required for 26 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:

B23004A B24007A B24009A B28003A B32202A B32202B B32202C B36307A
B37004A B61012A B62001B B74304B B74304C B74401F B74401R B91004A
B95032A B95069A B95069B BA1101B BC2001D BC3009C

EA3004D was graded passed by Evaluation and Processing Modification as directed by the AVO. The test requires that either pragma INLINE is obeyed for the invocation of a function in each of three contexts and that thus three library units are made obsolete by the re-compilation of the inlined function's body, or else the pragma is ignored completely. This implementation obeys the pragma except when the invocation is within a package specification. When the test's files are processed in the given order, only two units are made obsolete; thus, the expected error at line 27 of file EA3004D6M is not valid and is not flagged. To confirm that indeed the pragma is not obeyed in this one case, the test was also processed with the files re-ordered so that the re-compilation follows only the package declaration (and thus the other library units will not be made obsolete, as they are compiled later); a "NOT APPLICABLE" result wroduced, as expected. The revised order of files was 0-1-4-5-2-3-6.

BA2001E was graded passed by Evaluation Modification as directed by the AVO. The test expects that duplicate names of subunits with a common ancestor will be detected as compilation errors; this implementation detects the errors at link time, and the AVO ruled that this behavior is acceptable.

CD2A53A was graded inapplicable by Evaluation Modification as directed by the AVO. The test contains a specification of power-of-10 value as small for a fixed-point type. The AVO ruled that, under ACVC 1.11, support of decimal smalls may be omitted.

CE3202A was graded inapplicable by Evaluation Modification as directed by the AVO. The test will abort with an unhandled exception (USE_ERROR) when function NAME is invoked for the standard input file. The AVO ruled that this behavior is acceptable pending a resolution of the issue by the ISO WG-9 Ada Rapporteur Group. A modified version of this test was processed so that the raising of USE_ERROR could be confirmed and a "NOT APPLICABLE" result recorded: the two if statements which invoke NAME were encapsulated in blocks with exception handlers for USE_ERROR; these handlers called REPORT.NOT_APPLICABLE.

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 for technical information about this Ada implementation system, see:

Didier De Bie
Alsys SA
29, Avenue Lucien-Rene Duchesne
78170 La Celle Saint-Cloud
France

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

Bob Lamkin
Alsys Inc
67 South Bedford Street
Burlington
MA 01803-5152
U.S.A.

Philippe Loutrel
Alsys SA
29, Avenue Lucien-Rene Duchesne
78170 La Celle Saint-Cloud
France

John Stewart
Alsys Ltd
Partridge House
Newtown Road
Henley-on-Thames
Oxon,RG91EN
U.K.

Kurt Wey
Alsys Gmbh
Am Ruppurer Schoss 7
D-7500 Karlsruhe 51
Germany

Jun Shimura
Alsys-KKE Co. Ltd
Techno-Wave 16F
1-1-25 Shin-Urashima-cho
Kanagawa-Ku
Yokohama
Japan

Orjan Leringe
Alys AB
Patron Pehr Vag 10
Box 1085
141 22 Huddinge
Stockholm
Sweden

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	3802	
b)	Total Number of Withdrawn Tests	81	
c)	Processed Inapplicable Tests	86	
d)	Non-Processed I/O Tests	0	
e)	Non-Processed Floating-Point Precision Tests	201	
f)	Total Number of Inapplicable Tests	287	(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 287 tests were inapplicable to this implementation. All inapplicable tests were processed during validation testing except for 201 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 Data Cartridge Tape containing the customized test suite (see section 1.3) was taken on-site by the validation team for processing. The contents of the Data Cartridge 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.

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.

Test output, compiler and linker listings, and job logs were captured on Data Cartridge Tape 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 in two tables. The first table lists the values that are defined in terms of the maximum input-line length, which is the value for $MAX_IN_LEN--also listed here. These values are expressed here as Ada string aggregates, where "V" represents the maximum input-line length.

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

MACRO PARAMETERS

The following table lists all of the other macro parameters and their respective values.

Macro Parameter	Macro Value
$MAX_IN_LEN	255
ACC_SIZE	32
$ALIGNMENT	2
$COUNT_LAST	2147483647
$DEFAULT_MEM_SIZE	2**32
$DEFAULT_STOR_UNIT	8
$DEFAULT_SYS_NAME	MC680X0
$DELTA_DOC	2#1.0#E-31
$ENTRY_ADDRESS	INTERRUPT_ADDRESS
$ENTRY_ADDRESS1	INTERRUPT_ADDRESS1
$ENTRY_ADDRESS2	INTERRUPT_ADDRESS2
$FIELD_LAST	255
$FILE_TERMINATOR	CHARACTER'VAL(32)
$FIXED_NAME	NO_SUCH_TYPE
$FLOAT_NAME	NO_SUCH_TYPE
$FORM_STRING	""
$FORM_STRING2	"CANNOT_RESTRICT_FILE_CAPACITY"
$GREATER_THAN_DURATION	100_000.0
$GREATER_THAN_DURATION_BASE_LAST	100_000_000.0
$GREATER_THAN_FLOAT_BASE_LAST	2.0E128
$GREATER_THAN_FLOAT_SAFE_LARGE	2#1.11111111111111111111111#E127

$GREATER_THAN_SHORT_FLOAT_SAFE_LARGE 1.0E308

$HIGH_PRIORITY 16

$ILLEGAL_EXTERNAL_FILE_NAME1 /~/*/f1

$ILLEGAL_EXTERNAL_FILE_NAME2 /~/*/f2

$INAPPROPRIATE_LINE_LENGTH -1

$INAPPROPRIATE_PAGE_LENGTH -1

$INCLUDE_PRAGMA1 PRAGMA INCLUDE ("A28006D1.TST")

$INCLUDE_PRAGMA2 PRAGMA INCLUDE ("B28006D1.TST")

$INTEGER_FIRST -2147483648

$INTEGER_LAST 2147483647

$INTEGER_LAST_PLUS_1 2_147_483_648

$INTERFACE_LANGUAGE C

$LESS_THAN_DURATION -100_000.0

$LESS_THAN_DURATION_BASE_FIRST -100_000_000.0

$LINE_TERMINATOR ASCII.LF

$LOW_PRIORITY 1

$MACHINE_CODE_STATEMENT NULL;

$MACHINE_CODE_TYPE NO_SUCH_TYPE

$MANTISSA_DOC 31

$MAX_DIGITS 15

$MAX_INT 2147483647

$MAX_INT_PLUS_1 2_147_483_648

$MIN_INT -2147483648

$NAME	SHORT_SHORT_INTEGER
$NAME_LIST	MC680X0
$NAME_SPECIFICATION1	/tmp/X2120A
$NAME_SPECIFICATION2	/tmp/X2120B
$NAME_SPECIFICATION3	/tmp/X3119A
$NEG_BASED_INT	16#FFFFFFFE#
$NEW_MEM_SIZE	2**32
$NEW_STOR_UNIT	8
$NEW_SYS_NAME	MC680X0
$PAGE_TERMINATOR	ASCII.FF
$RECORD_DEFINITION	new INTEGER;
$RECORD_NAME	NO_SUCH_MACHINE_CODE_TYPE
$TASK_SIZE	32
$TASK_STORAGE_SIZE	4096
$TICK	0.02
$VARIABLE_ADDRESS	OBJECT_ADDRESS
$VARIABLE_ADDRESS1	OBJECT_ADDRESS1
$VARIABLE_ADDRESS2	OBJECT_ADDRESS2
$YOUR_PRAGMA	INTERFACE_NAME

APPENDIX B

OPTIONS

The 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.

COMPILATION SYSTEM OPTIONS

ERRORS=999: allow 999 errors before terminating compilation (default 50).
CALLS=INLINED: allow inline insertion of code for subprograms and take pragma INLINE into account. (default NORMAL).
REDUCTION=PARTIAL: perform some high level optimizations on checks and loops. (default NONE).
EXPRESSIONS=PARTIAL: perform some low level optimisations on common subexpressions and register allocations (default NONE).
FLOAT=MC68881: Floating point operations use the MC6888x arithmetic processor (default SOFTWARE).

NOWARNING: Do not generate warning messages
NODETAIL: Do not include extra detail in error messages
SHOW=NONE: No banner header on listing pages, no error summary at end of listing.
FILEWIDTH=120: Listing file has 120 characters per line
NOFILELENGTH: Unpaginated listing file

For tests rejected at compile time:
TEXT: compilation listing including full source text (with embedded error messages)

For tests compiled without errors:
NOTEXT: compilation listing including only source text for lines containg errors
(i.e. empty listing if no errors)

LINKER OPTIONS

FLOAT=MC68881: Floating point operations use the MC6888x arithmetic processor (default SOFTWARE).
NOWARNING: Do not generate warning messages
FILEWIDTH=80: Listing file has 80 characters per line
NOFILELENGTH:aginated listing file

APPENDIX C

APPENDIX F OF THE Ada STANDARD

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

```
package STANDARD is

    type SHORT_SHORT_INTEGER   is range  -2**7..2**7-1;
    type SHORT_INTEGER         is range -2**15..2**15-1;
    type INTEGER               is range -2**31..2**31-1;

    type FLOAT is digits 6 range
       -(2.0 - 2.0**(-23)) * 2.0**127..
       +(2.0 - 2.0**(-23)) * 2.0**127;

   type LONG_FLOAT is digits 15 range
       -(2.0 - 2.0**(-52)) * 2.0**1023..
       +(2.0 - 2.0**(-52)) * 2.0**1023;

   type DURATION is delta 2.0**(-14) range -86_400.00..86_400.00;

end STANDARD;
```

Alsys Ada Compiler

APPENDIX F

for MOTOROLA 68K based Workstations under UNIX

Version 5.3

Alsys S.A.
29, Avenue Lucien-René Duchesne
78170 La Celle St. Cloud, France

Alsys Inc.
67 South Bedford Street
Burlington, MA 01803-5152, U.S.A.

Alsys Ltd
Partridge House, Newtown Road
Henley-on-Thames,
Oxfordshire RG9 1EN, U.K.

Alsys AB
Patron Pehr Väg 10
Box 1085
141 22 Huddinge, Stockholm, Sweden

Copyright 1990 by Alsys

All rights reserved. No part of this document may be reproduced in any form or by any means without permission in writing from Alsys.

Printed: September 25, 1990

Alsys reserves the right to make changes in specifications and other information contained in this publication without prior notice. Consult Alsys to determine whether such changes have been made.

HP-UX is a trademark of Hewlett Packard
Sun Workstation is a registered trademark of Sun Microsystems, Inc.
UNIX is a registered trademark of AT&T BELL LABORATORIES.
Apollo and DOMAIN are registered trademarks of Apollo Computer Inc.
AEGIS and DOMAIN/IX are registered trademarks of Apollo Computer Inc.
Macintosh, Mac, and A/UX are trademarks of Apple Computer, Inc.
Cetia, Unigraph and Unigraph/X are registered trademarks of Cetia.
Alsys, AdaWorld, AdaProbe, AdaTune, AdaXref, AdaReformat, and AdaMake are registered trademarks of Alsys.

TABLE OF CONTENTS

PREFACE		v
1	**IMPLEMENTATION-DEPENDENT PRAGMAS**	**7**
1.1	The Pragma INTERFACE	7
1.2	The Pragma INTERFACE_NAME	7
1.3	The Pragma INLINE	8
1.4	The Pragma EXPORT	8
1.5	The Pragma EXTERNAL_NAME	9
1.6	The Pragma INDENT	9
1.7	The Pragma IMPROVE	10
1.8	The other Pragmas	10
1.9	Pragmas with no Effect	10
2	**IMPLEMENTATION-DEPENDENT ATTRIBUTES**	**11**
2.1	Attributes used in Record Representation Clauses	12
2.2	Limitations on the use of the Attribute ADDRESS	12
2.3	The Attribute IMPORT	12
3	**THE PACKAGE SYSTEM**	**13**
4	**TYPE REPRESENTATION CLAUSES**	**18**
4.1	Enumeration Types	18
4.2	Integer Types	19
4.3	Floating Point Types	20
4.4	Fixed Point Types	21
4.5	Access Types	22
4.6	Task Types	23
4.7	Array Types	23
4.8	Record Types	24

5	**IMPLEMENTATION-DEPENDENT COMPONENTS**	**27**
6	**ADDRESS CLAUSES**	**28**
6.1	Address Clauses for Objects	28
6.2	Address Clauses for Program Units	28
6.3	Address Clauses for Entries	28
7	**UNCHECKED CONVERSIONS**	**29**
8	**INPUT-OUTPUT CHARACTERISTICS**	**30**
8.1	Introduction	30
8	The Parameter FORM	31

PREFACE

This is the "Appendix F, Implementation-Dependent Characteristics" of the Reference Manual for the Ada Programming Language, ISO/8652-1987.

CHAPTER 1

IMPLEMENTATION-DEPENDENT PRAGMAS

1.1 The Pragma INTERFACE

Programs written in Ada can interface with external subprograms written in another language, by use of the pragma INTERFACE. The format of the pragma is:

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

The *language_name* may be Assembler, C, Fortran or Pascal depending on the Alsys Ada compiler used (see *Application Developer's Guide*).

The *Ada_subprogram_name* is the name by which the subprogram is known in Ada.

Interfacing the Ada language with other languages is detailed in the *Application Developer's Guide*.

1.2 The Pragma INTERFACE_NAME

To name the external subprogram to which an Ada subprogram is interfaced, as defined in the other language, may require the use of non-Ada naming conventions, such as special characters, or case sensitivity. For this purpose the implementation-dependent pragma INTERFACE_NAME may be used in conjunction with the pragma INTERFACE.

pragma INTERFACE_NAME (*Ada_subprogram_name*, *name_string*);

The *name_string* is a string, which denotes the name of the external subprogram as defined in the other language. The *Ada_subprogram_name* is the name by which the subprogram is known in Ada.

The pragma INTERFACE_NAME may be used anywhere in an Ada program where INTERFACE is allowed (see [13.9]). It must occur after the corresponding pragma INTERFACE and within the same declarative part or package specification.

1.3 The Pragma INLINE

Pragma INLINE is fully supported; however, it is not possible to inline a subprogram in a declarative part.

Note that inlining facilities are also provided by use of the command COMPILE with the option IMPROVE (see the *User's Guide*).

1.4 The Pragma EXPORT

The pragma EXPORT takes a language name and an Ada identifier as arguments. This pragma allows an object defined in Ada to be visible to external programs written in the specified language.

pragma EXPORT (*language_name, Ada_identifier*)

Example:

```
package MY_PACKAGE is

  THIS_OBJECT : INTEGER;
  pragma EXPORT (PASCAL, THIS_OBJECT);
  ......
end MY_PACKAGE;
```

Limitations on the use of pragma EXPORT

- This pragma must occur in a declarative part and applies only to objects declared in this same declarative part, that is, generic instantiated objects or renamed objects are excluded.
- The pragma is only to be used for objects with direct allocation mode, which are declared in a library package. The ALSYS implementation gives indirect allocation mode to dynamic objects, and objects that have a significant size (see Section 2.1 of the *Application Developer's Guide*).

1.5 The Pragma EXTERNAL_NAME

To name an exported Ada object as it is identified in the other language may require the use of non-Ada naming conventions, such as special characters, or case sensitivity. For this purpose the implementation-dependent pragma EXTERNAL_NAME may be used in conjunction with the pragma EXPORT:

> **pragma** EXTERNAL_NAME (*Ada_identifier, name_string*);

The *name_string* is a string which denotes the name of the identifier defined in the other language. The *Ada_identifier* denotes the exported Ada object.

The pragma EXTERNAL_NAME may be used anywhere in an Ada program where pragma EXPORT is allowed. It must occur after the corresponding pragma EXPORT and within the same library package.

Example:

```
package MY_PACKAGE is

    THIS_OBJECT : INTEGER;
    pragma EXPORT (PASCAL, THIS_OBJECT);
    pragma EXTERNAL_NAME (THIS_OBJECT, "ThisObject");
    ......

end MY_PACKAGE;
```

1.6 The Pragma INDENT

This pragma is only used by AdaReformat. This tool offers the functionalities of a pretty-printer in an Ada environment.

The pragma is placed in the source file and interpreted by AdaReformat.

> **pragma** INDENT(OFF) causes AdaReformat not to modify the source lines after this pragma.

> **pragma** INDENT(ON) causes AdaReformat to resume its action after this pragma.

1.7 The Pragma IMPROVE

This pragma is used to suppress implicit components from a record type.

pragma IMPROVE (TIME | SPACE, [ON =>] simple_name);

See Section 4 8 for the complete description.

1.8 The other Pragmas

Pragma PACK is discussed in detail in the section on representation clauses and records (Chapter 4).

Pragma PRIORITY is accepted with the range of priorities running from 1 to 16 (see the definition of the predefined package SYSTEM in Section 3). Undefined priority (no pragma PRIORITY) is treated as though it were less than every defined priority value.

In addition to pragma SUPPRESS, it is possible to suppress all checks in a given compilation by the use of the Compiler option CHECKS. (See Chapter 4 of the *User's Guide*.)

1.9 Pragmas with no Effect

The following pragmas have no effect:

CONTROLLED
MEMORY_SIZE
STORAGE_UNIT
SYSTEM_NAME
OPTIMIZE

For optimization, certain facilities are provided through use of the command COMPILE with the option IMPROVE (see the *User's Guide*).

CHAPTER 2

IMPLEMENTATION-DEPENDENT ATTRIBUTES

Throughout this chapter and the remaining chapters of this document three special types of integer are used in the text. They are used where the number of bits used to store the integer is important.

The three types used are defined as:

INTEGER_8; an integer stored in 8 bits,

INTEGER_16; an integer stored in 16 bits,

INTEGER_32; an integer stored in 32 bits.

and can be respectively declared, with representation clauses, thus:

```
type INTEGER_8 is new INTEGER range -2**7 .. 2**7 -1;
for INTEGER_8'SIZE use 8;

type INTEGER_16 is new INTEGER range -2**15 .. 2**15 -1;
for INTEGER_16'SIZE use 16;

type INTEGER_32 is new INTEGER range -2**31 .. 2**31 -1;
for INTEGER_32'SIZE use 32;
```

The user gains complete control over the data storage by using these forms of declaration, as opposed to those defined in package STANDARD over which the user has no control. (Refer to Chapter 3 of this document.)

2.1 Attributes used in Record Representation Clauses

In addition to the Representation Attributes of [13.7.2] and [13.7.3], the following five attributes are used to form names of indirect and implicit components for use in record representation clauses, as described in Section 4.8.

'OFFSET
'RECORD_SIZE
'VARIANT_INDEX
'ARRAY_DESCRIPTOR
'RECORD_DESCRIPTOR

2.2 Limitations on the use of the Attribute ADDRESS

The attribute ADDRESS is implemented for all prefixes that have meaningful addresses.

Note: The value returned by the attribute ADDRESS is undefined before the elaboration of the subprogram body (when 'ADDRESS is applied to a subprogram).

The following entities do not have meaningful addresses and will therefore cause a compilation error if used as prefix to ADDRESS:

- A constant that is implemented as an immediate value, i.e., does not have any space allocated for it
- A package specification that is not a library unit
- A package body that is not a library unit or subunit
- A function that renames an enumeration literal.

2.3 The Attribute IMPORT

This attribute is a function which takes two literal strings as arguments; the first one denotes a language name and the second one denotes an external symbol name. It yields the address of this external symbol. The prefix of this attribute must be SYSTEM.ADDRESS. The value of this attribute is of the type SYSTEM.ADDRESS. The syntax is:

SYSTEM.ADDRESS'IMPORT ("*Language_name*", "*external_symbol_name*")

CHAPTER 3

THE PACKAGE SYSTEM

This section contains information on the visible part of the specification of the package SYSTEM

```
package SYSTEM is

-- Standard Ada definitions

   type NAME is (MC680X0) ;
    SYSTEM_NAME      : constant NAME := MC680X0;
    STORAGE_UNIT     : constant := 8 ;
    MEMORY_SIZE      : constant := 2**32 ;
    MIN_INT          : constant := -(2**31) ;
    MAX_INT          : constant := 2**31-1 ;
    MAX_DIGITS       : constant := 15 ;
    MAX_MANTISSA     : constant := 31 ;
    FINE_DELTA       : constant := 2#1.0#e-31 ;
    TICK             : constant := 0.02 ;

   type ADDRESS is private;
   NULL_ADDRESS : constant ADDRESS;

   subtype PRIORITY is INTEGER range 1..16;
```

```
-- Address operations

   function VALUE (LEFT : in STRING) return ADDRESS;
   -----------------------------------------------------------
   -- Converts a string into an address.
   -- The string can represent either an unsigned address ie.
   -- "16#XXXXXXXX#" where XXXXXXXX is in the range
   -- 0..FFFFFFFF, or a signed address ie.
   -- "-16#XXXXXXXX#" where XXXXXXXX is in the range
   -- 0..7FFFFFFF.
   -- A CONSTRAINT_ERROR is raised if the string does not conform to
   -- this syntax
   -----------------------------------------------------------

   ADDRESS_WIDTH : constant := 3 + 8 + 1;
   subtype ADDRESS_STRING is STRING(1..ADDRESS_WIDTH);

   function IMAGE( LEFT : in ADDRESS ) return ADDRESS_STRING;
   -----------------------------------------------------------
   -- Converts an address to a string. The syntax of the returned string is
   -- described in the VALUE function above. Refer to the unsigned
   -- representation.
   -----------------------------------------------------------

   type OFFSET is range -2**31 .. 2**31 -1;
   -- This type is used to measure a number of storage units (bytes).
   -- The type is an Ada integer type.

   function SAME_SEGMENT (LEFT, RIGHT : in ADDRESS)
         return BOOLEAN;
   ---------------------------------------------------------------- On a segmented architecture
   the function returns TRUE if the two
   -- addresses have the same segment value. On a non-segmented
   -- architecture it always returns TRUE.------------------------------------------------------
   --------
```

```
ADDRESS_ERROR : exception;
--------------------------------------------------------------
-- This exception is raised by "<", "<=", ">", ">=" and "-" if the two
-- addresses do not have the same segment value. This exception is
-- never raised on a non-segmented machine.
-- The exception CONSTRAINT_ERROR can be raised by "+" and "-".
--------------------------------------------------------------

function "+" (LEFT : in OFFSET; RIGHT : in ADDRESS)
      return ADDRESS;
function "+" (LEFT : in ADDRESS; RIGHT : in OFFSET)
return ADDRESS;function "-" (LEFT : in ADDRESS; RIGHT : in OFFSET)
return ADDRESS;--------------------------------------------------------------
-- These routines provide support to perform address computations. The
-- meaning of the "+" and "-" operators is architecture dependent. For
-- example on a segmented machine the OFFSET parameter is added to,
-- or subtracted from the offset part of the address, the segment
-- remaining unaltered.
--------------------------------------------------------------

function "-" (LEFT : in ADDRESS; RIGHT : in ADDRESS)
      return OFFSET;
--------------------------------------------------------------
-- Returns the distance between the given addresses. The result is
-- signed. The exception ADDRESS_ERROR is raised on a segmented
-- architecture if the two addresses do not have the same segment value.
--------------------------------------------------------------

function "<"  (LEFT, RIGHT : in ADDRESS)
      return BOOLEAN;
function "<=" (LEFT , RIGHT : in ADDRESS)
      return BOOLEAN;
function ">"  (LEFT , RIGHT : in ADDRESS)
      return BOOLEAN;
```

```
function ">=" (LEFT , RIGHT : in ADDRESS)
      return BOOLEAN;
-------------------------------------------------------------
-- Perform a comparison on addresses, or on the offset part of addresses
-- for a segmented machine. The comparison is unsigned on all
-- machines except the Transputer.
-------------------------------------------------------------

function "mod" (LEFT : in ADDRESS; RIGHT : in POSITIVE)
      return NATURAL;
-------------------------------------------------------------
-- Returns the offset of LEFT relative to the memory block immediately
-- below it starting at a multiple of RIGHT storage units. On a
-- segmented machine, the segment part is ignored.
-------------------------------------------------------------

type ROUND_DIRECTION is (DOWN, UP);

function ROUND (   VALUE : in ADDRESS;
                   DIRECTION : in ROUND_DIRECTION;
                   MODULUS : in POSITIVE ) return ADDRESS;
-------------------------------------------------------------
-- Returns the given address rounded to a specific value.
-------------------------------------------------------------

generic
      type TARGET is private;
function FETCH_FROM_ADDRESS ( A : in ADDRESS)
       return TARGET;
generic
      type TARGET is private;
procedure ASSIGN_TO_ADDRESS ( A : in ADDRESS;
       T : in TARGET);
-------------------------------------------------------------
-- These routines are provided to perform READ/WRITE operations in
-- memory. These routines may give unexpected results if used with
-- unconstrained types.
-------------------------------------------------------------
```

```
type OBJECT_LENGTH is range 0 .. 2**31 - 1;
-- This type is used to designate the size of an object in storage units.

procedure MOVE (    TO : in ADDRESS;
                    FROM : in ADDRESS;
                    LENGTH : in OBJECT_LENGTH );
-------------------------------------------------------------------
-- Copies LENGTH storage units starting at the address FROM to the
-- address TO. The source and destination may overlap.
-- Use of this procedure with optimizers may lead to unexpected
-- results.
-------------------------------------------------------------------

private

    -- private part of the system

end SYSTEM;
```

CHAPTER 4

TYPE REPRESENTATION CLAUSES

The aim of this section is to explain how objects are represented and allocated by the Alsys Ada compiler for MC680X0 machines and how it is possible to control this using representation clauses.

The representation of an object is closely connected with its type. For this reason this section addresses successively the representation of enumeration, integer, floating point, fixed point, access, task, array and record types. For each class of type the representation of the corresponding objects is described.

Except in the case of array and record types, the description for each class of type is independent of the others. To understand the representation of an array type it is necessary to understand first the representation of its components. The same rule applies to record types.

Apart from implementation defined pragmas, Ada provides three means to control the size of objects:

- a (predefined) pragma PACK, when the object is an array, an array component, a record or a record component
- a record representation clause, when the object is a record or a record component
- a size specification, in any case.

For each class of types the effect of a size specification alone is described. Interference between size specifications, packing and record representation clauses is described under array and record types.

4.1 Enumeration Types

Internal codes of enumeration literals

When no enumeration representation clause applies to an enumeration type, the internal code associated with an enumeration literal is the position number of the

enumeration literal. Then, for an enumeration type with n elements, the internal codes are the integers 0, 1, 2, .. , n-1.

An enumeration representation clause can be provided to specify the value of each internal code as described in [13.3]. The Alsys compiler fully implements enumeration representation clauses.

As internal codes must be machine integers the internal codes provided by an enumeration representation clause must be in the range $-2^{31} .. 2^{31}-1$.

Encoding of enumeration values

An enumeration value is always represented by its internal code in the program generated by the compiler.

When an enumeration type is not a boolean type c s a boolean type with an enumeration representation clause, binary code is used to represent internal codes. Negative codes are then represented using two's complement.

When a boolean type has no enumeration representation clause, the internal code 0 is represented by a succession of 0s and the internal code 1 is represented by a succession of 1s. The length of this pattern of 0s or of 1s is the size of the boolean value.

4.2 Integer Types

Predefined integer types

There are three predefined integer types in the Alsys implementation for MC680X0 machines:

```
type SHORT_SHORT_INTEGER  is range -2**7 .. 2**7 -1;
type SHORT_INTEGER        is range -2**15 .. 2**15 -1;
type INTEGER              is range -2**31 .. 2**31 -1;
```

Selection of the parent of an integer type

An integer type declared by a declaration of the form:

```
type T is range L .. R;
```

is implicitly derived from a predefined integer type. The compiler automatically selects the predefined integer type whose range is the shortest that contains the values L to R inclusive.

Encoding of integer values

Binary code is used to represent integer values. Negative numbers are represented using two's complement.

4.3 Floating Point Types

Predefined floating point types

There are two predefined floating point types in the Alsys implementation for MC680X0 machines:

```
type FLOAT is
    digits 6 range -(2.0 - 2.0**(-23))*2.0**127 .. (2.0 - 2.0**(-23))*2.0**127;

type LONG_FLOAT is
    digits 15 range -(2.0 - 2.0**(-51))*2.0**1023 .. (2.0 - 2.0**(-51))*2.0**1023;
```

Selection of the parent of a floating point type

A floating point type declared by a declaration of the form:

```
type T is digits D [range L .. R];
```

is implicitly derived from a predefined floating point type. The compiler automatically selects the smallest predefined floating point type whose number of digits is greater than or equal to D and which contains the values L to R inclusive.

Encoding of floating point values

In the program generated by the compiler, floating point values are represented using the IEEE standard formats for single and double floats.

The values of the predefined type FLOAT are represented using the single float format. The values of the predefined type LONG_FLOAT are represented using the double float format. The values of any other floating point type are represented in the same way as the values of the predefined type from which it derives, directly or indirectly.

4.4 Fixed Point Types

Small of a fixed point type

If no specification of small applies to a fixed point type, then the value of small is determined by the value of delta as defined by [3.5.9].

A specification of small can be used to impose a value of small. The value of small is required to be a power of two.

Predefined fixed point types

To implement fixed point types, the Alsys compiler for MC680X0 machines uses a set of anonymous predefined types of the form:

> **type** *FIXED_8* **is delta** *D* **range** *(-2**07-1)*S .. 2**07*S;*
> **for** *FIXED_8'SMALL* **use** *S;*
>
> **type** *FIXED_16* **is delta** *D* **range** *(-2**15-1)*S .. 2**15*S;*
> **for** *FIXED_16'SMALL* **use** *S;*
>
> **type** *FIXED_32* **is delta** *D* **range** *(-2**31-1)*S .. 2**31*S;*
> **for** *FIXED_32'SMALL* **use** *S;*

where D is any real value and S any power of two less than or equal to D.

Encoding of fixed point values

In the program generated by the compiler, a safe value V of a fixed point subtype F is represented as the integer:

V / F'BASE'SMALL

4.5 Access Types

Collection Size

When no specification of collection size applies to an access type, no storage space is reserved for its collection, and the value of the attribute STORAGE_SIZE is then 0.

As described in [13.2], a specification of collection size can be provided in order to reserve storage space for the collection of an access type. The Alsys compiler fully implements this kind of specification.

Encoding of access values.

Access values are machine addresses.

4.6 Task Types

Storage for a task activation

When no length clause is used to specify the storage space to be reserved for a task activation, the storage space indicated at bind time is used for this activation.

As described in [13.2], a length clause can be used to specify the storage space for the activation of each of the tasks of a given type. In this case the value indicated at bind time is ignored for this task type, and the length clause is obeyed.

Encoding of task values.

Encoding of a task value is not described here.

4.7 Array Types

Size of an array subtype

The size of an array subtype is obtained by multiplying the number of its components by the sum of the size of the components and the size of the gaps (if any). If the subtype is unconstrained, the maximum number of components is considered.

The size of an array subtype cannot be computed at compile time

- if it has non-static constraints or is an unconstrained array type with non-static index subtypes (because the number of components can then only be determined at run time).
- if the components are records or arrays and their constraints or the constraints of their subcomponents (if any) are not static (because the size of the components and the size of the gaps can then only be determined at run time).

As has been indicated above, the effect of a pragma PACK on an array type is to suppress the gaps and to reduce the size of the components. The consequence of packing an array type is thus to reduce its size.

If the components of an array are records or arrays and their constraints or the constraints of their subcomponents (if any) are not static, the compiler ignores any

pragma PACK applied to the array type but issues a warning message. Apart from this limitation, array packing is fully implemented by the Alsys compiler.

A size specification applied to an array type or first named subtype has no effect. The only size that can be specified using such a length clause is its usual size. Nevertheless, such a length clause can be useful to verify that the layout of an array is as expected by the application.

4.8 Record Types

Implicit components

In some circumstances, access to an object of a record type or to its components involves computing information which only depends on the discriminant values. To avoid useless recomputation the compiler stores this information in the record objects, updates it when the values of the discriminants are modified and uses it when the objects or its components are accessed. This information is stored in special components called implicit components.

An implicit component may contain information which is used when the record object or several of its components are accessed. In this case the component will be included in any record object (the implicit component is considered to be declared before any variant part in the record type declaration). There can be two components of this kind; one is called RECORD_SIZE and the other VARIANT_INDEX.

On the other hand an implicit component may be used to access a given record component. In that case the implicit component exists whenever the record component exists (the implicit component is considered to be declared at the same place as the record component). Components of this kind are called ARRAY_DESCRIPTORs or RECORD_DESCRIPTORs.

- *RECORD_SIZE*

This implicit component is created by the compiler when the record type has a variant part and its discriminants are defaulted. It contains the size of the storage space necessary to store the current value of the record object (note that the storage effectively allocated for the record object may be more than this).

The value of a RECORD_SIZE component may denote a number of bits or a number of storage units. In general it denotes a number of storage units, but if any component

clause specifies that a component of the record type has an offset or a size which cannot be expressed using storage units, then the value designates a number of bits.

The implicit component RECORD_SIZE must be large enough to store the maximum size of any value of the record type. The compiler evaluates an upper bound MS of this size and then considers the implicit component as having an anonymous integer type whose range is 0 .. MS.

If R is the name of the record type, this implicit component can be denoted in a component clause by the implementation generated name R'RECORD_SIZE.

- *VARIANT_INDEX*

This implicit comp‿nent is created by the compiler when the record type has a variant part. It indicates the set of components that are present in a record value. It is used when a discriminant check is to be done.

- *ARRAY_DESCRIPTOR*

An implicit component of this kind is associated by the compiler with each record component whose subtype is an anonymous array subtype that depends on a discriminant of the record. It contains information about the component subtype.

The structure of an implicit component of kind ARRAY_DESCRIPTOR is not described in this documentation. Nevertheless, if a programmer is interested in specifying the location of a component of this kind using a component clause, he can obtain the size of the component using the ASSEMBLY parameter in the COMPILE command.

The compiler treats an implicit component of the kind ARRAY_DESCRIPTOR as having an anonymous array type. If C is the name of the record component whose ubtype is described by the array descriptor, then this implicit component can be denoted in a component clause by the implementation generated name C'ARRAY_DESCRIPTOR.

- *RECORD_DESCRIPTOR*

An implicit component of this kind is associated by the compiler with each record component whose subtype is an anonymous record subtype that depends on a discriminant of the record. It contains information about the component subtype.

The structure of an implicit component of kind RECORD_DESCRIPTOR is not described in this documentation. Nevertheless, if a programmer is interested in

specifying the location of a component of this kind using a component clause, he can obtain the size of the component using the ASSEMBLY parameter in the COMPILE command.

The compiler treats an implicit component of the kind RECORD_DESCRIPTOR as having an anonymous array type. if C is the name of the record component whose subtype is described by the record descriptor, then this implicit component can be denoted in a component clause by the implementation generated name C'RECORD_DESCRIPTOR.

Size of a record subtype

Unless a component clause specifies that a component of a record type has an offset or a size which cannot be expressed using storage units, the size of a record subtype is rounded up to the a whole number of storage units.

The size of a constrained record subtype is obtained by adding the sizes of its components and the sizes of its gaps (if any). This size is not computed at compile time

- when the record subtype has non-static constraints,
- when a component is an array or a record and its size is not computed at compile time.

The size of an unconstrained record subtype is obtained by adding the sizes of the components and the sizes of the gaps (if any) of its largest variant. If the size of a component or of a gap cannot be evaluated exactly at compile time an upper bound of this size is used by the compiler to compute the subtype size.

A size specification applied to a record type or first named subtype has no effect. The only size that can be specified using such a length clause is its usual size. Nevertheless, such a length clause can be useful to verify that the layout of a record is as expected by the application.

Alignment of a record subtype

When no record representation clause applies to its base type, a record subtype is even byte aligned if it contains a component whose subtype is even byte aligned. Otherwise the record subtype is byte aligned.
When a record representation clause that does not contain an alignment clause applies to its base type, a record subtype is even byte aligned if it contains a component whose

subtype is even byte aligned and whose offset is a multiple of 16 bits. Otherwise the record subtype is byte aligned.

When a record representation clause that contains an alignment clause applies to its base type, a record subtype has an alignment that obeys the alignment clause. An alignment clause can specify that a record type is byte aligned or even byte aligned.

CHAPTER 5

IMPLEMENTATION-DEPENDENT COMPONENTS

The following forms of implementation-generated names [13.4(8)] are used to denote implementation-dependent record components, as described in Section 4.8 in the paragraph on indirect and implicit components:

C'OFFSET
R'RECORD_SIZE
R'VARIANT_INDEX
R'ARRAY_DESCRIPTORs
R'RECORD_DESCRIPTORs

where C is the name of a record component and R the name of a record type.

CHAPTER 6

ADDRESS CLAUSES

An address clause can be used to specify the address of an object, a program unit or an entry.

6.1 Address Clauses for Objects

An address clause can be used to specify an address for an object as described in [13.5]. When such a clause applies to an object no storage is allocated for it in the program generated by the compiler. The program accesses the object by using the address specified in the clause.

An address clause is not allowed for task objects, for unconstrained records whose size is greater than 8 kb, or for a constant.

Note that the function SYSTEM.VALUE, defined in the package SYSTEM, is available to convert a STRING value into a value of type SYSTEM.ADDRESS, also, the IMPORT attribute is available to provide the address of an external symbol. *(Refer to Chapter* 3 and section 2.3)

6.2 Address Clauses for Program Units

Address clauses for program units are not implemented in the current version of the compiler.

6.3 Address Clauses for Entries

An address clause may be used to associate an entry with a UNIX signal. (See *Application Developer's Guide* for detailed information.)

CHAPTER 7

UNCHECKED CONVERSIONS

Unchecked type conversions are described in [13.10.2]. The following restrictions apply to their use.

Unconstrained arrays are not allowed as target types. Unconstrained record types without defaulted discriminants are not allowed as target types. Access types to unconstrained arrays are not allowed as target or source types. Note also that UNCHECKED_CONVERSION cannot be used for an access to an unconstrained string.

However, if the source and the target types are each scalar or access types, the sizes of the objects of the source and target types must be equal.

If a composite type is used either as source type or as target type this restriction on the size does not apply.

If the source and the target types are each of scalar or access type or if they are both of composite type, the effect of the function is to return the operand.

In other cases the effect of unchecked conversion can be considered as a copy:

- If an unchecked conversion is achieved of a scalar or access source type to a composite target type, the result of the function is a copy of the source operand. The result has the size of the source.
- If an unchecked conversion is achieved of a composite source type to a scalar or access target type, the result of the function is a copy of the source operand. The result has the size of the target.

CHAPTER 8

INPUT-OUTPUT CHARACTERISTICS

In this part of the Appendix the implementation-specific aspects of the input-output system are described.

8.1 Introduction

In Ada, input-output operations are considered to be performed on *objects* of a certain file type rather than being performed directly on external files. An external file is anything external to the program that can produce a value to be read or receive a value to be written. Values transferred for a given file must be all of one type.

Generally, in Ada documentation, the term *file* refers to an object of a certain file type, whereas a physical manifestation is known as an *external file*. An external file is characterized by

- its NAME, which is a string defining a legal path name under the current version of the operating system
- its FORM, which gives implementation-dependent information on file characteristics.

Both the NAME and the FORM appear explicitly as parameters of the Ada procedures CREATE and OPEN. Though a file is an object of a certain file type, ultimately the object has to correspond to an external file. Both CREATE and OPEN associate a NAME of an external file (of a certain FORM) with a program file object.

Ada input-output operations are provided by means of standard packages ([14]):

SEQUENTIAL_IO A generic package for sequential files of a single element type.

DIRECT_IO A generic package for direct (random) access files.

TEXT_IO A generic package for human-readable files (text, ASCII).

(Please note that trying to apply TEXT_IO.NAME or TEXT_IO.FORM to STANDARD_INPUT or STANDARD_OUTPUT will raise USE_ERROR. Though it may surprise the user, [14.4(5)] allows this behavior.)

IO_EXCEPTIONS A package which defines the exceptions needed by the above three packages.

The generic package LOW_LEVEL_IO is not implemented in this version.

The upper bound for index values in DIRECT_IO and for line, column and page numbers in TEXT_IO is given by

COUNT'LAST = 2**31 -1

The upper bound for field widths in TEXT_IO is given by

FIELD'LAST = 255

8.2 The Parameter FORM

The parameter FORM of both the procedures CREATE and OPEN in Ada specifies the characteristics of the external file involved.

The procedure CREATE establishes a new external file, of a given NAME and FORM, and associates it with a specified program file object. The external file is created (and the file object set) with a specified (or default) file mode. If the external file already exists, the file will be erased. The exception USE_ERROR is raised if the file mode is IN_FILE.

Example:

```
CREATE(F, OUT_FILE, NAME => "MY_FILE",
              FORM =>
              "WORLD => READ, OWNER => READ_WRITE");
```

The procedure OPEN associates an existing external file, of a given NAME and FORM, with a specified program file object. The procedure also sets the current file mode. If there is an inadmissible change of mode, then the exception USE_ERROR is raised.

The parameter FORM is a string, formed from a list of attributes, with attributes separated by commas. The string is not case sensitive (so that, for example, *HERE* and *here* are treated alike). (FORM attributes are distinct from Ada attributes.) The attributes specify:

- File protection
- File sharing
- File structure
- Buffering
- Appending
- Blocking
- Terminal input

The general form of each attribute is a keyword followed by => and then a qualifier. The arrow and qualifier may sometimes be omitted. The format for an attribute specifier is thus either of

KEYWORD

KEYWORD => QUALIFIER