2

AVF Control Number: AVF-VSR-178.0988
88-02-11-RWL

# Ada COMPILER
# VALIDATION SUMMARY REPORT:

Certificate Number. 880613W1.09090
Rockwell International Corporation
DDC-Based Ada/CAPS Compiler, Version 2.0
VAX 8650 to CAPS/AAMP

Completion of On-Site Testing:
15 June 1988

Prepared By:
Ada Validation Facility
ASD/SCEL
Wright-Patterson AFB OH 45433-6503

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

UNCLASSIFIED

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

| | |
|---|---|
| **REPORT DOCUMENTATION PAGE** | READ INSTRUCTIONS<br>BEFORE COMPLETEING FORM |
| 1. REPORT NUMBER \| 2. GOVT ACCESSION NO. | 3. RECIPIENT'S CATALOG NUMBER |
| 4. TITLE *(and Subtitle)*<br>Ada Compiler Validation Summary Report: Rockwell International Corporation, DDC-Based Ada/CAPS Compiler, Version 2.0, VAX 8650 (Host) to CAPS/AAMP (Target). (880613W1.09090) | 5. TYPE OF REPORT & PERIOD COVERED<br>15 June 1988 to 15 June 1989<br>6. PERFORMING ORG. REPORT NUMBER |
| 7. AUTHOR*(s)*<br>Wright-Patterson Air Force Base, Dayton, Ohio, U.S.A. | 8. CONTRACT OR GRANT NUMBER*(s)* |
| 9. PERFORMING ORGANIZATION AND ADDRESS<br>Wright-Patterson Air Force Base, Dayton, Ohio, U.S.A. | 10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS |
| 11. CONTROLLING OFFICE NAME AND ADDRESS<br>Ada Joint Program Office<br>United States Department of Defense<br>Washington, DC 20301-3081 | 12. REPORT DATE<br>15 June 1988<br>13. NUMBER OF PAGES<br>42 p. |
| 14. MONITORING AGENCY NAME & ADDRESS*(If different from Controlling Office)*<br>Wright-Patterson Air Force Base, Dayton, Ohio, U.S.A. | 15. SECURITY CLASS *(of this report)*<br>UNCLASSIFIED<br>15a. DECLASSIFICATION/DOWNGRADING SCHEDULE<br>N/A |

16. DISTRIBUTION STATEMENT *(of this Report)*

Approved for public release; distribution unlimited.

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

UNCLASSIFIED

18. SUPPLEMENTARY NOTES

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

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

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

DDC-Based Ada Compiler, Version 2.0, Rockwell International Corporation, Wright-Patterson Air Force Base, VAX 8650 under VMS, 4.7 (Host) to CAPS/AAMP (bare machine) (Target), ACVC 1.9.

UNCLASSIFIED

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

## Ada Compiler Validation Summary Report:

Compiler Name: DDC-Based Ada/CAPS Compiler, Version 2.0

Certificate Number: 880613W1.09090

| Host: | Target: |
|---|---|
| VAX 8650 under | CAPS/AAMP |
| VMS, 4.7 | bare machine |

Testing Completed 15 June 1988 Using ACVC 1.9

This report has been reviewed and is approved.

Steven P. Wilson

Ada Validation Facility
Steven P. Wilson
Technical Director
ASD/SCEL
Wright-Patterson AFB OH 45433-6503

John F Kramer

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

Virginia L. Castor

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

# TABLE OF CONTENTS

# CHAPTER 1

## INTRODUCTION

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

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

The information in this report is derived from the test results produced during validation testing. The validation process includes submitting a suite of standardized tests, the ACVC, as inputs to an Ada compiler and evaluating the results. The purpose of validating is to ensure conformity of the compiler to the Ada Standard by testing that the compiler properly implements legal language constructs and that it identifies and rejects illegal language constructs. The testing also identifies behavior 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 behavior is allowed by the Ada Standard

Testing of this compiler was conducted by SofTech, Inc. under the direction of the AVF according to procedures established by the Ada Joint Program Office and administered by the Ada Validation Organization (AVO). On-site testing was completed 15 June 1988 at Rockwell International, 400 Collins Road NE, Cedar Rapids, IA 52498.

## 1.2 USE OF THIS VALIDATION SUMMARY REPORT

Consistent with the national laws of the originating country, the AVO may make full and free public disclosure of this report. In the United States, this is provided in accordance with the "Freedom of Information Act" (5 U.S.C. #552). The results of this validation apply only to the computers, operating systems, and compiler versions identified in this report.

The organizations represented on the signature page of this report do not represent or warrant that all statements set forth in this report are accurate and complete, or that the subject compiler has no nonconformities to the Ada Standard other than those presented. Copies of this report are available to the public from:

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

or from:

Ada Validation Facility
ASD/SCEL
Wright-Patterson AFB OH 45433-6503

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

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

## 1.3 REFERENCES

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

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

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

4. Ada Compiler Validation Capability User's 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.

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

Applicant The agency requesting validation.

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

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 compiler is exceeded, the test is classified as inapplicable. If a Class D test compiles successfully, it is self-checking and produces a PASSED or FAILED message during execution.

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

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

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

# CONFIGURATION INFORMATION

## 2.1 CONFIGURATION TESTED

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

Compiler: DDC-Based Ada/CAPS Compiler, Version 2.0

ACVC Version: 1.9

Certificate Number: 880613W1.09090

Host Computer:

| | |
|---|---|
| Machine: | VAX 8650 |
| Operating System: | VMS, 4.7 |
| Memory Size: | 16 megabytes |

Target Computer:

| | |
|---|---|
| Machine: | CAPS/AAMP |
| Operating System: | bare machine |
| Memory Size: | 256K words |

Communications Network: Ethernet

## 2.2 IMPLEMENTATION CHARACTERISTICS

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

- Capacities.

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

- Universal integer calculations.

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

- Predefined types.

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

- Based literals.

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

- Expression evaluation.

  Apparently all default initialization expressions for record components are evaluated before any value is checked to belong to a component's subtype. (See test C32117A.)

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

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

Sometimes 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.)

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

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

. Rounding.

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

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

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

. Array types.

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

Declaration of an array type or subtype declaration with more than SYSTEM.MAX_INT components raises NUMERIC_ERROR when the array is two-dimensional with second dimension larger than the first, and raises CONSTRAINT_ERROR otherwise. (See test C36003A.)

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

CONSTRAINT_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 CONSTRAINT_ERROR when the array type is declared. (See test C52103X.)

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

A null array with one dimension of length greater than INTEGER'LAST may raise NUMERIC_ERROR or CONSTRAINT_ERROR either when declared or assigned. Alternatively, an implementation may accept the declaration. However, lengths must match in array slice assignments. This implementation raises CONSTRAINT_ERROR when the array type is declared (See test E52103Y.)

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

- Discriminated types.

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

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

- Aggregates.

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

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

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

- Representation clauses.

An implementation might legitimately place restrictions on representation clauses used by some cf the tests. If a representation clause is used by a test in a way that violates a restriction, then the implementation must reject it.

For this implementation:

- Enumeration representation clauses containing noncontiguous values for enumeration types other than character and boolean types are not supported. (See tests C35502I..J, C35502M..N, and A39005F.)

- Enumeration representation clauses containing noncontiguous values for character types are not supported. (See tests C35507I..J, C35507M..N, and C55B16A.)

- Enumeration representation clauses for boolean types containing representational values other than (FALSE => 0, TRUE => 1) are not supported. (See tests C35508I..J and C35508M..N.)

- Length clauses with SIZE specifications for enumeration types are not supported. (See test A39005B.)

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

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

- Length clauses with SMALL specifications are not supported. (See tests A39005E and C87B62C.)

- Record representation alignment clauses are not supported. (See test A39005G.)

- Length clauses with SIZE specifications for derived integer types are not supported. (See test C87B62A.)

- Pragmas.

  The pragma INLINE is supported for procedures and functions. (See tests LA3004A, LA3004B, EA3004C, EA3004D, CA3004E, and CA3004F.)

- Input/output.

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

  The package DIRECT_IO can be instantiated with unconstrained array types and record types with discriminants without defaults. (See tests AE2101H, EE2401D, and EE2401G.)

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

- Generics.

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

Generic subprogram declarations and bodies as subunits cannot be compiled in separate compilations. (See tests CA2009F.)

Generic package declarations and bodies cannot be compiled in separate compilations. (See tests CA2009C, BC3204C, and BC3205D.)

Subunits of generic unit bodies can be compiled in separate compilations. (See test CA3011A.)

# CHAPTER 3

# TEST INFORMATION

## 3.1 TEST RESULTS

Version 1.9 of the ACVC comprises 3122 tests. When this compiler was tested, 27 tests had been withdrawn because of test errors. The AVF determined that 516 tests were inapplicable to this implementation. All inapplicable tests were processed during validation testing except for 285 executable tests that use floating-point precision exceeding that supported by the implementation and 174 executable tests that use file operations not supported by the implementation. Modifications to the code, processing, or grading for 9 tests were required to successfully demonstrate the test objective. (See section 3.6.)

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

## 3.2 SUMMARY OF TEST RESULTS BY CLASS

| RESULT | TEST CLASS | | | | | | TOTAL |
|---|---|---|---|---|---|---|---|
| | A | B | C | D | E | L | |
| Passed | 105 | 1046 | 1352 | 17 | 13 | 46 | 2579 |
| Inapplicable | 5 | 5 | 501 | 0 | 5 | 0 | 516 |
| Withdrawn | 3 | 2 | 21 | 0 | 1 | 0 | 27 |
| TOTAL | 113 | 1053 | 1874 | 17 | 19 | 46 | 3122 |

## 3.3 SUMMARY OF TEST RESULTS BY CHAPTER

| RESULT | CHAPTER | | | | | | | | | | | | | TOTAL |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | |
| Passed | 184 | 454 | 490 | 244 | 166 | 98 | 140 | 326 | 135 | 36 | 232 | 3 | 71 | 2579 |
| Inapplicable | 20 | 118 | 184 | 4 | 0 | 0 | 3 | 1 | 2 | 0 | 2 | 0 | 182 | 516 |
| Withdrawn | 2 | 14 | 3 | 0 | 0 | 1 | 2 | 0 | 0 | 0 | 2 | 1 | 2 | 27 |
| TOTAL | 206 | 586 | 677 | 248 | 166 | 99 | 145 | 327 | 137 | 36 | 236 | 4 | 255 | 3122 |

## 3.4 WITHDRAWN TESTS

The following 27 tests were withdrawn from ACVC Version 1.9 at the time of this validation:

```
A35902C    A74106C    AD1A01A    B28003A    BC3105A
C34004A    C35502P    C35904A    C35904B    C35A03E
C35A03R    C37213H    C37213J    C37215C    C37215E
C37215G    C37215H    C38102C    C41402A    C45332A
C45614C    C85018B    C87B04B    CC1311B    CE2401H
CE3208A    E28005C
```

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, 516 tests were inapplicable for the reasons indicated:

- C35502I..J (2 tests), C35502M..N (2 tests), C35507I..J (2 tests), C35507M..N (2 tests), C35508I..J (2 tests), C35508M..N (2 tests), A39005F, and C55B16A use enumeration representation clauses which are not supported by this compiler.

- C35702A uses SHORT_FLOAT which is not supported by this implementation.

- A39005B and C87B62A use length clauses with SIZE specifications for derived integer types or for enumeration types which are not supported by this compiler.

- A39005E and C87B62C use length clauses with SMALL specifications which are not supported by this implementation.

- A39005G uses a record representation clause which is not supported by this compiler.

- The following tests use SHORT_INTEGER, which is not supported by this compiler:

  C45231B  C45304B  C45502B  C45503B  C45504B
  C45504E  C45611B  C45613B  C45614B  C45631B
  C45632B  B52004E  C55B07B  B55B09D

- C45231D requires a macro substitution for any predefined numeric types other than INTEGER, SHORT_INTEGER, LONG_INTEGER, FLOAT, SHORT_FLOAT, and LONG_FLOAT. This compiler does not support any such types.

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

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

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

- C96005B requires the range of type DURATION to be different from that of its base type; in this implementation they are the same.

- CA2009F compiles generic subprogram declarations and bodies in separate compilations. This compiler requires that generic subprogram declarations and bodies be in a single compilation.

- CA2009C, BC3204C, and BC3205D compile generic package specifications and bodies in separate compilations. This compiler requires that generic package specifications and bodies be in a single compilation.

- The following 182 tests are inapplicable because sequential, text, and direct access files are not supported:

| | | | |
|---|---|---|---|
| CE2102C | CE2102G..H(2) | CE2102K | CE2104A..D(4) |
| CE2105A..B(2) | CE2106A..B(2) | CE2107A..I(9) | CE2108A..D(4) |
| CE2109A..C(3) | CE2110A..C(3) | CE2111A..E(5) | CE2111G..H(2) |
| CE2115A..B(2) | CE2201A..C(3) | EE2201D | EE2201E |
| CE2201F..G(2) | CE2204A..B(2) | CE2208B | CE2210A |
| CE2401A..C(3) | EE2401D | CE2401E..F(2) | EE2401G |
| CE2404A | CE2405B | CE2406A | CE2407A |
| CE2408A | CE2409A | CE2410A | CE2411A |
| AE3101A | CE3102B | EE3102C | CE3103A |
| CE3104A | CE3107A | CE3108A..B(2) | CE3109A |
| CE3110A | CE3111A..E(5) | CE3112A..B(2) | CE3114A..B(2) |
| CE3115A | CE3201A | CE3202A | CE3203A |
| CE3301A..C(3) | CE3302A | CE3305A | CE3402A..D(4) |
| CE3403A..C(3) | CE3403E..F(2) | CE3404A..C(3) | CE3405A..D(4) |
| CE3406A..D(4) | CE3407A..C(3) | CE3408A..C(3) | CE3409A |
| CE3409C..F(4) | CE3410A | CE3410C..F(4) | CE3411A |
| CE3411C | CE3412A | CE3412C | CE3413A |
| CE3413C | CE3602A..D(4) | CE3603A | CE3604A |
| CE3605A..E(5) | CE3606A..B(2) | CE3704A..B(2) | CE3704D..F(3) |
| CE3704M..O(3) | CE3706D | CE3706F | CE3804A..E(5) |
| CE3804G | CE3804I | CE3804K | CE3804M |
| CE3805A..B(2) | CE3806A | CE3806D..E(2) | CE3905A..C(3) |
| CE3905L | CE3906A..C(3) | CE3906E..F(2) | |

- The following 285 tests require a floating-point accuracy that exceeds the maximum of 9 digits supported by this implementation:

| | |
|---|---|
| C24113F..Y (20 tests) | C35705F..Y (20 tests) |
| C35706F..Y (20 tests) | C35707F..Y (20 tests) |
| C35708F..Y (20 tests) | C35802F..Z (21 tests) |
| C45241F..Y (20 tests) | C45321F..Y (20 tests) |
| C45421F..Y (20 tests) | C45521F..Z (21 tests) |
| C45524F..Z (21 tests) | C45621F..Z (21 tests) |
| C45641F..Y (20 tests) | C46012F..Z (21 tests) |

## 3.6 TEST, PROCESSING, AND EVALUATION MODIFICATIONS

It is expected that some tests will require modifications of code, processing, or evaluation in order to compensate for legitimate implementation 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 9 Class B tests.

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

B33301A B55A01A B67001A B67001C B67001D
B97102A BC1109A BC1109C BC1109D

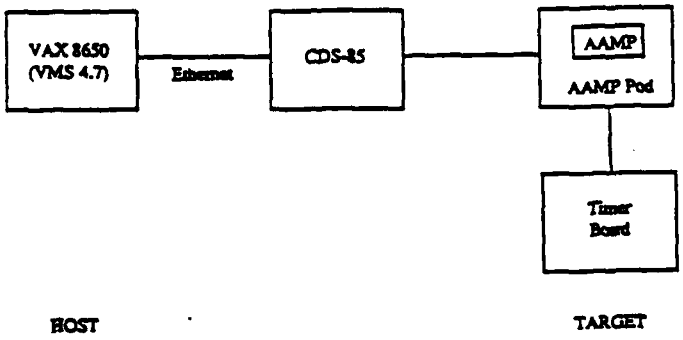
## 3.7 ADDITIONAL TESTING INFORMATION

### 3.7.1 Prevalidation

Prior to validation, a set of test results for ACVC Version 1.9 produced by the DDC-Based Ada/CAPS Compiler, Version 2.0, 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 DDC-Based Ada/CAPS Compiler, Version 2.0, using ACVC Version 1.9 was conducted on-site by a validation team from the AVF. The configuration consisted of a VAX 8650 host operating under VMS, Version 4.7, and an Advanced Architecture Microprocessor (AAMP) bare machine target (see figure below). A CDS-85 Computer Development Station was used to facilitate running the executable tests. An executable image for each test was transferred from the VAX to the CDS-85 using Ethernet. An Ada Symbolic Debugger, Version 3.6, was used to load each image into CDS-85 memory from which the program was executed by the AAMP. The Timer Board provided the real-time clock used by the AAMP. Test output was captured by the CDS-85.



A magnetic tape containing all tests except for withdrawn tests, tests requiring unsupported floating-point precisions, and executable tests that use unsupported file operations 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 included in their modified form on the magnetic tape.

The body of package REPORT was modified to use a package SIMPLE_IO instead of TEXT_IO because package TEXT_IO is implemented in such a way that an exception is raised for all file operations. A set of executable tests was run to verify that the modified body of package REPORT operated correctly.

The contents of the magnetic tape were loaded directly onto the host computer. After the test files were loaded to disk, the full set of tests was compiled on the VAX 8650, and all executable tests were run on the CAPS/AAMP. Object files were linked on the host computer, and executable images were transferred to the target computer. The transferred executable images did not include those portions of the run-time system that are identical for every test. The run-time system was loaded once for each chapter and used by each test. This had the effect of significantly reducing the time needed for downloading the tests. Results were printed from the host computer, with results being transferred to the host computer via Ethernet.

The compiler was tested using command scripts provided by Rockwell International Corporation and reviewed by the validation team. The compiler was tested using all default switch settings except for the following:

| Switch | Effect |
|---|---|
| /LIST | Generate a source listing |
| /NODEBUG | Suppress debugger information |
| /NOOBJECT | Suppress object code generation (for Class B and L tests) |

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

### 3.7.3 Test Site

Testing was conducted at Rockwell International, 400 Collins Road NE, Cedar Rapids, IA 52498 and was completed on 15 June 1988.

# APPENDIX A

## DECLARATION OF CONFORMANCE

Rockwell International Corporation has submitted the following Declaration of Conformance concerning the DDC-Based Ada/CAPS Compiler, Version 2.0.

## DECLARATION OF CONFORMANCE

Compiler Implementor: Rockwell International
Ada Validation Facility: ASD/SCEL, Wright-Patterson AFB, OH 45433-6503
Ada Compiler Validation Capability (ACVC) Version: 1.9

### Base Configuration

| | | | |
|---|---|---|---|
| Base Compiler Name: | DDC-Based Ada/CAPS Compiler | Version: | 2.0 |
| Host Architecture ISA: | VAX 8650 | OS&VER #: | VMS 4.7 |
| Target Architecture ISA: | CAPS/AAMP | OS&VER #: | bare machine |

### Implementor's Declaration

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

Date: 6/15/88

Rockwell International Corporation
C. E. Kress, Manager of Processor Technology Department

### Owner's Declaration

I, the undersigned, representing Rockwell International Corporation, take full responsibility for implementation and maintenance of the Ada compiler 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.

CE Kress Date: 6/15/88

Rockwell International Corporation
C. E. Kress, Manager of Processor Technology Department

# APPENDIX B

## APPENDIX F OF THE Ada STANDARD

The only allowed implementation dependencies correspond to implementation-dependent pragmas, to certain machine-dependent conventions as mentioned in chapter 13 of the Ada Standard, and to certain allowed restrictions on representation clauses. The implementation-dependent characteristics of the DDC-Based Ada/CAPS Compiler, 2.0, are described in the following sections, taken from Appendix F of the Ada Standard. Implementation-specific portions of the package STANDARD are also included in this appendix.

```
package STANDARD is

     ...

     type INTEGER is range -32_768 .. 32_767;
     type LONG_INTEGER is range -2_147_483_646 .. 2_147_483_647;

     type FLOAT is digits 6 range -16#0.7FFF_FF8#E+32 .. 16#0.7FFF_FF8#E+32;
     type LONG_FLOAT is digits 9
          range -16#0.7FFF_FFFF_FF8#E+32 .. 16#0.7FFF_FFFF_FF8#E+32;

     type DURATION is delta 0.0001 range -131_072.0000 .. 131_071.999938965;


     ...

end STANDARD;
```

# IMPLEMENTATION-DEPENDENT CHARACTERISTICS

This appendix describes the implementation-dependent characteristics of the DDC-Based Ada/CAPS Compiler.

## F.1 Implementation-Dependent Pragmas.

### F.1.1 Pragma EXPORT.

Takes an identifier denoting a subprogram or an object, and optionally takes a string literal (the name of a CAPS object module entry/external name) as arguments. This pragma is only allowed at the place of a declarative item and must apply to a subprogram or object declared by an earlier declarative item in the same declarative part or package specification. The pragma must occur in the same compilation unit as the subprogram body to export a subprogram, and in the same compilation unit as the declaration to export an object. The subprogram to be exported may not be nested within anything but a library_unit package specification or body. The pragma is not allowed for an access or a task object. The object exported must be a static object. Generally, objects declared in a package specification or body are static; objects declared local to a subprogram are not.

This pragma allows the export of a procedure, function, or object to a non-Ada environment.

```
pragma EXPORT(internal_name [, external_name]);

internal_name ::= identifier

external_name ::= string_literal
```

If external_name is not specified, the internal_name is used as the external_name. If a string_literal is given, it is used. External_name must be an identifier that is acceptable to the CAPS linker, though it does not have to be a valid Ada identifier.

Exporting Subprograms:

In this case the pragma specifies that the body of the specified subprogram associated with an Ada subprogram specification may be called from another CAPS language (e.g., Jovial, or assembly).

Subprograms must be uniquely identified by their internal names. An exported subprogram must be a library unit or be declared in the outermost declarative part of a library package (specification or body). An EXPORT pragma is allowed for a subprogram which is a compilation unit only after the subprogram body in the compilation unit. It is allowed for a subprogram in a package body after the body of the subprogram. Pragma EXPORT is not allowed in a package specification.

Example:

```
  procedure AUTO_PILOT (MODE: in INTEGER) is
     ...
  end AUTO_PILOT;
  pragma EXPORT (AUTO_PILOT);
```

Exporting Objects:

In this case the pragma specifies that an Ada object is to be accessible by an external routine in an another CAPS language.

Objects must be uniquely identified by their internal names. An exported object must be a variable declared in the outermost declarative part of a library package (specification or body).

The object must be allocated to static storage. To guarantee this, the subtype indication for the object must denote one of the following:

- A scalar type or subtype.
- An array subtype with static index constraints whose component size is static.
- A simple record type or subtype.

Example:

```
  SYSTEM_STATUS: INTEGER;
  pragma EXPORT (SYSTEM_STATUS, "SYS$STS");  -- SYS$STS is a Jovial
                                             --   identifier.
```

## F.1.2 Pragma IMPORT.

Takes an internal name denoting a subprogram, and optionally takes an external name (the name of a CAPS object module entry/external name) as arguments. This pragma is only allowed at the place of a declarative item and must apply to a subprogram declared by an earlier declaration item in the same declarative part or package specification.

This pragma allows the import of a procedure or function from a non-Ada environment.

```
pragma IMPORT(internal_name [, external_name]);

internal_name ::= identifier | string_literal

external_name ::= identifier | string_literal
```

Internal_name may only be a string_literal when designating an operator function for import. If external_name is not specified, the internal_name is used as the external_name. If an identifier or string_literal is given, it is used. External_name must name an identifier that is acceptable to the Ada linker though it does not have to be a valid Ada identifier.

Importing Subprograms:

In this case the pragma specifies that the body of the specified subprogram associated with an Ada subprogram specification is to be provided by another CAPS language. The pragma INTERFACE must also be given for the internal_name earlier for the same declarative part or package specification. The use of the pragma INTERFACE implies that a corresponding body is not given.

Subprograms must be uniquely identified by their internal names. An imported subprogram must be a library unit or be declared in the outermost declarative part of a library package (specification or body). An import pragma is allowed only if either the body does not have a corresponding specification, or the specification and body occur in the same declarative part.

If a subprogram has been declared as a compilation unit, the pragma is only allowed after the subprogram declaration and before any subsequent compilation unit. This pragma may not be used for a subprogram that is declared by a generic instantiation of a predefined subprogram.

Example:

```
  function SIN (X: in FLOAT) return FLOAT;
  pragma INTERFACE (ASSEMBLY, SIN);
  pragma IMPORT (SIN, "SIN$$");
```

### F.1.3 Pragma STACK_SIZE.

This pragma has two arguments, a task type name and an integer expression. This pragma is allowed anywhere that a task storage specification is allowed. The effect of this pragma is to use the value of the expression as the number of storage units (words) to be allocated to the process stack of tasks of the associated task type.

Example:

```
  task type DISPLAY_UNIT is

    entry UPPER_DISPLAY;
    entry BOTTOM_LINE;

  end DISPLAY_UNIT;

  for DISPLAY_UNIT'STORAGE_SIZE use 20_000;
  pragma STACK_SIZE (DISPLAY_UNIT, 1000);
```

## F.2 Implementation-Dependent Attributes.

No implementation-dependent attributes are supported.

## F.3 Specification Of The Package SYSTEM.

```
package SYSTEM is

   type ADDRESS        is range 0..16#FF_FFFF#     -- 24 bit address
   subtype PRIORITY    is INTEGER range 1 .. 254;
   type NAME           is (VAX11,   AAMP,   CAPS6,   CAPS7,
                              CAPS8, CAPS10,  ACAPS);
   SYSTEM_NAME:        constant NAME    := AAMP;
   STORAGE_UNIT:       constant         := 16;
   MEMORY_SIZE:        constant         := 16_384 * 1024;
   MIN_INT:            constant         := -2_147_483_647-1;
   MAX_INT:            constant         := 2_147_483_647;
   MAX_DIGITS:         constant         := 9;
   MAX_MANTISSA:       constant         := 31;
   FINE_DELTA:         constant         := 2#1.0#E-30;
   TICK:               constant         := 0.000_1;

end SYSTEM;
```

## F.4 Representation Clause Restrictions.

### F.4.1 Representation Clauses.

In general, no representation clauses may given for a derived type. The representation clauses that are allowed for non-derived types are described in the following sections.

### F.4.2 Length Clauses.

The compiler accepts only length clauses that specify the number of storage units reserved for a collection or for a task data stack size via the 'STORAGE_SIZE clause. (See pragma STACK_SIZE for a complementary capability.) The 'SIZE clause has no effect for tasks.

### F.4.3 Enumeration Representation Clauses.

Enumeration representation clauses are not supported.

## F.5 Implementation-Generated Names.

Implementation-generated names for implementation-dependent components are not supported.

## F.6 Address Clause Expressions.

All address values are interpreted as the 24-bit address of a 16 bit word of memory, even for code addresses which are normally thought of as 8 bit byte addresses. All subprogram and task entry addresses are word aligned by the compiler.

## F.7 Unchecked Conversion Restrictions.

Unchecked conversion is only allowed between objects of the same size.

## F.8 I/O Package Implementation-Dependent Characteristics.

The target environment does not support a file system; therefore I/O procedure or function calls involving files will raise the predefined exception USE_ERROR or STATUS_ERROR.

### F.8.1 Package SEQUENTIAL_IO.

All procedures and functions raise STATUS_ERROR, except for CREATE and OPEN, which raise USE_ERROR, and IS_OPEN which always returns FALSE.

### F.8.2 Package DIRECT_IO.

All procedures and functions raise STATUS_ERROR, except for CREATE and OPEN, which raise USE_ERROR, and IS_OPEN which always returns FALSE.

### F.8.3 Package TEXT_IO.

All procedures and functions with a file parameter raise STATUS_ERROR, except for CREATE and OPEN which raise USE_ERROR, and IS_OPEN which always returns FALSE. All procedures and functions without a file parameter which operate on the current default input file raise STATUS_ERROR. All procedures and functions without a file parameter which operate on the current default output file raise STATUS_ERROR, except PUT, PUT_LINE, and NEW_LINE which communicate with external hardware/software (usually the terminal) via a memory-mapped I/O interface.

### F.8.4 Package LOW_LEVEL_IO.

Package LOW_LEVEL_IO is not provided.

## F.9 Other Implementation-Dependent Features.

### F.9.1 Predefined Types.

This section describes the implementation-dependent predefined types declared in the predefined package STANDARD, and the relevant attributes of these types.

#### F.9.1.1 Integer Types.

Two predefined integer types are implemented, INTEGER, and LONG_INTEGER. They have the following attributes:

| | | |
|---|---|---|
| INTEGER'FIRST | = | -32768 |
| INTEGER'LAST | = | 32767 |
| INTEGER'SIZE | = | 16 |
| LONG_INTEGER'FIRST | = | -2_147_483_648 |
| LONG_INTEGER'LAST | = | 2_147_483_647 |
| LONG_INTEGER'SIZE | = | 32 |

#### F.9.1.2 Floating Point Types.

Two predefined floating point types are implemented, FLOAT and LONG_FLOAT. They have the following attributes:

| | | |
|---|---|---|
| FLOAT'DIGITS | = | 6 |
| FLOAT'EMAX | = | 84 |
| FLOAT'EPSILON | = | 16#0.1000_000#E-04 |
| | ~= | 9.53674E-07 |
| FLOAT'FIRST | = | -16#0.7FFF_FF8#E+32 |
| | ~= | -1.70141E+38 |
| FLOAT'LARGE | = | 16#0.FFFF_F80#E+21 |
| | ~= | 1.93428E+25 |
| FLOAT'LAST | = | 16#0.7FFF_FF8#E+32 |
| | ~= | 1.70141E+38 |
| FLOAT'MACHINE_EMAX | = | 127 |
| FLOAT'MACHINE_EMIN | = | -127 |
| FLOAT'MACHINE_MANTISSA | = | 24 |
| FLOAT'MACHINE_OVERFLOWS | = | TRUE |
| FLOAT'MACHINE_RADIX | = | 2 |
| FLOAT'MACHINE_ROUNDS | = | TRUE |

| | | |
|---|---|---|
| FLOAT'MANTISSA | = | 21 |
| FLOAT'SAFE_EMAX | = | 127 |
| FLOAT'SAFE_LARGE | = | 16#0.7FFF_FC#E+32 |
| | ~= | 1.70141E+38 |
| FLOAT'SAFE_SMALL | = | 16#0.1000_000#E-31 |
| | ~= | 2.93874E-39 |
| FLOAT'SIZE | = | 32 |
| FLOAT'SMALL | = | 16#0.8000_000#E-21 |
| | ~= | 2.58494E-26 |

| | | |
|---|---|---|
| LONG_FLOAT'DIGITS | = | 9 |
| LONG_FLOAT'EMAX | = | 124 |
| LONG_FLOAT'EPSILON | = | 16#0.4000_0000_000#E-7 |
| | ~= | 9.31322575E-10 |
| LONG_FLOAT'FIRST | = | -16#0.7FFF_FFFF_FF8#E+32 |
| | ~= | -1.70141183E+38 |
| LONG_FLOAT'LARGE | = | 16#0.FFFF_FFFE_000#E+31 |
| | ~= | 2.12676479E+37 |
| LONG_FLOAT'LAST | = | 16#0.7FFF_FFFF_FF8#E+32 |
| | ~= | 1.70141183E+38 |
| LONG_FLOAT'MACHINE_EMAX | = | 127 |
| LONG_FLOAT'MACHINE_EMIN | = | -127 |
| LONG_FLOAT'MACHINE_MANTISSA | = | 40 |
| LONG_FLOAT'MACHINE_OVERFLOWS | = | TRUE |
| LONG_FLOAT'MACHINE_RADIX | = | 2 |
| LONG_FLOAT'MACHINE_ROUNDS | = | TRUE |
| LONG_FLOAT'MANTISSA | = | 31 |
| LONG_FLOAT'SAFE_EMAX | = | 127 |
| LONG_FLOAT'SAFE_LARGE | = | 16#0.7FFF_FFFF#E+32 |
| | ~= | 1.70141183E+38 |
| LONG_FLOAT'SAFE_SMALL | = | 16#0.1000_0000_000#E-31 |
| | ~= | 2.93873588E-39 |
| LONG_FLOAT'SIZE | = | 48 |
| LONG_FLOAT'SMALL | = | 16#0.8000_0000_000#E-31 |
| | ~= | 2.35098870E-38 |

### F.9.1.3 Fixed Point Types.

To implement fixed point numbers, Ada/CAPS uses two sets of anonymous, predefined, fixed point types, here named FIXED and LONG_FIXED. These names are not defined in package STANDARD, but are only used here for reference.

These types are of the following form:

```
    type FIXED_TYPE is delta SMALL range -M*SMALL .. (M-1)*SMALL;

    where SMALL = 2**n for -128 <= n <= 127,
    and M = 2**15 for FIXED, or M = 2**31 for LONG__FIXED.
```

For each of FIXED and LONG_FIXED there exists a virtual predefined type for each possible value of SMALL (cf. RM 3.5.9). SMALL may be any power of 2 which is representable by a LONG_FLOAT value. FIXED types are represented by 16 bits, while 32 bits are used for LONG_FIXED types.

A user-defined fixed point type is represented as that predefined FIXED or LONG_FIXED type which has the largest value of SMALL not greater than the user-specified DELTA, and which has the smallest range that includes the user-specified range.

As the value of SMALL increases, the range increases. In other words, the greater the allowable error (the value of SMALL), the larger the allowable range.

Example 1:

For a FIXED type, to get the smallest amount of error possible requires SMALL = 2**-128, but the range is constrained to:

```
    -(2**15)*(2**-128) .. ((2**15)-1)*(2**-128), which is
    -2**-113 .. ((2**-113) - (2**-128)).
```

Example 2:

For a FIXED type, to get the largest range possible requires SMALL = 2**127, i.e., the error may be as large as 2**127. The range is then:

```
    -(2**15)*(2**127) .. ((2**15)-1)*(2**127), which is
    -2**142 .. ((2**142) - (2**127)).
```

For any FIXED or LONG_FIXED type T:

| | |
|---|---|
| T'MACHINE_OVERFLOWS | = TRUE |
| T'MACHINE_ROUNDS | = FALSE |

### F.9.1.4 The Type DURATION.

The predefined fixed point type DURATION has the following attributes:

| | |
|---|---|
| DURATION'AFT | = 4 |
| DURATION'DELTA | = 0.0001 |
| DURATION'FIRST | = -131_072.0000 |
| DURATION'FORE | = 7 |
| DURATION'LARGE | = 131_071.999938965 |
| | = 2#1.0#E+17 - 2#1.0F-14 |
| DURATION'LAST | = DURATION'LARGE |
| DURATION'MANTISSA | = 31 |
| DURATION'SAFE_LARGE | = DURATION'LARGE |
| DURATION'SAFE_SMALL | = DURATION'SMALL |
| DURATION'SIZE | = 32 |
| DURATION'SMALL | = 6.103515625E-5 |

```
                              = 2#1.0#E-14
```

### F.9.2 Uninitialized Variables.

There is no check on the use of uninitialized variables. The effect of a program that uses the value of such a variable is undefined.

### F.9.3 Package MACHINE_CODE.

Machine code insertions (cf. RM 13.8) are supported by the Ada/CAPS compiler via the use of the predefined package MACHINE_CODE.

```
package MACHINE_CODE is

   type CODE is record
      INSTR: STRING (1 .. 80);
   end record;

end MACHINE_CODE;
```

Machine code insertions may be used only in a procedure body, and only if the body contains nothing but code statements, as in the following example:

```
with MACHINE_CODE;  -- Must apply to the compilation unit
                    -- containing DOUBLE.

procedure DOUBLE (VALUE: in INTEGER; DOUBLE_VALUE: out INTEGER);

procedure DOUBLE (VALUE: in INTEGER; DOUBLE_VALUE: out INTEGER) is

begin

  MACHINE_CODE.CODE' (INSTR => "REFSL 1");   -- Get VALUE.
  MACHINE_CODE.CODE' (INSTR => "DUP");       -- Make copy of VALUE.
  MACHINE_CODE.CODE' (INSTR => "LOCX");      -- Add copies together.
  MACHINE_CODE.CODE' (INSTR => "ASNSL 0");   -- Store result in
                                             --   DOUBLE_VALUE.
end DOUBLE;
```

The string value assigned to INSTR may be a CAPS assembly language instruction or macro. The file ADAMACS.MAC, located in a runtime subdirectory of the compiler system, defines the macros which are available to use. The macros may change with different releases and should be used cautiously as there is no guarantee that they will perform the same across all releases. The CAPS Macro Assembler User's Guide contains information on how to call macros and write assembly instructions.

## F.9.4 Compiler Limitations.

The following limitations apply to Ada programs in the DDC-Based Ada/CAPS Compiler System:

- A program (sum of all compilation units) may not contain more than 64K words of static data and stacks. Static data is allocated for variables declared in the specification or body of a package. A stack is allocated for each task including the main program. Some of the 64K maximum is used by the runtime system. Static data requirements exceeding the 64K word maximum may be permanently allocated to the heap at the cost of an additional indirect memory access.

- A compilation unit may not contain more than 64K bytes (32K words) of code.

- A compilation unit may not contain more than 32K words of data.

- A compilation unit may not contain more than 32K words of constants.

- It follows that any single object may be no larger than 32K words.

- No more than 500 subprograms may be defined in a single compilation unit, including any implicitly allocated by the compiler.

- The maximum nesting level for blocks is 100.

# APPENDIX C

## TEST PARAMETERS

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

| Name and Meaning | Value |
|---|---|
| $BIG_ID1<br>Identifier the size of the maximum input line length with varying last character. | (1..125 =>'A', 126 =>'1') |
| $BIG_ID2<br>Identifier the size of the maximum input line length with varying last character. | (1..125 =>'A', 126 =>'2') |
| $BIG_ID3<br>Identifier the size of the maximum input line length with varying middle character. | (1..62 \| 64..126 =>'A', 63 =>'3') |
| $BIG_ID4<br>Identifier the size of the maximum input line length with varying middle character. | (1..62 \| 64..126 =>'A', 63 =>'4') |
| $BIG_INT_LIT<br>An integer litera' of value 298 with enough leading zeroes so that it is the size of the maximum line length. | (1..123 =>'0', 124..126 =>'298') |

TEST PARAMETERS

| Name and Meaning | Value |
|---|---|
| $BIG_REAL_LIT<br>A universal real literal of value 690.0 with enough leading zeroes to be the size of the maximum line length. | (1..120 =>'0', 121..126 =>'69.0E1') |
| $BIG_STRING1<br>A string literal which when catenated with BIG_STRING2 yields the image of BIG_ID1. | (1..66 =>'A') |
| $BIG_STRING2<br>A string literal which when catenated to the end of BIG_STRING1 yields the image of BIG_ID1. | (1..59 =>'A', 60 +>'1') |
| $BLANKS<br>A sequence of blanks twenty characters less than the size of the maximum line length. | (1..106 =>' ') |
| $COUNT_LAST<br>A universal integer literal whose value is TEXT_IO.COUNT'LAST. | 2_147_483_647 |
| $FIELD_LAST<br>A universal integer literal whose value is TEXT_IO.FIELD'LAST. | 35 |
| $FILE_NAME_WITH_BAD_CHARS<br>An external file name that either contains invalid characters or is too long. | x}]!@#^&~Y |
| $FILE_NAME_WITH_WILD_CARD_CHAR<br>An external file name that either contains a wild card character or is too long. | XYZ* |
| $GREATER_THAN_DURATION<br>A universal real literal that lies between DURATION'BASE'LAST and DURATION'LAST or any value in the range of DURATION. | 76_536.0 |

| Name and Meaning | Value |
|---|---|
| $GREATER_THAN_DURATION_BASE_LAST<br>A universal real literal that is greater than DURATION'BASE'LAST. | 10_000_000.0 |
| $ILLEGAL_EXTERNAL_FILE_NAME1<br>An external file name which contains invalid characters. | BAD-CHARACTER*^ |
| $ILLEGAL_EXTERNAL_FILE_NAME2<br>An external file name which is too long. | MUCH-TOO-LONG-NAME-FOR-A-FILE |
| $INTEGER_FIRST<br>A universal integer literal whose value is INTEGER'FIRST. | -32_768 |
| $INTEGER_LAST<br>A universal integer literal whose value is INTEGER'LAST. | 32_767 |
| $INTEGER_LAST_PLUS_1<br>A universal integer literal whose value is INTEGER'LAST + 1. | 32_768 |
| $LESS_THAN_DURATION<br>A universal real literal that lies between DURATION'BASE'FIRST and DURATION'FIRST or any value in the range of DURATION. | -76_536.0 |
| $LESS_THAN_DURATION_BASE_FIRST<br>A universal real literal that is less than DURATION'BASE'FIRST. | -10_000_000.0 |
| $MAX_DIGITS<br>Maximum digits supported for floating-point types. | 9 |
| $MAX_IN_LEN<br>Maximum input line length permitted by the implementation. | 126 |
| $MAX_INT<br>A universal integer literal whose value is SYSTEM.MAX_INT. | 2_147_483_647 |
| $MAX_INT_PLUS_1<br>A universal integer literal whose value is SYSTEM.MAX_INT+1. | 2_147_483_648 |

| Name and Meaning | Value |
|---|---|
| $MAX_LEN_INT_BASED_LITERAL<br>A universal integer based literal whose value is 2#11# with enough leading zeroes in the mantissa to be MAX_IN_LEN long. | (1..2 =>"2:", 3..123 =>'0',<br>124..126 =>"11:") |
| $MAX_LEN_REAL_BASED_LITERAL<br>A universal real based literal whose value is 16:F.E: with enough leading zeroes in the mantissa to be MAX_IN_LEN long. | (1..3 =>"16:", 4..122 =>'0',<br>123..125 =>"F.E") |
| $MAX_STRING_LITERAL<br>A string literal of size MAX_IN_LEN, including the quote characters. | (1..124 =>'A') |
| $MIN_INT<br>A universal integer literal whose value is SYSTEM.MIN_INT. | -2_147_483_648 |
| $NAME<br>A name of a predefined numeric type other than FLOAT, INTEGER, SHORT_FLOAT, SHORT_INTEGER, LONG_FLOAT, or LONG_INTEGER. | [No such type is supported] |
| $NEG_BASED_INT<br>A based integer literal whose highest order nonzero bit falls in the sign bit position of the representation for SYSTEM.MAX_INT. | 16#FFFFFFFE# |

# APPENDIX D

## WITHDRAWN TESTS

Some tests are withdrawn from the ACVC because they do not conform to the Ada Standard. The following 27 tests had been withdrawn at the time of validation testing for the reasons indicated. A reference of the form "AI-ddddd" is to an Ada Commentary.

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

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

- C34004A: The expression in line 168 yields a value outside the range of the target type T, but there is no handler for CONSTRAINT_ERROR.

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

- A35902C: The assignment in line 17 of the nominal upper bound of a fixed-point type to an object raises CONSTRAINT_ERROR, for that value lies outside of the actual range of the type.

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

- C35904B: The subtype declaration that is expected to raise CONSTRAINT_ERROR when its compatibility is checked against that of various types passed as actual generic parameters, may, in fact, raise NUMERIC_ERROR or CONSTRAINT_ERROR for reasons not anticipated by the test.

- C35A03E and C35A03R: These tests assume that attribute 'MANTISSA returns 0 when applied to a fixed-point type with a null range, but the Ada Standard does not support this assumption.

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

- C37213J: The aggregate in line 451 incorrectly raises CONSTRAINT_ERROR.

- C37215C, C37215E, C37215G, and C37215H: Various discriminant constraints are incorrectly expected to be incompatible with type CONS.

- C38102C: The fixed-point conversion on line 23 wrongly raises CONSTRAINT_ERROR.

- C41402A: The attribute 'STORAGE_SIZE is incorrectly applied to an object of an access type.

- C45332A: The test expects that either an expression in line 52 will raise an exception or else MACHINE_OVERFLOWS is FALSE. However, an implementation may evaluate the expression correctly using a type with a wider range than the base type of the operands, and MACHINE_OVERFLOWS may still be TRUE.

- C45614C: The function call of IDENT_INT in line 15 uses an argument of the wrong type.

- A74106C, C85018B, C87B04B, and CC1311B: A bound specified in a fixed-point subtype declaration lies outside of that calculated for the base type, raising CONSTRAINT_ERROR. Errors of this sort occur at lines 37 & 59, 142 & 143, 16 & 48, and 252 & 253 of the four tests, respectively.

- BC3105A: Lines 159 through 168 expect error messages, but these lines are correct Ada.

- AD1A01A: The declaration of subtype SINT3 raises CONSTRAINT_ERROR for implementations which select INT'SIZE to be 16 or greater.

- CE2401H: The record aggregates in lines 105 and 117 contain the wrong values.

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