AD-A224 553

REPORT DOCUMENTATION PAGE

Form Approved
OPM No. 0704-0188

1. AGENCY USE ONLY *(Leave Blank)*

DTIC FILE COPY

2. REPORT DATE

3. REPORT TYPE AND DATES COVERED

Final 09 Jan 1990 to 09 Jan 1991

4. TITLE AND SUBTITLE Ada Compiler Validation Summary Report: STERIA, Ent_Alsy_01, Version 4.2, CETIA UNIGRAPH 6000 (Host) to Motorola MC68020 in PMF (Target), 900109A1.10239

5. FUNDING NUMBERS

(2)

6. AUTHOR(S)

AFNOR, Paris, FRANCE

DTIC ELECTE JUL 25 1990 S D D

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

AFNOR
Tour Europe, Cedex 7
F-92080 Paris la Defense
FRANCE

8. PERFORMING ORGANIZATION REPORT NUMBER

AVF-VSR-AFNOR-89-16

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Ada Joint Program Office
United States Department of Defense
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)*

STERIA, Ent_Alsy_01, Version 4.2, Paris, France, CETIA UNIGRAPH 6000 under Unigraph/X, release 3.0.4 (Host) to Motorola MC68020 in PMF with ARTK, Version 4.2 (bare machine), (Target), ACVC 1.10.

14. SUBJECT TERMS Ada programming language, Ada Compiler Validation Summary Report, Ada Compiler Validation Capability, Validation Testing, Ada Validation Office, Ada Validation Facility, ANSI/MIL-STD-1815A, Ada Joint Program Office

15. NUMBER OF PAGES

16. PRICE CODE

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

AVF Control Number: AVF-VSR-AFNOR-89-16

Ada COMPILER
VALIDATION SUMMARY REPORT:
Certificate Number: 900109A1.10239
STERIA
Ent_Alsy_01, Version 4.2
CETIA UNIGRAPH 6000 Host and Motorola MC68020 in PMF Target

Completion of On-Site Testing:
9 January 1990

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

Prepared For:
Ada Joint Program Office
United States Department of Defense
Washington DC 20301-3081

Ada Compiler Validation Summary Report:

Compiler Name: Ent_Alsy_01, Version 4.2

Cértificate Number: 900109A1.10239

Host: CETIA UNIGRAPH 6000 under Unigraph/X release 3.0.4

Target: Motorola MC68020 in PMF with ARTK, Version 4.2 (bare machine)

Testing Completed 9 January 1990 Using ACVC 1.10

This report has been reviewed and is approved.

AFNOR
Fabrice Garnier de Labareyre
Tour Europe
Cedex 7
F-92049 Paris la Défense

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

TABLE OF CONTENTS

CHAPTER 1 INTRODUCTION

1.1 PURPOSE OF THIS VALIDATION SUMMARY REPORT 4
1.2 USE OF THIS VALIDATION SUMMARY REPORT 5
1.3 REFERENCES. 6
1.4 DEFINITION OF TERMS 6
1.5 ACVC TEST CLASSES 7

CHAPTER 2 CONFIGURATION INFORMATION

2.1 CONFIGURATION TESTED. 9
2.2 IMPLEMENTATION CHARACTERISTICS. 9

CHAPTER 3 TEST INFORMATION

3.1 TEST RESULTS. 14
3.2 SUMMARY OF TEST RESULTS BY CLASS. 14
3.3 SUMMARY OF TEST RESULTS BY CHAPTER. 14
3.4 WITHDRAWN TESTS 15
3.5 INAPPLICABLE TESTS. 15
3.6 TEST, PROCESSING, AND EVALUATION MODIFICATIONS. 18
3.7 ADDITIONAL TESTING INFORMATION. 19
3.7.1 Prevalidation 19
3.7.2 Test Method 19
3.7.3 Test Site 20

APPENDIX A DECLARATION OF CONFORMANCE

APPENDIX B TEST PARAMETERS

APPENDIX C WITHDRAWN TESTS

APPENDIX D APPENDIX F OF THE Ada STANDARD

CHAPTER 1

INTRODUCTION

This Validation Summary Report (VSR) describes the extent to which a specific Ada compiler conforms to the Ada Standard, ANSI/MIL-STD-1815A. This report explains all technical terms used within it and thoroughly reports the results of testing this compiler using the Ada Compiler Validation Capability (ACVC). An Ada compiler must be implemented according to the Ada Standard, and any implementation-dependent features must conform to the requirements of the Ada Standard. The Ada Standard must be implemented in its entirety, and nothing can be implemented that is not in the Standard.

Even though all validated Ada compilers conform to the Ada Standard, it must be understood that some differences do exist between implementations. The Ada Standard permits some implementation dependencies--for example, the maximum length of identifiers or the maximum values of integer types. Other differences between compilers result from the characteristics of particular operating systems, hardware, or implementation strategies. All the dependencies observed during the process of testing this compiler are given in this report.

The information in this report is derived from the test results produced during validation testing. The validation process includes submitting a suite of standardized tests, the ACVC, as inputs to an Ada compiler and evaluating the results. The purpose of validating is to ensure conformity of the compiler to the Ada Standard by testing that the compiler properly implements legal language constructs and that it identifies and rejects illegal language constructs. The testing also identifies behavior that is implementation dependent, but is permitted by the Ada Standard. Six classes of tests are used. These tests are designed to perform checks at compile time, at link time, and during execution.

1.1 PURPOSE OF THIS VALIDATION SUMMARY REPORT

This VSR documents the results of the validation testing performed on an Ada compiler. Testing was carried out for the following purposes:

- To attempt to identify any language constructs supported by the compiler that do not conform to the Ada Standard

- To attempt to identify any language constructs not supported by the compiler but required by the Ada Standard

- To determine that the implementation-dependent behavior is allowed by the Ada Standard

Testing of this compiler was conducted by STERIA under the direction of the AVF according to procedures established by the Ada Joint Program Office and administered by the Ada Validation Organization (AVO). On-site testing was completed 9 January 1990 at STERIA, 26, Avenue de l'Europe, Velizy-Villacoublay FRANCE.

1.2 USE OF THIS VALIDATION SUMMARY REPORT

Consistent with the national laws of the originating country, the AVO 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 compiler has no nonconformities to the Ada Standard other than those presented. Copies of this report are available to the public from:

Ada Information Clearinghouse
Ada Joint Program Office
OUSDRE
The Pentagon, Rm 3D-139 (Fern Street)
Washington DC 20301-3081

or from:

AFNOR
Tour Europe
cedex 7
F-92049 Paris la Défense

Questions regarding this report or the validation test results should be directed to the AVF listed above or to:

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

1.3 REFERENCES

1. Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815A, February 1983, and ISO 8652-1987.

2. Ada Compiler Validation Procedures, Ada Joint Program Office, May 1989

3. Ada Compiler Validation Capability Implementers' Guide, SofTech, Inc., December 1986.

4. Ada Compiler Validation Capability User's Guide, January 1989

1.4 DEFINITION OF TERMS

ACVC — The Ada Compiler Validation Capability. The set of Ada programs that tests the conformity of an Ada compiler to the Ada programming language.

Ada Commentary — An Ada Commentary contains all information relevant to the point addressed by a comment on the Ada Standard. These comments are given a unique identification number having the form AI-ddddd.

Ada Standard — ANSI/MIL-STD-1815A, February 1983 and ISO 8652-1987.

Applicant — The agency requesting validation.

AVF — The Ada Validation Facility. The AVF is responsible for conducting compiler validations according to procedures contained in the Ada Compiler Validation Procedures.

AVO — The Ada Validation Organization. The AVO has oversight authority over all AVF practices for the purpose of maintaining a uniform process for validation of Ada compilers. The AVO provides administrative and technical support for Ada validations to ensure consistent practices.

Compiler — A processor for the Ada language. In the context of this report, a compiler is any language processor, including cross-compilers, translators, and interpreters.

Failed test — An ACVC test for which the compiler generates a result that demonstrates nonconformity to the Ada Standard.

Host — The computer on which the compiler resides.

Inapplicable test — An ACVC test that uses features of the language that a compiler is not required to support or may legitimately support in a way other than the one expected by the test.

Passed test — An ACVC test for which a compiler generates the expected result.

Target — The computer which executes the code generated by the compiler.

Test — A program that checks a compiler's conformity regarding a particular feature or a combination of features to the Ada Standard. In the context of this report, the term is used to designate a single test, which may comprise one or more files.

Withdrawn — An ACVC test found to be incorrect and not used to check test conformity to the Ada Standard. A test may be incorrect because it has an invalid test objective, fails to meet its test objective, or contains illegal or erroneous use of the language.

1.5 ACVC TEST CLASSES

Conformity to the Ada Standard is measured using the ACVC. The ACVC contains both legal and illegal Ada programs structured into s[illegible]: 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, and special program units are used to report their results during execution. Class B tests are expected to produce compilation errors. Class L tests are expected to produce errors because of the way in which a program library is used at link time.

Class A tests ensure the successful compilation and execution of legal Ada programs with certain language constructs which cannot be verified at run time. There are no explicit program components in a Class A test to check semantics. For example, a Class A test checks that reserved words of another language (other than those already reserved in the Ada language) are not treated as reserved words by an Ada compiler. A Class A test is passed if no errors are detected at compile time and the program executes to produce a PASSED message.

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 every syntax or semantic error in the test is detected. A Class B test is passed if every illegal construct that it contains is detected by the compiler.

Class C tests check the run time system to ensure that legal Ada programs can be correctly compiled and executed. Each Class C test is self-checking and produces a PASSED, FAILED, or NOT APPLICABLE message indicating the result when it is executed.

Class D tests check the compilation and execution capacities of a compiler. Since there are no capacity requirements placed on a compiler by the Ada Standard for some parameters--for example, the number of identifiers permitted in a compilation or the number of units in a library--a compiler may refuse to compile a Class D test and still be a conforming compiler. Therefore, if a Class D test fails to compile because the capacity of the compiler is exceeded, the test is classified as inapplicable. If a Class D test compiles successfully, it is self-checking and produces a PASSED or FAILED message during execution.

Class E tests are expected to execute successfully and check implementation-dependent options and resolutions of ambiguities in the Ada Standard. Each Class E test is self-checking and produces a NOT APPLICABLE, PASSED, or FAILED message when it is compiled and executed. However, the Ada Standard permits an implementation to reject programs containing some features addressed by Class E tests during compilation. Therefore, a Class E test is passed by a compiler if it is compiled successfully and executes to produce a PASSED message, or if it is rejected by the compiler for an allowable reason.

Class L tests check that incomplete or illegal Ada programs involving multiple, separately compiled units are detected and not allowed to execute. Class L tests are compiled separately and execution is attempted. A Class L test passes if it is rejected at link time--that is, an attempt to execute the main program must generate an error message before any declarations in the main program or any units referenced by the main program are elaborated. In some cases, an implementation may legitimately detect errors during compilation of the test.

Two library units, the package REPORT and the procedure CHECK_FILE, support the self-checking features of the executable tests. The package REPORT provides the mechanism by which executable tests report PASSED, FAILED, or NOT APPLICABLE results. It 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 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. These tests produce messages that are examined to verify that the units are operating correctly. If these units are not operating correctly, then the validation is not attempted.

The text of each test in the ACVC follows conventions that are intended to ensure that the tests are reasonably portable without modification. For example, the tests make use of only the basic set of 55 characters, contain lines with a maximum length of 72 characters, use small numeric values, and place features that may not be supported by all implementations in separate tests. However, some tests contain values that require the test to be customized according to implementation-specific values--for example, an illegal file name. A list of the values used for this validation is provided in Appendix C.

A compiler must correctly process each of the tests in the suite and demonstrate conformity to the Ada Standard by either meeting the pass criteria given for the test or by showing that the test is inapplicable to the implementation. The applicability of a test to an implementation is considered each time the implementation is validated. A test that is inapplicable for one validation is not necessarily inapplicable for a subsequent validation. Any test that was determined to contain an illegal language construct or an erroneous language construct is withdrawn from the ACVC and, therefore, is not used in testing a compiler. The tests withdrawn at the time of this validation are given in Appendix D.

CHAPTER 2

CONFIGURATION INFORMATION

2.1 CONFIGURATION TESTED

The candidate compilation system for this validation was tested under the following configuration:

Compiler: Ent_Alsy_01, Version 4.2

ACVC Version: 1.10

Certificate Number: 900109A1.10239

Host Computer:

Machine:	CETIA UNIGRAPH 6000
Operating System:	Unigraph/X release 3.0.4
Memory Size:	8 Mb

Target Computer:

Machine:	Motorola MC68020 in PMF
CPU:	Motorola MC68020
Bus:	VME
I/O:	Intel 8251A
Timer:	Intel 8254
Coprocessor:	Motorola MC68881
Run-Time System:	ARTK, Version 4.2
Memory Size:	1 Mb

Communications Network: RS 232 serial connection

2.2 IMPLEMENTATION CHARACTERISTICS

One of the purposes of validating compilers is to determine the behavior of a compiler in those areas of the Ada Standard that permit implementations to differ. Class D and E tests specifically check for such implementation differences. However, tests in other classes also characterize an implementation. The tests demonstrate the following characteristics:

a. Capacities.

(1) The compiler correctly processes a compilation containing 723 variables in the same declarative part. (See test D29002K.)

(2) The compiler correctly processes tests containing loop statements nested to 65 levels. (See tests D55A03A..H (8 tests).)

(3) The compiler correctly processes a test containing block statements nested to 65 levels. (See test D56001B.)

(4) The compiler correctly processes tests containing recursive procedures separately compiled as subunits nested to 10 levels. (See tests D64005E..G (3 tests).)

b. Predefined types.

(1) This implementation supports the additional predefined types, SHORT_INTEGER, LONG_INTEGER, LONG_FLOAT in the package STANDARD. (See tests B86001T..Z (7 tests).)

c. Based literals.

(1) An implementation is allowed to raise NUMERIC_ERROR or CONSTRAINT_ERROR when a value exceeds SYSTEM.MAX_INT . This implementation raises NUMERIC_ERROR during execution. (See test E24201A.)

d. Expression evaluation.

The order in which expressions are evaluated and the time at which constraints are checked are not defined by the language. While the ACVC tests do not specifically attempt to determine the order of evaluation of expressions, test results indicate the following:

(1) None of the default initialization expressions for record components are evaluated before any value is checked for membership in a component's subtype. (See test C32117A.)

(2) Assignments for subtypes are performed with the same precision as the base type. (See test C35712B.)

(3) This implementation uses no extra bits for extra precision. This implementation uses all extra bits for extra range. (See test C35903A.)

(4) NUMERIC_ERROR is raised when an integer literal operand in a comparison or membership test is outside the range of the base type. (See test C45232A.)

(5) NUMERIC_ERROR is raised when a literal operand in a fixed-point comparison or membership test is outside the range of the base type. (See test C45252A.)

(6) Underflow is gradual. (See tests C45524A..Z.) (26 tests)

e. Rounding.

The method by which values are rounded in type conversions is not defined by the language. While the ACVC tests do not specifically attempt to determine the method of rounding, the test results indicate the following:

(1) The method used for rounding to integer is round to even. (See tests C46012A..Z.) (26 tests)

(2) The method used for rounding to longest integer is round to even. (See tests C46012A..Z.) (26 tests)

(3) The method used for rounding to integer in static universal real expressions is round to even. (See test C4A014A.)

f. Array types.

An implementation is allowed to raise NUMERIC_ERROR or CONSTRAINT_ERROR for an array having a 'LENGTH that exceeds STANDARD.INTEGER'LAST and/or SYSTEM.MAX_INT. For this implementation:

(1) Declaration of an array type or subtype declaration with more than SYSTEM.MAX_INT components raises NUMERIC_ERROR . (See test C36003A.)

(2) NUMERIC_ERROR is raised when 'LENGTH is applied to an array type with INTEGER'LAST + 2 components. (See test C36202A.)

(3) NUMERIC_ERROR is raised when an array type with SYSTEM.MAX_INT + 2 components is declared. (See test C36202B.)

(4) A packed BOOLEAN array having a 'LENGTH exceeding INTEGER'LAST raises no exception. (See test C52103X.)

(5) A packed two-dimensional BOOLEAN array with more than INTEGER'LAST components raises CONSTRAINT_ERROR when the length of a dimension is calculated and exceeds INTEGER'LAST. (See test C52104Y.)

(6) In assigning one-dimensional array types, the expression is evaluated in its entirety before CONSTRAINT_ERROR is raised when checking whether the expression's subtype is compatible with the target's subtype. (See test C52013A.)

(7) In assigning two-dimensional array types, the expression is not evaluated in its entirety before CONSTRAINT_ERROR is raised when checking whether the expression's subtype is compatible with the target's subtype. (See test C52013A.)

g. A null array with one dimension of length greater than INTEGER'LAST may raise NUMERIC_ERROR or CONSTRAINT_ERROR either when declared or assigned. Alternatively, an implementation may accept the declaration. However, lengths must match in array slice assignments. This implementation raises no exception. (See test E52103Y.)

h. Discriminated types.

(1) In assigning record types with discriminants, the expression is evaluated in its entirety before CONSTRAINT_ERROR is raised when checking whether the expression's subtype is compatible with the target's subtype. (See test C52013A.)

i. Aggregates.

(1) In the evaluation of a multi-dimensional aggregate, the test results indicate that all choices are evaluated before checking against the index type. (See tests C43207A and C43207B.)

(2) In the evaluation of an aggregate containing subaggregates, not all choices are evaluated before being checked for identical bounds. (See test E43212B.)

(3) CONSTRAINT_ERROR is raised after all choices are evaluated when a bound in a non-null range of a non-null aggregate does not belong to an index subtype. (See test E43211B.)

j. Pragmas.

(1) The pragma INLINE is supported for functions or procedures, but not functions called inside a package specification. (See tests LA3004A..B, EA3004C..D, and CA3004E..F.)

k. Generics.

(1) Generic specifications and bodies can be compiled in separate compilations. (See tests CA1012A, CA2009C, CA2009F, BC3204C, and BC3205D.)

(2) Generic subprogram declarations and bodies can be compiled in separate compilations. (See tests CA1012A and CA2009F.)

(3) Generic library subprogram specifications and bodies can be compiled in separate compilations. (See test CA1012A.)

(4) Generic non-library package bodies as subunits can be compiled in separate compilations. (See test CA2009C.)

(5) Generic non-library subprogram bodies can be compiled in separate compilations from their stubs. (See test CA2009F.)

(6) Generic unit bodies and their subunits can be compiled in separate compilations. (See test CA3011A.)

(7) Generic package declarations and bodies can be compiled in separate compilations. (See tests CA2009C, BC3204C, and BC3205D.)

(8) Generic library package specifications and bodies can be compiled in separate compilations. (See tests BC3204C and BC3205D.)

(9) Generic unit bodies and their subunits can be compiled in separate compilations. (See test CA3011A.)

1. Input and output.

(3) The director, AJPO, has determined (AI-00332) that every call to OPEN and CREATE must raise USE_ERROR or NAME_ERROR if file input/output is not supported. This implementation exhibits this behavior for SEQUENTIAL_IO, DIRECT_IO, and TEXT_IO.

CHAPTER 3

TEST INFORMATION

3.1 TEST RESULTS

Version 1.10 of the ACVC comprises 3717 tests. When this compiler was tested, 44 tests had been withdrawn because of test errors. The AVF determined that 572 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. Modifications to the code, processing, or grading for 53 tests were required. (See section 3.6.)

The AVF concludes that the testing results demonstrate acceptable conformity to the Ada Standard.

3.2 SUMMARY OF TEST RESULTS BY CLASS

RESULT	TEST CLASS						TOTAL
	A	B	C	D	E	L	
Passed	129	1133	1761	16	16	46	3101
Inapplicable	0	5	554	1	12	0	572
Withdrawn	1	2	35	0	6	0	44
TOTAL	130	1140	2350	17	34	46	3717

3.3 SUMMARY OF TEST RESULTS BY CHAPTER

RESULT	CHAPTER													TOTAL
	2	3	4	5	6	7	8	9	10	11	12	13	14	
Passed	198	577	555	248	171	99	161	332	137	36	252	259	76	3101
Inappl	14	72	125	0	1	0	5	0	0	0	0	110	245	572
Wdrn	1	1	0	0	0	0	0	2	0	0	1	35	4	44
TOTAL	213	650	680	248	172	99	166	334	137	36	253	404	325	3717

3.4 WITHDRAWN TESTS

The following 44 tests were withdrawn from ACVC Version 1.10 at the time of this validation:

A39005G	B97102E	BC3009B	C97116A	CD2A62D	CD2A63A	CD2A63B	CD2A63C	CD2A63D
CD2A66A	CD2A66B	CD2A66C	CD2A66D	CD2A73A	CD2A73B	CD2A73C	CD2A73D	CD2A76A
CD2A76B	CD2A76C	CD2A76D	CD2A81G	CD2A83G	CD2A84M	CD2A84N	CD2D11B	CD2B15C
CD5007B	CD5011O	CD7105A	CD7203B	CD7204B	CD7205C	CD7205D	CE2107I	CE3111C
CE3301A	CE3411B	E28005C	ED7004B	ED7005C	ED7005D	ED7006C	ED7006D	

See Appendix D for the reason that each of these tests was withdrawn.

3.5 INAPPLICABLE TESTS

Some tests do not apply to all compilers because they make use of features that a compiler is not required by the Ada Standard to support. Others may depend on the result of another test that is either inapplicable or withdrawn. The applicability of a test to an implementation is considered each time a validation is attempted. A test that is inapplicable for one validation attempt is not necessarily inapplicable for a subsequent attempt. For this validation attempt, 572 tests were inapplicable for the reasons indicated:

- The following 201 tests are not applicable because they have floating-point type declarations requiring more digits than System.Max_Digits:

C24113L..Y (14 tests)	C35705L..Y (14 tests)
C35706L..Y (14 tests)	C35707L..Y (14 tests)
C35708L..Y (14 tests)	C35802L..Z (15 tests)
C45241L..Y (14 tests)	C45321L..Y (14 tests)
C45421L..Y (14 tests)	C45521L..Z (15 tests)
C45524L..Z (15 tests)	C45621L..Z (15 tests)
C45641L..Y (14 tests)	C46012L..Z (15 tests)

- C35702A and B86001T are not applicable because this implementation supports no predefined type Short_Float.

- C45531M..P (4 tests) and C45532M..P (4 tests) are not applicable because the value of System.Max_Mantissa is less than 32.

- D64005G is not applicable because this implementation does not support nesting 17 levels of recursive procedure calls.

- C86001F, is not applicable because recompilation of Package SYSTEM is not allowed.

- B86001X, C45231D, and CD7101G are not applicable because this implementation does not support any predefined integer type with a name other than Integer, Long_Integer, or Short_Integer.

- B86001Y is not applicable because this implementation supports no predefined fixed-point type other than Duration.

- B86001Z is not applicable because this implementation supports no predefined floating-point type with a name other than Float, Long_Float, or Short_Float.

- BD5006D is not applicable because address clause for packages is not supported by this implementation.

- The following 10 tests are not applicable because size clause on float is not supported by this implementation:

CD1009C	CD2A41A..B (2 tests)
CD2A41E	CD2A42A..B (2 tests)
CD2A42E..F (2 tests)	CD2A42I..J (2 tests)

- CD1C04B, CD1C04E, CD4051A..D (4 tests) are not applicable because representation clause on derived records or derived tasks is not supported by this implementation.

- CD2A84B..I (8 tests), CD2A84K..L (2 tests) are not applicable because size clause on access type is not supported by this implementation.

- The following 28 tests are not applicable because size clause for derived private type is not supported by this implementation:

CD1C04A	CD2A21C..D (2 tests)
CD2A22C..D (2 tests)	CD2A22G..H (2 tests)
CD2A31C..D (2 tests)	CD2A32C..D (2 tests)
CD2A32G..H (2 tests)	CD2A41C..D (2 tests)
CD2A42C..D (2 tests)	CD2A42G..H (2 tests)
CD2A51C..D (2 tests)	CD2A52C..D (2 tests)
CD2A52G..H (2 tests)	CD2A53D
CD2A54D	CD2A54H

- The following 29 tests are not applicable because of the way this implementation allocates storage space for one component, size specification clause for an array type or for a record type requires compression of the storage space needed for all the components (without gaps).

CD2A61A..D (4 tests)	CD2A61F
CD2A61H..L (5 tests)	CD2A62A..C (3 tests)
CD2A71A..D (4 tests)	CD2A72A..D (4 tests)
CD2A74A..D (4 tests)	CD2A75A..D (4 tests)

- CD4041A is not applicable because alignment "at mod 8" is not supported by this implementation.

- The following 21 tests are not applicable because address clause for a constant is not supported by this implementation:

CD5011B,D,F,H,L,N,R (7 tests)	CD5012C,D,G,H,L (5 tests)
CD5013B,D,F,H,L,N,R (7 tests)	CD5014U,W (2 tests)

- CD5012J, CD5013S, CD5014S are not applicable because address clause for a task is not supported by this implementation.

- CE2103A is not applicable because USE_ERROR is raised on a CREATE of an instantiation of SEQUENTIAL_IO with an ILLEGAL EXTERNAL FILE NAME.

- CE2103B is not applicable because USE_ERROR is raised on a CREATE of an instantiation of DIRECT_IO with an ILLEGAL EXTERNAL FILE NAME.

- CE3107A is not applicable because USE_ERROR is raised on a CREATE of a file of type TEXT_IO.FILE_TYPE with an ILLEGAL EXTERNAL FILE NAME.

- The following 242 tests are inapplicable because sequential, text, and direct access files are not supported:

CE2102A..C (3 tests)	CE2102G..H (2 tests)
CE2102K	CE2102N..Y (12 tests)
CE2103C..D (2 tests)	CE2104A..D (4 tests)
CE2105A..B (2 tests)	CE2106A..B (2 tests)
CE2107A..H (8 tests)	CE2107L
CE2108A..H (8 tests)	CE2109A..C (3 tests)
CE2110A..D (4 tests)	CE2111A..I (9 tests)
CE2115A..B (2 tests)	CE2201A..C (3 tests)
EE2201D..E (2 tests)	CE2201F..N (9 tests)
CE2204A..D (4 tests)	CE2205A
CE2208B	CE2401A..C (3 tests)
EE2401D	EE2401G
CE2401E..F (2 tests)	CE2401H..L (5 tests)
CE2404A..B (2 tests)	CE2405B
CE2406A	CE2407A..B (2 tests)
CE2408A..B (2 tests)	CE2409A..B (2 tests)
CE2410A..B (2 tests)	CE2411A
CE3102A..B (2 tests)	EE3102C
CE3102F..H (3 tests)	CE3102J..K (2 tests)
CE3103A	CE3104A..C (3 tests)
CE3107B	CE3108A..B (2 tests)
CE3109A	CE3110A
CE3111A..B (2 tests)	CE3111D..E (2 tests)
CE3112A..D (4 tests)	CE3114A..B (2 tests)
CE3115A	EE3203A
CE3208A	EE3301B
CE3302A	CE3305A
CE3402A	EE3402B
CE3402C..D (2 tests)	CE3403A..C (3 tests)
CE3403E..F (2 tests)	CE3404B..D (3 tests)
CE3405A	EE3405B
CE3405C..D (2 tests)	CE3406A..D (4 tests)
CE3407A..C (3 tests)	CE3408A..C (3 tests)
CE3409A	CE3409C..E (3 tests)
EE3409F	CE3410A
CE3410C..E (3 tests)	EE3410F
CE3411A	CE3411C
CE3412A	
EE3412C	CE3413A
CE3413C	CE3602A..D (4 tests)
CE3603A	CE3604A..B (2 tests)
CE3605A..E (5 tests)	CE3606A..B (2 tests)
CE3704A..F (6 tests)	CE3704M..O (3 tests)
CE3706D	CE3706F..G (2 tests)
CE3804A..P (16 tests)	CE3805A..B (2 tests)
CE3806A..B (2 tests)	CE3806D..E (2 tests)
CE3806G..H (2 tests)	CE3905A..C (3 tests)
CE3905L	CE3906A..C (3 tests)
CE3906E..F (2 tests)	

TEST, PROCESSING, AND EVALUATION MODIFICATIONS

It is expected that some tests will require modifications of code, processing, or evaluation in order to compensate for legitimate implementation behavior. Modifications are made by the AVF in cases where legitimate implementation behavior prevents the successful completion of an (otherwise) applicable test. Examples of such modifications include: adding a length clause to alter the default size of a collection; splitting a Class B test into subtests so that all errors are detected; and confirming that messages produced by an executable test demonstrate conforming behavior that wasn't anticipated by the test (such as raising one exception instead of another).

Modifications were required for 53 tests.

The following 27 tests were split because syntax errors at one point resulted in the compiler not detecting other errors in the test:
B23004A B24007A B24009A B28003A B32202A B32202B B32202C B33001A B36307A B37004A
B49003A B49005A B61012A B62001B B74304B B74304C B74401F B74401R B91004A B95032A
B95069A B95069B BA1101B BC2001D BC3009A BC3009C BD5005B

The following 21 tests were split in order to show that the compiler was able to find the representation clause indicated by the comment --N/A =>ERROR :

CD2A61A CD2A61B CD2A61F CD2A61I CD2A61J CD2A62A CD2A62B CD2A71A CD2A71B CD2A72A
CD2A72B CD2A75A CD2A75B CD2A84B CD2A84C CD2A84D CD2A84E CD2A84F CD2A84G CD2A84H
CD2A84I

The test EA3004D when run as it is, the implementation fails to detect an error on line 27 of test file EA3004D6M (line 115 of "cat -n ea3004d*"). This is because the pragma INLINE has no effect when its object is within a package specification. However, the results of running the test as it is does not confirm that the pragma had no effect, only that the package was not made obsolete. By re-ordering the compilations so that the two subprograms are compiled after file D5 (the re-compilation of the "with"ed package that makes the various earlier units obsolete), we create a test that shows that indeed pragma INLINE has no effect when applied to a subprogram that is called within a package specification: the test then executes and produces the expected NOT_APPLICABLE result (as though INLINE were not supported at all). The re-ordering of EA3004D test files is 0-1-4-5-2-3-6.

BA2001E requires that duplicate names of subunits with a common ancestor be detected and rejected at compile time. This implementation detects the error at link time, and the AVO ruled that this behavior is acceptable.

Modified version was produced for C87B62B, in order to have the test run to completion and fully exhibit the test behavior:
An explicit STORAGE_SIZE clause was added for the access type declared at line 68. This allows the test to execute without raising STORAGE_ERROR and to meet its objective (test overloading resolution in expression within length clause). The test then produces the expected PASSED result.

Modified versions were produced for CD2C11A and CD2C11B, in order to have the test run to completion and fully exhibit the test behavior:
Because the given STORAGE_SIZE is too small for the implementation, the length clause was changed from 1024 to 4096 at line 43 and 46, respectively, . The same change was made also at line 95 and 98 on the identity function IDENT_INT. This allows the test to execute without raising STORAGE_ERROR and to meet its objective (test if a task storage size specification can be given for a task type). The test then produces the expected PASSED result.

3.7 ADDITIONAL TESTING INFORMATION

3.7.1 Prevalidation

Prior to validation, a set of test results for ACVC Version 1.10 produced by the Ent_Alsy_01, Version 4.2 compiler was submitted to the AVF by the applicant for review. Analysis of these results demonstrated that the compiler successfully passed all applicable tests, and the compiler exhibited the expected behavior on all inapplicable tests.

3.7.2 Test Method

Testing of the Ent_Alsy_01, Version 4.2 compiler using ACVC Version 1.10 was conducted on-site by a validation team from the AVF. The configuration in which the testing was performed is described by the following designations of hardware and software components:

Host computer:	CETIA UNIGRAPH 6000
Host operating system:	Unigraph/X release 3.0.4
Target computer:	Motorola MC68020 in PMF
Compiler:	Ent_Alsy_01, Version 4.2
Pre-linker:	built-in and Alsys proprietary
Linker:	Editeur de lien 68020-PMF, Version 2.0
Loader/Downloader:	built-in and Alsys proprietary
Target Run-Time system:	ARTK, Version 4.2 (bare machine)

A data cartridge containing all tests except for withdrawn tests and tests requiring unsupported floating-point precisions was taken on-site by the validation team for processing. Tests that make use of implementation-specific values were customized before being written to the data cartridge. Tests requiring modifications during the prevalidation testing were included in their modified form on the data cartridge.

The contents of the data cartridge were loaded directly onto the host computer.

After the test files were loaded to disk, the full set of tests was compiled and linked on the CETIA UNIGRAPH 6000. Then all executable images were transferred to a CETIA UNIGRAPH 3000 via Ethernet. Then all executable images were transferred from CETIA UNIGRAPH 3000 to the Motorola MC68020 in PMF via RS 232 serial connection and run. Results were printed from the host computer.

The compiler was tested using command scripts provided by STERIA and reviewed by the validation team. The compiler was tested using all default option settings except for the following:

OPTION	EFFECT
CALLS=INLINED	Allow inline insertion of code for subprograms and take pragma INLINE are taken into account
EXPRESSION=EXTENSIVE	Optimization are achieved on common subexpression and register allocation
REDUCTION=PARTIAL	Perform some high level optimizations on checks and loops
OBJECT=PEEPHOLE	Local optimization during code generation is made
GENERIC=INLINED	Generics are inlined
FLOAT=MC68881	Floating point operations use the MC68881 arithmetic coprocessor

Tests were compiled, linked, and executed (as appropriate) using a single host and target computer. Test output, compilation listings, and job logs were captured on data cartridge and archived at the AVF. The listings examined on-site by the validation team were also archived.

3.7.3 Test Site

Testing was conducted at STERIA 26, Avenue de l'Europe, Velizy-Villacoublay FRANCE, and was completed on 9 January 1990.

APPENDIX A

DECLARATION OF CONFORMANCE

STERIA and ALSYS have submitted the following Declaration of Conformance concerning the Ent_Alsy_01, Version 4.2 compiler.

DECLARATION OF CONFORMANCE

Customer: STERIA

Licensor: ALSYS

Ada Validation Facility: AFNOR, Tour Europe, Cedex 7
F-92049 PARIS LA DEFENSE

ACVC Version: 1.10

Ada Implementation:

Compiler Name and Version: Ent_Alsy_01, Version 4.2

Host Computer System: CETIA UNIGRAPH 6000 under
Unigraph/X release 3.0.4

Target Computer System: Motorola MC68020 in PMF under
ARTK, Version 4.2 (bare machine)

Customer's Declaration

I, the undersigned, representing STERIA, declare that STERIA has no knowledge of deliberate deviations from the Ada Language Standard ANSI/MIL-STD-1815A in the implementation listed in this declaration.

Date: January 9 1990

Luc Secouard, Project Manager
STERIA
Velizy-Villacoublay, France

Licensor's Declaration

I, the undersigned, representing ALSYS, declare that STERIA is the licensee of the above implementation and the certificate shall be awarded in the name of STERIA.

Date:

Etienne Morel, Managing Director
ALSYS
La Celle Saint-Cloud, France

APPENDIX B

TEST PARAMETERS

Certain tests in the ACVC make use of implementation-dependent values, such as the maximum length of an input line and invalid file names. A test that makes use of such values is identified by the extension .TST in its file name. Actual values to be substituted are represented by names that begin with a dollar sign. A value must be substituted for each of these names before the test is run. The values used for this validation are given below.

Name and Meaning	Value
$ACC_SIZE An integer literal whose value is the number of bits sufficient to hold any value of an access type.	32
$BIG_ID1 Identifier the size of the maximum input line length with varying last character.	(254 * 'A') & '1'
$BIG_ID2 Identifier the size of the maximum input line length with varying last character.	(254 * 'A) & '2'
$BIG_ID3 Identifier the size of the maximum input line length with varying middle character.	(126 * 'A') & '3' & (128 * 'A')
$BIG_ID4 Identifier the size of the maximum input line length with varying middle character.	(126 * 'A') & '4' & (128 * 'A')

Name and Meaning	Value
$BIG_INT_LIT An integer literal of value 298 with enough leading zeroes so that it is the size of the maximum line length.	(252 * '0') & '298'
$BIG_REAL_LIT A universal real literal of value 690.0 with enough leading zeroes to be the size of the maximum line length.	(250 * '0') & '690.0'
$BIG_STRING1 A string literal which when catenated with BIG_STRING2 yields the image of BIG_ID1.	'"' & (127 * 'A') & '"'
$BIG_STRING2 A string literal which when catenated to the end of BIG_STRING1 yields the image of BIG_ID1.	'"' & (127 * 'A') & '1"'
$BLANKS A sequence of blanks twenty characters less than the size of the maximum line length.	(235 * ' ')
$COUNT_LAST A universal integer literal whose value is TEXT_IO.COUNT'LAST.	2147483647
$DEFAULT_MEM_SIZE An integer literal whose value is SYSTEM.MEMORY_SIZE.	2**32
$DEFAULT_STOR_UNIT An integer literal whose value is SYSTEM.STORAGE_UNIT.	8
$DEFAULT_SYS_NAME The value of the constant SYSTEM.SYSTEM_NAME.	ARTK
$DELTA_DOC A real literal whose value is SYSTEM.FINE_DELTA.	2#1.0#E-31

Name and Meaning	Value
$FIELD_LAST A universal integer literal whose value is TEXT_IO.FIELD'LAST.	255
$FIXED_NAME The name of a predefined fixed-point type other than DURATION.	NO_SUCH_TYPE
$FLOAT_NAME The name of a predefined floating-point type other than FLOAT, SHORT_FLOAT, or LONG_FLOAT.	NO_SUCH_TYPE
$GREATER_THAN_DURATION A universal real literal that lies between DURATION'BASE'LAST and DURATION'LAST or any value in the range of DURATION.	-100_000_000.0
$GREATER_THAN_DURATION_BASE_LAST A universal real literal that is greater than DURATION'BASE'LAST.	100_000_000.0
$HIGH_PRIORITY An integer literal whose value is the upper bound of the range for the subtype SYSTEM.PRIORITY.	24
$ILLEGAL_EXTERNAL_FILE_NAME1 An external file name specifying a non existent directory	/~/*/f1
$ILLEGAL_EXTERNAL_FILE_NAME2 An external file name different from $ILLEGAL_EXTERNAL_FILE_NAME1	/~/*/f2
$INTEGER_FIRST A universal integer literal whose value is INTEGER'FIRST.	-32768
$INTEGER_LAST A universal integer literal whose value is INTEGER'LAST.	32767
$INTEGER_LAST_PLUS_1 A universal integer literal whose value is INTEGER'LAST + 1.	32768

Name and Meaning	Value
$LESS_THAN_DURATION A universal real literal that lies between DURATION'BASE'FIRST and DURATION'FIRST or any value in the range of DURATION.	-100_00.0
$LESS_THAN_DURATION_BASE_FIRST A universal real literal that is less than DURATION'BASE'FIRST.	-3_000_000.0
$LOW_PRIORITY An integer literal whose value is the lower bound of the range for the subtype SYSTEM.PRIORITY.	1
$MANTISSA_DOC An integer literal whose value is SYSTEM.MAX_MANTISSA.	31
$MAX_DIGITS Maximum digits supported for floating-point types.	15
$MAX_IN_LEN Maximum input line length permitted by the implementation.	255
$MAX_INT A universal integer literal whose value is SYSTEM.MAX_INT.	2147483647
$MAX_INT_PLUS_1 A universal integer literal whose value is SYSTEM.MAX_INT+1.	2_147_483_648
$MAX_LEN_INT_BASED_LITERAL A universal integer based literal whose value is 2:11: with enough leading zeroes in the mantissa to be MAX_IN_LEN long.	'2:' & (250 * '0') & '11:'
$MAX_LEN_REAL_BASED_LITERAL A universal real based literal whose value is 16: F.E: with enough leading zeroes in the mantissa to be MAX_IN_LEN long.	'16:' & (248 * '0') & 'F.E:'

Name and Meaning	Value
$MAX_STRING_LITERAL A string literal of size MAX_IN_LEN, including the quote characters.	'"' & (253 * 'A') & '"'
$MIN_INT A universal integer literal whose value is SYSTEM.MIN_INT.	-2147483648
$MIN_TASK_SIZE An integer literal whose value is the number of bits required to hold a task object which has no entries, no declarations, and NULL;" as the only statement in its body.	32
$NAME A name of a predefined numeric type other than FLOAT, INTEGER, SHORT_FLOAT, SHORT_INTEGER, LONG_FLOAT, or LONG_INTEGER.	NO_SUCH_TYPE
$NAME_LIST A list of enumeration literals in the type SYSTEM.NAME, separated by commas.	ARTK
$NEG_BASED_INT A based integer literal whose highest order nonzero bit falls in the sign bit position of the representation for SYSTEM.MAX_INT.	16#FFFFFFFE#
$NEW_MEM_SIZE An integer literal whose value is a permitted argument for pragma memory_size, other than DEFAULT_MEM_SIZE. If there is no other value, then use DEFAULT_MEM_SIZE.	2**32
$NEW_STOR_UNIT An integer literal whose value is a permitted argument for pragma storage_unit, other than DEFAULT_STOR_UNIT. If there is no other permitted value, then use value of SYSTEM.STORAGE_UNIT.	8

Name and Meaning	Value
$NEW_SYS_NAME A value of the type SYSTEM.NAME, other than $DEFAULT_SYS_NAME. If there is only one value of that type, then use that value.	ARTK
$TASK_SIZE An integer literal whose value is the number of bits required to hold a task object which has a single entry with one inout parameter.	32
$TICK A real literal whose value is SYSTEM.TICK.	1.0

APPENDIX C

WITHDRAWN TESTS

Some tests are withdrawn from the ACVC because they do not conform to the Ada Standard. The following 44 tests had been withdrawn at the time of validation testing for the reasons indicated. A reference of the form AI-ddddd is to an Ada Commentary.

E28005C
This test expects that the string "-- TOP OF PAGE. --63" of line 204 will appear at the top of the listing page due to a pragma PAGE in line 203; but line 203 contains text that follows the pragma, and it is this that must appear at the top of the page.

A39005G
This test unreasonably expects a component clause to pack an array component into a minimum size (line 30).

B97102E
This test contains an unitended illegality: a select statement contains a null statement at the place of a selective wait alternative (line 31).

C97116A
This test contains race conditions, and it assumes that guards are evaluated indivisibly. A conforming implementation may use interleaved execution in such a way that the evaluation of the guards at lines 50 & 54 and the execution of task CHANGING_OF_THE_GUARD results in a call to REPORT.FAILED at one of lines 52 or 56.

BC3009B
This test wrongly expects that circular instantiations will be detected in several compilation units even though none of the units is illegal with respect to the units it depends on; by AI-00256, the illegality need not be detected until execution is attempted (line 95).

CD2A62D
This test wrongly requires that an array object's size be no greater than 10 although its subtype's size was specified to be 40 (line 137).

CD2A63A..D, CD2A66A..D, CD2A73A..D, CD2A76A..D [16 tests]
These tests wrongly attempt to check the size of objects of a derived type (for which a 'SIZE length clause is given) by passing them to a derived subprogram (which implicitly converts them to the parent type (Ada standard 3.4:14)). Additionally, they use the 'SIZE length clause and attribute, whose interpretation is considered problematic by the WG9 ARG.

CD2A81G, CD2A83G, CD2A84N & M, & CD50110 [5 tests]
These tests assume that dependent tasks will terminate while the main program executes a loop that simply tests for task termination; this is not the case, and the main program may loop indefinitely (lines 74, 85, 86 & 96, 86 & 96, and 58, resp.).

CD2B15C & CD7205C

These tests expect that a 'STORAGE_SIZE length clause provides precise control over the number of designated objects in a collection; the Ada standard 13.2:15 allows that such control must not be expected.

CD2D11B

This test gives a SMALL representation clause for a derived fixed-point type (at line 30) that defines a set of model numbers that are not necessarily represented in the parent type; by Commentary AI-00099, all model numbers of a derived fixed-point type must be representable values of the parent type.

CD5007B

This test wrongly expects an implicitly declared subprogram to be at the the address that is specified for an unrelated subprogram (line 303).

ED7004B, ED7005C & D, ED7006C & D [5 tests]

These tests check various aspects of the use of the three SYSTEM pragmas; the AVO withdraws these tests as being inappropriate for validation.

CD7105A

This test requires that successive calls to CALENDAR.CLOCK change by at least SYSTEM.TICK; however, by Commentary AI-00201, it is only the expected frequency of change that must be at least SYSTEM.TICK--particular instances of change may be less (line 29).

CD7203B, & CD7204B

These tests use the 'SIZE length clause and attribute, whose interpretation is considered problematic by the WG9 ARG.

CD7205D

This test checks an invalid test objective: it treats the specification of storage to be reserved for a task's activation as though it were like the specification of storage for a collection.

CE2107I

This test requires that objects of two similar scalar types be distinguished when read from a file--DATA_ERROR is expected to be raised by an attempt to read one object as of the other type. However, it is not clear exactly how the Ada standard 14.2.4:4 is to be interpreted; thus, this test objective is not considered valid. (line 90)

CE3111C

This test requires certain behavior, when two files are associated with the same external file, that is not required by the Ada standard.

CE3301A

This test contains several calls to END_OF_LINE & END_OF_PAGE that have no parameter: these calls were intended to specify a file, not to refer to STANDARD_INPUT (lines 103, 107, 118, 132, & 136).

CE3411B

This test requires that a text file's column number be set to COUNT'LAST in order to check that LAYOUT_ERROR is raised by a subsequent PUT operation. But the former operation will generally raise an exception due to a lack of available disk space, and the test would thus encumber validation testing.

APPENDIX D

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 the Ent_Alsy_01, Version 4.2 compiler, as described in this Appendix, are provided by STERIA and ALSYS. Unless specifically noted otherwise, references in this appendix are to compiler documentation and not to this report. Implementation-specific portions of the package STANDARD, which are not a part of Appendix F, are:

```
package STANDARD is

 ...

 type SHORT_INTEGER is range -128 .. 127;

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

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

 type FLOAT is digits 6 range
  -2#1.111_1111_1111_1111_1111_1111#E+127
..
   2#1.111_1111_1111_1111_1111_1111#E+127;

 type LONG_FLOAT is digits 15 range
-2#1.1111_1111_1111_1111_1111_1111_1111_1111_1111_1111_1111_1111_1111#E1023
 ..
 2#1.1111_1111_1111_1111_1111_1111_1111_1111_1111_1111_1111_1111_1111#E1023;

 type DURATION is delta 2.0**(-14) range -86_400.0 ..86_400.0;

 ...

 end STANDARD;
```

steria

Alsys Ada

ADA Cross Compiler for PMF

APPENDIX F

Version 4.2

Alsys S.A.
29, Avenue de Versailles
78170 La Celle St. Cloud, France

Alsys Inc.
1432 Main Street
Waltham, MA 02154, U.S.A.

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

steria

Printed: November 1989

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.

TABLE OF CONTENTS

1 INTERFACING THE LANGUAGE ADA WITH OTHER LANGUAGES 2
1.1 Calling External Subprograms, General Aspects 3
1.2 Calling Assembler Language Subprograms 3
1.3 Calling C Language Subprograms 3

2 IMPLEMENTATION-DEPENDENT PRAGMAS 3

3 IMPLEMENTATION-DEPENDENT ATTRIBUTES 3

4 PACKAGES SYSTEM AND STANDARD 3

5 TYPE REPRESENTATION CLAUSES 6
5.1 Enumeration Types 6
5.2 Integer Types 6
5.3 Floating Point Types 6
5.4 Fixed Point Types 6
5.5 Access Types 6
5.6 Task Types 7
5.7 Array Types 7
5.8 Record Types 7

6 ADDRESS CLAUSES 8
6.1 Address Clauses for Objects 8
6.2 Address Clauses for Program Units 8
6.3 Address Clauses for Entries 8

7 UNCHECKED CONVERSIONS 8

8 INPUT-OUTPUT CHARACTERISTICS 8
8.1 Introduction 8
8.2 The FORM Parameter 9

APPENDIX F

1 INTERFACING THE LANGUAGE ADA WITH OTHER LANGUAGES

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

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

where the *language_name* can be any of

- ASSEMBLER
- C

The convention used for C parameter passing should be compatible with most C standard compilers.

The *Ada_subprogram_name* is the name by which the subprogram is known in Ada. For example, to call a subprogram known as FAST_FOURIER in Ada, written in C, the INTERFACE pragma is:

pragma INTERFACE (C, FAST_FOURIER);

To relate the name used in Ada with the name used in the original language, the Alsys Ada compiler converts this name to lower case and truncates it to 32 significant characters.

To avoid naming conflict with routines of the Alsys Ada Executive, external routine names should not begin with the letters *alsy* (whether in lower or upper case or a combination of both).

To allow the use of non-Ada naming conventions, such as special characters, or case sensitivity, an implementation-dependent pragma INTERFACE_NAME has been introduced:

pragma INTERFACE_NAME (*Ada_subprogram_name, name_string*);

so that, for example,

pragma INTERFACE_NAME (FAST_FOURIER, "fft");

will associate the FAST_FOURIER subprogram in Ada with the C subprogram fft.

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

For example:

```
package SAMPLE_LIB is
      function SAMPLE_DEVICE (X : INTEGER) return INTEGER;
      function PROCESS_SAMPLE (X : INTEGER) return INTEGER;
private
      pragma INTERFACE (ASSEMBLER, SAMPLE_DEVICE);
      pragma INTERFACE (C, PROCESS_SAMPLE);
      pragma INTERFACE_NAME (SAMPLE_DEVICE, "dev10");
      pragma INTERFACE_NAME (PROCESS_SAMPLE, "sample");
end SAMPLE_LIB;
```

1.1 Calling External Subprograms, General Aspects

Subprogram parameters of external subprograms, in the Alsys implementation, are passed on the stack in the reverse order of their declaration, that is, the first parameter is at the top of the stack. This is the ordering of parameters for C.

If a subprogram is written in assembler it is the programmer's responsibility to follow this convention.

There are no checks for consistency between the subprogram parameters (as declared in Ada) and the corresponding external subprogram parameters. As external subprograms have no notion of parameter modes, parameters passed by reference are not protected from modification by the external subprogram. That is, even if a parameter is declared to be only of mode **in** (and not **in out**) the value of the Ada actual parameter can still be modified.

In the Alsys implementation of external subprogram interfaces, scalar and access parameters of mode **in** are passed by value. All other parameters of mode **in** are passed by reference. Parameters of mode **out** or **in out** are always passed by reference.

Values of the following types cannot be passed as parameters to an external subprogram:

- Task types [9.1 9.2]

Only scalar types are allowed for the result returned by an external subprogram.

The user should be very careful to establish the exact nature of the types of parameters to be passed. The bit representations of these types can be different in crucial respects between the Alsys implementation and other languages, or between different implementations of the same language. Stacked values must occupy equal space in the two languages.

A discussion of data types in general terms is given below, before dealing with each of the languages mentioned above.

Integer Types

In the Alsys implementation, all integers are represented in two's complement form. Short integers are stored as 8-bit numbers, integers as 16-bit numbers, and long integers as 32-bit numbers.

When an integer is passed by value, a copy of the integer value is pushed on the stack, with the sign extended to satisfy the requirements of the external subprogram.

Boolean Types

Booleans are represented in the Alsys implementation as 8-bit values. FALSE is represented by all 8 bits being set to 0, and TRUE is represented by all 8 bits being set to 1.

When a Boolean is passed by value, a copy is pushed on the stack. The Ada Boolean value occupies one byte, and it is up to the user to make sure that the Boolean is in a suitable format for the external subprogram.

Enumeration Types

Values of an enumeration type [3.5.1] without enumeration representation clause [13.3] are represented internally as nonnegative integers representing the value's position in the list of enumeration literals defining the type. The first literal in the list corresponds to an integer value of 0. There can be no more than 32767 elements in an enumeration type and, as the program representation of the position number of an element starts at 0, the maximum value has to be 32766.

Values of enumeration types with less than 128 elements are represented as short (8 bit) integers, whereas values of other enumeration types are represented as normal (16 bit) integers. When passed by value, the copy of the integer is treated as if it were an unsigned integer, with any necessary extension to 64 bits to satisfy the requirements of the external subprogram.

Consequently, the values of the predefined type CHARACTER are represented as 8-bit values in the range 0..127.

Real Types

Ada fixed point types [3.5.9, 3.5.10] are not supported as parameters or results of external subprograms.

Floating point values [3.5.7, 3.5.8], in the Alsys implementation, are stored on 32 bits (FLOAT) or stored on 64 bits (LONG_FLOAT). These two types conform to the conventions defined in the document *A Proposed Standard for Binary Floating-Point Arithmetic*, IEEE P754, draft 10.0.

When passed by value, a copy is pushed on the stack, possibly extended to 64 bits according to the external subprogram conventions.

Access Types

A value of an access type [3.8] has an internal representation as the 32-bit address of the underlying designated object (an *access_address*, say, stored in an *access_address_location*). When passed by value, therefore, a copy of this 32-bit *access_address* is pushed on the stack. If the type is passed by reference, however, a 32-bit address pointer to the *access_address_location* is pushed on the stack (a double-indirect address).

Array Types

In the Alsys implementation, arrays [3.6] are always passed by reference. The value pushed on the stack is the address of the first element of the array. Multidimensional arrays are stored so that successive values of the *last* index correspond to successive elements in store.

When an array is passed as a parameter to an external subprogram, the usual checks on the consistency of array bounds between calling program and called subprogram are not enforced. This means that the programmer must be responsible for ensuring that the subprogram keeps within the proper array bounds.

Values of the predefined type STRING [3.6.3] are a special case of arrays, and are passed in the same way. The address of the first character in the string is pushed on the stack.

Record Types

Records [3.7] are always passed by reference in the Alsys implementation, pushing the address of the record on the stack. Unlike arrays, however, the individual components of a record may be reordered internally by the Alsys Ada Compiler. In addition, if a record contains discriminants or composite components of dynamic size, the compiler may add implicit components to the record.

The exact internal structure of a record in memory cannot be known directly at the time of coding.

As direct assignment to a discriminant of a record is not allowed [3.7.1], a discriminant cannot be passed as an actual parameter of mode **out** or **in out**. This restriction applies equally to Ada subprograms and to external subprograms.

1.2 Calling Assembler Language Subprograms

Scalar and access parameters of mode **in** are passed by value: the value of the parameter object is copied and pushed on the stack. All other types of **in** parameters (arrays, records), and parameters of mode **out** or **in out** are passed by reference: the address of the parameter object is pushed on the stack.

The results returned by functions are expected in the register **D0** if the result is a scalar, or in **A0** if the result is an access value.

LONG_FLOAT values that are represented as 64 bits are returned in two registers: **D0** contains the low-level word and **D1** contains the high-level word.

Integer Types

When passed by value to an assembler subprogram, values of type LONG_INTEGER and INTEGER are copied, and pushed on the stack without alteration. Values of type SHORT_INTEGER are copied and pushed on the stack as the most significant (leftmost) 8 bits of a 16-bit field; the low-order 8 bits may have any values.

When passed by reference, in each case the value is not altered and a 32-bit address pointer is pushed on the stack.

Boolean Types

Boolean values are represented as 8 bits in Ada (FALSE corresponds to 2#0000_0000#, and TRUE to 2#1111_1111#). When passed by value to an assembler subprogram, values of type BOOLEAN are copied, and pushed on the stack as the most significant (leftmost) 8 bits of a 16-bit field; the low-order 8 bits may have any values.
When passed by reference, in each case the value is not altered and a 32-bit address pointer is pushed on the stack.

Enumeration Types

When passed by value to an assembler subprogram, values of enumeration types represented by 16 bits are copied, and pushed on the stack without alteration. Values of types represented by 8 bits are copied and pushed on the stack as the most significant (leftmost) 8 bits of a 16-bit field; the low-order 8 bits may have any values.
When passed by reference, in each case the value is not altered and a 32-bit address pointer is pushed on the stack.

Real Types

When passed by value to an assembler subprogram, values of types FLOAT and LONG_FLOAT are copied, and pushed on the stack without alteration. When passed by reference, in each case the value is not altered and a 32-bit address pointer is pushed on the stack.

Access, Array, and Record Types

See *Calling External Subprograms, General Aspects*, above

steria

1.3 Calling C Language Subprograms

For an C external subprogram, scalar and access parameters of mode **in** are passed by value: the value of the parameter object is copied and pushed on the stack. All other types of **in** parameters (arrays, records) and parameters of mode **out** or **in out** are passed by reference: the address of the parameter object is pushed on the stack.

Integer Types

When passed by value to a C external subprogram, all integer values [3.5.4] are extended to 32 bits to conform to C parameter passing conventions. Alsys Ada LONG_INTEGER (32 bit) values are pushed on the stack unchanged: INTEGER (16 bit) and SHORT_INTEGER (8 bit) values are sign extended to 32 bits.

Integers passed by reference, however, are not changed: the values are not sign-extended to 32 bits. The value pushed on the stack is the 32 bit address of the appropriate integer.

In passing integers by reference, therefore, it should be recognized that an Ada SHORT_INTEGER has no representation in C, while Ada INTEGER corresponds to C short integer, and Ada LONG_INTEGER corresponds to C integer.

If a SHORT_INTEGER is passed by reference from an Alsys Ada program then the C external subprogram must treat the reference as being a pointer to a type *char*, and not as a pointer to short.

Ada SHORT_INTEGER has no representation in C, while Ada INTEGER corresponds to C short integer, and Ada LONG_INTEGER corresponds to C integer.

Boolean Types

There are no Boolean types in C, and the Ada representation does not correspond to anything in C. The 'POS attribute can be used to convert the boolean to an integer.

Enumeration Types

When an enumeration value is passed by an Alsys Ada program to a C external subprogram, the underlying integer value is passed according to the convention given above for integer values. When passed by value, a copy is pushed on the stack, automatically extended to 32 bits, and when passed by reference, a 32 bit address pointer is pushed on the stack.

Enumeration values in C are represented in the same general way as in Ada, in that the values are always expressed as 16 bit integers (no matter how many elements). When pushed on the stack as parameters, the values are extended to 32 bits, with the high order 16 bits disregarded. The automatic extension to 32 bits means that enumeration values, whether stored in 8 bits or in 16 bits in Ada, are in the correct form for C.

When passed by reference, however, the original values are not extended; therefore 8 bit enumeration objects have no representation in C.

However, the Ada predefined type CHARACTER corresponds directly to the type *char* in C, whether passed by value or by reference.

Real Types

C uses the IEEE P754 floating-point conventions for both 32 bit and 64 bit floating point numbers. All 32 bit values are automatically extended to 64 bit values when passed as parameters.

Floating point parameters can be passed either by value or by reference, with the automatic extension of 32 bit numbers with passing by value.

Access Types

Ada access types have no meaning in C. An object of the designated type can, however, be passed to a C external subprogram, subject to the rules for that type.

Array Types

See *Calling External Subprograms, General Aspects*, above.

Alsys Ada strings do not end with an ASCII null character (\0), as required in C. The programmer must therefore append a null character to the end of a string, if the string is to be passed to a C external subprogram. It might also be necessary to remove the excess character on returning from the subprogram.

Record Types

See *Calling External Subprograms, General Aspects*, above.

2 IMPLEMENTATION-DEPENDENT PRAGMAS

Pragma INTERFACE

This pragma has been described in detail in the previous section.

Pragma IMPROVE and Pragma PACK

These pragmas are discussed in detail in sections 5.7 and 5.8 on representation clauses for arrays and records.

Note that packing of record types is done systematically by the compiler. The pragma pack will affect the mapping of each component onto storage. Each component will be allocated on the logical size of the subtype.

Example:

```
type R is
    record
        C1 : BOOLEAN; C2 : INTEGER range 1 .. 10;
    end record;
pragma PACK(R);
-- the attribute R'SIZE returns 5
```

Pragma INDENT

This pragma is only used with the Alsys *Reformatter*; this tool offers the functionalities of a pretty-printer in an Ada environment.

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

```
pragma INDENT(OFF)
```

causes the Reformatter not to modify the source lines after this pragma.

```
pragma INDENT(ON)
```

causes the Reformatter to resume its action after this pragma.

Pragmas not implemented

The following pragmas are not implemented:

CONTROLLED
MEMORY_SIZE
OPTIMIZE
STORAGE_UNIT
SYSTEM_NAME

3 IMPLEMENTATION-DEPENDENT ATTRIBUTES

In addition to the Representation Attributes of [13.7.2] and [13.7.3], there are four attributes which are listed under F.5 below, for use in record representation clauses.

Limitations on the use of the attribute ADDRESS

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

4 PACKAGES SYSTEM AND STANDARD

This section contains information on two predefined library packages:

- a complete listing of the specification of the package SYSTEM
- a list of the implementation-dependent declarations in the package STANDARD.

The package SYSTEM

The specification of the predefined library package SYSTEM is as follows:

```
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 := 1.0 ;                -- unused

-- The real basic clock cycle depends on the current hardware
-- and corresponding board support package.

    type ADDRESS is private ;
    NULL_ADDRESS : constant ADDRESS ;

    subtype PRIORITY is INTEGER range 1..24 ;   -- 1..248 for VRTX

-- Address arithmetic

    function TO_LONG_INTEGER (LEFT : ADDRESS)
        return LONG_INTEGER ;
    function TO_ADDRESS (LEFT : LONG_INTEGER)
        return ADDRESS ;

    function "+" (LEFT : LONG_INTEGER ; RIGHT : ADDRESS)
        return ADDRESS ;
    function "+" (LEFT : ADDRESS ; RIGHT : LONG_INTEGER)
        return ADDRESS ;

    function "-" (LEFT : ADDRESS ; RIGHT : ADDRESS)
        return LONG_INTEGER ;
    function "-" (LEFT : ADDRESS ; RIGHT : LONG_INTEGER)
        return ADDRESS ;
```

```
function "mod" (LEFT : ADDRESS ; RIGHT : POSITIVE)
      return NATURAL ;

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

function IS_NULL (LEFT : ADDRESS)
      return BOOLEAN ;

function WORD_ALIGNED (LEFT : ADDRESS)
      return BOOLEAN ;

function ROUND (LEFT : ADDRESS)
      return ADDRESS ;
-- Returns the given address rounded to the next lower even value

procedure COPY (FROM: ADDRESS ;TO: ADDRESS ;SIZE: NATURAL) ;
-- Copies SIZE storage units. The result is undefined if the two areas
-- overlap.

-- Direct memory access

generic
      type ELEMENT_TYPE is private ;
function FETCH (FROM : ADDRESS) return ELEMENT_TYPE ;
-- Returns the bit pattern stored at address FROM, as a value of the
-- specified ELEMENT_TYPE. This function is not implemented
-- for unconstrained array types.

generic
      type ELEMENT_TYPE is private ;
procedure STORE (INTO : ADDRESS ; OBJECT : ELEMENT_TYPE) ;
-- Stores the bit pattern representing the value of OBJECT, at
-- address INTO. This function is not implemented for
-- unconstrained array types.

end SYSTEM ;
```

The package STANDARD

The following are the implementation-dependent aspects of the package STANDARD:

```
type SHORT_INTEGER is range -(2**7) .. (2**7 -1);
type INTEGER is range -(2**15) .. (2**15 -1);
type LONG_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**(-51)) * 2.0**1023 ..
        +(2.0 - 2.0**(-51)) * 2.0**1023;

type DURATION is delta 2.0**(-14) range -86_400.0 .. 86_400.0;
```

5 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.

5.1 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 RM 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 or is 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.

Minimum size of an enumeration subtype

The minimum size of an enumeration subtype is the minimum number of bits that is necessary for representing the internal codes of the subtype values in normal binary form.

For a static subtype, if it has a null range its minimum size is 1. Otherwise, if m and M are the values of the internal codes associated with the first and last enumeration values of the subtype, then its minimum size L is determined as follows. For m >= 0, L is the smallest positive integer such that $M <= 2^{L}-1$. For $m < 0$, L is the smallest positive integer such that $-2^{L-1} <= m$ and $M <= 2^{L-1}-1$.

```
type COLOR is (GREEN, BLACK, WHITE, RED, BLUE, YELLOW);
-- The minimum size of COLOR is 3 bits.

subtype BLACK_AND_WHITE is COLOR range BLACK .. WHITE;
-- The minimum size of BLACK_AND_WHITE is 2 bits.

subtype BLACK_OR_WHITE is BLACK_AND_WHITE range X .. X;
-- Assuming that X is not static, the minimum size of BLACK_OR_WHITE is
-- 2 bits (the same as the minimum size of its type mark BLACK_AND_WHITE).
```

Size of an enumeration subtype

When no size specification is applied to an enumeration type or first named subtype, the objects of that type or first named subtype are represented as signed machine integers. The machine provides 8, 16 and 32 bit integers, and the compiler selects automatically the smallest signed machine integer which can hold each of the internal codes of the enumeration type (or subtype). The size of the enumeration type and of any of its subtypes is thus 8, 16 or 32 bits.

When a size specification is applied to an enumeration type, this enumeration type and each of its subtypes has the size specified by the length clause. The same rule applies to a first named subtype. The size specification must of course specify a value greater than or equal to the minimum size of the type or subtype to which it applies:

```
type EXTENDED is
  (        -- The usual American ASCII characters.
      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,
      -- Extended characters
      LEFT_ARROW,
      RIGHT_ARROW,
      UPPER_ARROW,
      LOWER_ARROW,
      UPPER_LEFT_CORNER,
      UPPER_RIGHT_CORNER,
      LOWER_RIGHT_CORNER,
      LOWER_LEFT_CORNER
      );

for EXTENDED'SIZE use 8;
-- The size of type EXTENDED will be one byte. Its objects will be represented
-- as unsigned 8 bit integers.
```

The Alsys compiler fully implements size specifications. Nevertheless, as enumeration values are coded using integers, the specified length cannot be greater than 32 bits.

Size of the objects of an enumeration subtype

Provided its size is not constrained by a record component clause or a pragma PACK, an object of an enumeration subtype has the same size as its subtype.

Alignment of an enumeration subtype

An enumeration subtype is byte aligned if the size of the subtype is less than or equal to 8 bits, it is otherwise even byte aligned.

Address of an object of an enumeration subtype

Provided its alignment is not constrained by a record representation clause or a pragma PACK, the address of an object of an enumeration subtype is even when its subtype is even byte aligned.

5.2 Integer Types

Predefined integer types

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

```
type SHORT_INTEGER    is range -2**07 .. 2**07-1;
type INTEGER          is range -2**15 .. 2**15-1;
type LONG_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.

Minimum size of an integer subtype

The minimum size of an integer subtype is the minimum number of bits that is necessary for representing the internal codes of the subtype values in normal binary form.

For a static subtype, if it has a null range its minimum size is 1. Otherwise, if m and M are the lower and upper bounds of the subtype, then its minimum size L is determined as follows. For $m >= 0$, L is the smallest positive integer such that $M <= 2^{L}-1$. For $m < 0$, L is the smallest positive integer that $-2^{L-1} <= m$ and $M <= 2^{L-1}-1$.

```
subtype S is INTEGER range 0 .. 7;
-- The minimum size of S is 3 bits.

subtype D is S range X .. Y;
-- Assuming that X and Y are not static, the minimum size of
-- D is 3 bits (the same as the minimum size of its type mark S).
```

Size of an integer subtype

The sizes of the predefined integer types SHORT_INTEGER, INTEGER and LONG_INTEGER are respectively 8, 16 and 32 bits.

When no size specification is applied to an integer type or to its first named subtype (if any), its size and the size of any of its subtypes is the size of the predefined type from which it derives, directly or indirectly. For example:

```
type S is range 80 .. 100;
-- S is derived from SHORT_INTEGER, its size is 8 bits.

type J is range 0 .. 255;
-- J is derived from INTEGER, its size is 16 bits.

type N is new J range 80 .. 100;
-- N is indirectly derived from INTEGER, its size is 16 bits.
```

When a size specification is applied to an integer type, this integer type and each of its subtypes has the size specified by the length clause. The same rule applies to a first named subtype. The size specification must of course specify a value greater than or equal to the minimum size of the type or subtype to which it applies:

```
type S is range 80 .. 100;
for S'SIZE use 32;
-- S is derived from SHORT_INTEGER, but its size is 32 bits
-- because of the size specification.

type J is range 0 .. 255;
for J'SIZE use 8;
-- J is derived from INTEGER, but its size is 8 bits because
-- of the size specification.

type N is new J range 80 .. 100;
-- N is indirectly derived from INTEGER, but its size is 8 bits
-- because N inherits the size specification of J.
```

The Alsys compiler fully implements size specifications. Nevertheless, as integers are implemented using machine integers, the specified length cannot be greater than 32 bits.

Size of the objects of an integer subtype

Provided its size is not constrained by a record component clause or a pragma PACK, an object of an integer subtype has the same size as its subtype.

Alignment of an integer subtype

An integer subtype is byte aligned if the size of the subtype is less than or equal to 8 bits, it is otherwise even byte aligned.

Address of an object of an integer subtype

Provided its alignment is not constrained by a record representation clause or a pragma PACK, the address of an object of an integer subtype is even when its subtype is even byte aligned.

5.3 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.

Minimum size of a floating point subtype

The minimum size of a floating point subtype is 32 bits if its base type is FLOAT or a type derived from FLOAT; it is 64 bits if its base type is LONG_FLOAT or a type derived from LONG_FLOAT.

Size of a floating point subtype

The sizes of the predefined floating point types FLOAT and LONG_FLOAT are respectively 32 and 64 bits.

The size of a floating point type and the size of any of its subtypes is the size of the predefined type from which it derives directly or indirectly.

The only size that can be specified for a floating point type or first named subtype using a size specification is its usual size (32 or 64 bits).

Size of the objects of a floating point subtype

An object of a floating point subtype has the same size as its subtype.

Alignment of a floating point subtype

A floating point subtype is always even byte aligned.

Address of an object of a floating point subtype

Provided its alignment is not constrained by a record representation clause or a pragma PACK, the address of an object of a floating point subtype is always even, since its subtype is even byte aligned.

5.4 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 RM 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 SHORT_FIXED     is delta D range (-2**07-1)*S .. 2**07*S;
for SHORT_FIXED'SMALL use S;

type FIXED           is delta D range (-2**15-1)*S .. 2**15*S;
for FIXED'SMALL use S;

type LONG_FIXED      is delta D range (-2**31-1)*S .. 2**31*S;
for LONG_FIXED'SMALL use S;
```

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

Selection of the parent of a fixed point type

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

type T **is delta** D **range** L .. R;

possibly with a small specification:

for T'SMALL **use** S;

is implicitly derived from a predefined fixed point type. The compiler automatically selects the predefined fixed point type whose small and delta are the same as the small and delta of T and whose range is the shortest that includes the values L to R inclusive.

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

Minimum size of a fixed point subtype

The minimum size of a fixed point subtype is the minimum number of binary digits that is necessary for representing the values of the range of the subtype using the small of the base type.

For a static subtype, if it has a null range its minimum size is 1. Otherwise, s and S being the bounds of the subtype, if i and I are the integer representations of m and M, the smallest and the greatest model numbers of the base type such that s < m and M < S, then the minimum size L is determined as follows. For i >= 0, L is the smallest positive integer such that $I <= 2^{L}-1$. For i < 0, L is the smallest positive integer such that $-2^{L-1} <= i$ and $I <= 2^{L-1}-1$.

type F **is delta** 2.0 **range** 0.0 .. 500.0;
-- The minimum size of F is 8 bits.

subtype S **is** F **delta** 16.0 **range** 0.0 .. 250.0;
-- The minimum size of S is 7 bits.

subtype D **is** S **range** X .. Y;
-- Assuming that X and Y are not static, the minimum size of D is 7 bits
-- (the same as the minimum size of its type mark S).

Size of a fixed point subtype

The sizes of the predefined fixed point types *SHORT_FIXED*, *FIXED* and *LONG_FIXED* are respectively 8, 16 and 32 bits.

When no size specification is applied to a fixed point type or to its first named subtype, its size and the size of any of its subtypes is the size of the predefined type from which it derives directly or indirectly. For example:

```
type S is delta 0.01 range 0.8 .. 1.0;
-- S is derived from an 8 bit predefined fixed type, its size is 8 bits.

type F is delta 0.01 range 0.0 .. 2.0;
-- F is derived from a 16 bit predefined fixed type, its size is 16 bits.

type N is new F range 0.8 .. 1.0;
-- N is indirectly derived from a 16 bit predefined fixed type, its size is 16 bits.
```

When a size specification is applied to a fixed point type, this fixed point type and each of its subtypes has the size specified by the length clause. The same rule applies to a first named subtype. The size specification must of course specify a value greater than or equal to the minimum size of the type or subtype to which it applies:

```
type S is delta 0.01 range 0.8 .. 1.0;
for S'SIZE use 32;
-- S is derived from an 8 bit predefined fixed type, but its size is 32 bits
-- because of the size specification.

type F is delta 0.01 range 0.0 .. 2.0;
for F'SIZE use 8;
-- F is derived from a 16 bit predefined fixed type, but its size is 8 bits
-- because of the size specification.

type N is new F range 0.8 .. 1.0;
-- N is indirectly derived from a 16 bit predefined fixed type, but its size is
-- 8 bits because N inherits the size specification of F.
```

The Alsys compiler fully implements size specifications. Nevertheless, as fixed point objects are represented using machine integers, the specified length cannot be greater than 32 bits.

Size of the objects of a fixed point subtype

Provided its size is not constrained by a record component clause or a pragma PACK, an object of a fixed point type has the same size as its subtype.

Alignment of a fixed point subtype

A fixed point subtype is byte aligned if its size is less than or equal to 8 bits, and is otherwise even byte aligned.

Address of an object of a fixed point subtype

Provided its alignment is not constrained by a record representation clause or a pragma PACK, the address of an object of a fixed point subtype is even when its subtype is even byte aligned.

5.5 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 RM 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.

Minimum size of an access subtype

The minimum size of an access subtype is 32 bits.

Size of an access subtype

The size of an access subtype is 32 bits, the same as its minimum size.

The only size that can be specified for an access type using a size specification is its usual size (32 bits).

Size of an object of an access subtype

An object of an access subtype has the same size as its subtype, thus an object of an access subtype is always 32 bits long.

Alignment of an access subtype.

An access subtype is always even byte aligned.

Address of an object of an access subtype

Provided its alignment is not constrained by a record representation clause or a pragma PACK, the address of an object of an access subtype is always even, since its subtype is even byte aligned.

5.6 Task Types

Storage for a task activation

A stack is allocated to each Ada task for the duration of its lifetime. Stacks are allocated on the heap.

A length clause STORAGE_SIZE may be given for a task. It defines the stack size for execution of the task. If the specified size is too small, an exception STORAGE_ERROR will be raised at runtime.

Encoding of task values.

Encoding of a task value is not described here.

Minimum size of a task subtype

The minimum size of a task subtype is 32 bits.

Size of a task subtype

The size of a task subtype is 32 bits, the same as its minimum size.

A size specification has no effect on a task type. The only size that can be specified using such a length clause is its minimum size.

Size of the objects of a task subtype

An object of a task subtype has the same size as its subtype. Thus an object of a task subtype is always 32 bits long.

Alignment of a task subtype

A task subtype is always even byte aligned.

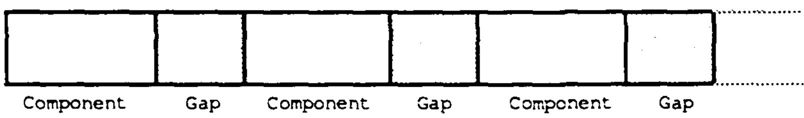
Address of an object of a task subtype

Provided its alignment is not constrained by a record representation clause, the address of an object of a task subtype is always even, since its subtype is even byte aligned.

5.7 Array Types

Layout of an array

Each array is allocated in a contiguous area of storage units. All the components have the same size. A gap may exist between two consecutive components (and after the last one). All the gaps have the same size.



COMPONENTS

If the array is not packed, the size of the components is the size of the subtype of the components:

```
type A is array (1 .. 8) of BOOLEAN;
-- The size of the components of A is the size of the type BOOLEAN: 8 bits.

type DECIMAL_DIGIT is range 0 .. 9;
for DECIMAL_DIGIT'SIZE use 4;
type BINARY_CODED_DECIMAL is
      array (INTEGER range <>) of DECIMAL_DIGIT;
-- The size of the type DECIMAL_DIGIT is 4 bits. Thus in an array of
-- type BINARY_CODED_DECIMAL each component will be represented on
-- 4 bits as in the usual BCD representation.
```

If the array is packed and its components are neither records nor arrays, the size of the components is the minimum size of the subtype of the components:

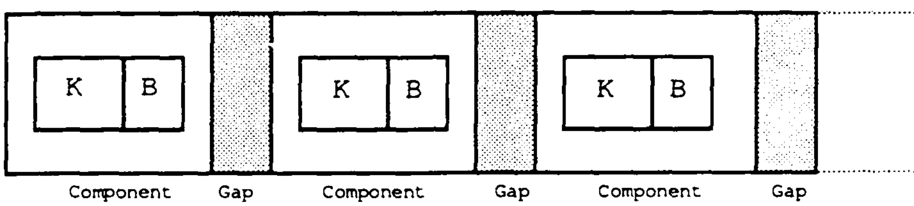
```
type A is array (1 .. 8) of BOOLEAN;
pragma PACK(A);
-- The size of the components of A is the minimum size of the type BOOLEAN:
-- 1 bit.

type DECIMAL_DIGIT is range 0 .. 9;
for DECIMAL_DIGIT'SIZE use 32;
type BINARY_CODED_DECIMAL is
      array (INTEGER range <>) of DECIMAL_DIGIT;
pragma PACK(BINARY_CODED_DECIMAL);
-- The size of the type DECIMAL_DIGIT is 32 bits, but, as
-- BINARY_CODED_DECIMAL is packed, each component of an array of this
-- type will be represented on 4 bits as in the usual BCD representation.
```

Packing the array has no effect on the size of the components when the components are records or arrays.

GAPS

If the components are records or arrays, no size specification applies to the subtype of the components and the array is not packed, then the compiler may choose a representation with a gap after each component; the aim of the insertion of such gaps is to optimize access to the array components and to their subcomponents. The size of the gap is chosen so that the relative displacement of consecutive components is a multiple of the alignment of the subtype of the components. This strategy allows each component and subcomponent to have an address consistent with the alignment of its subtype:

```
type R is
    record
        K : INTEGER; -- INTEGER is even byte aligned.
        B : BOOLEAN; -- BOOLEAN is byte aligned.
    end record;
-- Record type R is even byte aligned. Its size is 24 bits.

type A is array (1 .. 10) of R;
-- A gap of one byte is inserted after each component in order to respect the
-- alignment of type R. The size of an array of type A will be 320 bits.
```

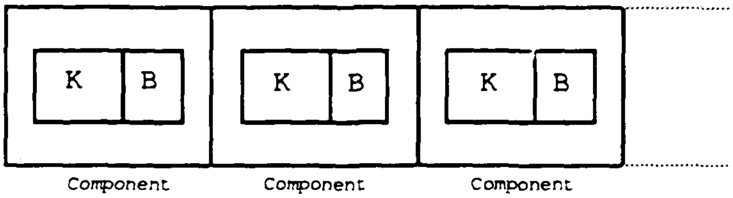


Array of type A: each subcomponent K has an even offset.

If a size specification applies to the subtype of the components or if the array is packed, no gaps are inserted:

```
type R is
    record
        K : INTEGER;
        B : BOOLEAN;
    end record;

type A is array (1 .. 10) of R;
pragma PACK(A);
-- There is no gap in an array of type A because
-- A is packed.
-- The size of an object of type A will be 240 bits.

type NR is new R;
for NR'SIZE use 24;
```

```
type B is array (1 .. 10) of NR;
-- There is no gap in an array of type B because
-- NR has a size specification.
-- The size of an object of type B will be 240 bits.
```



Array of type A or B: a subcomponent K can have an odd offset.

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.

Size of the objects of an array subtype

The size of an object of an array subtype is always equal to the size of the subtype of the object.

Alignment of an array subtype

If no pragma PACK applies to an array subtype and no size specification applies to its components, the array subtype is even byte aligned if the subtype of its components is even byte aligned. Otherwise it is byte aligned.

If a pragma PACK applies to an array subtype or if a size specification applies to its components (so that there are no gaps), the alignment of the array subtype is as given in the following table:

		relative displacement of components		
		even number of bytes	odd number of bytes	not a whole number of bytes
Component subtype alignment	even byte	even byte	byte	bit
	byte	byte	byte	bit
	bit	bit	bit	bit

Address of an object of an array subtype

Provided its alignment is not constrained by a record representation clause, the address of an object of an array subtype is even when its subtype is even byte aligned.

5.8 Record Types

Layout of a record

Each record is allocated in a contiguous area of storage units. The size of a record component depends on its type. Gaps may exist between some components.

The positions and the sizes of the components of a record type object can be controlled using a record representation clause as described in RM 13.4. In the Alsys implementation for MC680X0 machines there is no restriction on the position that can be specified for a component of a record. If a component is not a record or an array, its size can be any size from the minimum size to the size of its subtype. If a component is a record or an array, its size must be the size of its subtype:

```
type INTERRUPT_MASK is array (0 .. 2) of BOOLEAN;
pragma PACK(INTERRUPT_MASK);
-- The size of INTERRUPT_MASK is 3 bits.

type CONDITION_CODE is 0 .. 1;
-- The size of CONDITION_CODE is 8 bits, its minimum size is 1 bit.
```

```
type STATUS_BIT is new BOOLEAN;
for STATUS_BIT'SIZE use 1;
-- The size and the minimum size of STATUS_BIT are 1 bit.

SYSTEM   : constant := 0;
USER     : constant := 1;

type STATUS_REGISTER is
      record
            T : STATUS_BIT;               -- Trace
            S : STATUS_BIT;               -- Supervisor
            I : INTERRUPT_MASK;           -- Interrupt mask
            X : CONDITION_CODE;           -- Extend
            N : CONDITION_CODE;           -- Negative
            Z : CONDITION_CODE;           -- Zero
            V : CONDITION_CODE;           -- Overflow
            C : CONDITION_CODE;           -- Carry
      end record;
-- This type can be used to map the status register of a MC68000 processor:

for STATUS_REGISTER use
      record at mod 2;
            T    at SYSTEM    range 0 .. 0;
            S    at SYSTEM    range 2 .. 2;
            I    at SYSTEM    range 5 .. 7;
            X    at USER      range 3 .. 3;
            N    at USER      range 4 .. 4;
            Z    at USER      range 5 .. 5;
            V    at USER      range 6 .. 6;
            C    at USER      range 7 .. 7;
      end record;
```

A record representation clause need not specify the position and the size for every component.

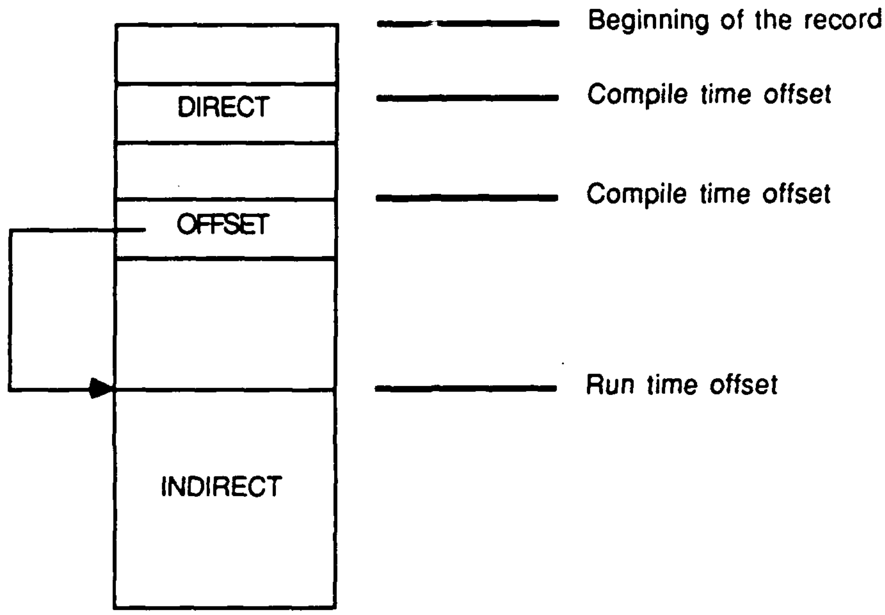
If no component clause applies to a component of a record, its size is the size of its subtype. Its position is chosen by the compiler so as to optimize access to the components of the record: the offset of the component is chosen as a multiple of 8 bits if the objects of the component subtype are usually byte aligned, but a multiple of 16 bits if these objects are usually even byte aligned. Moreover, the compiler chooses the position of the component so as to reduce the number of gaps and thus the size of the record objects.

Because of these optimizations, there is no connection between the order of the components in a record type declaration and the positions chosen by the compiler for the components in a record object.

In the current version, it is not possible to apply a record representation clause to a derived type. The same storage representation is used for an object of a derived type as for an object of the parent type.

Indirect components

If the offset of a component cannot be computed at compile time, this offset is stored in the record objects at run time and used to access the component. Such a component is said to be indirect while other components are said to be direct:

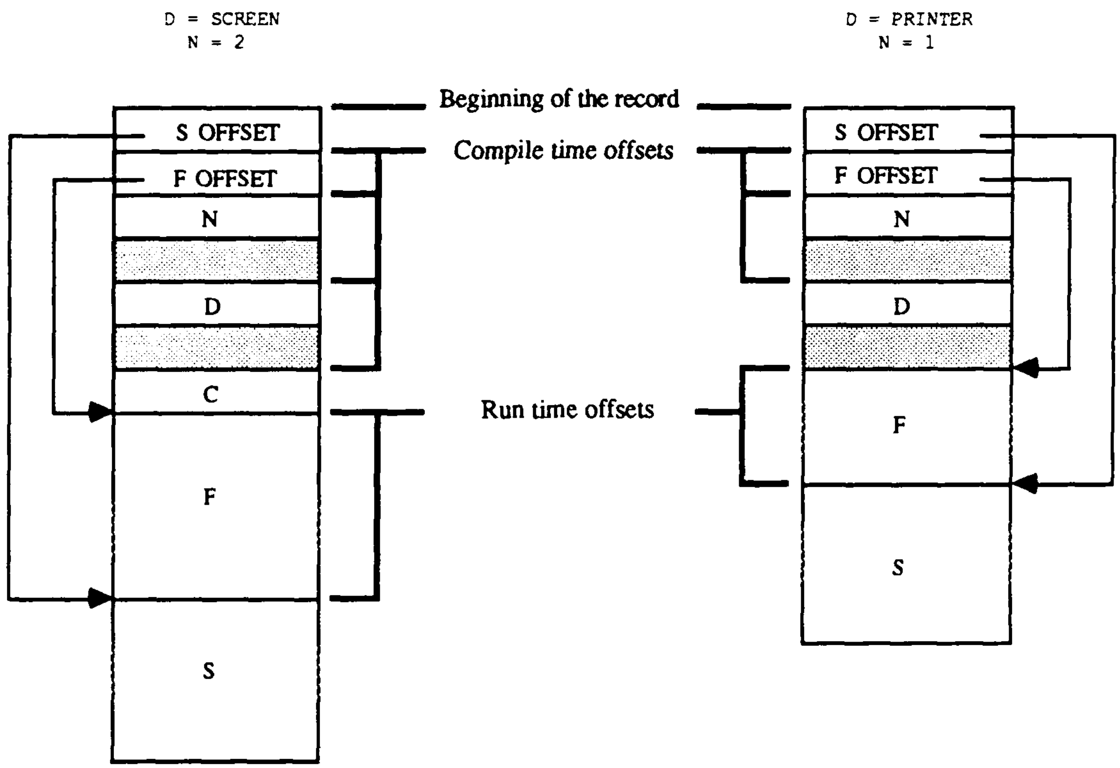


A direct and an indirect component

If a record component is a record or an array, the size of its subtype may be evaluated at run time and may even depend on the discriminants of the record. We will call these components dynamic components:

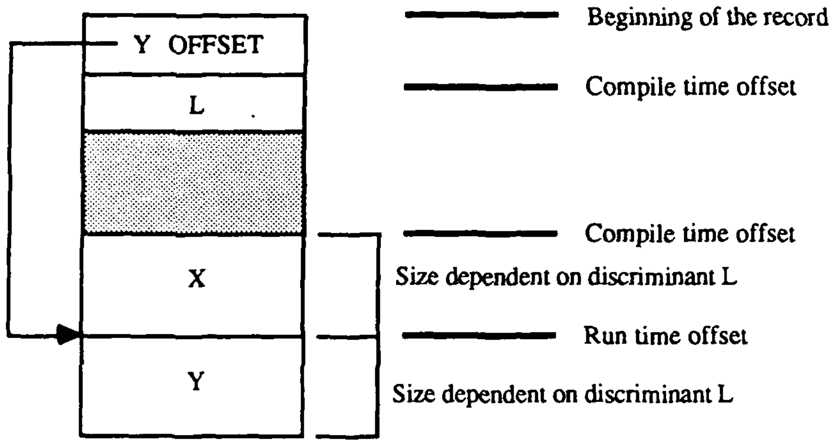
```
type DEVICE is (SCREEN, PRINTER);

type COLOR is (GREEN, RED, BLUE);

type SERIE is array (POSITIVE range <>) of INTEGER;

type GRAPH (L : NATURAL) is
      record
            X : SERIE(1 .. L); -- The size of X depends on L
            Y : SERIE(1 .. L); -- The size of Y depends on L
      end record;

Q : POSITIVE;

type PICTURE (N : NATURAL; D : DEVICE) is
      record
            F : GRAPH(N); -- The size of F depends on N
            S : GRAPH(Q); -- The size of S depends on Q
            case D is
                  when SCREEN =>
                        C : COLOUR;
                  when PRINTER =>
                        null;
            end case;
      end record;
```

Any component placed after a dynamic component has an offset which cannot be evaluated at compile time and is thus indirect. In order to minimize the number of indirect components, the compiler groups the dynamic components together and places them at the end of the record:



The record type PICTURE: F and S are placed at the end of the record

Thanks to this strategy, the only indirect components are dynamic components. But not all dynamic components are necessarily indirect: if there are dynamic components in a component list which is not followed by a variant part, then exactly one dynamic component of this list is a direct component because its offset can be computed at compilation time (the only dynamic components that are direct components are in this situation):



The record type GRAPH; the dynamic component X is a direct component.

The offset of an indirect component is always expressed in storage units.

The space reserved for the offset of an indirect component must be large enough to store the size of any value of the record type (the maximum potential offset). The compiler evaluates an upper bound MS of this size and treats an offset as a component having an anonymous integer type whose range is 0 .. MS.

If C is the name of an indirect component, then the offset of this component can be denoted in a component clause by the implementation generated name C'OFFSET.

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 component 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.

Component lists that do not contain a variant part are numbered. These numbers are the possible values of the implicit component VARIANT_INDEX.

```
type VEHICLE is (AIRCRAFT, ROCKET, BOAT, CAR);

type DESCRIPTION (KIND : VEHICLE := CAR) is
      record
            SPEED : INTEGER;
            case KIND is
                  when AIRCRAFT | CAR =>
                        WHEELS : INTEGER;
                        case KIND is
                              when AIRCRAFT =>        -- 1
                                    WINGSPAN : INTEGER;
                              when others =>   -- 2
                                    null;
                        end case;
                  when BOAT =>                 -- 3
                        STEAM  : BOOLEAN;
                  when ROCKET =>               -- 4
                        STAGES : INTEGER;
            end case;
      end record;
```

The value of the variant index indicates the set of components that are present in a record value:

Variant Index	Set
1	{KIND, SPEED, WHEELS, WINGSPAN}
2	{KIND, SPEED, WHEELS}
3	{KIND, SPEED, STEAM}
4	{KIND, SPEED, STAGES}

A comparison between the variant index of a record value and the bounds of an interval is enough to check that a given component is present in the value:

Component	Interval
KIND	- -
SPEED	- -
WHEELS	1 .. 2
WINGSPAN	1 .. 1
STEAM	3 .. 3
STAGES	4 .. 4

The implicit component VARIANT_INDEX must be large enough to store the number V of component lists that don't contain variant parts. The compiler treats this implicit component as having an anonymous integer type whose range is 1 .. V.

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'VARIANT_INDEX.

- *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 subtype 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.

Suppression of implicit components

The Alsys implementation provides the capability of suppressing the implicit components RECORD_SIZE and/or VARIANT_INDEX from a record type. This can be done using an implementation defined pragma called IMPROVE. The syntax of this pragma is as follows:

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

The first argument specifies whether TIME or SPACE is the primary criterion for the choice of the representation of the record type that is denoted by the second argument.

If TIME is specified, the compiler inserts implicit components as described above. If on the other hand SPACE is specified, the compiler only inserts a VARIANT_INDEX or a RECORD_SIZE component if this component appears in a record representation clause that applies to the record type. A record representation clause can thus be used to keep one implicit component while suppressing the other.

A pragma IMPROVE that applies to a given record type can occur anywhere that a representation clause is allowed for this type.

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.

Size of an object of a record subtype

An object of a constrained record subtype has the same size as its subtype.

An object of an unconstrained record subtype has the same size as its subtype if this size is less than or equal to 8 kb. If the size of the subtype is greater than this, the object has the size necessary to store its current value; storage space is allocated and released as the discriminants of the record change.

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.

Address of an object of a record subtype

Provided its alignment is not constrained by a representation clause, the address of an object of a record subtype is even when its subtype is even byte aligned.

6 ADDRESS CLAUSES

6.1 Address Clauses for Objects

An address clause can be used to specify an address for an object as described in RM 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 using the address specified in the clause.

An address clause is not allowed for task objects, nor for unconstrained records whose size is greater than 8 kb.

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 interrupt can be associated with an Ada task entry so that generation of the interrupt causes a call of the corresponding entry (see RM section 13.5.1).

Interrupt Entries

Several interrupt entries may be declared in the specification of a task. Furthermore:

- An interrupt entry may have several **in** parameters.
- Interrupt entries are software callable.
- Interrupts can be persistent or volatile (conditional or normal entry calls).
- Prioritized interrupts are supported.

Execution of the accept body of an interrupt entry.is performed at priority level 24 + hardware priority level.

An Interrupt Handling Example

To illustrate interrupt handling, consider the following example of a basic handler:

```
with SYSTEM;
procedure MAIN is
    task HANDLER is
        pragma PRIORITY(10);
        -- assume that the following entry is
        -- called by an interrupt with hardware level 3
        entry INTERRUPT    (STATUS     : BOOLEAN;
                            DATA       : INTEGER);
        for INTERRUPT use at SYSTEM.TO_ADDRESS(16#160#);
    end HANDLER;

    task body HANDLER is
    begin
        loop
            ...
            -- this code executes at priority 10
            accept INTERRUPT    (STATUS : BOOLEAN;
                                 DATA       : INTEGER) do
                ...
                -- this code executes at priority
                -- level 24+3 = 27
            end INTERRUPT;
            ...
            -- this code executes at priority 10
        end loop;
    end HANDLER;
end MAIN;
```

7 UNCHECKED CONVERSIONS

Unconstrained arrays are not allowed as target types. Unconstrained record types without defaulted discriminants are not allowed as target types. Access to unconstrained arrays are not allowed as target or source types.

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.

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 (IO) 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 FORM appear explicitly in the Ada CREATE and OPEN procedures. 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 IO 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 (text, ASCII) files.
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 generic packages SEQUENTIAL_IO and DIRECT_IO are implemented, but the ALSYS RUNTIME KERNEL does not offer file management. However, the user can adapt the *board support package* to offer a full implementation of input-output features.

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 fields widths in TEXT_IO is given by

FIELD'LAST = 255

8.2 The FORM Parameter

The FORM parameter to both the CREATE and OPEN procedures in Ada specifies the characteristics of the external file involved. A complete description is given here, but as indicated above, the kernel must be adapted to offer a full implementation. With the basic kernel, the FORM parameter is decoded, but not used.

The CREATE procedure 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 certain 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.

The OPEN procedure 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 an Ada USE_ERROR is raised.

The FORM parameter 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). The attributes specify:

- File sharing
- File structuring
- Appending

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

KEYWORD

KEYWORD => QUALIFIER

We will discuss each attribute in order.

File Sharing

An external file can be shared, that means associated simultaneously with several logical file objects created by the OPEN and CREATE procedures.

The file_sharing attribute may restrict or suppress this capability by specifying one of the following access modes:

NOT_SHARED
- exclusive access - no other logical file may be associated with the external file

SHARED => READERS
only logical files opened with IN mode are allowed

SHARED => SINGLE_WRITER
only logical files opened with IN mode and at most one with INOUT or OUT mode are allowed

SHARED => ANY
no restriction

The default value for the file-sharing attribute is *SHARED => ANY.*

File Structuring

. Text Files

There is no FORM parameters to define the structure of text files.

A text file consists in a sequence of bytes holding the ASCII code of characters.

The representation of Ada-terminators depends on the file's mode (IN or OUT) and whether it is associated with a terminal device or a mass-storage file:

. Mass-storage files

end of line: ASCII.CR
end of page: ASCII.CR ASCII.FF
end of file: ASCII.CR ASCII.FF ASCII.EOT

. Terminal device / IN mode

end of line: ASCII.CR
end of page: ASCII.FF
end of file: ASCII.EOT

. Terminal device / OUT mode

end of line: ASCII.CR
end of page: ASCII.CR ASCII.FF
end of file: ASCII.CR ASCII.FF ASCII.EOT

. Binary files

Two FORM attributes, *RECORD_SIZE* and *RECORD_UNIT*, control the structure of binary files.

Such a file can be viewed as a sequence (sequential access) or a set (direct access) of consecutive RECORDS.

The structure of a record is

[HEADER] OBJECT [UNUSED_PART]

formed from up to three items:

- an OBJECT with the exact binary representation of the ADA object in the executable program, possibly including an object descriptor

- a HEADER consisting of two fields (each of 32 bits)
  the length of the object in bytes
  the length of the descriptor in bytes

- an UNUSED_PART of variable size to permit full control of the record's size

The HEADER is implemented only if the actual parameter of the instantiation of the IO package is unconstrained

The file structure attributes take the form:

RECORD_SIZE => size_in_bytes

RECORD_UNIT => size_in_bytes

Their meaning depends on the object's type (constrained or not) and the file access mode (sequential or direct access):

- If the object's type is constrained

  the *RECORD_UNIT* attribute is illegal

  if the *RECORD_SIZE* attribute is omitted, no UNUSED_PART will be implemented: the default *RECORD_SIZE* is the object's size

  if present, the *RECORD_SIZE* attribute must specify a record size greater than or equal to the object's size, otherwise the exception USE_ERROR will be raised

- If the object's type is unconstrained and the file access mode is direct i/o

  the *RECORD_UNIT* attribute is illegal

  the *RECORD_SIZE* attribute has no default value, and if not specified, a USE_ERROR will be raised

  An attempt to input or output an object larger than the given *RECORD_SIZE* will raise a DATA_ERROR exception

- If the object's type is unconstrained and the file access mode is sequential i/o

  the *RECORD_SIZE* attribute is illegal

  the default value of the *RECORD_UNIT* attribute is 1 (byte)

  the record size will be the smallest multiple of the specified (or default) RECORD_UNIT that holds the object and its length. This is the only case where records of a file may have different sizes.

Appending

Only to be used with the procedure OPEN, the format of this attribute is simply

APPEND

and any output will be placed at the end of the named external file.

In normal circumstances, when an external file is opened, an index is set which points to the beginning of the file. If the *APPEND* attribute is present for a sequential or for a text file, then data transfer will commence at the end of the file. For a direct access file, the value of the index is set to one more than the number of records in the external file.

This attribute is not applicable to terminal devices.