### Ada Compiler Validation Summary Report

**Ada Compiler Validation Summary Report**: Alsys Limited, AlsyCOMP 006, Version 4.1, IBM 9370 Model 90 (host) to IBM 370 3084Q (target) 8B1216N1.10015

**Performing Organization and Address**
National Computing Centre Limited, Manchester, United Kingdom.

**Controlled Office Name and Address**
Ada Joint Program Office
United States Department of Defense
Washington, DC 20301-3081

**Monitoring Agency Name and Address (if different from Controlling Office)**
National Computing Centre Limited, Manchester, United Kingdom.

**Distribution Statement (of this report)**
Approved for public release; distribution unlimited.

---


---

**Abstract**
Alsys Limited, AlsyCOMP 006, Version 4.1, National Computing Centre, IBM 9370 Model 90 under VM/1S CMS release 5.1 (host) to IBM 370 3084Q under MVS 3.2 (target), ACVC 1.10
AVF Control Number: AVF-VSR-90502-45

Ada COMPILER
VALIDATION SUMMARY REPORT:
Certificate Number: #881216N1.10015
Alsys Limited
AlsyCOMP.006 Version 4.1
IBM 9370 Model 90 / IBM 370 3084Q

Completion of On-Site Testing:
16th December 1988

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

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

Compiler Name : AlsyCOMP_006 Version 4.1
Certificate Number : #881216NI.10015
Host : IBM 9370 Model 90 under VM/IS CMS release 5.1
Target : IBM 370 3084Q under MVS 3.2

Testing Completed 16th December 1988 Using ACVC 1.10

This report has been reviewed and is approved.

J. Pink
The National Computing Centre Limited
Jane Pink
Testing Services
Oxford Road
Manchester
M1 7ED
United Kingdom

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

Ada Joint Program Office
Dr. John Solomond
Director
Washington D.C. 20301
Ada Compiler Validation Summary Report:

Compiler Name: AlsyCOMP_006 Version 4.1
Certificate Number: F881216N1.10015
Host: IBM 9370 Model 90 under VM/IS CMS release 5.1
Target: IBM 370 3084Q under MVS 3.2

Testing Completed 16th December 1988 Using ACVC 1.10

This report has been reviewed and is approved.

J. Pink
The National Computing Centre Limited
Jane Pink
Testing Services
Oxford Road
Manchester
M1 7ED
United Kingdom

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

Ada Joint Program Office
Dr. John Solomond
Director
Washington D.C. 20301
# TABLE OF CONTENTS

## CHAPTER 1

**INTRODUCTION**

1.1 PURPOSE OF THIS VALIDATION SUMMARY REPORT .......................... 2
1.2 USE OF THIS VALIDATION SUMMARY REPORT .............................. 2
1.3 REFERENCES .................................................. 3
1.4 DEFINITION OF TERMS .......................................... 3
1.5 ACVC TEST CLASSES ............................................. 5

## CHAPTER 2

**CONFIGURATION INFORMATION**

2.1 CONFIGURATION TESTED ........................................... 1
2.2 IMPLEMENTATION CHARACTERISTICS ................................ 2

## CHAPTER 3

**TEST INFORMATION**

3.1 TEST RESULTS .................................................. 1
3.2 SUMMARY OF TEST RESULTS BY CLASS ................................ 1
3.3 SUMMARY OF TEST RESULTS BY CHAPTER ............................... 1
3.4 WITHDRAWN TESTS .............................................. 2
3.5 INAPPLICABLE TESTS ............................................. 2
3.6 TEST, PROCESSING, AND EVALUATION MODIFICATIONS ..................... 6
3.7 ADDITIONAL TESTING INFORMATION ................................ 7
   3.7.1 Prevalidation ............................................ 7
   3.7.2 Test Method ............................................. 7
   3.7.3 Test Site ............................................... 10

## APPENDIX A

DEARATION OF CONFORMANCE

DEARATION OF CONFORMANCE ........................................ 2

## APPENDIX B

APPENDIX F OF THE Ada STANDARD

package STANDARD ................................................. 1
APPENDIX F OF THE Ada STANDARD ................................ 2

## APPENDIX C

TEST PARAMETERS

MACRO DEFINITIONS ................................................ 1
APPENDIX D

WITHDRAWN TESTS

WITHDRAWN TEST LIST .................................. 1
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 language constructs not supported by the compiler but 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 The National Computing Centre Limited according to procedures established by the Ada Joint Program Office and administered by the Ada Validation Organization (AVO). On-site testing was completed 16th December 1988 at Alsys Limited, Partridge House, Newtown Road, Henley-on-Thames, Oxfordshire, RG9 1EN, United Kingdom.

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
INTRODUCTION

or from:

The National Computing Centre Limited
Testing Services
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,

2. Ada Compiler Validation Procedures and Guidelines,

3. Ada Compiler Validation Capability Implementers' Guide,

   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 to the point addressed by a comment on the Ada Standard. These comments are given a unique identification number having the form AI-ddddd.

Applicant: The agency requesting validation.

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

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

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

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

Host: The computer on which the compiler resides.

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

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

Target: The computer for which a compiler generates code.

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

Class A tests check that legal Ada programs can be successfully compiled and executed. There are no explicit program components in a Class A test to check semantics. For example, a Class A test checks that reserved words of another language (other than those already reserved in the Ada language) are not treated as reserved words by an Ada compiler. A Class A test is passed if no errors are detected at compile time and the program executes to produce a PASSED message.

Class B tests check that a compiler detects illegal language usage. Class B tests are not executable. Each test in this class is compiled and the resulting compilation listing is examined to verify that every syntax or semantic error in the test is detected. A Class B test is passed if every illegal construct that it contains is detected by the compiler.

Class C tests check 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
INTRODUCTION

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. In some cases an implementation may legitimately detect errors during compilation of the test.

Two library units, the package REPORT and the procedure CHECK_FILE, support the self-checking features of the executable tests. The package REPORT provides the mechanism by which executable tests report PASSED, FAILED, or NOT APPLICABLE results. It also provides a set of identity functions used to defeat some compiler optimizations allowed by the Ada Standard that would circumvent a test objective. The procedure CHECK_FILE is used to check the contents of text files written by some of the Class C tests for chapter 14 of the Ada Standard. The operation of REPORT and CHECK_FILE is checked by a set of executable tests. These tests produce messages that are examined to verify that the units are operating correctly. If these units are not operating correctly, then the validation is not attempted.

The text of the tests in the ACVC follow conventions that are intended to ensure that the tests are reasonably portable without modification. For example, the tests make use of only the basic set of 55 characters, contain lines with a maximum length of 72 characters, use small numeric values, and place features that may not be supported by all implementations in separate tests. However, some tests contain values that require the test to be customized according to implementation-specific values—for example, an illegal file name. A list of the values used for this validation is provided in Appendix C.
A compiler must correctly process each of the tests in the suite and demonstrate conformity to the Ada Standard by either meeting the pass criteria given for the test or by showing that the test is inapplicable to the implementation. The applicability of a test to an implementation is considered each time the implementation is validated. A test that is inapplicable for one validation is not necessarily inapplicable for a subsequent validation. Any test that was determined to contain an illegal language construct or an erroneous language construct is withdrawn from the ACVC and, therefore, is not used in testing a compiler. The tests withdrawn at the time of this validation are given in Appendix D.
CHAPTER 2

2.1 CONFIGURATION TESTED

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

Compiler : AlsyCOMP_006 Version 4.1
ACVC Version : 1.10
Certificate Number : #881216N1.10015
Host Computer :
  Machine : IBM 9370 Model 90
  Operating System : VM/IS CMS release 5.1
  Memory Size : 8 Mbytes
Target Computer :
  Machine : IBM 370 3084Q
  Operating System : MVS 3.2
  Memory Size : 1 Mbyte
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 a compilation containing 723 variables in the same declarative part. (See test D29002K.)
  The compiler correctly processes tests containing loop statements nested to 65 levels. (See tests D55A03A..H (8 tests).)
  The compiler correctly processes tests containing block statements nested to 65 levels. (See test D56001B.)
  The compiler correctly processes tests containing recursive procedures separately compiled as subunits nested to 17 levels. (See tests D64005E..G (3 tests).)

- Predefined types.
  This implementation supports the additional predefined types SHORT INTEGER, SHORT SHORT INTEGER, SHORT FLOAT and LONG_FLOAT in the package STANDARD. (See tests B86001T..Z (7 tests).)

- Expression evaluation.
  The order in which expressions are evaluated and the time at which constraints are checked are not defined by the language. While the ACVC tests do not specifically attempt to determine the order of evaluation of expressions, test results indicate the following:
  No 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 and uses all extra bits for extra range. (See test C35903A.)

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.)

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.)

Underflow is not gradual. (See tests C45524A..Z.)

- Rounding.

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

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

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

The method used for rounding to integer in static universal real expressions is 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.)
CONFIGURATION INFORMATION

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

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 subtypes are 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. (See test C52013A.)

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.

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.)

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

- Pragmas.
  
The pragma INLINE is supported for function or procedure calls within a body. The pragma INLINE for function calls within a declaration is not supported. (See tests LA3004A..B, EA3004C..D, and CA3004E..F.)

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

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

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

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

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

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

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

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

- Input and 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 or 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..E, CE2102N, and CE2102P.)

Modes IN_FILE, OUT_FILE, and INOUT_FILE are supported for DIRECT_IO. (See tests CE2102F, CE2102I..J, CE2102R, CE2102T, and CE2102V.)

Modes IN_FILE and OUT_FILE are supported for text files. (See tests CE3102E and CE3102I..K.)

RESET and DELETE operations are supported for SEQUENTIAL_IO. (See tests CE2102G and CE2102X.)

RESET and DELETE operations are supported for DIRECT_IO. (See tests CE2102K and CE2102Y.)

RESET and DELETE operations are supported for text files. (See tests CE3102F..G, CE3104C, CE3110A, and CE3114A.)

Overwriting to a sequential file does not truncate the file. (See test CE2208B.)
Temporary sequential files are given names and deleted when closed. (See test CE2108A.)

Temporary direct files are given names and deleted when closed. (See test CE2108C.)

Temporary text files are given names and deleted when closed. (See test CE3112A.)

More than one internal file can be associated with each external file for sequential files when reading only. (See tests CE2107A..E, CE2102L, CE2110B, and CE2111D.)

More than one internal file can be associated with each external file for direct files when reading only. (See tests CE2107F..I, CE2110D and CE2111H.)

More than one internal file can be associated with each external file for text files when reading only. (See tests CE3111A..E, CE3114B, and CE3115A.)
CHAPTER 3

TEST INFORMATION

3.1 TEST RESULTS

Version 1.10 of the ACVC comprises 3717 tests. When this compiler was tested, 36 tests had been withdrawn because of test errors. The AVF determined that 342 tests were inapplicable to this implementation. All inapplicable tests were processed during validation testing except for 159 executable tests that use floating-point precision exceeding that supported by the implementation. Modifications to the code, processing, or grading for 45 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

<table>
<thead>
<tr>
<th>RESULT</th>
<th>TEST CLASS</th>
<th>TOTAL</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>Passed</td>
<td>128</td>
<td>1133</td>
</tr>
<tr>
<td>Inapplicable</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>Withdrawn</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>TOTAL</td>
<td>130</td>
<td>1140</td>
</tr>
</tbody>
</table>

3.3 SUMMARY OF TEST RESULTS BY CHAPTER

<table>
<thead>
<tr>
<th>RESULT</th>
<th>CHAPTER</th>
<th>TOTAL</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>Passed</td>
<td>202</td>
<td>593</td>
</tr>
<tr>
<td>Inappl</td>
<td>11</td>
<td>56</td>
</tr>
<tr>
<td>Withd</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>TOTAL</td>
<td>213</td>
<td>650</td>
</tr>
</tbody>
</table>
3.4 WITHDRAWN TESTS

The following 36 tests were withdrawn from ACVC Version 1.10 at the time of this validation:

A39005G       B97102E
BC3009B        CD2A62D
CD2A73A..D (4 tests) CD2A76A..D (4 tests)
CD2A81G        CD2A83G
CD2A84M..N (2 tests) CD50110
CD2B15C        CD7205C
CD5007B        CD7105A
CD7203B        CD7204B
CD7205D        CE2107I
CE3111C        CE3301A
CE3411B

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, 342 tests were inapplicable for the reasons indicated:

- The following 159 tests are not applicable because they have floating-point type declarations requiring more digits than SYSTEM.MAX_DIGITS:

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

- The following 16 tests are not applicable because this implementation does not support a predefined type LONG_INTEGER:

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

- C45531M..P (4 tests), C45532M..P (4 tests) are not applicable because the size of a mantissa of a fixed point type is limited to 31 bits.

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

- B86001Z is not applicable because this implementation supports no predefined floating-point type with a name other than FLOAT, LONG_FLOAT, or SHORT_FLOAT.

- C86001F is not applicable because, for this implementation, the package TEXT_IO is dependent upon package SYSTEM. This test redefines package SYSTEM, making package TEXT_IO, and hence package REPORT, obsolete.

- CD1009C, CD2A41A..E (5 tests) and CD2A42A..J (10 tests) are not applicable because SIZE clause on FLOAT is not supported.

- The following 26 tests are all inapplicable for this implementation because length clauses on a type derived from a private type are not supported outside the defining package.


- CD1C04B, CD1C04E and CD4051A..D (4 tests) are not applicable because representation clauses on derived records or derived tasks are not supported.
The following 25 tests are inapplicable because LENGTH clause on an array or record would require change of representation of the components or elements.

CD2A61A..D (4 tests)  CD2A61F
CD2A61H..L (5 tests)  CD2A62A..C (3 tests)
CD2A71A..D (4 tests)  CD2A72C..D (2 tests)
CD2A74A..D (4 tests)  CD2A75C..D (2 tests)

CD2A84B..I (8 tests) and CD2A84K..L (2 tests) are not applicable because the minimum size for a 'SIZE clause applied to the access type is 32 bits.

The following 30 tests are not applicable because ADDRESS clauses for constants are not supported.

CD5011B  CD5011D  CD5011F  CD5011H  CD5011L
CD5011N  CD5011R  CD5011S  CD5012C  CD5012D
CD5012G  CD5012H  CD5012L  CD5013B  CD5013D
CD5013F  CD5013H  CD5013L  CD5013N  CD5013R
CD5014B  CD5014D  CD5014F  CD5014H  CD5014J
CD5014L  CD5014N  CD5014R  CD5014U  CD5014W

CD5012J, CD5013S and CD5014S are not applicable because ADDRESS clauses for tasks are not supported.

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

CE2102D is inapplicable because this implementation supports CREATE with IN_FILE mode for SEQUENTIAL_IO.

CE2102E is inapplicable because this implementation supports CREATE with OUT_FILE mode for SEQUENTIAL_IO.

CE2102F is inapplicable because this implementation supports CREATE with INOUT_FILE mode for DIRECT_IO.

CE2102I is inapplicable because this implementation supports CREATE with IN_FILE mode for DIRECT_IO.

CE2102J is inapplicable because this implementation supports CREATE with OUT_FILE mode for DIRECT_IO.

CE2102N is inapplicable because this implementation supports OPEN with IN_FILE mode for SEQUENTIAL_IO.
TEST INFORMATION

- CE21020 is inapplicable because this implementation supports RESET with IN_FILE mode for SEQUENTIAL_IO.
- CE2102P is inapplicable because this implementation supports open with OUT_FILE mode for SEQUENTIAL_IO.
- CE2102Q is inapplicable because this implementation supports RESET with OUT_FILE mode for SEQUENTIAL_IO.
- CE2102R is inapplicable because this implementation supports open with INOUT_FILE mode for DIRECT_IO.
- CE2102S is inapplicable because this implementation supports RESET with INOUT_FILE mode for DIRECT_IO.
- CE2102T is inapplicable because this implementation supports open with IN_FILE mode for DIRECT_IO.
- CE2102U is inapplicable because this implementation supports RESET with IN_FILE mode for DIRECT_IO.
- CE2102V is inapplicable because this implementation supports open with OUT_FILE mode for DIRECT_IO.
- CE2102W is inapplicable because this implementation supports RESET with OUT_FILE mode for DIRECT_IO.

- CE2107B..E (4 tests), CE2107L, CE2110B and CE2111D are not applicable because multiple internal files cannot be associated with the same external file when one or more files is writing for sequential files. The proper exception is raised when multiple access is attempted.

- CE2107G..H (2 tests), CE2110D and CE2111H are not applicable because multiple internal files cannot be associated with the same external file when one or more files is writing for direct files. The proper exception is raised when multiple access is attempted.

- CE3102E is inapplicable because this implementation supports CREATE with IN_FILE mode for text files.
- CE3102F is inapplicable because this implementation supports RESET for text files.
- CE3102G is inapplicable because this implementation supports deletion of an external file for text files.
CE3102I is inapplicable because this implementation supports CREATE with OUT_FILE mode for text files.

CE3102J is inapplicable because this implementation supports OPEN with IN_FILE mode for text files.

CE3102K is inapplicable because this implementation supports OPEN with OUT_FILE mode for text files.

CE3111B, CE3111D..E (2 tests), CE3114B, and CE3115A are not applicable because multiple internal files cannot be associated with the same external file when one or more files is writing for text files. The proper exception is raised when multiple access is attempted.

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

Modifications were required for 45 tests.

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

- B23004A
- B24007A
- B24009A
- B26005A
- B28003A
- B28003C
- B32202A
- B32202B
- B32202C
- B33001A
- B37004A
- B45102A
- B61012A
- B62001B
- B62001C
- B62001D
- B74304A
- B74401F
- B74401R
- B91004A
- B95069A
- B95069B
- B97103A
- BA1101B
- BC2001D
- BC3009C
- BD5005B

The following tests were split in order to show that features not supported caused errors to be raised.

- CD2A62A..B (2 tests)
- CD2A72A..B (2 tests)
- CD2A75A..B (2 tests)
- CD2A84B..I (8 tests)
EA3004D, when processed, produces only two of the expected three errors: the implementation fails to detect an error on line 27 of file EA3004D6M. This is because the pragma INLINE has no effect when its object is within a package specification. The task was reordered to compile files D2 and D3 after file D5 (the re-compilation of the "with"ed package that makes the various earlier units obsolete), the re-ordered test executed and produced the expected NOT APPLICABLE result (as though INLINE were not supported at all). The re-ordering of EA3004D test files was: 0-1-4-5-2-3-6. The AVO ruled that the test should be counted as passed.

3.7 ADDITIONAL TESTING INFORMATION

3.7.1 Prevalidation

Prior to validation, a set of test results for ACVC Version 1.10 produced by the AlsyCOMP_006 Version 4.1 was submitted to the AVF by the applicant for review. Analysis of these results demonstrated that the compiler successfully passed all applicable tests, and the compiler exhibited the expected behavior on all inapplicable tests.

3.7.2 Test Method

Testing of the AlsyCOMP_006 using ACVC Version 1.10 was conducted on-site by a validation team from the AVF. The configuration in which the testing was performed is described by the following designations of hardware and software components:

Host computer : IBM 9370
Host operating system : VM/IS CMS release 5.1
Target computer : IBM 9370
Target operating system : VM/IS CMS release 5.1
Compiler : AlsyCOMP_006 Version 4.1
Pre-linker : AlsyCOMP_006 Version 4.1
Assembler : AlsyCOMP_006 Version 4.1
Linker : VM/IS CMS release 5.1
Runtime System : AlsyCOMP_006 Version 4.1

A magnetic tape containing all tests except for withdrawn tests and tests requiring unsupported floating-point precisions was taken on-site by the validation team for processing. Tests that make use of implementation-specific values were customized before being written to the 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 files were loaded onto a Sun 3/160 computer (using UNIX BSD 4.2) which then did all the required splits to the relevant tests using the UNIX ED editor. These tests were then loaded onto magnetic tape and transferred to the host machine.

The host machine then proceeded to compile, bind and the object images were transferred to the target machine where they were run. The output was transferred back to the Sun machine for printing. The procedure used to transfer the files back to the Sun machine was provided by Sun-IBM file transfer software.

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

<table>
<thead>
<tr>
<th>OPTION</th>
<th>EFFECT</th>
</tr>
</thead>
<tbody>
<tr>
<td>SOURCE =&gt; source_name</td>
<td>expects the file 'source_file' to contain Ada source code.</td>
</tr>
<tr>
<td>LIBRARY =&gt; library_name</td>
<td>expects this to reference the Ada library.</td>
</tr>
<tr>
<td>ERRORS =&gt; 999</td>
<td>maximum number of compilation errors permitted before the compiler terminates the compilation.</td>
</tr>
<tr>
<td>LEVEL =&gt; CODE</td>
<td>a complete compilation takes place, transferring source code into object code.</td>
</tr>
<tr>
<td>CHECKS =&gt; ALL</td>
<td>all run time checks are performed.</td>
</tr>
<tr>
<td>GENERICS =&gt; INLINE</td>
<td>places the code generic instantiations inline in the same unit as the unit that contains the instantiation.</td>
</tr>
<tr>
<td>OUTPUT =&gt; filename</td>
<td>writes the output to a file with name filename.</td>
</tr>
<tr>
<td>Parameter</td>
<td>Value</td>
</tr>
<tr>
<td>-----------------</td>
<td>---------</td>
</tr>
<tr>
<td>WARNING</td>
<td>NO</td>
</tr>
<tr>
<td>TEXT</td>
<td>YES</td>
</tr>
<tr>
<td>DETAIL</td>
<td>YES</td>
</tr>
<tr>
<td>ASSEMBLY</td>
<td>NONE</td>
</tr>
<tr>
<td>CALLS</td>
<td>NORMAL</td>
</tr>
<tr>
<td>REDUCTION</td>
<td>NONE</td>
</tr>
<tr>
<td>OBJECT</td>
<td>PEEPHOLE</td>
</tr>
<tr>
<td>TREE</td>
<td>NO</td>
</tr>
<tr>
<td>STACK</td>
<td>1024</td>
</tr>
<tr>
<td>GLOBAL</td>
<td>1024</td>
</tr>
<tr>
<td>UNNESTED</td>
<td>16</td>
</tr>
<tr>
<td>SHOW</td>
<td>NONE</td>
</tr>
<tr>
<td>FILE_WIDTH</td>
<td>80</td>
</tr>
<tr>
<td>FILE_LENGTH</td>
<td>NO</td>
</tr>
</tbody>
</table>

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

Testing was conducted at Alsys Limited, Partridge House, Newtown Road, Henley-on-Thames, Oxfordshire, RG9 1EN, and was completed on 16th December 1988.
APPENDIX A

DECLARATION OF CONFORMANCE

Alsys Limited has submitted the following Declaration of Conformance concerning the AlsyCOMP_006.
DECLARATION OF CONFORMANCE

Compiler Implementor: Alsys Limited
Partridge House
Newtown Road
Henley-on-Thames
Oxfordshire RG9 1EN

Ada Validation Facility: The National Computing Centre Limited,
Oxford Road
Manchester
M1 7ED
United Kingdom

Ada Compiler Validation Capability (ACVC) Version: 1.10

Base Configuration

Base Compiler Name: AlsyCOMP_006 version 4.1
Host Architecture ISA: IBM 9370 Model 90
Host Operating System: VM/IS CMS release 5.1
Target Architecture ISA: IBM 370 3084Q
Target Operating System: 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.

Signed ___________________________ Date 15/12/85

Title ___________________________
Owner's Declaration

I, the undersigned, representing Alsys Limited, 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 declare that all of the Ada language compilers listed, and their host/target performance, are in compliance with the Ada Language Standard ANSI/MIL-STD-1815A.

Signed ___________________________ Date ____________

Title ____________________________
APPENDIX F OF THE Ada STANDARD

The only allowed implementation dependencies correspond to implementation-dependent pragmas, to certain machine-dependent conventions as mentioned in chapter 13 of the Ada Standard, and to certain allowed restrictions on representation clauses. The implementation-dependent characteristics of the AlsyCOMP 006 Version 4.1, are described in this Appendix provided by Alsys Limited for this compiler. Any reference in this appendix are to compiler documentation and not to this report. Implementation-specific portions of the package STANDARD--not a part of Appendix F--are:

package STANDARD is

... 

type INTEGER is range -2147483648 .. 2147483647;
type SHORT_INTEGER is range -32768 .. 32767;
type SHORT_SHORT_INTEGER is range -128 .. 127;

type FLOAT is digits 15 range -7.24E+75 .. 7.24E+75;
type SHORT_FLOAT is digits 6 range -7.24E+75 .. 7.24E+75;
type LONG_FLOAT is digits 18 range -7.24E+75 .. 7.24E+75;

type DURATION is delta 2.0**-14 range -86400.0 .. 86400.0;

...

end STANDARD;
Alsys IBM 370 Ada* Compiler

APPENDIX F
for VM/CMS and MVS
(including MVS/XA)
Implementation - Dependent Characteristics

Version 4.0

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.

*Ada is a registered trademark of the U.S. Government, Ada Joint Program Office
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 VM/CMS, MVS, or MVS/XA.

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 CMS and MVS operating systems, and has access to the following IBM documents:

- **CMS Command and Macro Reference**, Release 3, SC19-6209
- **OS/VS2 MVS Overview**, GC28-0984
- **OS/VS2 System Programming Library: Job Management**, GC28-1303
- **MVS/370 JCL Reference**, GC28-1350
- **IBM System/370 Principles of Operation**, GA22-7000
- **IBM System/370 System Summary**, GA22-7001
# TABLE OF CONTENTS

## APPENDIX F

### 1 Implementation-Dependent Pragmas

1.1 INLINE 2
1.2 INTERFACE 2
1.2.1 Calling Conventions 2
1.2.2 Parameter-Passing Conventions 3
1.2.3 Parameter Representations 4
1.2.4 Restrictions on Interfaced Subprograms 6
1.3 INTERFACE_NAME 6
1.4 INDENT 7
1.5 RMODE 8
1.6 Other Pragmas 8

### 2 Implementation-Dependent Attributes 10

### 3 Specification of the Package SYSTEM 11

### 4 Restrictions on Representation Clauses 12

4.1 Enumeration Types 12
4.2 Integer Types 15
4.3 Floating Point Types 17
4.4 Fixed Point Types 18
4.5 Access Types 21
4.6 Task Types 22
4.7 Array Types 23
4.8 Record Types 26

### 5 Conventions for Implementation-Generated Names 36

### 6 Address Clauses 37

6.1 Address Clauses for Objects 37
6.2 Address Clauses for Program Units 37
6.3 Address Clauses for Entries 37
7 Restrictions on Unchecked Conversions

8 Input-Output Packages

8.1 NAME Parameter
8.1.1 VM/CMS
8.1.2 MVS
8.2 FORM Parameter
8.3 STANDARD_INPUT and STANDARD_OUTPUT
8.4 USE_ERROR
8.5 Text Terminators
8.6 EBCDIC and ASCII
8.7 Characteristics of Disk Files
8.7.1 TEXT_IO
8.7.2 SEQUENTIAL_IO
8.7.3 DIRECT_IO

9 Characteristics of Numeric Types

9.1 Integer Types
9.2 Floating Point Type Attributes
9.3 Attributes of Type DURATION

10 Other Implementation-Dependent Characteristics

10.1 Characteristics of the Heap
10.2 Characteristics of Tasks
10.3 Definition of a Main Program
10.4 Ordering of Compilation Units
10.5 Implementation Defined Packages
10.5.1 Package EBCDIC
10.5.2 Package SYSTEM_ENVIRONMENT
10.5.3 Package RECORD_IO
10.5.4 Package STRINGS

INDEX
APPENDIX F

Implementation-Dependent Characteristics

This appendix summarises the implementation-dependent characteristics of the Alsys IBM 370 Ada Compiler for VM/CMS, MVS and MVS/XA. This document should be considered as the Appendix F to the Reference Manual for the Ada Programming Language ANSI/MIL-STD 1815 - January 1983, as appropriate to the Alsys Ada implementation for the IBM 370 under VM/CMS, MVS and MVS/XA.

Sections 1 to 8 of this appendix correspond to the various information required in Appendix F [F]*; sections 9 and 10 provide other information relevant to the Alsys implementation. The contents of these sections is described below:

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 [13.7].

4. The list of all restrictions on representation clauses [13.1].

5. The conventions used for any implementation-generated names denoting implementation-dependent components [13.4].

6. The interpretation of expressions that appear in address clauses, including those for interrupts [13.5].

7. Any restrictions on unchecked conversions [13.10.2].

8. Any implementation-dependent characteristics of the input-output packages [14].


10. Other implementation-dependent characteristics.

Throughout this appendix, 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 control, I/O, and other utility functions.

1 Implementation-Dependent Pragmas

1.1 INLINE

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

1.2 INTERFACE

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

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.

The language name 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.

1.2.1 Calling Conventions

The following calling conventions are required for code to be interfaced to Ada by use of the pragma interface to ASSEMBLER.

The contents of the general purpose registers 12 and 13 must be restored to their original values by the interfaced code before returning to Ada.

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 program 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. On entry to the interfaced code exponent underflow and significance interruptions are suppressed. Note that the program mask should be restored to its original value (i.e. X'C') before returning to Ada code.

1.2.2 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 most significant (sign) bit set to indicate the end of the list.

For formal parameters of mode in, which are of scalar or access type, the address passed is that of a copy of the value of the actual parameter. For all other parameters the address passed is the address of the actual parameter itself.

Since all non-scalar and non-access parameters to interfaced subprograms are passed by address, they cannot be protected from modification by the called subprogram, even though they may be formally declared to be of mode in. It is the programmer's responsibility to ensure that the semantics of the Ada parameter modes are honoured in these cases.

If the address of an Ada object is passed explicitly as a parameter to an interfaced subprogram (i.e. to a formal parameter of type SYSTEM.ADDRESS) it is the address of the address which is passed in the parameter list: a value of type SYSTEM.ADDRESS being treated identically to any other scalar value.

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

Appendix F, Implementation-Dependent Characteristics
An example of an interfaced subprogram is:

* 64-bit integer addition:

* type DOUBLE is
  * record
    * HIGH : INTEGER;
    * LOW : INTEGER;
  * end record
  * for DOUBLE use
  * record
    * HIGH at 0 range 0..31;
    * LOW at 4 range 0..31;
  * end record;
* 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,'F1'  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

1.2.3 Parameter Representations

This section describes the representation of values of the types that can be passed as parameters to an interfaced subprogram. The discussion assumes no representation clauses have been used to alter the default representations of the types involved. Chapter 4 describes the effect of representation clauses on the representation of values.

Integer Types [3.5.4]

Ada integer types are represented in two's complement form and occupy 8 (SHORT_SHORT_INTEGER), 16 (SHORT_INTEGER) or 32 (INTEGER) bits.

Boolean Types [3.5.3]

Booleans are represented as 8 bit values. FALSE is represented by the value 0, and TRUE is represented by the value 1.
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 between 256 and 65536 ($2^{16}$) elements in 16 bits and all others in 32 bits. The maximum number of values an enumeration type can include is $2^{31}$.

Consequently, 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:

$$2^{**} \text{MANTISSA} \geq \max (\text{abs (upper_bound)}, \text{abs (lower_bound)}) / \text{small}$$

The size of a fixed point type is:

<table>
<thead>
<tr>
<th>MANTISSA</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 .. 15</td>
<td>16 bits</td>
</tr>
<tr>
<td>16 .. 31</td>
<td>32 bits</td>
</tr>
</tbody>
</table>

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 any bits of this word, including those 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. The elements of the array are allocated by row. 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; the value passed is the address of the first component of the record. Components of a record are aligned on their natural boundaries (e.g. INTEGER on a word boundary) and the components may be re-ordered by the compiler so as to minimise the total size of objects of the record type. If a record contains discriminants or components having a dynamic size, implicit components may be added to the record. Thus the default layout of the internal structure of the record may not be inferred directly from its Ada declaration. The use of a representation clause to control the layout of any record type whose values are to be passed to interfaced subprograms is recommended.

1.2.4 Restrictions on Interfaced Subprograms

The Ada Run-Time Executive uses the SPIE and ESPIE macros (SVC 14). 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.

1.3 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 SAMPLEDEVICE (X : INTEGER) return INTEGER;
  function PROCESSSAMPLE (X : INTEGER) return INTEGER;
private
  pragma INTERFACE (ASSEMBLER, SAMPLEDEVICE);
  pragma INTERFACE (ASSEMBLER, PROCESSSAMPLE);
  pragma INTERFACE_NAME (PROCESSSAMPLE, "PSAMPLE");
end SAMPLE_DATA;
```

### 1.4 INDENT

This pragma is only used with the Alsys Reformatter (*AdaReformat*); this tool offers the functionalities of a source reformatter in an Ada environment. The Alsys Reformatter is currently available only with the Alsys IBM 370 Ada compiler hosted under VM/CMS.

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

```
pragma INDENT(OFF)
```

The Reformatter does not modify the source lines after the OFF pragma INDENT.

```
pragma INDENT(ON)
```

The Reformatter resumes its action after the ON pragma INDENT. Therefore any source lines that are bracketed by the OFF and ON pragma INDENTs are not modified by the Alsys Reformatter.
1.5 RMODE

Pragma RMODE associates a residence mode with the objects designated by the access values belonging to a given access type.

This pragma takes the form:

```plaintext
pragma RMODE (access_type_name, residence_mode);
```

where

- `access_type_name` is the name of the access type defining the collection of objects whose residence mode is to be specified.
- `residence_mode` is the residence mode to be associated with the designated objects.

A24: Indicates that the designated objects must reside within 24 bit addressable virtual storage (that is, below the 16 megabyte virtual storage line under MVS/XA).

ANY: Indicates that the designated objects may reside anywhere in virtual storage (that is, either above or below the 16 megabyte virtual storage line under MVS/XA).

Under CMS or MVS on non-extended architecture machines the pragma is effectively ignored, since only 16 megabytes of virtual address space are available and all virtual addresses implicitly meet the A24 residence mode criteria.

Under MVS/XA the pragma is significant for data whose residence mode must be explicitly controlled, e.g. data which is to be passed to non-Ada code via the pragma INTERFACE.

In the absence of the pragma RMODE, the default residence mode associated with the objects designated by an access type is ANY.

The `access_type_name` must be a simple name. The pragma RMODE and the access type declaration to which it refers must both occur immediately within the same declarative part, package specification or task specification; the declaration must occur before the pragma.

1.6 Other Pragmas

Pragmas IMPROVE and PACK are discussed in detail in the section on representation clauses (Chapter 4).

Pragma PRIORITY is accepted with the range of priorities running from 1 to 10 (see the definition of the predefined package SYSTEM in Chapter 3). The undefined priority (no pragma PRIORITY) is treated as though it were less than any defined priority value.
In addition to pragma SUPPRESS, it is possible to suppress all checks in a given compilation by the use of the Compiler option CHECKS.

The following language defined pragmas are not implemented.

  CONTROLLED
  MEMORY_SIZE
  OPTIMIZE
  STORAGE_UNIT
  SYSTEM_NAME

Note that all access types are implemented by default as controlled collections as described in [4.8] (see section 10.1).
2 Implementation-Dependent Attributes

In addition to the Representation Attributes of [13.7.2] and [13.7.3], the four attributes listed in section 5 (Conventions for Implementation-Generated Names), for use in record representation clauses, and the attributes described below are provided:

**T'DESCRPTOR_SIZE** For a prefix T that denotes a type or subtype, this attribute yields the size (in bits) required to hold a descriptor for an object of the type T, allocated on the heap or written to a file. If T is constrained, T'DESCRPTOR_SIZE will yield the value 0.

**T'IS_ARRAY** For a prefix T that denotes a type or subtype, this attribute yields the value TRUE if T denotes an array type or an array subtype; otherwise, it yields the value FALSE.

Limitations on the use of the attribute ADDRESS

The attribute ADDRESS is implemented for all prefixes that have meaningful addresses. The following entities do not have meaningful addresses and will therefore cause a compilation error if used as a prefix to ADDRESS:

- A constant that is implemented as an immediate value i.e. does not have any space allocated for it.
- A package specification that is not a library unit.
- A package body that is not a library unit or subunit.
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**31;

       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.

On the non-extended architecture (AMODE 24) the top byte of a value of type address
is ignored (i.e. does not form part of the address). On an extended architecture (31 bit
addressing) the top bit of a value of type address is similarly ignored.

Appendix F. Implementation-Dependent Characteristics
4 Restrictions on Representation Clauses

This section explains how objects are represented and allocated by the Alsys IBM 370 Ada Compiler and how it is possible to control this using representation clauses.

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

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

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

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

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

Representation clauses on derived record types or derived task types is not supported.

Size representation clauses on types derived from private types are not supported when the derived type is declared outside the private part of the defining package.

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

4.1 Enumeration Types

Internal codes of enumeration literals

When no enumeration representation clause applies to an enumeration type, the internal code associated with an enumeration literal is the position number of the enumeration literal. Then, for an enumeration type with n elements, the internal codes are the integers 0, 1, 2, .., n-1.

An enumeration representation clause can be provided to specify the value of each internal code as described in RM 13.3. The Alsys compiler fully implements enumeration representation clauses.
As internal codes must be machine integers the internal codes provided by an enumeration representation clause must be in the range $-2^{31} \ldots 2^{31}-1$.

**Encoding of enumeration values**

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

**Minimum size of an enumeration subtype**

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

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

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

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

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

**Size of an enumeration subtype**

When no size specification is applied to an enumeration type or first named subtype, the objects of that type or first named subtype are represented as signed machine integers. The machine provides 8, 16 and 32 bit integers, and the compiler selects automatically the smallest signed machine integer which can hold each of the internal codes of the enumeration type (or subtype). The size of the enumeration type and of any of its subtypes is thus 8, 16 or 32 bits.
When a size specification is applied to an enumeration type, this enumeration type and each of its subtypes has the size specified by the length clause. The same rule applies to a first named subtype. The size specification must of course specify a value greater than or equal to the minimum size of the type or subtype to which it applies:

```
type EXTENDED is
  (NUL, SOH, STX, ETX, EOT, ENQ, ACK, BEL,
   BS, HT, LF, VT, FF, CR, SO, SI,
   DLE, DC1, DC2, DC3, DC4, NAK, SYN, ETB,
   CAN, EM, SUB, ESC, FS, GS, RS, US,
   '(', ')', '*', '+', ',', '.', '<', '>', '=', '/',
   '0', '1', '2', '3', '4', '5', '6', '7',
   '8', '9', '
   '@', 'A', 'B', 'C', 'D', 'E', 'F', 'G',
   'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O',
   'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W',
   'X', 'Y', 'Z', '
   ', '
   'a', 'b', 'c', 'd', 'e', 'f', 'g',
   'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o',
   'p', 'q', 'r', 's', 't', 'u', 'v', 'w',
   'x', 'y', 'z', '
   ')
   -- Extended characters
   LEFT_ARROW,
   RIGHT_ARROW,
   UPPER_ARROW,
   LOWER_ARROW,
   UPPER_LEFT_CORNER,
   UPPER_RIGHT_CORNER,
   LOWER_RIGHT_CORNER,
   LOWER_LEFT_CORNER,
   ...
);```

for EXTENDED'SIZE use 8;

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

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

**Size of the objects of an enumeration subtype**

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

An enumeration subtype is byte aligned if the size of the subtype is less than or equal to 8 bits, halfword aligned if the size of the subtype is less than or equal to 16 bits and word aligned otherwise.

Address of an object of an enumeration subtype

Provided its alignment is not constrained by a record representation clause or a pragma PACK, the address of an object of an enumeration subtype is a multiple of the alignment of the corresponding subtype.

4.2 Integer Types

Predefined integer types

There are three predefined integer types in the Alsys implementation for IBM 370 machines:

- type SHORT_SHORT_INTEGER is range \(-2^{07}..2^{07}-1\);
- type SHORT_INTEGER is range \(-2^{15}..2^{15}-1\);
- type INTEGER is range \(-2^{31}..2^{31}-1\);

Selection of the parent of an integer type

An integer type declared by a declaration of the form:

\[
\text{type } T \text{ is range } L .. R;
\]

is implicitly derived from either the SHORT INTEGER or INTEGER predefined integer type. The compiler automatically selects the predefined integer type whose range is the shortest that contains the values L to R inclusive. Note that the SHORT_SHORT_INTEGER representation is never automatically selected by the compiler.

Encoding of integer values

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

Minimum size of an integer subtype

The minimum size of an integer subtype is the minimum number of bits that is necessary for representing the internal codes of the subtype values in normal binary form.
For a static subtype, if it has a null range its minimum size is 1. Otherwise, if m and M are the lower and upper bounds of the subtype, then its minimum size L is determined as follows. For m >= 0, L is the smallest positive integer such that M <= 2^L - 1. For m < 0, L is the smallest positive integer that -2^L-1 <= m and M <= 2^L-1 - 1.

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

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

Size of an integer subtype

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

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

type S is range 80 .. 100;
-- S is derived from SHORT_INTEGER, its size is 16 bits.

type J is range 0 .. 65535;
-- J is derived from INTEGER, its size is 32 bits.

type N is new J range 80 .. 100;
-- N is indirectly derived from INTEGER, its size is 32 bits.

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

```ada
import S
for S'SIZE use 32;
-- S is derived from SHORT_INTEGER, but its size is 32 bits
-- because of the size specification.

import J
for J'SIZE use 8;
-- J is derived from SHORT_INTEGER, but its size is 8 bits because
-- of the size specification.

import N
for N'SIZE use 8;
-- N is indirectly derived from SHORT_INTEGER, but its size is 8 bits
-- because N inherits the size specification of J.
```

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

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

Alignment of an integer subtype

An integer subtype is byte aligned if the size of the subtype is less than or equal to 8 bits, halfword aligned if the size of the subtype is less than or equal to 16 bits and word aligned otherwise.

Address of an object of an integer subtype

Provided its alignment is not constrained by a record representation clause or a pragma PACK, the address of an object of an integer subtype is a multiple of the alignment of the corresponding subtype.

4.3 Floating Point Types

Predefined floating point types

There are three predefined floating point types in the Alsys implementation for IBM 370 machines:

- type SHORT_FLOAT is
digits 6 range -2.0**252*(1.0-2.0**-24) .. 2.0**252*(1.0-2.0**-24);
- type FLOAT is
digits 15 range -2.0**252*(1.0-2.0**-56) .. 2.0**252*(1.0-2.0**-56);
- type LONG_FLOAT is
digits 18 range -2.0**252*(1.0-2.0**-112) .. 2.0**252*(1.0-2.0**-112);

Selection of the parent of a floating point type

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

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

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

Encoding of floating point values

In the program generated by the compiler, floating point values are represented using the IBM 370 data formats for single precision, double precision and extended precision floating point values respectively.
Values of the predefined type SHORT_FLOAT are represented using the single precision format, values of the predefined type FLOAT are represented using the double precision format and values of the predefined type LONG_FLOAT are represented using the extended precision format. The values of any other floating point type are represented in the same way as the values of the predefined type from which it derives, directly or indirectly.

Minimum size of a floating point subtype

The minimum size of a floating point subtype is 32 bits if its base type is SHORT_FLOAT or a type derived from SHORT_FLOAT, 64 bits if its base type is FLOAT or a type derived from FLOAT and 128 bits if its base type is LONG_FLOAT or a type derived from LONG_FLOAT.

Size of a floating point subtype

The sizes of the predefined floating point types SHORT_FLOAT, FLOAT and LONG_FLOAT are respectively 32 and 64 and 128 bits.

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

The only size that can be specified for a floating point type or first named subtype using a size specification is its usual size (32, 64 or 128 bits).

Size of the objects of a floating point subtype

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

Alignment of a floating point subtype

A floating point subtype is word aligned if its size is 32 bits and double word aligned otherwise.

Address of an object of a floating point subtype

Provided its alignment is not constrained by a record representation clause or a pragma PACK, the address of an object of a floating point subtype is a multiple of the alignment of the corresponding subtype.

4.4 Fixed Point Types

Small of a fixed point type

If no specification of small applies to a fixed point type, then the value of small is determined by the value of delta as defined by RM 3.5.9.
A specification of small can be used to impose a value of small. The value of small is required to be a power of two.

**Predefined fixed point types**

To implement fixed point types, the Alsys compiler for IBM 370 machines uses a set of anonymous predefined types of the form:

```vhl
    type FIXED is delta D range (-2**15-1)*S .. 2**15*S;
    for FIXED'_SMALL use S;
    type LONG_FIXED is delta D range (-2**31-1)*S .. 2**31*S;
    for LONG_FIXED'_SMALL use S;
```

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

**Selection of the parent of a fixed point type**

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

```vhl
    type T is delta D range L .. R;
    for T'SMALL use S;
```

possibly with a small specification:

```vhl
    for T'_SMALL use S;
```

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

**Encoding of fixed point values**

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

```
    V / F'BASE'_SMALL
```

**Minimum size of a fixed point subtype**

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

For a static subtype, if it has a null range its minimum size is 1. Otherwise, s and S being the bounds of the subtype, if i and I are the integer representations of m and M, the smallest and the greatest model numbers of the base type such that s < m and M < S, then the minimum size L is determined as follows. For i >= 0, L is the smallest positive integer such that I <= 2^L-1. For i < 0, L is the smallest positive integer such that -2^L-1 <= i and I <= 2^L-1-1.
type F is delta 2.0 range 0.0 .. 500.0;
-- The minimum size of F is 8 bits.

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

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

Size of a fixed point subtype

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

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

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

type L is delta 0.01 range 0.0 .. 300.0;
-- L is derived from a 32 bit predefined fixed type, its size is 32 bits.

type N is new L range 0.0 .. 2.0;
-- N is indirectly derived from a 32 bit predefined fixed type, its size is 32 bits.

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

type F is delta 0.01 range 0.0 .. 2.0;
for F'SIZE use 32;
-- F is derived from a 16 bit predefined fixed type, but its size is 32 bits
-- because of the size specification.

type L is delta 0.01 range 0.0 .. 300.0;
for F'SIZE use 16;
-- F is derived from a 32 bit predefined fixed type, but its size is 16 bits
-- because of the size specification.

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

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

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

Alignment of a fixed point subtype

A fixed point subtype is byte aligned if its size is less than or equal to 8 bits, halfword aligned if the size of the subtype is less than or equal to 16 bits and word aligned otherwise.

Address of an object of a fixed point subtype

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

4.5 Access Types

Collection Size

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

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

Encoding of access values.

Access values are machine addresses.

Minimum size of an access subtype

The minimum size of an access subtype is 32 bits.

Size of an access subtype

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

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

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

Alignment of an access subtype.

An access subtype is always word aligned.

Address of an object of an access subtype

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

4.6 Task Types

Storage for a task activation

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

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

It is not allowed to apply such a length clause to a derived type. The same storage space is reserved for the activation of a task of a derived type as for the activation of a task of the parent type.

Encoding of task values.

Task values are machine addresses.

Minimum size of a task subtype

The minimum size of a task subtype is 32 bits.

Size of a task subtype

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

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

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

Alignment of a task subtype

A task subtype is always word aligned.

Address of an object of a task subtype

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

4.7 Array Types

Layout of an array

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

```
Component   Gap   Component   Gap   Component   Gap
```

Components

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

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

type DECIMAL_DIGIT is range 0 .. 9;
for DECIMAL_DIGIT'SIZE use 4;
type BINARY_CODED_DECIMAL is
   array (INTEGER range <>) of DECIMAL_DIGIT;
   -- The size of the type DECIMAL_DIGIT is 4 bits. Thus in an array of
   -- type BINARY_CODED_DECIMAL each component will be represented on
   -- 4 bits as in the usual BCD representation.
```
If the array is packed and its components are neither records nor arrays, the size of the components is the minimum size of the subtype of the components:

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

type DECIMAL_DIGIT is range 0 .. 9;
type BINARY_CODED_DECIMAL is
   array (INTEGER range <>) of DECIMAL_DIGIT;
pragma PACK(BINARY_CODED_DECIMAL);
-- The size of the type DECIMAL_DIGIT is 16 bits, but, as
-- BINARY_CODED_DECIMAL is packed, each component of an array of this
-- type will be represented on 4 bits as in the usual BCD representation.
```

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

**Gaps**

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

```ada
type R is
   record
      K : INTEGER; -- INTEGER is word aligned.
      B : BOOLEAN; -- BOOLEAN is byte aligned.
   end record;
-- Record type R is word aligned. Its size is 40 bits.

type A is array (1 .. 10) of R;
-- A gap of three bytes is inserted after each component in order to respect the
-- alignment of type R. The size of an array of type A will be 640 bits.
```

![Array of type A: each subcomponent K has a word offset.](image)
If a size specification applies to the subtype of the components or if the array is packed, no gaps are inserted:

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

type A is array (1 .. 10) of R;
pragma PACK(A);
-- There is no gap in an array of type A because
-- A is packed.
-- The size of an object of type A will be 400 bits.

type NR is new R;
for NR'SIZE use 40;

type B is array (1 .. 10) of NR;
-- There is no gap in an array of type B because
-- NR has a size specification.
-- The size of an object of type B will be 400 bits.
```

Array of type A or B: a subcomponent K can have any byte offset.

**Size of an array subtype**

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

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

- if it has non-static constraints or is an unconstrained array type with non-static index subtypes (because the number of components can then only be determined at run time).
- if the components are records or arrays and their constraints or the constraints of their subcomponents (if any) are not static (because the size of the components and the size of the gaps can then only be determined at run time).
As has been indicated above, the effect of a pragma PACK on an array type is to suppress the gaps and to reduce the size of the components. The consequence of packing an array type is thus to reduce its size.

If the components of an array are records or arrays and their constraints or the constraints of their subcomponents (if any) are not static, the compiler ignores any pragma PACK applied to the array type but issues a warning message. Apart from this limitation, array packing is fully implemented by the Alsys compiler.

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

**Size of the objects of an array subtype**

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

**Alignment of an array subtype**

If no pragma PACK applies to an array subtype and no size specification applies to its components, the array subtype has the same alignment as the subtype of its components.

If a pragma PACK applies to an array subtype or if a size specification applies to its components (so that there are no gaps), the alignment of the array subtype is the lesser of the alignment of the subtype of its components and the relative displacement of the components.

**Address of an object of an array subtype**

Provided its alignment is not constrained by a record representation clause, the address of an object of an array subtype is a multiple of the alignment of the corresponding subtype.

### 4.8 Record Types

**Layout of a record**

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

The positions and the sizes of the components of a record type object can be controlled using a record representation clause as described in RM 13.4. In the Alsys implementation for IBM 370 machines there is no restriction on the position that can be specified for a component of a record. If a component is not a record or an array, its size can be any size from the minimum size to the size of its subtype. If a component is a record or an array, its size must be the size of its subtype.
type ACCESS KEY is range 0..15;
-- The size of ACCESS KEY is 16 bits, the minimum size is 4 bits

type CONDITIONS is (ZERO, LESS_THAN, GREATER_THAN, OVERFLOW);
-- The size of CONDITIONS is 8 bits, the minimum size is 2 bits

type PROG_EXCEPTION is (FIX_OVFL, DEC_OVFL, EXP_UNDFL, SIGNIF);
type PROG_MASK is array (PROG_EXCEPTION) of BOOLEAN;
pragma PACK (PROGMASK);
-- The size of PROGMASK is 4 bits

type ADDRESS is range 0..2**24-1;
for ADDRESS'SIZE use 24;
-- ADDRESS represents a 24 bit memory address

type PSW is
  record
    PER_MASK : BOOLEAN;
    DAT_MODE : BOOLEAN;
    IO_MASK : BOOLEAN;
    EXTERNAL_MASK : BOOLEAN;
    PSW_KEY : ACCESS KEY;
    EC_MODE : BOOLEAN;
    MACHINE_CHECK : BOOLEAN;
    WAIT_STATE : BOOLEAN;
    PROBLEM_STATE : BOOLEAN;
    ADDRESS_SPACE : BOOLEAN;
    CONDITION_CODE : CONDITIONS;
    PROGRAM_MASK : PROGMASK;
    INSTR_ADDRESS : ADDRESS;
  end record;
-- This type can be used to map the program status word of the IBM 370

for PSW use
  record at mod 8;
    PER_MASK at 0 range 1..1;
    DAT_MODE at 0 range 5..5;
    IO_MASK at 0 range 6..6;
    EXTERNAL_MASK at 0 range 7..7;
    PSW_KEY at 1 range 0..3;
    EC_MODE at 1 range 4..4;
    MACHINE_CHECK at 1 range 5..5;
    WAIT_STATE at 1 range 6..6;
    PROBLEM_STATE at 1 range 7..7;
    ADDRESS_SPACE at 2 range 0..0;
    CONDITION_CODE at 2 range 2..3;
    PROGRAM_MASK at 2 range 4..7;
    INSTR_ADDRESS at 5 range 0..23;
  end record;

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

Appendix F. Implementation-Dependent Characteristics
If no component clause applies to a component of a record, its size is the size of its subtype. Its position is chosen by the compiler so as to optimise access to the components of the record: the offset of the component is chosen as a multiple of the alignment of the component subtype. Moreover, the compiler chooses the position of the component so as to reduce the number of gaps and thus the size of the record objects.

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

Pragma PACK has no further effect on records. The Alsys compiler always optimises the layout of records as described above.

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

**Indirect components**

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

![Diagram showing direct and indirect components]

A direct and an indirect component

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

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

type COLOR is (GREEN, RED, BLUE);
```
type SERIES is array (POSITIVE range <>) of INTEGER;

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

Q : POSITIVE;

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

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

\[
\begin{align*}
D &= \text{SCREEN} \\
N &= 2 \\
\text{S OFFSET} &\quad \text{F OFFSET} \\
\text{F} &\quad \text{N} \\
\text{N} &\quad \text{D} \\
\text{D} &\quad \text{C} \\
\text{C} &\quad \text{F} \\
\text{F} &\quad \text{S} \\
\text{S} &
\end{align*}
\]

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

Appendix F. Implementation-Dependent Characteristics 29
Thanks to this strategy, the only indirect components are dynamic components. But not all dynamic components are necessarily indirect: if there are dynamic components in a component list which is not followed by a variant part, then exactly one dynamic component of this list is a direct component because its offset can be computed at compilation time (the only dynamic components that are direct components are in this situation):

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

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

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

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

Implicit components

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

An implicit component may contain information which is used when the record object or several of its components are accessed. In this case the component will be included in any record object (the implicit component is considered to be declared before any variant part in the record type declaration). There can be two components of this kind; one is called RECORD_SIZE and the other VARIANT_INDEX.
On the other hand an implicit component may be used to access a given record component. In that case the implicit component exists whenever the record component exists (the implicit component is considered to be declared at the same place as the record component). Components of this kind are called ARRAY DESCRIPTORs or RECORD DESCRIPTORs.

- **RECORD SIZE**

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

The value of a RECORD SIZE component may denote a number of bits or a number of storage units. In general it denotes a number of storage units, but if any component clause specifies that a component of the record type has an offset or a size which cannot be expressed using storage units, then the value designates a number of bits.

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

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

- **VARIANT INDEX**

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

Component lists that do not contain a variant part are numbered. These numbers are the possible values of the implicit component VARIANT_INDEX.
type VEHICLE is (AIRCRAFT, ROCKET, BOAT, CAR);

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

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

<table>
<thead>
<tr>
<th>Variant Index</th>
<th>Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(KIND, SPEED, WHEELS, WINGSPAN)</td>
</tr>
<tr>
<td>2</td>
<td>(KIND, SPEED, WHEELS)</td>
</tr>
<tr>
<td>3</td>
<td>(KIND, SPEED, STEAM)</td>
</tr>
<tr>
<td>4</td>
<td>(KIND, SPEED, STAGES)</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Component</th>
<th>Interval</th>
</tr>
</thead>
<tbody>
<tr>
<td>KIND</td>
<td>--</td>
</tr>
<tr>
<td>SPEED</td>
<td>--</td>
</tr>
<tr>
<td>WHEELS</td>
<td>1 .. 2</td>
</tr>
<tr>
<td>WINGSPAN</td>
<td>1 .. 1</td>
</tr>
<tr>
<td>STEAM</td>
<td>3 .. 3</td>
</tr>
<tr>
<td>STAGES</td>
<td>4 .. 4</td>
</tr>
</tbody>
</table>

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

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

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

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

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

**RECORD_DESCRIPTOR**

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

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

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

**Suppression of implicit components**

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

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

The first argument specifies whether TIME or SPACE is the primary criterion for the choice of the representation of the record type that is denoted by the second argument.
If \texttt{TIME} is specified, the compiler inserts implicit components as described above. If on
the other hand \texttt{SPACE} is specified, the compiler only inserts a \texttt{VARIANT_INDEX} or a
\texttt{RECORD_SIZE} component if this component appears in a record representation clause
that applies to the record type. A record representation clause can thus be used to keep
one implicit component while suppressing the other.

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

\textit{Size of a record subtype}

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

The size of a constrained record subtype is obtained by adding the sizes of its
components and the sizes of its gaps (if any). This size is not computed at compile time
- when the record subtype has non-static constraints,
- when a component is an array or a record and its size is not computed at
  compile time.

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

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

\textit{Size of an object of a record subtype}

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

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

\textit{Alignment of a record subtype}

When no record representation clause applies to its base type, a record subtype has the
same alignment as the component with the highest alignment requirement.

When a record representation clause that does not contain an alignment clause applies to
its base type, a record subtype has the same alignment as the component with the highest
alignment requirement which has not been overridden by its component clause.
When a record representation clause that contains an alignment clause applies to its base type, a record subtype has an alignment that obeys the alignment clause.

Address of an object of a record subtype

Provided its alignment is not constrained by a representation clause, the address of an object of a record subtype is a multiple of the alignment of the corresponding subtype.
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 an 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 storage 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'VARIANT_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 an 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 a 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.
6 Address Clauses

6.1 Address Clauses for Objects

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

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

6.2 Address Clauses for Program Units

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

6.3 Address Clauses for Entries

Address clauses for entries are not implemented in the current version of the compiler.
7 Restrictions on Unchecked Conversions

Unconstrained arrays are not allowed as target types.

Unconstrained record types without defaulted discriminants are not allowed as target types.

If the source and the target types are each scalar or access types, the sizes of the objects of the source and target types must be equal. If a composite type is used either as the source type or as the target type this restriction on the size does not apply.

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

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

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

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

8.1 NAME Parameter

8.1.1 VM/CMS

The NAME parameter supplied to the Ada procedures CREATE or OPEN [14.2.1] may represent a CMS file name or DDNAME specified using a FILEDEF command.

The syntax of a CMS file name as specified in the Ada NAME parameter is as follows:

\[ file\_name := fn \ [ \ ft \ [ \ fm \ ] ] \ | \ %ddname \]

where

- \( fn \) is the CMS filename
- \( ft \) is the CMS filetype
- \( fm \) is the CMS filemode

If the filenames or filetypes exceed 8 characters then they are truncated. As indicated above, the filetype and filemode fields are not mandatory components of the NAME parameter. If the filemode is omitted, it defaults to "AI" for files being created; for files being opened all accessed minidisks are searched and the CMS filemode is set to that of the first file with the appropriate filename and filetype. If, in addition, the filetype is omitted it defaults to "FILE". The case of the characters of the filename is not significant.

The NAME parameter may also be a DDNAME. If the file name parameter starts with a % character, the remainder of the string (excluding trailing blanks) is taken as a DDNAME previously specified using the FILEDEF command. If the DDNAME has not been specified using FILEDEF, NAME_ERROR will be raised.

The effect of calling CREATE and DELETE for a file opened using a DDNAME is as if OPEN or CLOSE (respectively) had been called.

8.1.2 MVS

The NAME parameter supplied to the Ada procedures CREATE or OPEN [14.2.1] may represent an MVS dataset name or DDNAME.
The syntax of an MVS dataset name as specified in the Ada NAME parameter is as follows:

```
dataset_name ::= [&]dsname[(member)] | 'dsname[(member)]' | %ddname
```

where

- `dsname` is the MVS dataset name. If prefixed by an ampersand (&) the system assigns a temporary dataset name.
- `member` is the MVS member, generation or area name.

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

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 apostrophes) 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 NAME parameter may also be a DDNAME. If the file name parameter starts with a % character, the remainder of the string (excluding trailing blanks) is taken as a DDNAME previously allocated. If the DDNAME has not been allocated, NAME_ERROR will be raised.

The effect of calling CREATE and DELETE for a file opened using a DDNAME is as if OPEN or CLOSE (respectively) had been called.

8.2 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:
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:

\[
\text{NOT\_SHARED | SHARED} \Rightarrow \text{access\_mode}
\]

where

\[
\text{access\_mode} ::= \text{READERS | SINGLE\_WRITER | ANY}
\]

A file sharing attribute of:

\[
\text{NOT\_SHARED}
\]

implies only one internal file may access the external file.

\[
\text{SHARED} \Rightarrow \text{READERS}
\]

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.

\[
\text{SHARED} \Rightarrow \text{SINGLE\_WRITER}
\]

is as \(\text{SHARED} \Rightarrow \text{READERS}\), but in addition allows a single internal file of mode OUT\_FILE or INOUT\_FILE.

\[
\text{SHARED} \Rightarrow \text{ANY}
\]

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 \(\text{SHARED} \Rightarrow \text{READERS}\), for modes INOUT\_FILE and OUT\_FILE the default is \(\text{NOT\_SHARED}\).

Record format attribute

This attribute controls the record format (RECFM) of an external file created in Ada. The attribute may only be used in the FORM parameter of the CREATE command; if used in the FORM parameter of the OPEN command, USE_ERROR will be raised.
By default, files are created according to the following rules:

- for **TEXT_IO** and instantiations of **SEQUENTIAL_IO** of unconstrained types, variable-length record files (RECFM = V) are created.

- for **DIRECT_IO** and instantiations of **SEQUENTIAL_IO** of constrained types, fixed-length record files (RECFM = F) are created.

The syntax of the record format attribute is as follows:

\[
\text{RECFM} \Rightarrow V | F
\]

**Record size attribute**

This attribute controls the logical record length (LRECL) of an external file created in Ada. The attribute may only be used in the FORM parameter of the CREATE command; if used in the FORM parameter of the OPEN command, USE_ERROR will be raised.

In the case of RECFM F files (see record format attribute) the record size attribute specifies the record length of each record; in the case of RECFM V files, the record size attribute specifies the maximum record length.

In the case of **DIRECT_IO**.CREATE for unconstrained types the user is required to specify the RECORD_SIZE attribute (otherwise USE_ERROR will be raised by the OPEN or CREATE procedures).

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

In the case of **DIRECT_IO** and **SEQUENTIAL_IO** for unconstrained types the value given must not be smaller than \(\text{ELEMENT_TYPE'DESCRIP'TOR_SIZE} / \text{SYSTEM.STORAGE_UNIT}\) plus the size of the largest record which is to be read or written. 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:

\[
\text{RECORD_SIZE} | \text{LRECL} \Rightarrow \text{natural}
\]

where natural is a size in bytes.

For input-output of constrained types the default is:

\[
\text{RECORD_SIZE} \Rightarrow \text{element_length}
\]
where

\[ \text{element\_length} = \frac{\text{ELEMENT\_TYPE\_SIZE}}{\text{SYSTEM\_STORAGE\_UNIT}} \]

For input-output of unconstrained types other than via \texttt{DIRECT\_IO}, in which case the \texttt{RECORD\_SIZE} attribute must be provided by the user, variable-size records are used (\texttt{RECFM V}).

**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) or must obey the same rules for specifying the record size.

The default is

\[ \text{BLOCK\_SIZE} \Rightarrow \text{record\_size} \]

for \texttt{RECFM F} files and

\[ \text{BLOCK\_SIZE} \Rightarrow 4096 \]

for \texttt{RECFM V} files.

**Carriage control**

This attribute applies to \texttt{TEXT\_IO} only, and is intended for files destined to be sent to a printer.

For a file of mode \texttt{OUT\_FILE}, this attribute causes the output procedures of \texttt{TEXT\_IO} to place a carriage control character as the first character of every output record; 'l' (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 \texttt{TEXT\_IO}.

For a file of mode \texttt{IN\_FILE}, this attribute causes the input procedures of \texttt{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 \texttt{END\_OF\_PAGE}.

The user should naturally be careful to ensure the carriage control attribute of a file of mode \texttt{IN\_FILE} has the same value as that specified when creating the file.

The syntax of the carriage control attribute is as follows:

\[ \text{CARRIAGE\_CONTROL} [ \Rightarrow \text{boolean} ] \]
The default is set according to the filetype of the file: if the filetype is LISTING, the default is CARRIAGE_CONTROL => TRUE otherwise the default is CARRIAGE_CONTROL => FALSE. If the attribute alone is specified without a boolean value it defaults to TRUE.

Truncate

This attribute applies to TEXT_IO files of mode IN_FILE, and causes the input procedures of TEXT_IO to remove trailing blanks from records read.

The syntax of the TRUNCATE attribute is as follows:

    TRUNCATE [ => boolean ]

The default is TRUNCATE => FALSE.

Note that truncation is always performed for TEXT_IO files for which the record size attribute is set (i.e. RECFM = F). If the attribute alone is specified without a boolean value it defaults to TRUE.

Append

This attribute may only be used in the FORM parameter of the OPEN command; if used in the FORM parameter of the CREATE command, USE_ERROR will be raised.

The affect of this attribute is to cause writing to commence at the end of the existing file.

The syntax of the APPEND attribute is as follows:

    APPEND [ => boolean ]

The default is APPEND => FALSE. If the attribute alone is specified without a boolean value it defaults to TRUE.

Eof string

This attribute applies only to files associated with the terminal opened using TEXT_IO, and controls the logical end_of_file string. If a line equal to the logical end_of_file string is typed in, END_OF_FILE will become TRUE. If an attempt is made to read from a file for which END_OF_FILE is TRUE, END_ERROR will be raised.

The syntax of the EOF_STRING attribute is as follows:

    EOF_STRING => sequence_of_characters

The default is EOF_STRING => */

The EOF_STRING may not contain commas or spaces.
If the END_OF_FILE function is called, a "look-ahead read" will be required. This means that (for example) a question-and-answer session at the terminal coded as follows:

```plaintext
while not END_OF_FILE loop
    PUT_LINE ("Enter value.");
    GET_LINE (...);
end loop;
```

will cause the prompt to appear only after the first value has been input. If the example is recoded without the explicit call to END_OF_FILE (but perhaps within a handler for END_ERROR) the behaviour will be appropriate.

The following additional FORM parameter attributes apply only to programs run under MVS.

**Unit attribute**

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

```plaintext
UNIT => unit_name
```

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

The default is the local installation specific default.

**Volume attribute**

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

```plaintext
VOLUME => volume_name
```

where `volume_name` specifies the volume serial number.

The default is the local installation specific default.

**Primary attribute**

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

```plaintext
PRIMARY => natural
```

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

The default is the local installation specific default.
Secondary attribute

This attribute allows control over the secondary space allocation for a 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 installation specific default.

8.3 STANDARD_INPUT and STANDARD_OUTPUT

The Ada internal files STANDARD_INPUT and STANDARD_OUTPUT are associated with the external files %SYSIN and %SYSOUT, respectively. By default under CMS the DDNAMEs SYSIN and SYSOUT are defined to be the terminal, but the user may redefine their assignments using the FILEDEF command before running any program. Under MVS, the DDNAMEs must be allocated before any program is run, whether or not the corresponding Ada internal files are used.

The Ada internal files STANDARD_INPUT and STANDARD_OUTPUT are associated with the DD names SYSIN and SYSOUT, respectively. These DD names must be defined before any program can be run.

8.4 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 EOF_STRING FORM parameter attribute for files other than TEXT_IO files of mode IN_FILE.
- Specifying the CARRIAGE_CONTROL FORM parameter attribute for files other than TEXT_IO files.
- 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 instantiation 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 for constrained types.
- Violating the file sharing rules stated above.
For CMS, attempting to write a zero length record to other than the terminal.

Errors detected whilst reading or writing (e.g. writing to a file on a read-only disk).

8.5 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. Note that for terminal input a line consisting of the EOF_STRING (see 8.1.1) is interpreted as a file terminator. Thus, entering such a line to satisfy a read from the terminal will raise the END_ERROR exception.

The user should avoid the explicit output of the character ASCII.FF [C], as this will not cause a page break to be emitted. If the user explicitly outputs the character ASCII.LF, this is treated as a call of NEW_LINE [14.3.4].

The following characters have special meaning for VM/SP; this should be borne in mind when reading from the display terminal:

<table>
<thead>
<tr>
<th>Character</th>
<th>Default VM/SP meaning</th>
<th>May be changed using</th>
</tr>
</thead>
<tbody>
<tr>
<td>#</td>
<td>logical line end symbol</td>
<td>CP TERMINAL LINEND</td>
</tr>
<tr>
<td>\</td>
<td>logical escape character</td>
<td>CP TERMINAL ESCAPE</td>
</tr>
<tr>
<td>@</td>
<td>logical character delete symbol</td>
<td>CP TERMINAL CHARDEL</td>
</tr>
</tbody>
</table>

8.6 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 8.7).

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 Character Code Translation Tables in the Compiler User's Guide).
The specification of this package is given in section 10.5.1.

8.7 Characteristics of Disk Files

A disk file that has already been created and is opened takes 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 below.

8.7.1 TEXT_IO

- A carriage control character is placed in column 1 if the CARRIAGE control attribute is specified in the FORM parameter.
- Data is translated between ASCII and EBCDIC so that the external file is readable using other System/370 tools.
- Under MVS, TEXT_IO files are implemented as DSORG PS (QSAM) datasets.

8.7.2 SEQUENTIAL_IO

- 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 descriptor in the case of unconstrained types.
- Under MVS, SEQUENTIAL_IO files are implemented as DSORG PS (QSAM) datasets.

8.7.3 DIRECT_IO

- 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 descriptor in the case of unconstrained types.
- Under CMS DIRECT_IO files may be read using SEQUENTIAL_IO (and vice-versa if a RECORD_SIZE component is specified).
- Under MVS, DIRECT_IO files are implemented as DSORG DA (BDAM) datasets.
9 Characteristics of Numeric Types

9.1 Integer Types

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

- **SHORT_SHORT_INTEGER**  -128 .. 127  -- \(2^{15} - 1\)
- **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**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
<th>Approximate Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>DIGITS</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>MANTISSA</td>
<td>21</td>
<td></td>
</tr>
<tr>
<td>EMAX</td>
<td>84</td>
<td></td>
</tr>
<tr>
<td>EPSILON</td>
<td>$2.0 \times 10^{-20}$</td>
<td>9.54E-07</td>
</tr>
<tr>
<td>SMALL</td>
<td>$2.0 \times 10^{-85}$</td>
<td>2.58E-26</td>
</tr>
<tr>
<td>LARGE</td>
<td>$2.0 \times 10^{-84} \times (1.0 - 2.0 \times 10^{-21})$</td>
<td>1.93E+25</td>
</tr>
<tr>
<td>SAFE_EMAX</td>
<td>252</td>
<td></td>
</tr>
<tr>
<td>SAFE_SMALL</td>
<td>$2.0 \times 10^{-253}$</td>
<td>6.91E-77</td>
</tr>
<tr>
<td>SAFE_LARGE</td>
<td>$2.0 \times 10^{-205} \times (1.0 - 2.0 \times 10^{-21})$</td>
<td>1.70E+38</td>
</tr>
<tr>
<td>FIRST</td>
<td>$-2.0 \times 10^{-245} \times (1.0 - 2.0 \times 10^{-24})$</td>
<td>-7.24E+75</td>
</tr>
<tr>
<td>LAST</td>
<td>$2.0 \times 10^{-243} \times (1.0 - 2.0 \times 10^{-24})$</td>
<td>7.24E+75</td>
</tr>
<tr>
<td>MACHINE_RADIX</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>MACHINE_MANTISSA</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>MACHINE_EMAX</td>
<td>63</td>
<td></td>
</tr>
<tr>
<td>MACHINE_EMIN</td>
<td>-64</td>
<td></td>
</tr>
<tr>
<td>MACHINE_ROUNDS</td>
<td>FALSE</td>
<td></td>
</tr>
<tr>
<td>MACHINE_OVERFLOWS</td>
<td>TRUE</td>
<td></td>
</tr>
<tr>
<td>SIZE</td>
<td>32</td>
<td></td>
</tr>
</tbody>
</table>

**FLOAT**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
<th>Approximate Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>DIGITS</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>MANTISSA</td>
<td>51</td>
<td></td>
</tr>
<tr>
<td>EMAX</td>
<td>204</td>
<td></td>
</tr>
<tr>
<td>EPSILON</td>
<td>$2.0 \times 10^{-50}$</td>
<td>8.88E-16</td>
</tr>
<tr>
<td>SMALL</td>
<td>$2.0 \times 10^{-205}$</td>
<td>1.94E-62</td>
</tr>
<tr>
<td>LARGE</td>
<td>$2.0 \times 10^{-204} \times (1.0 - 2.0 \times 10^{-51})$</td>
<td>2.57E+61</td>
</tr>
<tr>
<td>SAFE_EMAX</td>
<td>252</td>
<td></td>
</tr>
<tr>
<td>SAFE_SMALL</td>
<td>$2.0 \times 10^{-253}$</td>
<td>6.91E-77</td>
</tr>
<tr>
<td>SAFE_LARGE</td>
<td>$2.0 \times 10^{-205} \times (1.0 - 2.0 \times 10^{-51})$</td>
<td>7.24E+75</td>
</tr>
<tr>
<td>FIRST</td>
<td>$-2.0 \times 10^{-205} \times (1.0 - 2.0 \times 10^{-56})$</td>
<td>-7.24E+75</td>
</tr>
<tr>
<td>LAST</td>
<td>$2.0 \times 10^{-205} \times (1.0 - 2.0 \times 10^{-56})$</td>
<td>7.24E+75</td>
</tr>
<tr>
<td>MACHINE_RADIX</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>MACHINE_MANTISSA</td>
<td>14</td>
<td></td>
</tr>
<tr>
<td>MACHINE_EMAX</td>
<td>63</td>
<td></td>
</tr>
<tr>
<td>MACHINE_EMIN</td>
<td>-64</td>
<td></td>
</tr>
<tr>
<td>MACHINE_ROUNDS</td>
<td>FALSE</td>
<td></td>
</tr>
<tr>
<td>MACHINE_OVERFLOWS</td>
<td>TRUE</td>
<td></td>
</tr>
<tr>
<td>SIZE</td>
<td>64</td>
<td></td>
</tr>
</tbody>
</table>
LONG_FLOAT

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 ** -243 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

Appendix F. Implementation-Dependent Characteristics
10 Other Implementation-Dependent Characteristics

10.1 Characteristics of the Heap

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

All objects on the heap belonging to a given collection have their storage reclaimed on exit from the innermost block statement, subprogram body or task body that encloses the access type declaration associated with the collection. For access types declared at the library level, this deallocation occurs only on completion of the main program.

There is no further automatic storage reclamation performed, i.e. in effect all access types are deemed to be controlled [4.8]. The explicit deallocation of the object designated by an access value can be achieved by calling an appropriate instantiation of the generic procedure UNCHECKED_DEALLOCATION.

Space for the heap is initially claimed from the system on program start up and additional space may be claimed as required when the initial allocation is exhausted. The size of both the initial allocation and the size of the individual increments claimed from the system may be controlled by the Binder options SIZE and INCREMENT. Corresponding run-time options also exist.

On an extended architecture machine space allocated from the program heap may be above or below the 16 megabyte virtual storage line. The implementation defined pragma RMODE (see section 1.4) is provided to control the residence mode of objects allocated from the program heap.

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. A corresponding run-time option also exists.

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 multiple of 10 milliseconds. A corresponding run-time option also exists. 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 .. 0. 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.

10.5 Implementation Defined Packages

The following packages are defined by the Alsys Ada implementation for the IBM 370 under VM/CMS and MVS

10.5.1 Package EBCDIC

The implementation-defined package EBCDIC provides the user with access to the ASCII to EBCDIC and EBCDIC to ASCII translation facilities used by the TEXT_IO, SYSTEM_ENVIRONMENT and RECORD_IO packages.

The specification of package EBCDIC is as follows:

```
package EBCDIC is

  type EBCDIC_CHARACTER is (-- 0 = 0h
    nul, -- 1 = 1h
    soh, -- 2 = 2h
    stx, -- 3 = 3h
    etx, E_6,-- 5 = 5h
    ht,  E_6,-- 7 = 7h
    del,--

Appendix F. Implementation-Dependent Characteristics 53
```
<table>
<thead>
<tr>
<th>Code</th>
<th>Name</th>
<th>Hex</th>
</tr>
</thead>
<tbody>
<tr>
<td>E_9</td>
<td>11</td>
<td>0Bh</td>
</tr>
<tr>
<td>E_A</td>
<td>12</td>
<td>0Ch</td>
</tr>
<tr>
<td>vt</td>
<td>13</td>
<td>00h</td>
</tr>
<tr>
<td>np</td>
<td>14</td>
<td>0Eh</td>
</tr>
<tr>
<td>cr</td>
<td>15</td>
<td>0Fh</td>
</tr>
<tr>
<td>so</td>
<td>16</td>
<td>10h</td>
</tr>
<tr>
<td>si</td>
<td>17</td>
<td>11h</td>
</tr>
<tr>
<td>die</td>
<td>18</td>
<td>12h</td>
</tr>
<tr>
<td>ac1</td>
<td>19</td>
<td>13h</td>
</tr>
<tr>
<td>n1</td>
<td>20</td>
<td>15h</td>
</tr>
<tr>
<td>bs</td>
<td>21</td>
<td>16h</td>
</tr>
<tr>
<td>E_17</td>
<td>22</td>
<td>18h</td>
</tr>
<tr>
<td>can</td>
<td>23</td>
<td>19h</td>
</tr>
<tr>
<td>mm</td>
<td>24</td>
<td>1Ah</td>
</tr>
<tr>
<td>E_1A</td>
<td>25</td>
<td>1Bh</td>
</tr>
<tr>
<td>E_1B</td>
<td>26</td>
<td>1Ch</td>
</tr>
<tr>
<td>E_1C</td>
<td>27</td>
<td>1Dh</td>
</tr>
<tr>
<td>E_1D</td>
<td>28</td>
<td>1Eh</td>
</tr>
<tr>
<td>E_1E</td>
<td>29</td>
<td>1Fh</td>
</tr>
<tr>
<td>E_1F</td>
<td>30</td>
<td>20h</td>
</tr>
<tr>
<td>E_20</td>
<td>31</td>
<td>21h</td>
</tr>
<tr>
<td>E_21</td>
<td>32</td>
<td>22h</td>
</tr>
<tr>
<td>E_22</td>
<td>33</td>
<td>23h</td>
</tr>
<tr>
<td>E_23</td>
<td>34</td>
<td>24h</td>
</tr>
<tr>
<td>E_24</td>
<td>35</td>
<td>25h</td>
</tr>
<tr>
<td>E_25</td>
<td>36</td>
<td>26h</td>
</tr>
<tr>
<td>E_26</td>
<td>37</td>
<td>27h</td>
</tr>
<tr>
<td>E_27</td>
<td>38</td>
<td>28h</td>
</tr>
<tr>
<td>E_28</td>
<td>39</td>
<td>29h</td>
</tr>
<tr>
<td>E_29</td>
<td>40</td>
<td>2Ah</td>
</tr>
<tr>
<td>E_2A</td>
<td>41</td>
<td>2Bh</td>
</tr>
<tr>
<td>E_2B</td>
<td>42</td>
<td>2Ch</td>
</tr>
<tr>
<td>E_2C</td>
<td>43</td>
<td>2Dh</td>
</tr>
<tr>
<td>E_2D</td>
<td>44</td>
<td>2Eh</td>
</tr>
<tr>
<td>E_2E</td>
<td>45</td>
<td>2Fh</td>
</tr>
<tr>
<td>E_2F</td>
<td>46</td>
<td>30h</td>
</tr>
<tr>
<td>E_30</td>
<td>47</td>
<td>31h</td>
</tr>
<tr>
<td>E_31</td>
<td>48</td>
<td>32h</td>
</tr>
<tr>
<td>E_32</td>
<td>49</td>
<td>33h</td>
</tr>
<tr>
<td>E_33</td>
<td>50</td>
<td>34h</td>
</tr>
<tr>
<td>E_34</td>
<td>51</td>
<td>35h</td>
</tr>
<tr>
<td>E_35</td>
<td>52</td>
<td>36h</td>
</tr>
<tr>
<td>E_36</td>
<td>53</td>
<td>37h</td>
</tr>
<tr>
<td>E_37</td>
<td>54</td>
<td>38h</td>
</tr>
<tr>
<td>E_38</td>
<td>55</td>
<td>39h</td>
</tr>
<tr>
<td>E_39</td>
<td>56</td>
<td>3Ah</td>
</tr>
<tr>
<td>E_3A</td>
<td>57</td>
<td>3Bh</td>
</tr>
<tr>
<td>E_3B</td>
<td>58</td>
<td>3Ch</td>
</tr>
<tr>
<td>E_3C</td>
<td>59</td>
<td>3Dh</td>
</tr>
<tr>
<td>E_3D</td>
<td>60</td>
<td>3Eh</td>
</tr>
</tbody>
</table>

*Alsys IBM 370 Ada Compiler, Appendix F for VM/CMS and MVS, v4.0*
nal,
E_3E, -- 61 = 3OH
sub,
' @, -- 63 = 3FH
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 = 68H
' %', -- 108 = 6CH
' ', -- 109 = 6DH
' >', -- 110 = 6EH
' ?', -- 111 = 6FH
E_70,
E_71,
<table>
<thead>
<tr>
<th>ASCII</th>
<th>Hex</th>
</tr>
</thead>
<tbody>
<tr>
<td>E_72</td>
<td>121</td>
</tr>
<tr>
<td>E_73</td>
<td>122</td>
</tr>
<tr>
<td>E_74</td>
<td>123</td>
</tr>
<tr>
<td>E_75</td>
<td>124</td>
</tr>
<tr>
<td>E_76</td>
<td>125</td>
</tr>
<tr>
<td>E_77</td>
<td>126</td>
</tr>
<tr>
<td>E_78</td>
<td>127</td>
</tr>
<tr>
<td>E_79</td>
<td>128</td>
</tr>
<tr>
<td>E_80</td>
<td>129</td>
</tr>
<tr>
<td>E_81</td>
<td>130</td>
</tr>
<tr>
<td>E_82</td>
<td>131</td>
</tr>
<tr>
<td>E_83</td>
<td>132</td>
</tr>
<tr>
<td>E_84</td>
<td>133</td>
</tr>
<tr>
<td>E_85</td>
<td>134</td>
</tr>
<tr>
<td>E_86</td>
<td>135</td>
</tr>
<tr>
<td>E_87</td>
<td>136</td>
</tr>
<tr>
<td>E_88</td>
<td>137</td>
</tr>
<tr>
<td>E_89</td>
<td>138</td>
</tr>
<tr>
<td>E_90</td>
<td>139</td>
</tr>
<tr>
<td>E_91</td>
<td>140</td>
</tr>
<tr>
<td>E_92</td>
<td>141</td>
</tr>
<tr>
<td>E_93</td>
<td>142</td>
</tr>
<tr>
<td>E_94</td>
<td>143</td>
</tr>
<tr>
<td>E_95</td>
<td>144</td>
</tr>
<tr>
<td>E_96</td>
<td>145</td>
</tr>
<tr>
<td>E_97</td>
<td>146</td>
</tr>
<tr>
<td>E_98</td>
<td>147</td>
</tr>
<tr>
<td>E_99</td>
<td>148</td>
</tr>
<tr>
<td>E_100</td>
<td>149</td>
</tr>
<tr>
<td>E_101</td>
<td>150</td>
</tr>
<tr>
<td>E_102</td>
<td>151</td>
</tr>
<tr>
<td>E_103</td>
<td>152</td>
</tr>
<tr>
<td>E_104</td>
<td>153</td>
</tr>
<tr>
<td>E_105</td>
<td>154</td>
</tr>
<tr>
<td>E_106</td>
<td>155</td>
</tr>
<tr>
<td>E_107</td>
<td>156</td>
</tr>
<tr>
<td>E_108</td>
<td>157</td>
</tr>
<tr>
<td>E_109</td>
<td>158</td>
</tr>
<tr>
<td>E_110</td>
<td>159</td>
</tr>
<tr>
<td>E_111</td>
<td>160</td>
</tr>
<tr>
<td>E_112</td>
<td>161</td>
</tr>
</tbody>
</table>

*Asys IBM 370 Ada Compiler, Appendix F for VM/CMS and MVS, v4.0*
'X', \quad -- 167 = 0A7h

'Y', \quad -- 168 = 0A8h

'Z', \quad -- 169 = 0A9h

E_AA, E_AB, E_AC,

'0', \quad -- 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,

'1', \quad -- 180 = 0BDh

E_BE, E_BF,

'2', \quad -- 192 = 0C0h

'A', \quad -- 193 = 0C1h

'B', \quad -- 194 = 0C2h

'C', \quad -- 195 = 0C3h

'D', \quad -- 196 = 0C4h

'E', \quad -- 197 = 0C5h

'F', \quad -- 198 = 0C6h

'G', \quad -- 199 = 0C7h

'H', \quad -- 200 = 0C8h

'I', \quad -- 201 = 0C9h

E_CA, E_CB, E_CC, E_CD, E_CE, E_CF,

'J', \quad -- 208 = 0D0h

'K', \quad -- 209 = 0D1h

'L', \quad -- 210 = 0D2h

'M', \quad -- 211 = 0D3h

'N', \quad -- 212 = 0D4h

'O', \quad -- 213 = 0D5h

'P', \quad -- 214 = 0D6h

'Q', \quad -- 215 = 0D7h

'R', \quad -- 216 = 0D8h

'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_6;
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;
E  := constant EBCDIC_CHARACTER := E_4;
PQC  := constant EBCDIC_CHARACTER := E_17;
USS  := constant EBCDIC_CHARACTER := E_1A;
CUT  := constant EBCDIC_CHARACTE := 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;
IMP  := 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;
CJS : constant EBCDIC_CHARACTER := E_3B;
SP : constant EBCDIC_CHARACTER := E_3C;
RSP : constant EBCDIC_CHARACTER := E_41;
CENT : constant EBCDIC_CHARACTER := E_44;
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 := NULL;
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 := DC1;
E_11 : 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;
<table>
<thead>
<tr>
<th>ASCII</th>
<th>EBCDIC</th>
<th>ESOIC</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>eot</code></td>
<td><code>eot</code></td>
<td><code>eot</code></td>
</tr>
<tr>
<td><code>de4</code></td>
<td><code>de4</code></td>
<td><code>de4</code></td>
</tr>
<tr>
<td><code>nak</code></td>
<td><code>nak</code></td>
<td><code>nak</code></td>
</tr>
<tr>
<td><code>sub</code></td>
<td><code>sub</code></td>
<td><code>sub</code></td>
</tr>
<tr>
<td><code>a</code></td>
<td><code>a</code></td>
<td><code>a</code></td>
</tr>
<tr>
<td><code>b</code></td>
<td><code>b</code></td>
<td><code>b</code></td>
</tr>
<tr>
<td><code>c</code></td>
<td><code>c</code></td>
<td><code>c</code></td>
</tr>
<tr>
<td><code>d</code></td>
<td><code>d</code></td>
<td><code>d</code></td>
</tr>
<tr>
<td><code>e</code></td>
<td><code>e</code></td>
<td><code>e</code></td>
</tr>
<tr>
<td><code>f</code></td>
<td><code>f</code></td>
<td><code>f</code></td>
</tr>
<tr>
<td><code>g</code></td>
<td><code>g</code></td>
<td><code>g</code></td>
</tr>
<tr>
<td><code>h</code></td>
<td><code>h</code></td>
<td><code>h</code></td>
</tr>
<tr>
<td><code>i</code></td>
<td><code>i</code></td>
<td><code>i</code></td>
</tr>
<tr>
<td><code>j</code></td>
<td><code>j</code></td>
<td><code>j</code></td>
</tr>
<tr>
<td><code>k</code></td>
<td><code>k</code></td>
<td><code>k</code></td>
</tr>
<tr>
<td><code>l</code></td>
<td><code>l</code></td>
<td><code>l</code></td>
</tr>
<tr>
<td><code>m</code></td>
<td><code>m</code></td>
<td><code>m</code></td>
</tr>
<tr>
<td><code>n</code></td>
<td><code>n</code></td>
<td><code>n</code></td>
</tr>
<tr>
<td><code>o</code></td>
<td><code>o</code></td>
<td><code>o</code></td>
</tr>
<tr>
<td><code>p</code></td>
<td><code>p</code></td>
<td><code>p</code></td>
</tr>
<tr>
<td><code>q</code></td>
<td><code>q</code></td>
<td><code>q</code></td>
</tr>
<tr>
<td><code>r</code></td>
<td><code>r</code></td>
<td><code>r</code></td>
</tr>
<tr>
<td><code>s</code></td>
<td><code>s</code></td>
<td><code>s</code></td>
</tr>
<tr>
<td><code>t</code></td>
<td><code>t</code></td>
<td><code>t</code></td>
</tr>
<tr>
<td><code>u</code></td>
<td><code>u</code></td>
<td><code>u</code></td>
</tr>
</tbody>
</table>
type EBCDIC_STRING is array (POSITIVE range <>) of EBCDIC_CHARACTER;

function ASCII_TO_EBCDIC (S : STRING) return EBCDIC_STRING;
function ASCII_TO_EBCDIC (C : CHARACTER) return EBCDIC_CHARACTER;
-- 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;
function EBCDIC_TO_ASCII (C : EBCDIC_CHARACTER) return CHARACTER;

-- 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;

EBCDIC_CHARACTER

The type EBCDIC_CHARACTER provides an Ada character type [3.5.2] following the EBCDIC character set encoding.

EBCDIC_STRING

The type EBCDIC_STRING provides a one dimensional array of the type EBCDIC_CHARACTER, indexed by values of the predefined type POSITIVE.

EBCDIC_STRING implements strings of EBCDIC_CHARACTER in the same way that the predefined type STRING implements strings of the predefined type CHARACTER.

In many ways EBCDIC_STRINGs may be manipulated exactly as the predefined type STRING; in particular, string literals and catenations are available.

ASCII_TO_EBCDIC

The subprograms ASCII_TO_EBCDIC convert ASCII encoded data to EBCDIC encoded data.

EBCDIC_TO_ASCII

The subprograms EBCDIC_TO_ASCII convert EBCDIC encoded data to ASCII encoded data.

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. 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.

The user may alter the ASCII to EBCDIC and EBCDIC to ASCII mappings used by the Alsyn IBM 370 Ada compiler, as described in the Installation Guides.
10.5.2 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 package SYSTEM_ENVIRONMENT is as follows:

```ada
package SYSTEM_ENVIRONMENT is

  MVS : constant BOOLEAN := boolean_value;

  subtype EXIT_STATUS is INTEGER;
  type STACK_MODE is (LIFO, FIFO);

  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;
  function ARG_COUNT return NATURAL;
  function ARG_VALUE (INDEX : in POSITIVE) return STRING;

  ARGUMENT_ERROR : exception;

  procedure SET_EXITSTATUS (STATUS : in EXIT_STATUS);
  function GET_EXITSTATUS return EXIT_STATUS;

  function EXECUTE_COMMAND (COMMAND : in STRING) return EXITSTATUS;
  procedure EXECUTE_COMMAND (COMMAND : in STRING);

  procedure STACK (COMMAND : in STRING;
                   MODE : in STACK_MODE := LIFO);
  function SENTRIES return NATURAL;

  procedure ABORT_PROGRAM (STATUS : in EXIT_STATUS);

  function SYSTIME return DURATION;
  function USRTIME return DURATION;

  function EXISTS (FILE : in STRING) return BOOLEAN;

  function LAST_EXCEPTION_NAME return STRING;

end SYSTEM_ENVIRONMENT;
```

MVS

The MVS boolean provides a convenient way for a user to query at run time whether a program is running under MVS or VM/CMS. This facility allows for the conditional execution of operating system specific code.
The boolean constant has the value TRUE under MVS and FALSE under VM/CMS.

ARG_LINE

The ARG_LINE subprograms give access to the CMS command line, the TSO command line parameters or the program PARM string as specified in the JCL used to run an MVS 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.

Under CMS the command line returned includes the name of the program executed, but not any run-time options specified.

Under MVS the name of the program executed is not available, but any run-time options specified are excluded, as under CMS.

ARG_START

The function ARG_START returns the index in the command line of the first parameter, i.e. ignoring the executed program name, for CMS; for MVS it always returns the value 1.

ARG_COUNT

The function ARG_COUNT returns the number of parameters in the command line of the program. The executed program name which is part of the command line as returned by ARG_LINE under CMS is not included in the count. Thus, ARG_COUNT for a program without parameters returns zero under both CMS and MVS.

ARG_VALUE

The function ARG_VALUE returns the specified parameter from the command line. Parameters are considered to be indexed from 1. The executed program name which is part of the command line as returned by ARG_LINE under CMS is not considered as a parameter, i.e. ARG_VALUE(1) returns the first user parameter. The exception ARGUMENT_ERROR is raised if the specified index is greater than ARG_COUNT.
SET_EXIT_STATUS

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 modify the exit status: the status finally returned being that specified by the last executed call to SET_EXIT_STATUS. 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, otherwise the value 0 is returned.

The following exit codes relate to unhandled exceptions:

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

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.
GET_EXIT_STATUS

The function GET_EXIT_STATUS returns the current exit status.

EXECUTE_COMMAND

Under CMS the EXECUTE_COMMAND subprograms with a non-null parameter execute the given CMS SUBSET command. The result of the EXECUTE_COMMAND function is the return code of the command. If a null string is given as the parameter, the program exits to the CMS subset level. This allows CMS SUBSET commands to be executed directly. Issuing the command RETURN from the CMS subset level will return to the Ada program. The return code of the EXECUTE_COMMAND function with a null COMMAND string is always zero.

Under MVS a call of the EXECUTE_COMMAND subprograms has no effect and the function always returns the value zero.

STACK

Under CMS the STACK procedure allows a command to be placed on the console stack: either last-in-first-out (LIFO) or first-in-first-out (FIFO).

Under MVS a call of the STACK procedure has no effect.

SENTRIES

Under CMS, the SENTRIES function returns the number of lines in the program stack.

Under MVS calls to the SENTRIES function always return the value 0.

ABORT_PROGRAM

The program may be aborted, returning the specified exit code, by a call of the ABORT_PROGRAM procedure.

SYSTIME, USRTIME

Under CMS the SYSTIME and USRTIME functions allow access to the amount of system and user time, respectively, used by the program since its execution.

Under MVS a call of either of these functions has no effect and returns the value 0.0.

EXISTS

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

The function \texttt{LAST\_EXCEPTION\_NAME} returns the name of the most recently raised exception in the current task. It may be used in handlers to identify the exception, e.g.:

\begin{verbatim}
... when others =>
    TEXT\_IO\_PUT (SYSTEM\_ENVIRONMENT.LAST\_EXCEPTION\_NAME);
    TEXT\_IO\_PUT\_LINE ("raised");
\end{verbatim}

10.5.3 Package RECORD\_IO

The implementation-defined package RECORD\_IO enables an Ada program to perform simple, record oriented I/O of an anonymous data type in an efficient manner.

RECORD\_IO provides similar facilities to the predefined packages SEQUENTIAL\_IO and DIRECT\_IO in a non-generic form. The package is therefore "typeless": the data on which I/O is being performed being specified via its address and length. It is the programmer's responsibility to see that the data manipulated by the facilities of RECORD\_IO is handled in a consistent manner.

The specification of package RECORD\_IO is as follows:

\begin{verbatim}
with SYSTEM, IO\_EXCEPTIONS;
package RECORD\_IO is
  -- *************
  -- * TYPES *
  -- *************
  type COUNT is range 0..INTEGER'LAST;
  subtype POSITIVE\_COUNT is COUNT range 1..COUNT'LAST;
  type FILE\_TYPE is limited private;
  type FILE\_MODE is (IN\_FILE, IN\_OUT\_FILE, OUT\_FILE);
  type FILE\_ORGANISATION is (SEQUENTIAL, DIRECT);
  -- ****************
  -- * FILE MANAGEMENT *
  -- ****************
  procedure CREATE (FILE : in out FILE\_TYPE;
                    MODE : in FILE\_MODE := OUT\_FILE;
                    NAME : in STRING := "";
                    FORM : in STRING := "";
                    ORGANISATION : in FILE\_ORGANISATION := SEQUENTIAL;
                    TRANSLATE : in BOOLEAN := FALSE);
\end{verbatim}

Appendix F. Implementation-Dependent Characteristics 67
procedure OPEN (FILE : in out FILE_TYPE;
    MODE : in FILE_MODE;
    NAME : in STRING;
    FORM : in STRING := "";
    ORGANISATION : in FILE_ORGANISATION := SEQUENTIAL;
    TRANSLATE : in BOOLEAN := FALSE);

procedure CLOSE (FILE : in out FILE_TYPE);
procedure DELETE (FILE : in out FILE_TYPE);
procedure RESET (FILE : in out FILE_TYPE;
    MODE : in FILE_MODE);
procedure RESET (FILE : in out FILE_TYPE);

function MODE (FILE : in FILE_TYPE) return FILE_MODE;
function NAME (FILE : in FILE_TYPE) return STRING;
function FORM (FILE : in FILE_TYPE) return STRING;

function IS_OPEN (FILE : in FILE_TYPE) return BOOLEAN;

-- * INPUT / OUTPUT *
-- ***************

procedure READ (FILE : in FILE_TYPE;
    ITEM : in SYSTEM.ADDRESS;
    LENGTH : in out NATURAL);

-- Only for DIRECT organisation files
procedure READ (FILE : in FILE_TYPE;
    ITEM : in SYSTEM.ADDRESS;
    LENGTH : in out NATURAL;
    FROM : in POSITIVE_COUNT);

procedure WRITE (FILE : in FILE_TYPE;
    ITEM : in SYSTEM.ADDRESS;
    LENGTH : in NATURAL);

-- Only for DIRECT organisation files
procedure WRITE (FILE : in FILE_TYPE;
    ITEM : in SYSTEM.ADDRESS;
    LENGTH : in NATURAL;
    TO : in POSITIVE_COUNT);

function END_OF_FILE (FILE : in FILE_TYPE) return BOOLEAN;

-- Only for DIRECT organisation files
procedure SET_INDEX (FILE : in FILE_TYPE;
    TO : in POSITIVE_COUNT);
function INDEX (FILE : in FILE_TYPE) return POSITIVE_COUNT;
function SIZE (FILE : in FILE_TYPE) return COUNT;
The specification of RECORD IO is similar to that of the predefined generic packages SEQUENTIAL IO and DIRECT IO. The file management facilities provided are analogous, with CREATE, OPEN, CLOSE, DELETE and RESET procedures, in addition to MODE, NAME, FORM and IS_OPEN functions as in the predefined packages. The syntax and semantics of file names and form strings are identical to those of the predefined packages, as described in section 8 of this appendix. The CREATE and OPEN procedures take two additional parameters, as below:

**ORGANISATION**

The ORGANISATION parameter specifies the organisation of the file being created or opened. Two types of organisations may be specified:

**SEQUENTIAL**

Sequential organised files correspond to files that could be created via an instantiation of the generic package SEQUENTIAL IO. Records in a sequential organisation file are variable length according to the length data written to them. A sequential organisation file is implemented as a QSAM file under MVS.

A sequential organised file must be written and read sequentially. An attempt to pass a FILE_TYPE value representing a sequential organised file to the READ or WRITE procedures with an explicit specification of the file record to be read or written (FROM or TO parameters), or to use those subprograms which directly manipulate the file index (SET_INDEX, etc.) will raise USE_ERROR.
DIRECT

Direct organised files correspond to files that could be created via an instantiation of the generic package DIRECT_IO. Records in a direct organisation file are fixed length according to the *record_length* form parameter, which the user must specify when creating a direct organisation file. Failure to specify the *record_length* form parameter on creating a direct organisation file will raise USE_ERROR, since there is no Ada type associated with the file to whose size the record length may default. A direct organisation file is implemented as a BDAM file under MVS.

TRANSLATE

The TRANSLATE parameter specifies whether ASCII to EBCDIC translation is to be performed on the data on output and whether EBCDIC to ASCII translation is to be correspondingly performed on input.

Use of the TRANSLATE parameter allows records of the external file to hold character data in an appropriate form for manipulation by other 370 tools expecting EBCDIC encoded character data.

The input output facilities themselves are represented by overloaded READ and WRITE procedures.

These procedures are analogous to those of SEQUENTIAL_IO and DIRECT_IO. The data is specified via its address (ITEM) and length (LENGTH).

The external file is characterised by its RECFM and LRECL attributes. These may be explicitly controlled via the FORM parameter (see section 8.2) or else default as below:

<table>
<thead>
<tr>
<th>Organisation</th>
<th>Attribute</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential</td>
<td>RECFM</td>
<td>V</td>
</tr>
<tr>
<td></td>
<td>LRECL</td>
<td>4096</td>
</tr>
<tr>
<td>Direct</td>
<td>RECFM</td>
<td>F</td>
</tr>
<tr>
<td></td>
<td>LRECL</td>
<td>No default</td>
</tr>
</tbody>
</table>

On output, LENGTH bytes of data are written to the appropriate record of the file from the address specified. If the LENGTH specified is greater than LRECL then DATA_ERROR is raised. If the LENGTH specified is less than LRECL and RECFM is F then the data is written at the start of the record. The remaining portion of the record will contain EBCDIC space characters (i.e. bytes whose value is 16#40#). If the LENGTH specified is less than LRECL and RECFM is V then a record of exactly LENGTH bytes is written, the LRECL specifying the maximum permissible record length.
On input, the appropriate record of the file is read at the address specified. If the length of the appropriate record of the file is less than \texttt{LENGTH} bytes, the entire record is read and the actual record length returned in \texttt{LENGTH}. If the length of the appropriate record of the file is greater than \texttt{LENGTH} bytes then \texttt{DATA\_ERROR} is raised. Note that for a direct organisation file the uninitialised portion of the record is considered to be part of the record length on input. It is the programmer's responsibility to read and write records via the facilities of \texttt{RECORD\_IO} in a consistent manner.

\texttt{READ} and \texttt{WRITE} procedures with explicit specification of the file record to be read or written (\texttt{FROM} and \texttt{TO} parameters) are only applicable to files opened or created with direct organisation. Application of these procedures to a sequential organisation file will raise \texttt{USE\_ERROR}.

The remaining input output facilities are analogous to the corresponding subprograms in \texttt{SEQUENTIAL\_IO} or \texttt{DIRECT\_IO}, with \texttt{END\_OF\_FILE}, \texttt{SET\_INDEX}, \texttt{INDEX} and \texttt{SIZE} subprograms. \texttt{END\_OF\_FILE} is applicable to both sequential and direct organisation files. The remainder, however, are only supported for files opened or created with direct organisation. Application of these procedures to a sequential organisation file will raise \texttt{USE\_ERROR}.

All other exceptional conditions raise the corresponding exceptions to those of the predefined I/O packages.

\subsection*{10.5.4 Package STRINGS}

The implementation-defined package \texttt{STRINGS} is a utility package providing the user with many commonly required string manipulation facilities.

The specification of package \texttt{STRINGS} is as follows:

\begin{verbatim}
with UNCHECKED_DEALLOCATION;
package STRINGS is
   -- **********
   -- * TYPES *
   -- **********
   type ACCESS_STRING is access STRING;
   procedure DEALLOCATE_STRING is new UNCHECKED_DEALLOCATION (STRING,
                                        ACCESS_STRING);
   -- **********
   -- * UTILITIES *
   -- **********
   function UPPER (C : in CHARACTER) return CHARACTER;
   function UPPER (S : in STRING) return STRING;
   procedure UPPER (S : in out STRING);
   function LOWER (C : in CHARACTER) return CHARACTER;
   function LOWER (S : in STRING) return STRING;
end STRINGS;
\end{verbatim}

\appendix{Implementation-Dependent Characteristics}
procedure LOWER (S : in out STRING);

function CAPITAL (S : in STRING) return STRING;
procedure CAPITAL (S : in out STRING);

function REMOVE_LEADING_BLANKS (S : in STRING) return STRING;
function REMOVE.TRAILING_BLANKS (S : in STRING) return STRING;
function TRIM (S : in STRING) return STRING;

function INDEX (C : in CHARACTER;
INTO : in STRING;
START : in POSITIVE := 1) return NATURAL;
function INDEX (S : in STRING;
INTO : in STRING;
START : in POSITIVE := 1) return NATURAL;

function NOT_INDEX (C : in CHARACTER;
INTO : in STRING;
START : in POSITIVE := 1) return NATURAL;
function NOT_INDEX (S : in STRING;
INTO : in STRING;
START : in POSITIVE := 1) return NATURAL;

function IS_AN_ABBREV (ABBREV : in STRING;
FULL_WORD : in STRING;
IGNORE_CASE : in BOOLEAN := TRUE) return BOOLEAN;

function MATCH_PATTERN (S : in STRING;
PATTERN : in STRING;
IGNORE_CASE : in BOOLEAN := TRUE) return BOOLEAN;

function ' & ' (LEFT : in STRING; RIGHT : in STRING) return STRING;
function ' & ' (LEFT : in STRING; RIGHT : in CHARACTER) return STRING;
function ' & ' (LEFT : in CHARACTER; RIGHT : in STRING) return STRING;
function ' & ' (LEFT : in CHARACTER; RIGHT : in CHARACTER) return STRING;

end STRINGS;

ACCESSSTRING

The ACCESSSTRING type is a convenient declaration of the commonly used access to string type.
DEALLOCATE_STRING

The DEALLOCATE_STRING procedure is an instantiation of UNCHECKED_DEALLOCATION for the type ACCESS_STRING. Note that since the type ACCESS_STRING is declared at the library level, the scope of the corresponding collection is only exited at program completion. For this reason, STRING objects belonging to this collection are never automatically deallocated. It is the programmer's responsibility to manage the deallocation of objects within this collection.

UPPER

The UPPER subprograms convert any lower case letters in their parameters to the corresponding upper case letters. Characters which are not lower case letters are unaffected. The procedure is more efficient than the corresponding function, as it does not make use of the program heap.

LOWER

The LOWER subprograms convert any upper case letters in their parameters to the corresponding lower case letters. Characters which are not upper case letters are unaffected. The procedure is more efficient than the corresponding function, as it does not make use of the program heap.

CAPITAL

The CAPITAL subprograms "capitalise" their parameters. That is they UPPER the first character and LOWER all subsequent characters of the string. The procedure is more efficient than the corresponding function, as it does not make use of the program heap.

REMOVE_LEADING_BLANKS

The REMOVE_LEADING_BLANKS function returns its parameter string with all leading spaces removed.

REMOVE_TRAILING_BLANKS

The REMOVE_TRAILING_BLANKS function returns its parameter string with all trailing spaces removed.

TRIM

The TRIM function returns its parameter string with all leading and all trailing spaces removed.

Appendix F. Implementation-Dependent Characteristics
INDEX

The INDEX subprograms return the index into the specified string (INTO) of the first character of the first occurrence of a given substring (S) or character (C). The search for the substring or character commences at the index specified by START. If the substring or character is not found, the functions return the value 0. Case is considered significant.

NOT_INDEX

The NOT_INDEX subprograms return the index into the specified string (INTO) of the first character which does not occur in the given string (S) or does not match the given character (C). The search for the non-matching character commences at the index specified by START. If all the characters of the string match, the functions return the value 0. Case is considered significant.

IS_AN_ABBREV

The IS_AN_ABBREV function determines whether the string ABBREV is an abbreviation for the string FULLWORD. Leading and trailing spaces in ABBREV are first removed and the trimmed string is then considered to be an abbreviation for FULLWORD if it is a proper prefix of FULLWORD.

The parameter IGNORE_CASE controls whether case is considered significant or not.

MATCH_PATTERN

The MATCH_PATTERN function determines whether the string S matches the pattern specified in PATTERN. A pattern is simply a string in which the character '*' is considered a wild-card which can match any number of any characters.

For example the string "ABCDEFG" is considered to match the pattern "A*G" and the pattern "ABCD*EFG*"

The parameter IGNORE_CASE controls whether case is considered significant or not.

The package STRINGS also provides overloaded subprograms designated by '&'. These are identical to the corresponding subprograms declared in package STANDARD, except that the concatenations are performed out of line. By performing concatenations out of line the size of the inline generated code is minimised at the expense of execution speed.
INDEX

%SYSIN 46
%SYSOUT 46

ABORT_PROGRAM procedure 66
ACCESS_STRING 72
Access_type_name 8
ADDRESS attribute 10
restrictions 10
Append attribute 44
ARG_COUNT function 64
ARG_LINE subprograms 64
ARG_START function 64
ARG_VALUE function 64
ARRAY_DESCRIPTOR attribute 36
ASCII 5, 6, 47, 48, 62
form feed 47
line feed 47
ASCII_TO_EBCDIC 62
ASSEMBLER 2
Attributes 10
ARRAY_DESCRIPTOR 36
DESCRIPTOR_SIZE 10
IS ARRAY 10
RECORD_DESCRIPTOR 36
RECORD_SIZE 36, 40
representation attributes 10
VARIANT_INDEX 36

BDAM 70
Binder 52
Binder options
SLICE 52
TASK 52
Block_size attribute 43, 46
Boolean types 4

CAPITAL subprograms 73
Carriage_control attribute 43, 46
CHARACTER 5, 47
Characteristics of disk files 48
CMS command line 64
CMS file name 39
CMS subset command 66
Compilation unit ordering 53
Console stack 66
CONSTRAINT_ERROR 65
COUNT 49

DD SYSIN 46

DD SYSOUT 46
DDNAME 39, 40
Deadlocking 65
DEALLOCATE_STRING subprogram 73
DESCRIPTOR_SIZE attribute 10, 42
DIRECT_IO 39, 47, 49
DURATION attributes 51

EBCDIC 47, 48, 53, 62
ASCII_TO_EBCDIC 62
EBCDIC_CHARACTER 62
EBCDIC_STRING 62
EBCDIC_TO_ASCII 62
EBCDIC_CHARACTER 53, 62
EBCDIC_STRING 62
EBCDIC_TO_ASCII 62
END_OF_FILE 45
Enumeration types 5
CHARACTER 5
EOF_STRING 47
Eof_string attribute 44, 46, 47
ESPIE 6
Exceptions 65
EXECUTE_COMMAND subprograms 66
EXISTS function 66
Exit status of program 65

FIELD 49
File sharing attribute 41
FILEDEF command 39
Fixed point types 5
DURATION 51
FLOAT 5, 50
Floating point types 5, 50
attributes 50
FLOAT 5, 50
LONG_FLOAT 5, 51
SHORT_FLOAT 5, 50
FORM parameter
for MVS 45
for VM/CMS 40
FORM parameter attributes
append 44
block_size attribute 43, 46
carriage_control 43, 46
eof_string 44, 46, 47

Index 75
file sharing attribute 41
primary attribute 45
record_format attribute 41
record_size attribute 42, 46
secondary attribute 46
truncate 44
unit attribute 45
volume attribute 45
Fully qualified name 40

GET_EXIT_STATUS function 66

Implementation-dependent attributes 10
Implementation-dependent characteristics
others 52
Implementation-dependent pragma 2
Implementation-generated names 36
IMPROVE 8
INDENT 7
INDEX subprograms 74
INLINE 2
Input-Output
MVS 39
VM/CMS 39
Input-Output packages 39
DIRECT_IO 39
IO_EXCEPTIONS 39
LOW_LEVEL_IO 39
SEQUENTIAL_IO 39
TEXT_IO 39
INTEGER 4, 49
Integer types 4, 49
COUNT 49
FIELD 49
INTEGER 4, 49
POSITIVE_COUNT 49
SHORT_INTEGER 4, 49
SHORT_SHORT_INTEGER 4, 49
INTERFACE 2
INTERFACE_NAME 2, 6
Interfaced subprograms
Restrictions 6
IO_EXCEPTIONS 39
IS_AN_ABBREV subprogram 74
IS_ARRAY attribute 10

Language_name 2
LAST_EXCEPTION_NAME function 67
LONG_FLOAT 5, 51
LOW_LEVEL_IO 39
LOWER subprograms 73

Main program
definition 53
MATCH_PATTERN subprogram 74
MVS dataset name 39, 40
MVS file name
PARM string 40
QUALIFIER parameter 40

NAME parameter
for MVS 39, 40
for VM/CMS 39
NOT_INDEX subprograms 74
NOT_SHARED 41
Numeric types
characteristics 49
Fixed point types 51
Floating point types 50
integer types 49
NUMERIC_ERROR 65

PACK 8
Parameter representations 4
Access types 5
Array types 6
Boolean types 4
Enumeration types 5
Fixed point types 5
Floating point types 5
Integer types 4
Record types 6
Parameter-passing conventions 3
PARM string 40, 64
POSITIVE_COUNT 49
Pragma INLINE 2
Pragma INTERFACE 2
ASSEMBLER 2
language_name 2
subprogram_name 2
Pragma INTERFACE_NAME 2
string_literal 7
subprogram_name 6
Pragma RMODE
access_type_name 8
residence_mode 8
Pragmas
IMPROVE 8
INDENT 7
INTERFACE 2
INTERFACE_NAME 6
PACK 8
PRIORITY 8, 52
RMODE 8
SUPPRESS 9
Primary attribute 45
PRIORITY 8
PRIORITY pragma 52
Program exit status 65
PROGRAM_ERROR 65

QSAM 69
QUALIFIER parameter 40

RECORD_DESCRIPTOR attribute 36
Record_format attribute 41
RECORD_IO 53, 67
ORGANISATION parameter 69
TRANSLATE parameter 70
RECORD_SIZE attribute 36, 40, 42, 46
REMOVE_LEADING_BLANKS
  subprogram 73
REMOVE_TRAILING_BLANKS
  subprogram 73
Representation attributes 10
Representation clauses 12
restrictions 12
Residence_mode 8
RMODE 8

Secondary attribute 46
SENTRIES function 66
SEQUENTIAL_IO 39, 47
SET_EXIT_STATUS function 65
SHARED 41
SHORT_FLOAT 5, 50
SHORT_INTEGER 4, 49
SHORT_SHORT_INTEGER 4, 49
SLICE option 52
SPIE 6
SPURIOUS_ERROR 65
STACK procedure 66
STANDARD_INPUT 46
STANDARD_OUTPUT 46
STORAGE_ERROR 65
STRING 6, 47
String literal 7
STRINGS 71
  ACCESS_STRING 72
  CAPITAL 73
  DEALLOCATE_STRING 73
  INDEX 74
  IS_AN_ABBREV 74
  LOWER 73
  MATCH_PATTERN 74
  NOT_INDEX 74

REMOVE_LEADING_BLANKS 73
REMOVE_TRAILING_BLANKS 73
TRIM 73
UPPER 73
Subprogram_name 2, 6
SUPPRESS 9
SYSTEM package 11
SYSTEM_ENVIRONMENT 47, 53, 63
  ABORT_PROGRAM 66
  ARG_COUNT 64
  ARG_LINE 64
  ARG_START 64
  ARG_VALUE 64
  EXECUTE_COMMAND 66
  EXISTS 66
  GET_EXIT_STATUS 66
  LAST_EXCEPTION_NAME 67
  MVS 63
  SENTRIES 66
  SET_EXIT_STATUS 65
  STACK 66
  SYSTIME 66
  USRTIME 66
SYSTIME function 66

TASK option 52
Tasks
  characteristics 52
  Timeslicing 52
Text terminators 47
TEXT_IO 39, 47, 49, 53
TRIM subprogram 73
Truncate attribute 44

Unchecked conversions 38
restrictions 38
Unit attribute 45
Unqualified name 40
UPPER subprograms 73
USE_ERROR 40, 46
USRTIME function 66

VARIANT_INDEX attribute 36
Volume attribute 45
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.

MACRO DEFINITIONS

<table>
<thead>
<tr>
<th>Name and Meaning</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$ACCSIZE</td>
<td>32</td>
</tr>
<tr>
<td>An integer literal whose value is the</td>
<td></td>
</tr>
<tr>
<td>number of bits sufficient to hold any</td>
<td></td>
</tr>
<tr>
<td>value of an access type.</td>
<td></td>
</tr>
<tr>
<td>$BIG_ID1</td>
<td>(1..254=&gt;'A', 255=&gt;1)</td>
</tr>
<tr>
<td>Identifier the size of the maximum</td>
<td></td>
</tr>
<tr>
<td>input line length with varying last</td>
<td></td>
</tr>
<tr>
<td>character.</td>
<td></td>
</tr>
<tr>
<td>$BIG_ID2</td>
<td>(1..254=&gt;'A', 255=&gt;2)</td>
</tr>
<tr>
<td>Identifier the size of the maximum</td>
<td></td>
</tr>
<tr>
<td>input line length with varying last</td>
<td></td>
</tr>
<tr>
<td>character.</td>
<td></td>
</tr>
<tr>
<td>$BIG_ID3</td>
<td>(1..127=&gt;'A', 128=&gt;3,</td>
</tr>
<tr>
<td>Identifier the size of the maximum</td>
<td>129..255=&gt;'A')</td>
</tr>
<tr>
<td>input line length with varying middle</td>
<td></td>
</tr>
<tr>
<td>character.</td>
<td></td>
</tr>
<tr>
<td>$BIG_ID4</td>
<td>(1..127=&gt;'A', 128=&gt;4,</td>
</tr>
<tr>
<td>Identifier the size of the maximum</td>
<td>129..255=&gt;'A')</td>
</tr>
<tr>
<td>input line length with varying middle</td>
<td></td>
</tr>
<tr>
<td>character.</td>
<td></td>
</tr>
<tr>
<td>$BIG_INT_LIT</td>
<td>(1..252=&gt;0,</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Name and Meaning

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>An integer literal of value 298 with enough leading zeroes so that it is the size of the maximum line length</td>
<td>253..255=&gt;298)</td>
</tr>
<tr>
<td>$BIG_REAL_LIT</td>
<td>A universal real literal of value 690.0 with enough leading zeroes to be the size of the maximum line length.</td>
</tr>
<tr>
<td>$BIG_STRING1</td>
<td>A string literal which when catenated with BIG_STRING2 yields the image of BIG_ID1.</td>
</tr>
<tr>
<td>$BIG_STRING2</td>
<td>A string literal which when catenated to the end of BIG_STRING1 yields the image of BIG_ID1.</td>
</tr>
<tr>
<td>$BLANKS</td>
<td>A sequence of blanks twenty characters less than the size of the maximum line length.</td>
</tr>
<tr>
<td>$COUNT_LAST</td>
<td>A universal integer literal whose value is TEXT_IO.COUNT'LAST.</td>
</tr>
<tr>
<td>$DEFAULT_MEM_SIZE</td>
<td>An integer literal whose value is SYSTEM.MEMORY_SIZE.</td>
</tr>
<tr>
<td>$DEFAULT_STOR_UNIT</td>
<td>An integer literal whose value is SYSTEM.STORAGE_UNIT.</td>
</tr>
<tr>
<td>$DEFAULT_SYS_NAME</td>
<td>The value of the constant SYSTEM.SYSTEM_NAME.</td>
</tr>
<tr>
<td>$DELTA_DOC</td>
<td>A real literal whose value is SYSTEM.FINE_DELTA.</td>
</tr>
<tr>
<td>Name and Meaning</td>
<td>Value</td>
</tr>
<tr>
<td>----------------------------------</td>
<td>--------------------</td>
</tr>
<tr>
<td><code>$FIELD_LAST</code></td>
<td>255</td>
</tr>
<tr>
<td>A universal integer literal whose value is TEXT_IO.FIELD'LAST.</td>
<td></td>
</tr>
<tr>
<td><code>$FIXED_NAME</code></td>
<td>NO_SUCH_TYPE</td>
</tr>
<tr>
<td>The name of a predefined fixed-point type other than DURATION.</td>
<td></td>
</tr>
<tr>
<td><code>$FLOAT_NAME</code></td>
<td>NO_SUCH_TYPE</td>
</tr>
<tr>
<td>The name of a predefined floating-point type other than FLOAT, SHORT_FLOAT, or LONG_FLOAT.</td>
<td></td>
</tr>
<tr>
<td><code>$GREATER_THAN_DURATION</code></td>
<td>1000000.0</td>
</tr>
<tr>
<td>A universal real literal that lies between DURATION'BASE'LAST and DURATION'LAST or any value in the range of DURATION.</td>
<td></td>
</tr>
<tr>
<td><code>$GREATER_THAN_DURATION_BASE_LAST</code></td>
<td>10000000.0</td>
</tr>
<tr>
<td>A universal real literal that is greater than DURATION'BASE'LAST.</td>
<td></td>
</tr>
<tr>
<td><code>$HIGH_PRIORITY</code></td>
<td>10</td>
</tr>
<tr>
<td>An integer literal whose value is the upper bound of the range for the subtype SYSTEM.PRIORITY.</td>
<td></td>
</tr>
<tr>
<td><code>$ILLEGAL_EXTERNAL_FILE_NAME1</code></td>
<td>T??????? LISTING A1</td>
</tr>
<tr>
<td>An external file name which contains invalid characters.</td>
<td></td>
</tr>
<tr>
<td><code>$ILLEGAL_EXTERNAL_FILE_NAME2</code></td>
<td>TOOLONGNAME TOOLONGTYPE TOOLONGMODE</td>
</tr>
<tr>
<td>An external file name which is too long.</td>
<td></td>
</tr>
<tr>
<td><code>$INTEGER_FIRST</code></td>
<td>-2147483648</td>
</tr>
<tr>
<td>A universal integer literal whose value is INTEGER'FIRST.</td>
<td></td>
</tr>
<tr>
<td><code>$INTEGER_LAST</code></td>
<td>2147483647</td>
</tr>
<tr>
<td>A universal integer literal whose value is INTEGER'LAST.</td>
<td></td>
</tr>
</tbody>
</table>
### Name and Meaning

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{INTEGER_LAST_PLUS_1}$</td>
<td>$2147483648$</td>
</tr>
<tr>
<td>A universal integer literal whose value is $\text{INTEGER'}\text{LAST}+1$.</td>
<td></td>
</tr>
<tr>
<td>$\text{LESS_THAN_DURATION}$</td>
<td>$-100000.0$</td>
</tr>
<tr>
<td>A universal real literal that lies between $\text{DURATION'}\text{BASE'}\text{FIRST}$ and $\text{DURATION'}\text{FIRST}$ or any value in the range of $\text{DURATION}$.</td>
<td></td>
</tr>
<tr>
<td>$\text{LESS_THAN_DURATION_BASE_FIRST}$</td>
<td>$-100000000.0$</td>
</tr>
<tr>
<td>A universal real literal that is less than $\text{DURATION'}\text{BASE'}\text{FIRST}$.</td>
<td></td>
</tr>
<tr>
<td>$\text{LOW_PRIORITY}$</td>
<td>$1$</td>
</tr>
<tr>
<td>An integer literal whose value is the lower bound of the range for the subtype $\text{SYSTEM_PRIORITY}$.</td>
<td></td>
</tr>
<tr>
<td>$\text{MANTISSA_DOC}$</td>
<td>$31$</td>
</tr>
<tr>
<td>An integer literal whose value is $\text{SYSTEM_MAX_MANTISSA}$.</td>
<td></td>
</tr>
<tr>
<td>$\text{MAX_DIGITS}$</td>
<td>$18$</td>
</tr>
<tr>
<td>Maximum digits supported for floating-point types.</td>
<td></td>
</tr>
<tr>
<td>$\text{MAX_IN_LEN}$</td>
<td>$255$</td>
</tr>
<tr>
<td>Maximum input line length permitted by the implementation.</td>
<td></td>
</tr>
<tr>
<td>$\text{MAX_INT}$</td>
<td>$2147483647$</td>
</tr>
<tr>
<td>A universal integer literal whose value is $\text{SYSTEM_MAX_INT}$.</td>
<td></td>
</tr>
<tr>
<td>$\text{MAX_INT_PLUS_1}$</td>
<td>$2147483648$</td>
</tr>
<tr>
<td>A universal integer literal whose value is $\text{SYSTEM_MAX_INT}+1$.</td>
<td></td>
</tr>
<tr>
<td>$\text{MAX_LEN_INT_BASE_LITERAL}$</td>
<td>($1..2=&gt;'2:','3..252=&gt;'0','253..255=&gt;'11:')</td>
</tr>
<tr>
<td>A universal integer based literal whose value is $2#11#$ with enough leading zeroes in the mantissa to be $\text{MAX_IN_LEN}$ long.</td>
<td></td>
</tr>
<tr>
<td>Name and Meaning</td>
<td>Value</td>
</tr>
<tr>
<td>------------------</td>
<td>-------</td>
</tr>
<tr>
<td>$\text{MAX_LEN_REAL_BASED_LITERAL}$</td>
<td>A universal real based literal whose value is $it:F.L:$ with enough leading zeroes in the mantissa to be MAX_IN_LEN long. (1..3=&gt;'16:', 4..251=&gt;'0', 252..255=&gt;'F.E:')</td>
</tr>
<tr>
<td>$\text{MAX_STRING_LITERAL}$</td>
<td>A string literal of size MAX_IN_LEN, including the quote characters. (1=&gt;'', 2..254=&gt;'A', 255=&gt;'''')</td>
</tr>
<tr>
<td>$\text{MIN_INT}$</td>
<td>A universal integer literal whose value is SYSTEM_MIN_INT. -2147483648</td>
</tr>
<tr>
<td>$\text{MIN_TASK_SIZE}$</td>
<td>An integer literal whose value is the number of bits required to hold a task object which has no entries, no declarations, and &quot;NULL;&quot; as the only statement in its body. 32</td>
</tr>
<tr>
<td>$\text{NAME}$</td>
<td>A name of a predefined numeric type other than FLOAT, INTEGER, SHORT_FLOAT, SHORT_INTEGER, LONG_FLOAT, or LONG_INTEGER. SHORT_SHORT_INTEGER</td>
</tr>
<tr>
<td>$\text{NAME_LIST}$</td>
<td>A list of enumeration literals in the type SYSTEM_NAME, separated by commas. IBM_370</td>
</tr>
<tr>
<td>$\text{NEG_BASED_INT}$</td>
<td>A based integer literal whose highest order nonzero bit falls in the sign bit position of the representation for SYSTEM_MAX_INT. 16#FFFFFFFF#</td>
</tr>
<tr>
<td>$\text{NEW_MEM_SIZE}$</td>
<td>An integer literal whose value is a permitted argument for pragma memory_size, other than $\text{DEFAULT_MEM_SIZE}$. If there is no other value, then use $\text{DEFAULT_MEM_SIZE}$. 2147483647</td>
</tr>
<tr>
<td>Name and Meaning</td>
<td>Value</td>
</tr>
<tr>
<td>----------------------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>$NEW_STOR_UNIT</td>
<td>8</td>
</tr>
<tr>
<td>An integer literal whose value is a permitted argument for pragma storage_unit, other than $DEFAULT_STOR_UNIT. If there is no other permitted value, then use value of SYSTEM.STORAGE_UNIT.</td>
<td></td>
</tr>
<tr>
<td>$NEW_SYS_NAME</td>
<td>IBM_370</td>
</tr>
<tr>
<td>A value of the type SYSTEM.NAME, other than $DEFAULT_SYS_NAME. If there is only one value of that type, then use that value.</td>
<td></td>
</tr>
<tr>
<td>$TASK_SIZE</td>
<td>32</td>
</tr>
<tr>
<td>An integer literal whose value is the number of bits required to hold a task object which has a single entry with one inout parameter.</td>
<td></td>
</tr>
<tr>
<td>$TICK</td>
<td>0.01</td>
</tr>
<tr>
<td>A real literal whose value is SYSTEM.TICK.</td>
<td></td>
</tr>
</tbody>
</table>
WITHDRAWN TESTS

Some tests are withdrawn from the ACVC because they do not conform to the Ada Standard. The following 36 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.

WITHDRAWN TEST LIST

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

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

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

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

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

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

- **CD2B15C & CD7205C**
  These tests expect that a 'STORAGE_SIZE length clause provides precise control over the number of designated objects in a collection; the Ada standard 13.2:15 allows that such control must not be expected.

- **CD5007B**
  This test wrongly expects an implicitly declared subprogram to be at the address that is specified for an unrelated subprogram (line 303).

- **CD7105A**
  This test requires that successive calls to CALENDAR.CLOCK change by at least SYSTEM.TICK; however, by Commentary AI-00201, it is only the expected frequency of change that must be at least SYSTEM.TICK--particular instances of change may be less (line 29).

- **CD72C3B & CD7204B**
  These tests use the 'SIZE length clause and attribute, whose interpretation is considered problematic by the WG9 ARG.

- **CD7205D**
  This test checks an invalid test objective: it treats the specification of storage to be reserved for a task's activation as though it were like the specification of storage for a collection.

- **CE2107I**
  This test requires that objects of two similar scalar types be distinguished when read from a file--DATA_ERROR is expected to be raised by an attempt to read one object as of the other type. However, it is not clear exactly how the Ada standard 14.2.4:4 is to be interpreted; thus, this test objective is not considered valid. (line 90)
This test requires certain behaviour, when two files are associated with the same external file, that is not required by the Ada standard.

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

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