AVF Control Number: AVF-VSR-90502/37

Ada* Compiler
VALIDATION SUMMARY REPORT:
Certificate Number: #871210N1.09012
Alsys Ltd
AlsyCOMP_013, Version 3.2
IBM PC/AT x IBM 370 3084Q

Completion of On-Site Testing:
10 December 1987

Prepared By:
The National Computing Centre Limited
Oxford Road
Manchester M1 7ED
United Kingdom

Prepared For:
Ada Joint Program Office
United States Department of Defense
Washington, D.C. 20301-3081

*Ada is a registered trademark of the United States Government (Ada Joint Program Office).

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

ADA199014

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER | 2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE *(and Subtitle)* Ada Compiler Validation Summary Report: Alsys Ltd., AlsyCOMP_013, Version 3.2, IBM PC/AT (Host) and IBM-370 3084Q (Target).	5. TYPE OF REPORT & PERIOD COVERED 10 Dec 1987 to 10 Dec 1988 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) National Computing Centre Limited, Manchester, United Kingdom.	8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION AND ADDRESS National Computing Centre Limited, Manchester, United Kingdom.	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Ada Joint Program Office United States Department of Defense Washington, DC 20301-3081	12. REPORT DATE 10 December 1988 13. NUMBER OF PAGES 72 p.
14. MONITORING AGENCY NAME & ADDRESS(*If different from Controlling Office*) National Computing Centre Limited, Manchester, United Kingdom.	15. SECURITY CLASS *(of this report)* UNCLASSIFIED 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A

16. DISTRIBUTION STATEMENT *(of this Report)*

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20. If different from Report)*

UNCLASSIFIED

18. SUPPLEMENTARY NOTES

19. KEYWORDS *(Continue on reverse side if necessary and identify by block number)*

Ada Programming language, Ada Compiler Validation Summary Report, Ada Compiler Validation Capability, ACVC, Validation Testing, Ada Validation Office, AVO, Ada Validation Facility, AVF, ANSI/MIL-STD-1815A, Ada Joint Program Office, AJPO

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

AlsyCOMP_013, Version 3.2, Alsys Ltd., National Computing Centre Limited, IBM PC/AT (Host) under PC/DOS Version 3.1 and IBM 370 3084Q under MVS Version 3.2 (Target), ,ACVC 1.9.

Ada* Compiler Validation Summary Report:

Compiler Name: AlsyCOMP_013, Version 3.2

Certificate Number: #871210N1.09012

Host:	Target:
IBM PC/AT under	IBM 370 3084Q under
PC/DOS	MVS
Version 3.1	Version 3.2

Testing Completed 10 December 1987 Using ACVC 1.9

This report has been reviewed and is approved.

A.E.J.Pink

The National Computing Centre Ltd
Jane Pink
Oxford Road
Manchester M1 7ED
United Kingdom

John F Kramer

Ada Validation Organization
Dr. John F. Kramer
Institute for Defense Analyses
Alexandria VA 22311

Virginia L. Castor

Ada Joint Program Office
Virginia L. Castor
Director
Department of Defense
Washington DC 20301

*Ada is a registered trademark of the United States Government (Ada Joint Program Office).

EXECUTIVE SUMMARY

This Validation Summary Report (VSR) summarizes the results and conclusions of validation testing performed on the AlsyCOMP_013, Version 3.2, using Version 1.9 of the Ada* Compiler Validation Capability (ACVC). The AlsyCOMP_013 is hosted on an IBM PC/AT operating under PC/DOS, Version 3.1. Programs processed by this compiler may be executed on an IBM 370 3084Q operating under MVS Version 3.2.

On-site testing was performed December 1987 through December 1987 at Alsys Ltd, Partridge House, Newtown Road, Henley on Thames under the direction of the NCC (AVF), according to Ada Validation Organisation (AVO) policies and procedures. At the time of testing, version 1.9 of the ACVC comprised 3122 tests of which 25 had been withdrawn. Of the remaining tests, 205 were determined to be inapplicable to this implementation. Not all of the inapplicable tests were processed during testing: 145 executable tests that use floating-point precision exceeding that supported by the implementation were not processed Results for processed Class A, C, D, and E tests were examined for correct execution. Compilation listings for Class B tests were analyzed for correct diagnosis of syntax and semantic errors. Compilation and link results of Class L tests were analyzed for correct detection of errors. There were 60 of the processed tests determined to be inapplicable. The remaining 2892 tests were passed. The results of validation are summarized in the following table:

RESULT	CHAPTER													TOTAL
	2	3	4	5	6	7	8	9	10	11	12	13	14	
Passed	193	516	564	245	166	98	141	327	135	36	234	3	234	2892
Failed	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Inapplicable	11	57	111	3	0	0	2	0	2	0	0	0	19	205
Withdrawn	2	13	2	0	0	1	2	0	0	0	2	1	2	25
TOTAL	206	586	677	248	166	99	145	327	137	36	236	4	255	3122

The AVF concludes that these results demonstrate acceptable conformity to ANSI/MIL-STD-1815A Ada.

*Ada is a registered trademark of the United States Government (Ada Joint Program Office).

TABLE OF CONTENTS

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 behaviour that is implementation dependent but 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 unsupported language constructs required by the Ada Standard
- To determine that the implementation-dependent behaviour is allowed by the Ada Standard

Testing of this compiler was conducted by NCC under the direction of the AVF according to policies and procedures established by the Ada Validation Organization (AVO). On-site testing was conducted from December 1987 through December at Alsys Ltd, Partridge House, Newtown Road, Henley-on-Thames.

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:-

The National Computing Centre Ltd
Oxford Road
Manchester M1 7ED
United Kingdom

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.

2. Ada Compiler Validation Procedures and Guidelines, Ada Joint Program Office, 1 January 1987.

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

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 and point addressed by a comment on the Ada Standard. Standard. These comments are given a unique identification number having the form AI-ddddd.
Ada Standard	ANSI/MIL-STD-1815A, February 1983.
Applicant	The agency requesting validation.
AVF	The Ada Validation Facility. In the context of this report, the AVF is responsible for conducting compiler validations according to established procedures.
AVO	The Ada Validation Organization. In the context of this report, the AVO is responsible for establishing procedures for compiler validations.
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	An AVCV 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.
Language Maintenance Panel	The Language Maintenance Panel (LMP) is a committee established by the Ada Board to recommend interpretations and possible changes to the ANSI/MIL-STD for Ada.
Passed test	An ACVC test for which a compiler generates the expected result.
Target	The computer for which a compiler generates code.
Test	An Ada 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 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 six test classes: A, B, C, D, E, and L. The first letter of a test name identifies the class to which it belongs. Class A, C, D, and E tests are executable, 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 link errors.

Class A tests check that legal Ada programs can be successfully compiled and executed. However, no checks are performed during execution to see if the test objective has been met. 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 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.

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.

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 these units 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 the tests in the ACVC follow conventions that are intended to ensure that the tests are reasonable 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 for 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 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: AlsyCOMP_013, Version 3.2

ACVC Version: 1.9

Certificate Number: #871210N1.09012

Host Computer:

Machine:	IBM PC/AT
Operating System:	PC/DOS Version 3.1
Memory Size:	640K (main) 4M (RAM disk)

Target Computer:

Machine:	IBM 370 3084Q
Operating System:	MVS Version 3.2
Memory Size:	1M partition

Communications Network: Magnetic media

2.2 IMPLEMENTATION CHARACTERISTICS

One of the purposes of validating compilers is to determine the behaviour 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:

- Capacities.

  The compiler correctly processes tests containing loop statements nested to 65 levels, block statements nested to 65 levels, and recursive procedures separately compiled as subunits nested to 17 levels. It correctly processes a compilation containing 723 variables in the same declarative part. (See tests D55A03A..H (8 tests), D56001B, D64005E..G (3 tests), and D29002K.)

- Universal integer calculations

  An implementation is allowed to reject universal integer calculations having values that exceed SYSTEM.MAX_INT. This implementation processes 64 bit integer calculations. (See tests D4A002A, D4A002B, D4A004A, and D4A004B).

- Predefined types.

  This implementation supports the additional predefined types SHORT_INTEGER, SHORT_FLOAT, LONG_FLOAT, in the package STANDARD. (See tests B86001C and B86001D.)

- Based literals

  An implementation is allowed to reject a based literal with a value exceeding SYSTEM.MAX_INT during compilation, or it may raise NUMERIC_ERROR or CONSTRAINT_ERROR during execution. This implementation raises NUMERIC_ERROR during execution. (See test E24101A.)

- Expression evaluation.

  Apparently some default initialization expressions for record components are evaluated before any value is checked

to belong to a component's subtype. (See test C32117A.)

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

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

Apparently 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.)

Sometimes 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.)

Apparently underflow is not gradual. (See tests C45524A..Z.)

. Rounding.

The method used for rounding to integer is apparently round away from zero. (See tests C46012A..Z.)

The method used for rounding to longest integer is apparently round away from zero. (See tests C46012A..Z.)

The method used for rounding to integer in static universal real expressions is apparently round away from zero. (See test C4A014A.)

. 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:

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

No exception is raised when LENGTH is applied to an array type with INTEGER'LAST + 2 components. NUMERIC_ERROR is raised when an array type with INTEGER'LAST + 2 components is declared. (See test C36202A.)

No exception is raised when 'LENGTH is applied to an array type with SYSTEM.MAX_INT + 2 components. NUMERIC_ERROR is

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

A packed BOOLEAN array having a 'LENGTH exceeding INTEGER'LAST raises NUMERIC_ERROR when the array type is declared. (See test C52103X.)

A packed two-dimensional BOOLEAN array with more than INTEGER'LAST components raises NUMERIC_ERROR when the array type is declared. (See test C52104Y.)

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 NUMERIC_ERROR when the array type is declared. (See test E52103Y.)

In assigning one-dimensional array types, the expression appears to be evaluated in its entirety before CONSTRAINT_ERROR is raised when checking whether the expression's subtype is compatible with the target's subtype. In assigning two-dimensional array types, the expression does not appear to be 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.)

- Discriminated types.

  During compilation, an implementation is allowed to either accept or reject an incomplete type with discriminants that is used in an access type definition with a compatible discriminant constraint. This implementation accepts such subtype indications. (See test E38104A.)

  In assigning record types with discriminants, the expression appears to be 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.)

- Aggregates.

  In the evaluation of a multi-dimensional aggregate, all choices appear to be evaluated before checking against the index type. (See tests C43207A and C43207B.)

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

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

. Representation clauses.

The Ada Standard does not require an implementation to support representation clauses. If a representation clause is not supported, then the implementation must reject it.

Enumeration representation clauses containing noncontiguous values for enumeration types other than character and boolean types are supported. (See tests C35502I..J. C35502M..N. and A39005F.)

Enumeration representation clauses containing noncontiguous values for character types are supported. (See tests C35507I..J, C35507M..N, and C55B16A.)

Enumeration representation clauses for boolean types containing representational values other than (FALSE => 0, TRUE => 1) are supported. (See tests C35508I..J and C35508M..N.)

Length clauses with SIZE specifications for enumeration types are supported. (See test A39005B.)

Length clauses with STORAGE_SIZE specifications for access types are supported. (See tests A39005C and C87B62B.)

Length clauses with STORAGE_SIZE specifications for task types are supported. (See tests A39005D and C87B62D.)

Length clauses with SMALL specifications are supported. (See tests A39005E and C87B62C.)

Record representation clauses are supported to the byte level only. (See test A39005G.)

Length clauses with SIZE specifications for derived integer types are supported. (See test C87B62A.)

. Pragmas.

The pragma INLINE is supported for procedure and function calls from within a body. The pragma INLINE for function

calls within a declaration is not supported. (See tests LA3004A, LA3004B, EA3004C, EA3004D, CA3004E, and CA3004F.)

. Input/output.

The package SEQUENTIAL_IO can be instantiated with unconstrained array types and record types with discriminants without defaults. (See tests AE2101C, EE2201D, and EE2201E.)

The package DIRECT_IO cannot be instantiated with unconstrained array types and record types with discriminants without defaults. (See tests AE2101H, EE2401D, and EE2401G.)

Modes IN_FILE and OUT_FILE) are supported for SEQUENTIAL_IO. (See tests CE2102D and CE2102E.)

Modes IN_FILE, OUT_FILE, and INOUT_FILE are supported for DIRECT_IO. (See tests CE2102F, CE2102I, and CE2102J.)

RESET and DELETE are supported for SEQUENTIAL_IO and DIRECT_IO. (See tests CE2102G and CE2102K.)

Dynamic creation and deletion of files are supported for SEQUENTIAL_IO and DIRECT_IO. (See tests CE2106A and CE2106B.)

Overwriting to a sequential file truncates the file to the last element written. (See test CE2208B.)

An existing text file can be opened in OUT_FILE mode, can be created in OUT_FILE mode, and can be created in IN_FILE mode. (See test EE3102C.)

More than one internal file can be associated with each external file for text I/O for reading only. (See tests CE2110B, CE2111D, CE3111A..E (5 tests), CE3114B, and CE3115A.)

More than one internal file can be associated with each external file for sequential I/O for reading only. (See tests CE2107A..D (4 tests) and CE2111D.)

More than one internal file can be associated with each external file for direct I/O for reading only. (See tests CE2107E..I (5 tests) and CE2111H.)

An external file associated with more than one internal file cannot be deleted for SEQUENTIAL_IO, DIRECT_IO, and TEXT_IO. (See test CE2110B.))

Temporary sequential files are given names. Temporary direct files are given names. Temporary files given names are deleted when they are closed. (See tests CE2108A and CE2108C.)

Generics.

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

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

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

CHAPTER 3

TEST INFORMATION

3.1 TEST RESULTS

At the time of testing, version 1.9 of the ACVC comprised 3122 tests of which 25 had been withdrawn. Of the remaining tests, 205 were determined to be inapplicable to this implementation. Not all of the inapplicable tests were processed during testing; 145 executable tests that use floating-point precision exceeding that supported by the implementation were not processed Modifications to the code, processing, or grading for 19 tests were required to successfully demonstrate the test objective. (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	108	1047	1659	17	15	46	2892
Failed	0	0	0	0	0	0	0
Inapplicable	2	4	196	0	3	0	205
Withdrawn	3	2	19	0	1	0	25
TOTAL	113	1053	1874	17	19	46	3122

3.3 SUMMARY OF TEST RESULTS BY CHAPTER

RESULT	CHAPTER													
	2	3	4	5	6	7	8	9	10	11	12	13	14	TOTAL
Passed	193	516	564	245	166	98	141	327	135	36	234	3	234	2892
Failed	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Inapplicable	11	57	111	3	0	0	2	0	2	0	0	0	19	205
Withdrawn	2	13	2	0	0	1	2	0	0	0	2	1	2	25
TOTAL	206	586	677	248	166	99	145	327	137	36	236	4	255	3122

3.4 WITHDRAWN TESTS

The following 25 tests were withdrawn from ACVC Version 1.9 at the time of this validation:

B28003A	E28005C	C34004A	C35502P	A35902C	C35904A
C35A03E	C35A03R	C37213H	C37213J	C37215C	C37215E
C37215G	C37215H	C38102C	C41402A	C45614C	A74106C
C85018B	C87B04B	CC1311B	BC3105A	AD1A01A	CE2401H
CE3208A					

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, 205 tests were inapplicable for the reasons indicated:

- A39005G uses a record representation clause at the bit level. This compiler only supports such clauses to the byte level.

. The following tests use LONG_INTEGER, which is not supported by this compiler:

C45231C	C45304C	C45502C	C45503C	C45504C
C45504F	C45611C	C45613C	C45631C	C45632C
B52004D	C55B07A	B55B09C		

. C45531M, C45531N, C45532M, and C45532N use fine 48 bit fixed point base types which are not supported by this compiler.

. C45531O, C45531P, C45532O, and C45532P use coarse 48 bit fixed point base types which are not supported by this compiler.

. B86001D & C45231D require a predefined numeric type other than those defined by the Ada language in package STANDARD. There is no such type for this implementation.

. C86001F redefines package SYSTEM, but TEXT_IO is made obsolete by this new definition in this implementation and the test cannot be executed since the package REPORT is dependent on the package TEXT_IO.

. BA2001E requires that duplicate names of subunits with a common ancestor be detected at compilation time. This compiler correctly detects the error at link time, and the AVO rules that such behaviour is acceptable.

. EA3004D This compiler only obeys the INLINE pragma for calls from an Ada statement within a body. This test calls an INLINE function within a declaration.

. AE2101H, EE2401D, and EE2401G use instantiations of package DIRECT_IO with unconstrained array types and record types having discriminants without defaults. These instantiations are rejected by this compiler.

. CE2107B..E(4 tests), CE2107G..I (3 tests), CE2110B, CE2111D, CE2111H, CE3111B..E (4 tests), CE3114B, and CE3115A are inapplicable because multiple internal files cannot be associated with the same external file when one file is open for writing. The proper exception is raised when multiple access is attempted.

. The following 159 tests require a floating-point accuracy that exceeds the maximum of 18 digits supported by this implementation:

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

3.6 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 behaviour. Modification are made with the approval of the AVO, and are made in cases where legitimate implementation behaviour 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 sub-tests so that all errors are detected; and confirming that messages produced by an executable test demonstrate conforming behaviour that wasn't anticipated by the test (such as raising one exception instead of another).

Modifications were required for 19 Class B tests.

The following Class B tests were split because syntax errors at one point resulted in the compiler not detecting other errors in the test:

B24007A	B24009A	B25002A	B26005A	B27005A
B32202A	B32202B	B32202C	B33001A	B37004A
B45102A	B61012A	B62001B	B62001C	B62001D
B91004A	B95069A	B95069B	BC3205C	

3.7 ADDITIONAL TESTING INFORMATION

3.7.1 Prevalidation

Prior to validation, a set of test results for ACVC Version 1.9 produced by the AlsyCOMP_013 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 behaviour on all inapplicable tests.

3.7.2 Test Method

Testing of the AlsyCOMP_013 using ACVC Version 1.9 was conducted on-site by a validation team from the AVF. The configuration consisted of an IBM PC/AT operating under PC/DOS Version 3.1 host and an IBM 370 3084Q target operating under MVS, Version 3.2. The host and target computers were linked via magnetic media.

A magnetic tape containing all tests 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 magnetic tape. Tests requiring modifications during the prevalidation testing were not included in their modified form on the magnetic tape.

The contents of the magnetic tape were not loaded directly onto the host computer.

The magnetic tape was loaded onto a DEC VAX 750 running VMS upon which tests requiring to be split were modified. The test files were then transferred, one chapter at a time, to the IBM PC/AT via an ethernet connection.

After the test files were loaded to disk, the full set of tests was compiled and bound on the IBM PC/AT.

All object files and compilation output were transferred via ethernet to the DEC VAX 750 where a magnetic tape in IBM compatible format was produced which contained all the object files. The tape was read onto a CMS mini disk on an IBM 370 3081K running VM/CMS 3.1.

The IBM 370 308/K machine was used to control the linking and execution of the object files by submitting batch jobs to the MVS 3.2 system running on an IBM 370 3084Q and retrieving the output of these jobs. The process was controlled by command scripts to submit object files and associated JCL to link and execute tests using 2 simultaneous batch streams on a single host computer. Once a test completed, its output was returned to the CMS system and processed by scripts that made the output easier to understand.

Run results were transferred to a SUN 3/160 running UNIX BSD 4.2, via a SUN 3/160 file transfer program IBMFTP, from where they were transferred via ethernet to the VAX 750. All test output was printed from the VAX 750.

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

Option | Switch	Effect
PAGE_LENGTH=45	Control length of compiler listing pages
PAGE_WIDTH=132	Control width of compiler listing pages
ERRORS=999	Control number of errors detected before compiler aborts
TEXT	Include full source code in listing

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

3.7.3 Test Site

The validation team arrived at Alsys Ltd, Partridge House, Newtown Road, Henley on Thames on 7 December 1987 and departed after testing was completed on 10 December 1987.

APPENDIX A

CONFORMANCE STATEMENT

Alsys Limited has submitted the following conformance statement concerning the AlsyCOMP_013

CONFORMANCE STATEMENT

DECLARATION OF CONFORMANCE

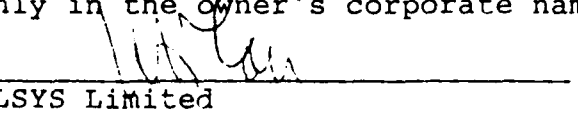
Compiler Implementor: Alsys Limited
Ada Validation Facility: The National Computing Centre Ltd
Ada Compiler Validation Capability (ACVC) Version: 1.9

Base Configuration

Base Compiler Name: AlsyCOMP_013	Version: 3.2
Host Architecture : IBM PC/AT	OS&VER : PC/DOS 3.1
Target Architecture : IBM 370 3084Q	OS&VER : MVS 3.2

Implementor's Declaration

I, the undersigned, representing Alsys Limited, have implemented no deliberate extensions to the Ada Language Standard ANSI/MIL-STD-1815A in the compiler(s) listed in this declaration. I declare that Alsys Limited is the owner of record of the Ada language compiler(s) listed above and, as such, is responsible for maintaining said compiler(s) in conformance to ANSI/MIL-STD-1815A. All certificates and registrations for Ada language compiler(s) listed in this declaration shall be made only in the owner's corporate name.



Date: 4/1/88

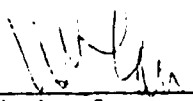
ALSYS Limited
M L J Jordan, Marketing Director

*Ada is a registered trademark of the United States Government (Ada Joint Program Office)

CONFORMANCE STATEMENT

Owner's Declaration

I, the undersigned, representing Alsys Ltd, take full responsibility for implementation and maintenance of the Ada compiler(s) listed above, and agree to the public disclosure of the final Validation Summary Report. I further agree to continue to comply with the Ada trademark policy, as defined by the Ada Joint Program Office. I declare that all of the Ada language compilers listed, and their host/target performance are in compliance with the Ada Language ANSI/MIL-STD-1815A.



Date: 4/1/89

Alsys Limited
M L J Jordan, Marketing Director

APPENDIX B

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 MIL-STD-1815A, and to certain allowed restrictions on representation clauses. The implementation-dependent characteristics of the AlsyCOMP_0013, version 3.2 are described in the following sections which discuss topics in Appendix F of the Ada Language Reference Manual (ANSI/MIL-STD-1815A). Implementation-specific portions of the package STANDARD are also included in this appendix.

Alsys IBM 370 Ada* Compiler

Appendix F
Implementation - Dependent Characteristics
for MVS

Version 3.2

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.

Printed: November 1987

PREFACE

This *Alsys IBM 370 Ada Compiler Appendix F* is for programmers, software engineers, project managers, educators and students who want to develop an Ada program for any IBM System/370 processor that runs MVS.

This appendix is a required part of the *Reference Manual for the Ada Programming Language*, ANSI/MIL-STD 1815A, February 1983 (throughout this appendix, citations in square brackets refer to this manual). It assumes that the user is already familiar with the MVS operating system, and has access to the following IBM documents:

OS/VS2 MVS Overview, GC28-0984

OS/VS2 System Programming Library: Job Management, GC28-1303

OS/VS2 MVS JCL, GC28-1350

IBM System/370: Principles of Operation, GA22-7000

IBM System/370 System Summary, GA22-7001

TABLE OF CONTENTS

Appendix F

Implementation-Dependent Characteristics

This appendix summarises the implementation-dependent characteristics of the Alsys IBM 370 Ada Compiler for MVS.

The sections of this appendix are as follows:

1. The form, allowed places, and effect of every implementation-dependent pragma.
2. The name and type of every implementation-dependent attribute.
3. The specification of the package SYSTEM.
4. The list of all restrictions on representation clauses.
5. The conventions used for any implementation-generated names denoting implementation-dependent components.
6. The interpretation of expressions that appear in address clauses, including those for interrupts.
7. Any restrictions on unchecked conversions.
8. Any implementation-dependent characteristics of the input-output packages.
9. Characteristics of numeric types.
10. Other implementation-dependent characteristics.
11. Compiler limitations.

The name *Ada Run-Time Executive* refers to the run-time library routines provided for all Ada programs. These routines implement the Ada heap, exceptions, tasking, IO, and other utility functions.

1 Implementation-Dependent Pragmas

Ada programs can interface with subprograms written in assembler or other languages through the use of the predefined pragma INTERFACE [13.9] and the implementation-defined pragma INTERFACE_NAME.

1.1 INTERFACE

Pragma INTERFACE specifies the name of an interfaced subprogram and the name of the programming language for which calling and parameter passing conventions

will be generated. Pragma INTERFACE takes the form specified in the *Reference Manual*:

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

where

- *language_name* is the name of the other language whose calling and parameter passing conventions are to be used.
- *subprogram_name* is the name used within the Ada program to refer to the interfaced subprogram.

The only language name currently accepted by pragma INTERFACE is ASSEMBLER.

The language name used in the pragma INTERFACE does not necessarily correspond to the language used to write the interfaced subprogram. It is used only to tell the Compiler how to generate subprogram calls, that is, which calling conventions and parameter passing techniques to use. ASSEMBLER is used to refer to the standard IBM 370 calling and parameter passing conventions. The programmer can use the language name ASSEMBLER to interface Ada subprograms with subroutines written in any language that follows the standard IBM 370 calling conventions.

Calling Conventions

The contents of the general purpose registers 12 and 13 must be left unchanged by the call. On entry to the subprogram, register 13 contains the address of a register save area provided by the caller.

Registers 15 and 14 contain the entry point address and return address, respectively, of the called subprogram.

The Ada Run-Time Executive treats any interruption occurring during the execution of the body of the subprogram as an exception being raised at the point of call of the subprogram. The exception raised following a program interruption in interfaced code is a NUMERIC_ERROR for the following cases:

Fixed-pt overflow *
Fixed-pt divide
Decimal overflow *
Decimal divide
Exponent overflow
Exponent underflow *
Significance *
Floating-pt divide

In other cases, PROGRAM_ERROR is raised. The classes of interruptions marked with an asterisk (*) may be masked by setting the program mask. Note that the program mask should be restored to its original value before returning to Ada code.

Parameter-Passing Conventions

On entry to the subprogram, register 1 contains the address of a parameter address list. Each word in this list is an address corresponding to a parameter. The last word in the list has its bit 0 (sign bit) set.

For actual parameters which are literal values, the address is that of a copy of the value of the parameter; for all other parameters it is the address of the parameter object. Interfaced subprograms have no notion of parameter modes; hence parameters whose addresses are passed are not protected from modification by the interfaced subprogram, even though they may be formally declared to be of mode in.

If the subprogram is a function, on exit register 0 is used to return the result. Scalar values are returned in register 0. Non-scalar values are returned by address in register 0.

No consistency checking is performed between the subprogram parameters declared in Ada and the corresponding parameters of the interfaced subprogram. It is the programmer's responsibility to ensure correct access to the parameters.

An example of an interfaced subprogram is:

```
* 64-bit integer addition: use an array rather than a record to
* represent the integer so as not to rely on record ordering if the
* components are accessed in Ada.
*
* type DOUBLE is array (1..2) of INTEGER;
* procedure ADD (LEFT, RIGHT : in  DOUBLE;
*                RESULT      : out DOUBLE);
ADD   CSECT
      USING  ADD,15
      STM    2,6,12(13)
      L      2,0(1)       Address of LEFT
      LM     3,4,0(2)     Value of LEFT
      L      2,4(1)       Address of RIGHT
      AL     4,4(2)       Add low-order components (no interruption)
      BC     12,$1        Branch if no carry
      A      3,=F'1'      Add carry (NUMERIC_ERROR possible)
$1    A      3,0(2)       Add high-order (NUMERIC_ERROR possible)
      L      2,8(1)       Address of RESULT
      STM    3,4,0(2)     Value of result
      LM     2,6,12(13)
      BR     14
      LTORG
      DROP
      END
```

Parameter Representations

This section describes the representation of values of the types that can be passed as parameters to an interfaced subprogram.

Integer Types [3.5.4]

Ada integer types occupy 16 (SHORT_INTEGER) or 32 (INTEGER) bits. An INTEGER subtype falling within the range of SHORT_INTEGER is implemented as a SHORT_INTEGER in 16 bits.

Enumeration Types [3.5.1]

Values of an Ada enumeration type are represented internally as unsigned values representing their position in the list of enumeration literals defining the type. The first literal in the list corresponds to a value of zero.

Enumeration types with 256 elements or fewer are represented in 8 bits, those with more than 256 elements in 16 bits. The maximum number of values an enumeration type can include is 65536 (2**16).

The Ada predefined type CHARACTER [3.5.2] is represented in 8 bits, using the standard ASCII codes [C].

Floating Point Types [3.5.7, 3.5.8]

Ada floating-point values occupy 32 (SHORT_FLOAT), 64 (FLOAT) or 128 (LONG_FLOAT) bits, and are held in IBM 370 (short, long or extended floating point) format.

Fixed Point Types [3.5.9, 3.5.10]

Ada fixed-point types are managed by the Compiler as the product of a signed *mantissa* and a constant *small*. The mantissa is implemented as a 16 or 32 bit integer value. *Small* is a compile-time quantity which is the power of two equal or immediately inferior to the delta specified in the declaration of the type.

The attribute MANTISSA is defined as the smallest number such that:

2 ** MANTISSA >= max (abs (upper_bound), abs (lower_bound)) / *small*

The size of a fixed point type is:

MANTISSA	Size
1 .. 15	16 bits
16 .. 31	32 bits

Fixed point types requiring a MANTISSA greater than 31 are not supported.

Access Types [3.8]

Values of access types are represented internally by the 31-bit address of the designated object held in a 32 bit word. Users should not alter the bits of this word,

which are ignored by the architecture on which the program is running. The value zero is used to represent **null**.

Array Types [3.6]

Ada arrays are passed by reference; the value passed is the address of the first element of the array. When an array is passed as a parameter to an interfaced subprogram, the usual consistency checking between the array bounds declared in the calling program and the subprogram is not enforced. It is the programmer's responsibility to ensure that the subprogram does not violate the bounds of the array.

Values of the predefined type STRING [3.6.3] are arrays, and are passed in the same way: the address of the first character in the string is passed. Elements of a string are represented in 8 bits, using the standard ASCII codes.

Record Types [3.7]

Ada records are passed by reference, by passing the address of the first component of the record. Components of a record are aligned on their natural boundaries (e.g. INTEGER on a four-byte boundary). If a record contains discriminants or components having a dynamic size, implicit components may be added to the record. Thus the exact internal structure of the record in memory may not be inferred directly from its Ada declaration.

Restrictions on Interfaced Subprograms

The Ada Run-Time Executive uses the SPIE (SVC 14) macro. Interfaced subprograms should avoid use of this facility, or else restore interruption processing to its original state before returning to the Ada program. Failure to do so may lead to unpredictable results.

Similarly, interfaced subprograms must not change the program mask in the Program Status Word (PSW) of the machine without restoring it before returning.

1.2 INTERFACE_NAME

Pragma INTERFACE_NAME associates the name of an interfaced subprogram, as declared in Ada, with its name in the language of origin. If pragma INTERFACE_NAME is not used, then the two names are assumed to be identical. This pragma takes the form

pragma INTERFACE_NAME (*subprogram_name*, *string_literal*);

where

- *subprogram_name* is the name used within the Ada program to refer to the interfaced subprogram.

- *string_literal* is the name by which the interfaced subprogram is referred to at link-time.

The use of INTERFACE_NAME is optional, and is not needed if a subprogram has the same name in Ada as in the language of origin. It is useful, for example, if the name of the subprogram in its original language contains characters that are not permitted in Ada identifiers. Ada identifiers can contain only letters, digits and underscores, whereas the IBM 370 linkage editor/loader allows external names to contain other characters, e.g. the plus or minus sign. These characters can be specified in the *string_literal* argument of the pragma INTERFACE_NAME.

The pragma INTERFACE_NAME is allowed at the same places of an Ada program as the pragma INTERFACE [13.9]. However, the pragma INTERFACE_NAME must always occur after the pragma INTERFACE declaration for the interfaced subprogram.

In order to conform to the naming conventions of the IBM 370 linkage editor/loader, the link-time name of an interfaced subprogram will be truncated to 8 characters and converted to upper case.

Example

```
package SAMPLE_DATA is
        function SAMPLE_DEVICE (X : INTEGER) return INTEGER;
        function PROCESS_SAMPLE (X : INTEGER) return INTEGER;
private
        pragma INTERFACE (ASSEMBLER, SAMPLE_DEVICE);
        pragma INTERFACE (ASSEMBLER, PROCESS_SAMPLE);
        pragma INTERFACE_NAME (PROCESS_SAMPLE, "PSAMPLE");
end SAMPLE_DATA;
```

1.3 Other Pragmas

No other implementation-dependent pragmas are supported in the current version of this compiler.

2 Implementation-Dependent Attributes

In a addition to the Representation tributes of [13.7.2] and [13.7.3], there are the four attributes listed in section ' (Conventions for Implementation-Generated Names), for use in record representation clauses. There also exists the restrictions given below on the use of the ADDRESS attribute.

Limitations on the use of the attribute ADDRESS

The attribute ADDRESS is implemented for all prefixes that have meaningful addresses. The following entries 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 specifiaction that is not a library unit.
- A package body that is not a library unit or subunit.

3 Specification of the Package SYSTEM

```
package SYSTEM is

    type NAME is (IBM_370);

    SYSTEM_NAME : constant NAME := NAME'FIRST;
    MIN_INT     : constant := -(2**31);
    MAX_INT     : constant := 2**31-1;
    MEMORY_SIZE : constant := 2**24;

    type ADDRESS is range MIN_INT .. MAX_INT;

    STORAGE_UNIT : constant := 8;
    MAX_DIGITS   : constant := 18;
    MAX_MANTISSA : constant := 31;
    FINE_DELTA   : constant := 2#1.0#e-31;
    TICK         : constant := 0.01;
    NULL_ADDRESS : constant ADDRESS := 0;

    subtype PRIORITY is INTEGER range 1 .. 10;

    -- These subprograms are provided to perform
    -- READ/WRITE operations in memory.

    generic
        type ELEMENT_TYPE is private;
    function FETCH (FROM : ADDRESS) return ELEMENT_TYPE;

    generic
        type ELEMENT_TYPE is private;
    procedure STORE (INTO : ADDRESS; OBJECT : ELEMENT_TYPE);

end SYSTEM;
```

The generic function FETCH may be used to read data objects from given addresses in store. The generic procedure STORE may be used to write data objects to given addresses in store.

4 Restrictions on Representation Clauses

This version of the Alsys IBM 370 Ada Compiler supports representation clauses [13.1] with the following exceptions:

- There is no bit level implementation for any of the representation clauses.
- Address clauses are not supported.
- Change of representation for RECORD types are not implemented.
- Machine code insertions are not supported.
- For the length clause:
  - Size specification: T'SIZE is not implemented for types declared in a generic unit.
  - Specification of *small* for a fixed point type: T'SMALL is restricted to a power of 2, and the absolute value of the exponent must be less than 31.
- The Enumeration Clause is not allowed if there is a range constraint on the parent subtype.
- The Record Clause is not allowed for a derived record type.
- The pragma PACK [13.1] is also not supported. However, its presence in a program does not in itself make the program illegal; the Compiler will simply issue a warning message and ignore the pragma.

5 Conventions for Implementation-Generated Names

Special record components are introduced by the compiler for certain record type definitions. Such record components are implementation-dependent: they are used by the compiler to improve the quality of the generated code for certain operations on the record types. The existence of these components is established by the compiler depending on implementation-dependent criteria. Attributes have been defined for referring to them in record representation clauses. An error message is issued by the compiler if the user refers to implementation-dependent attribute that does not exist. If the implementation-dependent component exists, the compiler checks that the storage location specified in the component clause is compatible with the treatment of this component and the stoarge locations of other components. An error message is issued if this check fails.

There are four such attributes:

T'RECORD_SIZE For a prefix T that denotes a record type. This attribute refers to the record component introduced by the compiler in a record to store the size of the record object. This component exists for objects of a record type with defaulted discriminants when the sizes of the record objects depend on the values of the discriminants.

T'VARIENT_INDEX For a prefix T that denotes a record type. This attribute refers to the record component introduced by the compiler in a record to assist in the efficient implementation of discriminant checks. This component exists for objects of a record type with variant type.

C'ARRAY_DESCRIPTOR For a prefix C that denotes a record component of array type whose component subtype definition depends on discriminants. This attribute refers to the record component introduced by the compiler in a record to store information on subtypes of components that depend on discriminants.

C'RECORD_DESCRIPTOR For a prefix C that denotes a record component of record type whose component subtype definition depends on discriminants. This attribute refers to the record component introduced by the compiler in a record to store information on subtypes of components that depend on discriminants.

There are four implementation-generated names:

RECORD_SIZE This is an implementation-specific record component. The component is introduced by the compiler in a record to store the size of the record object.

VARIANT_INDEX This is an implementation-specific record component. The component is introduced by the compiler in a record to assist in the efficient implementation of discriminant checks.

ARRAY_DESCRIPTIOR and RECORD_DESCRIPTOR Array and record descriptors are internal components which are used by the compiler to store information on subtypes or record components that depend upon discriminants.

Array descriptors are used for record components of array types, whereas record descriptors are used for record components of record types

6 Address Clauses

Address clauses [13.5] are not supported in this version of the Alsys IBM 370 Ada Compiler.

7 Restrictions on Unchecked Conversions

Unchecked conversions [13.10.2] are allowed only between types which have the same value for their 'SIZE attribute.

8 Input-Output Packages

The predefined input-output packages SEQUENTIAL_IO [14.2.3], DIRECT_IO [14.2.5], and TEXT_IO [14.3.10] are implemented as described in the Language Reference Manual, as is the package IO_EXCEPTIONS [14.5], which specifies the exceptions that can be raised by the predefined input-output packages.

The package LOW_LEVEL_IO [14.6], which is concerned with low-level machine-dependent input-output, has not been implemented.

TEXT_IO and SEQUENTIAL_IO files are implemented as DSORG PS (QSAM). DIRECT_IO files are implemented as DSORG DA (BDAM).

8.1 Specifying External Files

The NAME parameter supplied to the Ada procedures CREATE or OPEN [14.2.1] may represent an MVS dataset name (DSNAME) or a DDNAME.

Files

An MVS dataset name as specified in the Ada NAME parameter may be given in any of the following forms:

```
OPEN (F, NAME => "UNQUALIFIED.NAME", ...);

OPEN (F, NAME => "'FULLY.QUALIFIED.NAME'", ...);

OPEN (F, NAME => "UNQUALIFIED.PDS (MEMBER)", ...);

OPEN (F, NAME => "'FULLY.QUALIFIED.PDS (MEMBER)'", ...);
```

An unqualified name (not enclosed in apostrophes) is first prefixed by the name (if any) given as the QUALIFIER parameter in the program PARM string when the program is run.

The QUALIFIER parameter may be specified as in the following example:

```
//STEP20 EXEC PGM=IEB73,PARM='/QUALIFIER(PAYROLL.ADA)'
```

A fully qualified name (enclosed in single quotes) is not so prefixed. The result of the NAME function is always in the form of a fully qualified name, i.e. enclosed in single quotes.

The file name parameter may also be a DDNAME. If the file name parameter starts with a % character, the remainder of the string (including trailing blanks) is taken as a DDNAME which must be defined in the JCL used to execute the program; otherwise NAME_ERROR will be raised.

If DELETE is called for a file opened using a DDNAME, USE_ERROR will be raised, but the file will be closed.

FORM Parameter

The FORM parameter comprises a set of attributes formulated according to the lexical rules of [2], separated by commas. The FORM parameter may be given as a null string except when DIRECT_IO is instantiated with an unconstrained type: in this case the RECORD_SIZE attribute must be provided. Attributes are comma-separated; blanks may be inserted between lexical elements as desired. In the descriptions below the meanings of *natural*, *positive*, etc., are as in Ada; attribute keywords (represented in upper case) are identifiers [2.3] and as such may be specified without regard to case.

USE_ERROR is raised if the FORM parameter does not conform to these rules.

The attributes are as follows:

Unit attribute

This attribute allows control over the unit on which a file is allocated. The syntax is as follows:

UNIT => *unit_name*

where *unit_name* specifies a group name, a device type or a specific unit address.

The default is the local system generation default.

Volume attribute

This attribute allows control over the volume on which a file is allocated. The syntax is as follows:

VOLUME => *volume_name*

where *volume_name* specifies the volume serial number.

The default is the local system generation default.

Primary attribute

This attribute allows control over the primary space allocation for the file. The syntax is as follows:

PRIMARY => *natural*

where *natural* is the number of blocks allocated to the file.

The default is the local system generation default.

Secondary attribute

This attribute allows control over the secondary space allocation for the file. The syntax is as follows:

SECONDARY => *natural*

where *natural* is the number of additional blocks allocated to the file if more space is needed.

The default is the local system generation default.

Extension attribute

This attribute allows control over the manner in which DIRECT_IO (BDAM) files are extended. When a DIRECT_IO file is created it must be initialised to a certain extent (number of records). In order to extend the file beyond this point, the file must be recreated and copied. This recreation is automatic, but does imply an overhead in extending a file beyond its initial allocation. The EXTENSION attribute allows this overhead to be minimised by specifying both the initial number of records to be initialised when the file is created, and the additional number of

records to be intialised over and above that required, when a record greater than the current number of initialised records is written to. The syntax is as follows:

EXTENSION => *natural*

where *natural* is the number of records initialised on file creation and the number of additional records initialised on file extension.

The default is EXTENSION => 0.

Unpredictable results may occur if file sharing is attempted for DIRECT_IO files which are extended (see File sharing attribute).

File sharing attribute

This attribute allows control over the sharing of one external file between several internal files within a single program. In effect it establishes rules for subsequent OPEN and CREATE calls which specify the same external file. If such rules are violated or if a different file sharing attribute is specified in a later OPEN or CREATE call, USE_ERROR will be raised. The syntax is as follows:

NOT_SHARED | SHARED => *access_mode*

where

access_mode ::= READERS | SINGLE_WRITER | ANY

A file sharing attribute of:

NOT_SHARED

implies only one internal file may access the external file.

SHARED => READERS

imposes no restrictions on internal files of mode IN_FILE, but prevents any internal files of mode OUT_FILE or INOUT_FILE being associated with the external file.

SHARED => SINGLE_WRITER

is as SHARED => READERS, but in addition allows a single internal file of mode OUT_FILE or INOUT_FILE.

SHARED => ANY

places no restrictions on external file sharing.

If a file of the same name has previously been opened or created, the default is taken from that file's sharing attribute, otherwise the default depends on the mode

of the file: for mode IN_FILE the default is SHARED => READERS, for modes INOUT_FILE and OUT_FILE the default is NOT_SHARED.

Unpredictable results may occur if file sharing is attempted for DIRECT_IO files which are extended (see Extension attribute).

Record size attribute

This attribute controls the record format (RECFM) and logical record length (LRECL) of an external file.

By default, records are output according to the following rules (see section 8.5):

- for TEXT_IO and SEQUENTIAL_IO, variable-length record files (RECFM = V).
- for DIRECT_IO, fixed-length record files (RECFM = F).

The user can specify the record size attribute to force the representation of the Ada element in output records of a given byte size. If the record size attribute is specified, fixed-length records (RECFM = F) will be generated, with a record length (LRECL) as specified (see section 8.5).

In the case of DIRECT_IO and SEQUENTIAL_IO for constrained types the value given which must not be smaller than ELEMENT_TYPE'SIZE / SYSTEM.STORAGE_UNIT; USE_ERROR will be raised if this rule is violated.

In the case of DIRECT_IO for unconstrained types the user is required to specifiy the RECORD_SIZE attribute (otherwise USE_ERROR will be raised by the OPEN or CREATE procedures). The size specified must be large enough to accommodate the largest record which is to be read or written plus 4 bytes for the descriptor (see section 8.5). If a larger record is processed, DATA_ERROR will be raised by the READ or WRITE.

In the case of TEXT_IO, output lines will be padded to the requisite length with spaces; this fact should be borne in mind when re-reading files generated using TEXT_IO with the record size attribute set.

The syntax of the record size attribute is as follows:

RECORD_SIZE => *natural*

where *natural* is a size in bytes.

The default is

RECORD_SIZE => *element_length*

where

element_length = ELEMENT_TYPE'SIZE / SYSTEM.STORAGE_UNIT

for input-output of constrained types, and

RECORD_SIZE => 0

(meaning variable-length records) for input-output of unconstrained types other than via DIRECT_IO in which case the RECORD_SIZE attribute must be provided by the user.

Block size attribute

This attribute controls the block size of an external file. The block size must be at least as large as the record size (if specified) plus four bytes for a record descriptor word in the case of RECFM V files.

The default is

BLOCK_SIZE => *record_size*

for RECFM F files and

BLOCK_SIZE => 4096

for RECFM V files.

Carriage control

This attribute applies to TEXT_IO only, and is intended for files destined to be sent to a printer.

For a file of mode OUT_FILE, this attribute causes the output procedures of TEXT_IO to place a carriage control character as the first character of every output record; '1' (skip to channel 1) if the record follows a page terminator, or space (skip to next line) otherwise. Subsequent characters are output as normal as the result of calls of the output subprograms of TEXT_IO.

For a file of mode IN_FILE, this attribute causes the input procedures of TEXT_IO to interpret the first character of each record as a carriage control character, as described in the previous paragraph. Carriage control characters are not explicitly returned as a result of an input subprogram, but will (for example) affect the result of END_OF_PAGE.

The user should naturally be careful to ensure the carriage control attribute of a file of mode IN_FILE has the same value as that specified when creating the file.

The syntax of the carriage control attribute is as follows:

CARRIAGE_CONTROL => *boolean*

The default is CARRIAGE_CONTROL => FALSE.

8.2 USE_ERROR

The following conditions will cause USE_ERROR to be raised:

- Specifying a FORM parameter whose syntax does not conform to the rules given above.
- Specifying the BLOCK_SIZE FORM parameter attribute to have a value less than RECORD_SIZE.
- Specifying the RECORD_SIZE FORM parameter attribute to have a value of zero (or failing to specify RECORD_SIZE) for instantiations of DIRECT_IO for unconstrained types.
- Specifying a RECORD_SIZE FORM parameter attribute to have a value less than that required to hold the element for instantiations of DIRECT_IO and SEQUENTIAL_IO of constrained types.
- Violating the file sharing rules stated above.
- Attempting to delete a file opened by DDNAME.
- Errors detected whilst reading or writing.

8.3 Text Terminators

Line terminators [14.3] are not implemented using a character, but are implied by the end of physical record.

Page terminators [14.3] are implemented using the EBCDIC character 0C (hexadecimal).

File terminators [14.3] are not implemented using a character, but are implied by the end of physical file.

The user should avoid the explicit output of the character ASCII.FF [C]. If the user explicitly outputs the character ASCII.LF, this is treated as a call of NEW_LINE [14.3.4].

8.4 EBCDIC and ASCII

All I/O using TEXT_IO is performed using ASCII/EBCDIC translation. CHARACTER and STRING values are held internally in ASCII but represented in external files in EBCDIC. For SEQUENTIAL_IO and DIRECT_IO no translation takes place, and the external file contains a binary image of the internal representation of the Ada element (see section 8.5).

It should be noted that the EBCDIC character set is larger than the (7 bit) ASCII and that the use of EBCDIC and ASCII control characters may not produce the desired results when using TEXT_IO (the input and output of control characters is in any case not defined by the Ada language [14.3]). Furthermore, the user is advised to exercise caution in the use of BAR (|) and SHARP (#), which are part of the lexis of Ada; if their use is prevented by translation between ASCII and EBCDIC, EXCLAM (!) and COLON (:), respectively, should be used instead [2.10].

Various translation tables exist to translate between ASCII and EBCDIC. The predefined package EBCDIC is provided to allow access to the translation facilities used by TEXT_IO and SYSTEM_ENVIRONMENT (see User's Guide for VM/CMS, Appendix E).

The specification of this package is as follows:

```
package EBCDIC is

    type EBCDIC_CHARACTER is (
        nul,                    -- 0 = 0h
        soh,                    -- 1 = 1h
        stx,                    -- 2 = 2h
        etx,                    -- 3 = 3h
        E_4,
        ht,                     -- 5 = 5h
        E_6,
        del,                    -- 7 = 7h
        E_8,
        E_9,
        E_A,
        vt,                     -- 11 = 0Bh
        np,                     -- 12 = 0Ch
        cr,                     -- 13 = 0Dh
        so,                     -- 14 = 0Eh
        si,                     -- 15 = 0Fh
        dle,                    -- 16 = 10h
        dc1,                    -- 17 = 11h
        dc2,                    -- 18 = 12h
        dc3,                    -- 19 = 13h
        E_14,
        nl,                     -- 21 = 15h
        bs,                     -- 22 = 16h
        E_17,
        can,                    -- 24 = 18h
        em,                     -- 25 = 19h
        E_1A,
        E_1B,
        E_1C,
        gs,                     -- 29 = 1Dh
        rs,                     -- 30 = 1Eh
        us,                     -- 31 = 1Fh
        E_20,
        E_21,
        fs,                     -- 34 = 22h
        E_23,
        E_24,
        E_25,
        etb,                    -- 38 = 26h
        esc,                    -- 39 = 27h
        E_28,
        E_29,
```

```
    E_2A,
    E_2B,
    E_2C,
    enq,                    -- 45 = 2Dh
    ack,                    -- 46 = 2Eh
    bel,                    -- 47 = 2Fh
    E_30,
    E_31,
    syn,                    -- 50 = 32h
    E_33,
    E_34,
    E_35,
    E_36,
    eot,                    -- 55 = 37h
    E_38,
    E_39,
    E_3A,
    E_3B,
    dc4,                    -- 60 = 3Ch
    nak,                    -- 61 = 3Dh
    E_3E,
    sub,                    -- 63 = 3Fh
    ' ',                    -- 64 = 40h
    E_41,
    E_42,
    E_43,
    E_44,
    E_45,
    E_46,
    E_47,
    E_48,
    E_49,
    E_4A,
    '.',                    -- 75 = 4Bh
    '<',                    -- 76 = 4Ch
    '(',                    -- 77 = 4Dh
    '+',                    -- 78 = 4Eh
    '|',                    -- 79 = 4Fh
    '&',                    -- 80 = 50H
    E_51,
    E_52,
    E_53,
    E_54,
    E_55,
    E_56,
    E_57,
    E_58,
    E_59,
    '!',                    -- 90 = 5Ah
    '$',                    -- 91 = 5Bh
    '*',                    -- 92 = 5Ch
    ')',                    -- 93 = 5Dh
```

```
';',                                    -- 94 = 5Eh
'^',                                    -- 95 = 5Fh
'-',                                    -- 96 = 60h
'/',                                    -- 97 = 61h
E_62,
E_63,
E_64,
E_65,
E_66,
E_67,
E_68,
E_69,
E_6A,
',',                                    --107 = 6Bh
'%',                                    --108 = 6Ch
'_',                                    --109 = 6Dh
'>',                                    --110 = 6Eh
'?',                                    --111 = 6Fh
E_70,
E_71,
E_72,
E_73,
E_74,
E_75,
E_76,
E_77,
E_78,
'`',                                    --121 = 79h
':',                                    --122 = 7Ah
'#',                                    --123 = 7Bh
'@',                                    --124 = 7Ch
''',                                    --125 = 7Dh
'=',                                    --126 = 7Eh
'"',                                    --127 = 7Fh
E_80,
'a',                                    --129 = 81h
'b',                                    --130 = 82h
'c',                                    --131 = 83h
'd',                                    --132 = 84h
'e',                                    --133 = 85h
'f',                                    --134 = 86h
'g',                                    --135 = 87h
'h',                                    --136 = 88h
'i',                                    --137 = 89h
E_8A,
E_8B,
E_8C,
E_8D,
E_8E,
E_8F,
E_90,
'j',                                    --145 = 91h
```

```
    'k',                                    140 = 92h
    'l',                                    147 = 93h
    'm',                                    148 = 94h
    'n',                                    149 = 95h
    'o',                                    150 = 96h
    'p',                                    151 = 97h
    'q',                                    152 = 98h
    'r',                                    153 = 99h
    E_9A,
    E_9B,
    E_9C,
    E_9D,
    E_9E,
    E_9F,
    E_A0,
    '~',                                    161 = 0A1h
    's',                                    162 = 0A2h
    't',                                    163 = 0A3h
    'u',                                    164 = 0A4h
    'v',                                    165 = 0A5h
    'w',                                    166 = 0A6h
    'x',                                    167 = 0A7h
    'y',                                    168 = 0A8h
    'z',                                    169 = 0A9h
    E_AA,
    E_AB,
    E_AC,
    '[',                                    173 = 0ADh
    E_AE,
    E_AF,
    E_B0,
    E_B1,
    E_B2,
    E_B3,
    E_B4,
    E_B5,
    E_B6,
    E_B7,
    E_B8,
    E_B9,
    E_BA,
    E_BB,
    E_BC,
    ']',                                    189 = 0BDh
    E_BE,
    E_BF,
    '{',                                    192 = 0C0h
    'A',                                    193 = 0C1h
    'B',                                    194 = 0C2h
    'C',                                    195 = 0C3h
    'D',                                    196 = 0C4h
    'E',                                    197 = 0C5h
```

```
    'F',                                --198 = 0C6h
    'G',                                --199 = 0C7h
    'H',                                --200 = 0C8h
    'I',                                --201 = 0C9h
    E_CA,
    E_CB,
    E_CC,
    E_CD,
    E_CE,
    E_CF,
    '}',                                --208 = 0D0h
    'J',                                --209 = 0D1h
    'K',                                --210 = 0D2h
    'L',                                --211 = 0D3h
    'M',                                --212 = 0D4h
    'N',                                --213 = 0D5h
    'O',                                --214 = 0D6h
    'P',                                --215 = 0D7h
    'Q',                                --216 = 0D8h
    'R',                                --217 = 0D9h
    E_DA,
    E_DB,
    E_DC,
    E_DD,
    E_DE,
    E_DF,
    '\',                                --224 = 0E0h
    E_E1,
    'S',                                --226 = 0E2h
    'T',                                --227 = 0E3h
    'U',                                --228 = 0E4h
    'V',                                --229 = 0E5h
    'W',                                --230 = 0E6h
    'X',                                --231 = 0E7h
    'Y',                                --232 = 0E8h
    'Z',                                --233 = 0E9h
    E_EA,
    E_EB,
    E_EC,
    E_ED,
    E_EE,
    E_EF,
    '0',                                --240 = 0F0h
    '1',                                --241 = 0F1h
    '2',                                --242 = 0F2h
    '3',                                --243 = 0F3h
    '4',                                --244 = 0F4h
    '5',                                --245 = 0F5h
    '6',                                --246 = 0F6h
    '7',                                --247 = 0F7h
    '8',                                --248 = 0F8h
    '9',                                --249 = 0F9h
```

```
        E_FA,
        E_FB,
        E_FC,
        E_FD,
        E_FE,
        E_FF);

    SEL         : constant EBCDIC_CHARACTER := E_4;
    RNL         : constant EBCDIC_CHARACTER := E_6;
    GE          : constant EBCDIC_CHARACTER := E_8;
    SPS         : constant EBCDIC_CHARACTER := E_9;
    RPT         : constant EBCDIC_CHARACTER := E_A;
    RES         : constant EBCDIC_CHARACTER := E_4;
    ENP         : constant EBCDIC_CHARACTER := E_4;
    POC         : constant EBCDIC_CHARACTER := E_17;
    UBS         : constant EBCDIC_CHARACTER := E_1A;
    CU1         : constant EBCDIC_CHARACTER := E_1B;
    IFS         : constant EBCDIC_CHARACTER := E_1C;
    DS          : constant EBCDIC_CHARACTER := E_20;
    SOS         : constant EBCDIC_CHARACTER := E_21;
    WUS         : constant EBCDIC_CHARACTER := E_23;
    BYP         : constant EBCDIC_CHARACTER := E_24;
    INP         : constant EBCDIC_CHARACTER := E_24;
    LF          : constant EBCDIC_CHARACTER := E_25;
    SA          : constant EBCDIC_CHARACTER := E_28;
    SFE         : constant EBCDIC_CHARACTER := E_29;
    SM          : constant EBCDIC_CHARACTER := E_2A;
    SW          : constant EBCDIC_CHARACTER := E_2A;
    CSP         : constant EBCDIC_CHARACTER := E_2B;
    MFA         : constant EBCDIC_CHARACTER := E_2C;
    IR          : constant EBCDIC_CHARACTER := E_33;
    PP          : constant EBCDIC_CHARACTER := E_34;
    TRN         : constant EBCDIC_CHARACTER := E_35;
    NBS         : constant EBCDIC_CHARACTER := E_36;
    SBS         : constant EBCDIC_CHARACTER := E_38;
    IT          : constant EBCDIC_CHARACTER := E_39;
    RFF         : constant EBCDIC_CHARACTER := E_3A;
    CU3         : constant EBCDIC_CHARACTER := E_3B;
    SP          : constant EBCDIC_CHARACTER := ' ';
    RSP         : constant EBCDIC_CHARACTER := E_41;
    CENT        : constant EBCDIC_CHARACTER := E_4A;
    SHY         : constant EBCDIC_CHARACTER := E_CA;
    HOOK        : constant EBCDIC_CHARACTER := E_CC;
    FORK        : constant EBCDIC_CHARACTER := E_CE;
    NSP         : constant EBCDIC_CHARACTER := E_E1;
    CHAIR       : constant EBCDIC_CHARACTER := E_EC;
    EO          : constant EBCDIC_CHARACTER := E_FF;


    E_0         : constant EBCDIC_CHARACTER := nul;
    E_1         : constant EBCDIC_CHARACTER := soh;
    E_2         : constant EBCDIC_CHARACTER := stx;
```

```
E_3     : constant EBCDIC_CHARACTER := etx;
E_5     : constant EBCDIC_CHARACTER := ht;
E_7     : constant EBCDIC_CHARACTER := del;
E_B     : constant EBCDIC_CHARACTER := vt;
E_C     : constant EBCDIC_CHARACTER := np;
E_D     : constant EBCDIC_CHARACTER := cr;
E_E     : constant EBCDIC_CHARACTER := so;
E_F     : constant EBCDIC_CHARACTER := si;
E_10    : constant EBCDIC_CHARACTER := dle;
E_11    : constant EBCDIC_CHARACTER := dc1;
E_12    : constant EBCDIC_CHARACTER := dc2;
E_13    : constant EBCDIC_CHARACTER := dc3;
E_15    : constant EBCDIC_CHARACTER := nl;
E_16    : constant EBCDIC_CHARACTER := bs;
E_18    : constant EBCDIC_CHARACTER := can;
E_19    : constant EBCDIC_CHARACTER := em;
E_1D    : constant EBCDIC_CHARACTER := gs;
E_1E    : constant EBCDIC_CHARACTER := rs;
E_1F    : constant EBCDIC_CHARACTER := us;
E_22    : constant EBCDIC_CHARACTER := fs;
E_26    : constant EBCDIC_CHARACTER := etb;
E_27    : constant EBCDIC_CHARACTER := esc;
E_2D    : constant EBCDIC_CHARACTER := enq;
E_2E    : constant EBCDIC_CHARACTER := ack;
E_2F    : constant EBCDIC_CHARACTER := bel;
E_32    : constant EBCDIC_CHARACTER := syn;
E_37    : constant EBCDIC_CHARACTER := eot;
E_3C    : constant EBCDIC_CHARACTER := dc4;
E_3D    : constant EBCDIC_CHARACTER := nak;
E_3F    : constant EBCDIC_CHARACTER := sub;
E_40    : constant EBCDIC_CHARACTER := ' ';
E_4B    : constant EBCDIC_CHARACTER := '.';
E_4C    : constant EBCDIC_CHARACTER := '<';
E_4D    : constant EBCDIC_CHARACTER := '(';
E_4E    : constant EBCDIC_CHARACTER := '+';
E_4F    : constant EBCDIC_CHARACTER := '|';
E_50    : constant EBCDIC_CHARACTER := '&';
E_5A    : constant EBCDIC_CHARACTER := '!';
E_5B    : constant EBCDIC_CHARACTER := '$';
E_5C    : constant EBCDIC_CHARACTER := '*';
E_5D    : constant EBCDIC_CHARACTER := ')';
E_5E    : constant EBCDIC_CHARACTER := ';';
E_5F    : constant EBCDIC_CHARACTER := '^';
E_60    : constant EBCDIC_CHARACTER := '-';
E_61    : constant EBCDIC_CHARACTER := '/';
E_6B    : constant EBCDIC_CHARACTER := ',';
E_6C    : constant EBCDIC_CHARACTER := '%';
E_6D    : constant EBCDIC_CHARACTER := '_';
E_6E    : constant EBCDIC_CHARACTER := '>';
E_6F    : constant EBCDIC_CHARACTER := '?';
E_79    : constant EBCDIC_CHARACTER := '`';
E_7A    : constant EBCDIC_CHARACTER := ':';
```

```
E_7B        : constant EBCDIC_CHARACTER := '#';
E_7C        : constant EBCDIC_CHARACTER := '@';
E_7D        : constant EBCDIC_CHARACTER := '';
E_7E        : constant EBCDIC_CHARACTER := '=';
E_7F        : constant EBCDIC_CHARACTER := '"';
E_81        : constant EBCDIC_CHARACTER := 'a';
E_82        : constant EBCDIC_CHARACTER := 'b';
E_83        : constant EBCDIC_CHARACTER := 'c';
E_84        : constant EBCDIC_CHARACTER := 'd';
E_85        : constant EBCDIC_CHARACTER := 'e';
E_86        : constant EBCDIC_CHARACTER := 'f';
E_87        : constant EBCDIC_CHARACTER := 'g';
E_88        : constant EBCDIC_CHARACTER := 'h';
E_89        : constant EBCDIC_CHARACTER := 'i';
E_91        : constant EBCDIC_CHARACTER := 'j';
E_92        : constant EBCDIC_CHARACTER := 'k';
E_93        : constant EBCDIC_CHARACTER := 'l';
E_94        : constant EBCDIC_CHARACTER := 'm';
E_95        : constant EBCDIC_CHARACTER := 'n';
E_96        : constant EBCDIC_CHARACTER := 'o';
E_97        : constant EBCDIC_CHARACTER := 'p';
E_98        : constant EBCDIC_CHARACTER := 'q';
E_99        : constant EBCDIC_CHARACTER := 'r';
E_A1        : constant EBCDIC_CHARACTER := '~';
E_A2        : constant EBCDIC_CHARACTER := 's';
E_A3        : constant EBCDIC_CHARACTER := 't';
E_A4        : constant EBCDIC_CHARACTER := 'u';
E_A5        : constant EBCDIC_CHARACTER := 'v';
E_A6        : constant EBCDIC_CHARACTER := 'w';
E_A7        : constant EBCDIC_CHARACTER := 'x';
E_A8        : constant EBCDIC_CHARACTER := 'y';
E_A9        : constant EBCDIC_CHARACTER := 'z';
E_AD        : constant EBCDIC_CHARACTER := '[';
E_BD        : constant EBCDIC_CHARACTER := ']';
E_C0        : constant EBCDIC_CHARACTER := '{';
E_C1        : constant EBCDIC_CHARACTER := 'A';
E_C2        : constant EBCDIC_CHARACTER := 'B';
E_C3        : constant EBCDIC_CHARACTER := 'C';
E_C4        : constant EBCDIC_CHARACTER := 'D';
E_C5        : constant EBCDIC_CHARACTER := 'E';
E_C6        : constant EBCDIC_CHARACTER := 'F';
E_C7        : constant EBCDIC_CHARACTER := 'G';
E_C8        : constant EBCDIC_CHARACTER := 'H';
E_C9        : constant EBCDIC_CHARACTER := 'I';
E_D0        : constant EBCDIC_CHARACTER := '}';
E_D1        : constant EBCDIC_CHARACTER := 'J';
E_D2        : constant EBCDIC_CHARACTER := 'K';
E_D3        : constant EBCDIC_CHARACTER := 'L';
E_D4        : constant EBCDIC_CHARACTER := 'M';
E_D5        : constant EBCDIC_CHARACTER := 'N';
E_D6        : constant EBCDIC_CHARACTER := 'O';
E_D7        : constant EBCDIC_CHARACTER := 'P';
```

```
        E_D8        : constant EBCDIC_CHARACTER := 'Q';
        E_D9        : constant EBCDIC_CHARACTER := 'R';
        E_E0        : constant EBCDIC_CHARACTER := '\';
        E_E2        : constant EBCDIC_CHARACTER := 'S';
        E_E3        : constant EBCDIC_CHARACTER := 'T';
        E_E4        : constant EBCDIC_CHARACTER := 'U';
        E_E5        : constant EBCDIC_CHARACTER := 'V';
        E_E6        : constant EBCDIC_CHARACTER := 'W';
        E_E7        : constant EBCDIC_CHARACTER := 'X';
        E_E8        : constant EBCDIC_CHARACTER := 'Y';
        E_E9        : constant EBCDIC_CHARACTER := 'Z';
        E_F0        : constant EBCDIC_CHARACTER := '0';
        E_F1        : constant EBCDIC_CHARACTER := '1';
        E_F2        : constant EBCDIC_CHARACTER := '2';
        E_F3        : constant EBCDIC_CHARACTER := '3';
        E_F4        : constant EBCDIC_CHARACTER := '4';
        E_F5        : constant EBCDIC_CHARACTER := '5';
        E_F6        : constant EBCDIC_CHARACTER := '6';
        E_F7        : constant EBCDIC_CHARACTER := '7';
        E_F8        : constant EBCDIC_CHARACTER := '8';
        E_F9        : constant EBCDIC_CHARACTER := '9';

        type EBCDIC_STRING is array (POSITIVE range <>) of EBCDIC_CHARACTER;

        function ASCII_TO_EBCDIC (S : STRING) return EBCDIC_STRING;

        -- CONSTRAINT_ERROR is raised if E_STRING'LENGTH /= A_STRING'LENGTH;
        procedure ASCII_TO_EBCDIC (A_STRING : in STRING;
                                   E_STRING : out EBCDIC_STRING);

        function EBCDIC_TO_ASCII (S : EBCDIC_STRING) return STRING;

        -- CONSTRAINT_ERROR is raised if E_STRING'LENGTH /= A_STRING'LENGTH;
        procedure EBCDIC_TO_ASCII (E_STRING : in EBCDIC_STRING;
                                   A_STRING : out STRING);

    end EBCDIC;
```

The subprograms ASCII_TO_EBCDIC and EBCDIC_TO_ASCII convert between ASCII encoded STRINGs and EBCDIC_STRINGs as appropriate.

The procedures ASCII_TO_EBCDIC and EBCDIC_TO_ASCII are much more efficient than the corresponding functions, as they do not make use of the program heap. Note that if the in and out string parameters are of different lengths (i.e. A_STRING'LENGTH /= E_STRING'LENGTH), the procedures will raise the exception CONSTRAINT_ERROR.

Note that the user may alter the ASCII to EBCDIC and EBCDIC to ASCII mappings used by the Alsys IBM 370 Ada compiler, as described in the installation guide.

If SEQUENTIAL_IO is instantiated with the type EBCDIC_STRING, IO of arbitary EBCDIC strings is possible. Note also that in many mays EBCDIC_STRINGs may

be manipulated exactly as the predefined type STRING; in particular, string literals and catenations are available.

8.5 Characteristics of disk files

Disk files that are have already been created and are opened take on the characteristics that are already associated with that file.

The characteristics of disk files that are created using the predefined input-output packages are set up as described in the below.

TEXT_IO

- RECFM = V, unless the RECORD_SIZE FORM parameter component is specified in which case RECFM = F and the LRECL is as specified.
- A carriage control character is placed in column 1 if the CARRIAGE control component is specified.
- Data is translated between ASCII and EBCDIC so that the external file is readable using other system 370 tools.

SEQUENTIAL_IO

- RECFM = V, unless the RECORD_SIZE FORM parameter component is specified in which case RECFM = F and the LRECL is as specified.
- No translation is performed between ASCII and EBCDIC; the data in the external file is a memory image of the elements written, preceded by a 4-byte length count in the case of unconstrained types for which a RECORD_SIZE component has been specified.

DIRECT_IO

- RECFM=F and LRECL=ELEMENT_TYPE'SIZE/SYSTEM.STORAGE_UNIT for unconstrained types (unlesss overriden by a RECORD_SIZE FORM parameter component), LRECL is defined by the mandatory RECORD_SIZE FORM parameter component for unconstrained types.
- No translation is performed between ASCII and EBCDIC; the data in the external file is a memory image of the elements written, preceded by a 4-byte length count in the case of unconstrained types.
- DIRECT_IO files may be read using SEQUENTIAL_IO (vice-versa if a RECORD_SIZE component is specified).

9 Characteristics of Numeric Types

9.1 Integer Types

The ranges of values for integer types declared in package STANDARD are as follows:

```
SHORT_INTEGER     -32768 .. 32767                 -- 2**15 - 1

INTEGER           -2147483648 .. 2147483647       -- 2**31 - 1
```

For the packages DIRECT_IO and TEXT_IO, the ranges of values for types COUNT and POSITIVE_COUNT are as follows:

```
COUNT             0 .. 2147483647                 -- 2**31 - 1

POSITIVE_COUNT    1 .. 2147483647                 -- 2**31 - 1
```

For the package TEXT_IO, the range of values for the type FIELD is as follows:

```
FIELD             0 .. 255                        -- 2**8 - 1
```

9.2 Floating Point Type Attributes

SHORT_FLOAT

		Approximate value
DIGITS	6	
MANTISSA	21	
EMAX	84	
EPSILON	2.0 ** -20	9.54E-07
SMALL	2.0 ** -85	2.58E-26
LARGE	2.0 ** 84 * (1.0 - 2.0 ** -21)	1.93E+25
SAFE_EMAX	252	
SAFE_SMALL	2.0 ** -253	6.91E-77
SAFE_LARGE	2.0 ** 127 * (1.0 - 2.0 ** -21)	1.70E+38
FIRST	-2.0 ** 252 * (1.0 - 2.0 ** -24)	-7.24E+75
LAST	2.0 ** 252 * (1.0 - 2.0 ** -24)	7.24E+75
MACHINE_RADIX	16	
MACHINE_MANTISSA	6	
MACHINE_EMAX	63	
MACHINE_EMIN	-64	
MACHINE_ROUNDS	FALSE	
MACHINE_OVERFLOWS	TRUE	
SIZE	32	

FLOAT

		Approximate value
DIGITS	15	
MANTISSA	51	
EMAX	204	
EPSILON	2.0 ** -50	8.88E-16
SMALL	2.0 ** -205	1.94E-62
LARGE	2.0 ** 204 * (1.0 - 2.0 ** -51)	2.57E+61
SAFE_EMAX	252	
SAFE_SMALL	2.0 ** -253	6.91E-77
SAFE_LARGE	2.0 ** 252 * (1.0 - 2.0 ** 51)	7.24E+75
FIRST	-2.0 ** 252 * (1.0 - 2.0 ** -56)	-7.24E+75
LAST	2.0 ** 252 * (1.0 - 2.0 ** -56)	7.24E+75
MACHINE_RADIX	16	
MACHINE_MANTISSA	14	
MACHINE_EMAX	63	
MACHINE_EMIN	-64	
MACHINE_ROUNDS	FALSE	
MACHINE_OVERFLOWS	TRUE	
SIZE	64	

LONG_FLOAT

		Approximate value
DIGITS	18	
MANTISSA	61	
EMAX	244	
EPSILON	2.0 ** -60	8.67E-19
SMALL	2.0 ** -245	1.77E-74
LARGE	2.0 ** 244 * (1.0 - 2.0 ** -61)	2.83E+73
SAFE_EMAX	252	
SAFE_SMALL	2.0 ** -253	6.91E-77
SAFE_LARGE	2.0 ** 252 * (1.0 - 2.0 ** -61)	7.24E+75
FIRST	-2.0 ** 252 * (1.0 - 2.0 ** -112)	-7.24E+75
LAST	2.0 ** 252 * (1.0 - 2.0 ** -112)	7.24E+75
MACHINE_RADIX	16	
MACHINE_MANTISSA	28	
MACHINE_EMAX	63	
MACHINE_EMIN	-64	
MACHINE_ROUNDS	FALSE	
MACHINE_OVERFLOWS	TRUE	
SIZE	128	

9.3 Attributes of Type DURATION

DURATION'DELTA	2.0 ** -14
DURATION'SMALL	2.0 ** -14

DURATION'LARGE	131072.0
DURATION'FIRST	-86400.0
DURATION'LAST	86400.0

10 Other Implementation-Dependent Characteristics

10.1 Characteristics of the Heap

All objects created by allocators go into the heap. Also, portions of the Ada Run-Time Executive's representation of task objects, including the task stacks, are allocated in the heap.

All objects whose visibility is linked to a subprogram or block have their storage reclaimed at exit.

10.2 Characteristics of Tasks

The default task stack size is 16 Kbytes, but by using the Binder option TASK the size for all task stacks in a program may be set to any size from 4 Kbytes to 16 Mbytes.

Timeslicing is implemented for task scheduling. The default time slice is 1000 milliseconds, but by using the Binder option SLICE the time slice may be set to any period of 10 milliseconds or more. It is also possible to use this option to specify no timeslicing, i.e. tasks are scheduled only at explicit synchronisation points. Timeslicing is started only upon activation of the first task in the program, so the SLICE option has no effect for sequential programs.

Normal priority rules are followed for preemption, where PRIORITY values run in the range 1 .. 10. All tasks with "undefined" priority (no pragma PRIORITY) are considered to have a priority of 0.

The minimum timeable delay is 10 milliseconds.

The maximum number of active tasks is limited only by memory usage. Tasks release their storage allocation as soon as they have terminated.

The acceptor of a rendezvous executes the accept body code in its own stack. A rendezvous with an empty accept body (e.g. for synchronisation) does not cause a context switch.

The main program waits for completion of all tasks dependent on library packages before terminating. Such tasks may select a terminate alternative only after completion of the main program.

Abnormal completion of an aborted task takes place immediately, except when the abnormal task is the caller of an entry that is engaged in a rendezvous. Any such task becomes abnormally completed as soon as the rendezvous is completed.

If a global deadlock situation arises because every task (including the main program) is waiting for another task, the program is aborted and the state of all tasks is displayed.

10.3 Definition of a Main Program

A main program must be a non-generic, parameterless, library procedure.

10.4 Ordering of Compilation Units

The Alsys IBM 370 Ada Compiler imposes no additional ordering constraints on compilations beyond those required by the language. However, if a generic unit is instantiated during a compilation, its body must be compiled prior to the completion of that compilation [10.3].

10.5 Package SYSTEM_ENVIRONMENT

The implementation-defined package SYSTEM_ENVIRONMENT enables an Ada program to communicate with the environment in which it is executed.

The specification of this package is as follows:

```
package SYSTEM_ENVIRONMENT is

    subtype EXIT_STATUS is INTEGER;


    function ARG_LINE return STRING;

    function ARG_LINE_LENGTH return NATURAL;

    procedure ARG_LINE (LINE : out STRING;
                        LAST : out NATURAL);

    function ARG_START return NATURAL;

    procedure SET_EXIT_STATUS (STATUS : in EXIT_STATUS);


    procedure ABORT_PROGRAM (STATUS : in EXIT_STATUS);


    function EXISTS (FILE : in STRING) return BOOLEAN;

end SYSTEM_ENVIRONMENT;
```

The ARG_LINE subprograms give access to the program PARM string as specified in the JCL used to run the program. The procedure ARG_LINE is more efficient than the corresponding function, as it does not make use of the program heap. The

out parameter LAST specifies the character in LINE which holds the last character of the command line. Note, if LINE is not long enough to hold the command line given, CONSTRAINT_ERROR will be raised. The command line returned includes the name of the program executed, but not any run-time options specified.

The function ARG_START is included for compatibility with the VM/CMS implementation of SYSTEM_ENVIRONMENT; for MVS ARG_START always returns the value 1.

The exit status of the program (returned in register 15 on exit) can be set by a call of SET_EXIT_STATUS. Subsequent calls of SET_EXIT_STATUS will overwrite the exit status, which is by default 0. If SET_EXIT_STATUS is not called, a positive exit code may be set by the Ada Run-Time Executive if an unhandled exception is propagated out of the main subprogram, or if a deadlock situation is detected.

The following exit codes relate to unhandled exceptions:

Exception	**Code**	**Cause of exception**
NUMERIC_ERROR:		
	1	divide by zero
	2	numeric overflow
CONSTRAINT_ERROR:		
	3	discriminant error
	4	lower bound index error
	5	upper bound index error
	6	length error
	7	lower bound range error
	8	upper bound range error
	9	null access value
STORAGE_ERROR:		
	10	frame overflow (overflow on subprogram entry)
	11	stack overflow (overflow otherwise)
	12	heap overflow
PROGRAM_ERROR:		
	13	access before elaboration
	14	function left without return
SPURIOUS_ERROR:		
	15-20	<an erroneous program>
NUMERIC_ERROR	21	(other than for the above reasons)
CONSTRAINT_ERROR	22	(other than for the above reasons)
	23	anonymously raised exception (an exception re-raised using the raise statement without an exception name)
	24	<unused>
	25	static exception (an exception raised using the raise statement with an exception name)

Code 100 is used if a deadlocking situation is detected and the program is aborted as a result.

Codes 1000-1999 are used to indicate other anomalous conditions in the initialisation of the program, messages concerning which are displayed on the terminal.

The EXISTS functions returns a boolean to indicate whether the file specified by the file name string exists or not.

11 Limitations

11.1 Compiler Limitations

- The maximum identifier length is 255 characters.
- The maximum line length is 255 characters.
- The maximum number of unique identifiers per compilation unit is 1500.
- The maximum number of compilation units in a library is 1023.
- The maximum number of subunits per compilation unit is 100.
- The maximum size of the generated code for a single program unit (subprogram or task body) is 128 Kbytes.
- There is no limit (apart from machine addressing range) on the size of the generated code for a single compilation unit.
- There is no limit (apart from machine addressing range) on the size of a single array or record object.
- The maximum size of a single stack frame is 64 Kbytes including the data for inner package subunits which is "unnested" to the parent frame.
- The maximum amount of data in the global data area of a single compilation unit is 64 Kbytes, including compiler-generated data.

APPENDIX C

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
$BIG_ID1 Identifier the size of the maximum input line length with varying last character.	X2345678901234567890123456789012345678901234567890123 45678901234567890123456789012345 A....A1 |----| 199 characters
$BIG_ID2 Identifier the size of the maximum input line length with varying last character.	X2345678901234567890123456789012345678901234567890123 45678901234567890123456789012345 A....A2 |----| 199 characters
$BIG_ID3 Identifier the size of the maximum input line length with varying middle character.	X2345678901234567890123456789012345678901234567890123 45678901234567890123456789012345 A....A3A....A |-99-| |-100-| characters
$BIG_ID4 Identifier the size of the maximum input line length with varying middle character.	X2345678901234567890123456789012345678901234567890123 45678901234567890123456789012345 A....A4A....A |-99-| |-100-| characters
$BIG_INT_LIT An integer literal of value 298 with enough leading zeroes so that it is the size of the maximum length.	0....0298 |----| 252 characters
$BIG_REAL_LIT A universal real literal of value 690.0 with enough leading zeroes to be the size of the maximum line length.	0....0690.0 |----| 250 characters

Name and Meaning	Value
$BIG_STRING1 A string literal which when catenated with BIG_STRING2 yields the image of BIG_ID1.	"X2345678901234567890123456789012345678901234567890123456789012345 A....A" |----| 72 characters
$BIG_STRING2 A string literal which when catenated to the end of BIG_STRING1 yields the image of BIG_ID1.	"A....A1" |----| 127 characters
$BLANKS A sequence of blanks twenty characters less than the size of the maximum line length.	235 blanks
$COUNT_LAST A universal integer literal whose value is TEXT_IO.COUNT'LAST.	2147483647
$FIELD_LAST A universal integer literal whose value is TEXT_IO.FIELD'LAST.	255
$FILE_NAME_WITH_BAD_CHARS An external file name that either contains invalid character or is too long.	T??????? LISTING A1
$FILE_NAME_WITH_WILD_CARD_CHAR An external file name that either contains a wild card character or is too long.	TOOLONGNAME TOOLONGTYPE TOOLONGMODE
$GREATER_THAN_DURATION A universal real literal that lies between DURATION'BASE'LAST and DURATION'LAST or any value in the range of DURATION.	100000.0
$GREATER_THAN_DURATION_BASE_LAST A universal real literal that is greater than DURATION'BASE'LAST.	10000000.0
$ILLEGAL_EXTERNAL_FILE_NAME1 An external file name which contains invalid characters.	T??????? LISTING A1

Name and Meaning	Value
$GREATER_THAN_DURATION_BASE_LAST A universal real literal that is greater than DURATION'BASE' LAST.	10000000.0
$ILLEGAL_EXTERNAL_FILE_NAME1 An external file name which contains invalid characters.	T??????? LISTING A1
$ILLEGAL_EXTERNAL_FILE_NAME2 An external file name which is too long.	TOOLONGNAME TOOLONGTYPE TOOLONGMODE
$INTEGER_FIRST A universal integer literal whose value is INTEGER'FIRST.	-2147483648
$INTEGER_LAST A universal integer literal whose value is INTEGER'LAST.	2147483647
$INTEGER_LAST_PLUS_1 A universal integer literal whose value is INTEGER'LAST + 1.	2147483648
$LESS_THAN_DURATION A universal real literal that lies between DURATION'BASE' FIRST and DURATION'FIRST or any value in the range of DURATION.	-100000.0
$LESS_THAN_DURATION_BASE_FIRST A universal real literal that is less than DURATION'BASE' FIRST	-10000000.0
$MAX-DIGITS Maximum digits supported for floating-point types.	18
$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

Name and Meaning	Value
$MAX_INT_PLUS_1 A universal integer literal whose value is SYSTEM.MAX_INT+1.	2147483648
$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.	2:0....011: |----| 250 characters
$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.	16:0....0F.E: |----| 248 characters
$MAX_STRING_LITERAL A string literal of size MAX_IN_LEN, including the quote characters.	"X2345678901234567890123456789012345678901234567890123 A....A3" |----| 197 characters
$MIN_INT A universal integer literal whose value is SYSTEM.MIN_INT.	-2147483648
$NAME A name of a predefined numeric type other than FLOAT, INTEGER, SHORT_FLOAT, SHORT_INTEGER, LONG_FLOAT, or LONG_INTEGER.	NO_SUCH_TYPE
$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.	8#20000000000#

APPENDIX D

WITHDRAWN TESTS

Some tests are withdrawn from the ACVC because they do not conform to the Ada Standard. The following 25 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.

B28003A: A basic declaration (line 36) wrongly follows a later declaration.

E28005C: This test requires that 'PRAGMA LIST (ON);' not appear in a listing that has been suspended by a previous "pragma LIST (OFF);"; the Ada Standard is not clear on this point, and the matter will be reviewed by the ALMP.

C34004A: The expression in line 168 wrongly yields a value outside of the range of the target type T, raising CONSTRAINT_ERROR.

C35502P: The equality operators in lines 62 & 69 should be inequality operators.

A35902C: Line 17's assignment of the nominal upper bound of a fixed-point type to an object of that type raises CONSTRAINT_ERROR for that value lies outside of the actual range of the type.

C35904A: The elaboration of the fixed-point subtype on line 28 wrongly raises CONSTRAINT_ERROR, because its upper bound exceeds that of the type.

C35A03E, These tests assume that attribute 'MANTISSA returns 0 when applied to a fixed-point type with a null range, but the Ada Standard doesn't support this assumption.

C35A03R These tests assume that attribute 'MANTISSA returns 0 when applied to a fixed-point type with a null range, but the Ada Standard doesn't support this assumption.

C37213H The subtype declaration of SCONS in line 100 is wrongly expected to raise an exception when elaborated.

C37213J The aggregate in line 451 wrongly raises CONSTRAINT_ERROR.

C37215C Various discriminant constraints are wrongly expected to be incompatible with type CONS.

C37215E Various discriminant constraints are wrongly expected to be incompatible with type CONS.

C37215G Various discriminant constraints are wrongly expected to be incompatible with type CONS.

C37215H Various discriminant constraints are wrongly expected to be incompatible with type CONS.

C38102C The fixed-point conversion on line 25 wrongly raises CONSTRAINT_ERROR.

C41402A 'STORAGE-SIZE' is wrongly applied to an object of an access type.

C45614C REPORT.IDENT_INT has an argument of the wrong type (LONG_INTEGER).

A74016C A bound specified in a fixed-point subtype declaration lies outside that calculated for the base type, raising CONSTRAING_ERROR. Errors of this sort occur re lines 37 & 59, 142 & 143, 16 & 48 and 252 & 253 of the four tests, respectively (and possibly elsewhere).

C85018B A bound specified in a fixed-point subtype declaration lies outside that calculated for the base type, raising CONSTRAING_ERROR. Errors of this sort occur re lines 37 & 59, 142 & 143, 16 & 48 and 252 & 253 of the four tests, respectively (and possibly elsewhere).

C87B04B A bound specified in a fixed-point subtype declaration lies outside that calculated for the base type, raising CONSTRAING_ERROR. Errors of this sort occur re lines 37 & 59, 142 & 143, 16 & 48 and 252 & 253 of the four tests, respectively (and possibly elsewhere).

CC1311B A bound specified in a fixed-point subtype declaration lies outside that calculated for the base type, raising CONSTRAING_ERROR. Errors of this sort occur re lines 37 &

59, 142 & 143, 16 & 48 and 252 & 253 of the four tests, respectively (and possibly elsewhere).

BC3105A Lines 159..168 are wrongly expected to be incorrect; they are correct.

AD1A01A The declaration of subtype INT3 raises CONSTRAINT_ERROR for implementations that select INT'SIZE to be 16 or greater.

CE2401H The record aggregates in lines 105 & 117 contain the wrong values.

CE3208A This test expects that an attempt to open the default output file (after it was closed) with mode IN_FILE raises NAME_ERROR or USE_ERROR; by Commentary AI-00048, MODE_ERROR should be raised.