UNCLASSIFIED

DTIC FILE COPY

(2)

AD-A216 383

...ATION PAGE — READ INSTRUCTIONS BEFORE COMPLETING FORM

Field	Entry
2. GOVT ACCESSION NO.	
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE	**Ada Compiler Validation Summary Report:** SYSTEAM KG SYSTEAM Ada Compiler VAX/VMS x MC68020/BARE, VAX 8530 (host) to Motorola MC68020 (target), 890825I1.10176
5. TYPE OF REPORT & PERIOD COVERED	25 August 89 - 1 Dec 90
6. PERFORMING ORG. REPORT NUMBER	
7. AUTHOR(s)	IABG, Ottobrunn, Federal Republic of Germany.
8. CONTRACT OR GRANT NUMBER(s)	
9. PERFORMING ORGANIZATION AND ADDRESS	IABG, Ottobrunn, Federal Republic of Germany.
10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS	
11. CONTROLLING OFFICE NAME AND ADDRESS	Ada Joint Program Office, United States Department of Defense, Washington, DC 20301-3081
12. REPORT DATE	
13. NUMBER OF PAGES	
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)	IABG, Ottobrunn, Federal Republic of Germany.
15. SECURITY CLASS (of this report)	UNCLASSIFIED
15a. DECLASSIFICATION/DOWNGRADING SCHEDULE	N/A

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

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

UNCLASSIFIED

DTIC ELECTE JAN 0 3 1990 S D D

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)

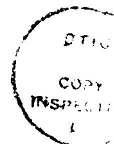
SYSTEAM KG, SYSTEAM Ada Compiler VAX/VMS x MC68020/BARE, IABG, West Germany, VAX 8530 under VMS, Version 4.7 (host) to Motorola MC68020 on MVME133XT board with MC68882 floating-point coprocessor (bare machine) (target), ACVC 1.10.

90 01 03 016

AVF Control Number: AVF-IABG-043

Ada COMPILER
VALIDATION SUMMARY REPORT:
Certificate Number: #890825I1.10176
SYSTEAM KG
SYSTEAM Ada Compiler VAX/VMS x MC68020/BARE
VAX 8530 Host and Motorola MC68020 Target

Completion of On-Site Testing:
25th August 1989



Prepared By:
IABG mbH, Abt SZT
Eirsteinstr 20
D8012 Ottobrunn
West Germany

Accesion For	
NTIS CRA&I	☑
DTIC TAB	☐
Unannounced	☐
Justification	
By	
Distribution /	
Availability Codes	
Dist	Avail and / or Special
A-1	

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

Ada Compiler Validation Summary Report:

Compiler Name: SYSTEAM Ada Compiler VAX/VMS x MC68020/BARE
Version 1.81

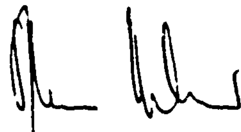
Certificate Number: #890825I1.10176

Host: VAX 8530 under VMS, Version 4.7

Target: Motorola MC68020 on MVME133XT board
with MC68882 floating-point coprocessor
(bare machine)

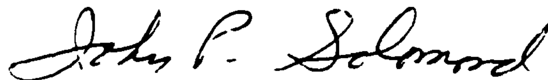
Testing Completed Friday 25th August 1989 Using ACVC 1.10

This report has been reviewed and is approved.



IABG mbH, Abt SZT
Dr S. Heilbrunner
Einsteinstr 20
D8012 Ottobrunn
West Germany

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



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

TABLE OF CONTENTS

CHAPTER 1 INTRODUCTION 1

1.1 PURPOSE OF THIS VALIDATION SUMMARY REPORT 1
1.2 USE OF THIS VALIDATION SUMMARY REPORT 2
1.3 REFERENCES 3
1.4 DEFINITION OF TERMS 3
1.5 ACVC TEST CLASSES 4

CHAPTER 2 CONFIGURATION INFORMATION 7

2.1 CONFIGURATION TESTED 7
2.2 IMPLEMENTATION CHARACTERISTICS 8

CHAPTER 3 TEST INFORMATION 13

3.1 TEST RESULTS 13
3.2 SUMMARY OF TEST RESULTS BY CLASS 13
3.3 SUMMARY OF TEST RESULTS BY CHAPTER 14
3.4 WITHDRAWN TESTS 14
3.5 INAPPLICABLE TESTS 14
3.6 TEST, PROCESSING, AND EVALUATION MODIFICATIONS . 17
3.7 ADDITIONAL TESTING INFORMATION
3.7.1 Prevalidation 18
3.7.2 Test Method 18
3.7.3 Test Site 19

APPENDIX A DECLARATION OF CONFORMANCE

APPENDIX B APPENDIX F OF THE Ada STANDARD

APPENDIX C TEST PARAMETERS

APPENDIX D WITHDRAWN TESTS

APPENDIX E COMPILER AND LINKER OPTIONS

CHAPTER 1

INTRODUCTION

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

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

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

1.1 PURPOSE OF THIS VALIDATION SUMMARY REPORT

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

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

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

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

Testing of this compiler was conducted by 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 Friday 25th August 1989 at SYSTEAM KG, Karlsruhe.

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

IABG mbH, Abt. SZT
Einsteinstr 20
D8012 Ottobrunn

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 which executes the code generated by the compiler.

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

Withdrawn 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 in which a program library is used at link time.

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

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

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

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

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

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

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

The text of each test in the ACVC follows conventions that are intended to ensure that the tests are reasonably portable without modification. For example, the tests make use of only the basic set of 55 characters, contain lines with a maximum length of 72 characters, use small numeric values, and

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: SYSTEAM Ada Compiler VAX/VMS x MC68020/BARE Version 1.81

ACVC Version: 1.10

Certificate Number: #890825I1.10176

Host Computer:

Machine: VAX 8530

Operating System: VMS Version 4.7

Memory Size: 32 MB

Target Computer:

Machine: Motorola MC68020 on MVME133XT board with MC68882 floating-point coprocessor

Operating System: BARE MACHINE

Memory Size: 4 MB

Communications Network: RS-232

2.2 IMPLEMENTATION CHARACTERISTICS

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

a. Capacities.

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

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

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

4) The compiler correctly processes tests containing recursive procedures separately compiled as subunits nested to 17 levels. (See tests D64005E..G (3 tests).)

b. Predefined types.

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

c. Expression evaluation.

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

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

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

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

4) No exception is raised when an integer literal operand in a comparison or membership test is outside the range of the base type. (See test C45232A.)

5) No exception is raised when a literal operand in a fixed-point comparison or membership test is outside the range of the base type. (See test C45252A.)

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

d. Rounding.

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

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

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

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

e. 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:

This implementation evaluates the 'LENGTH of each constrained array subtype during elaboration of the type declaration. This causes the declaration of a constrained array subtype with more than INTEGER'LAST (which is equal to SYSTEM.MAX_INT for this implementation) components to raise CONSTRAINT_ERROR. However, the optimisation mechanism of this implementation suppresses the evaluation of 'LENGTH if no object of the array type is declared depending on whether the bounds of the array are static, the visibility of the array type, and the presence of local subprograms. These general remarks apply to points (1) to (5), and (8).

1) Declaration of an array type or subtype declaration with more than SYSTEM.MAX_INT components raises no exception if the bounds of the array are static. (See test C36003A.)

2) CONSTRAINT_ERROR is raised when 'LENGTH is applied to an array type with INTEGER'LAST + 2 components if the bounds of the array are not static and if the subprogram declaring the array type contains no local subprograms. (See test C36202A.)

3) CONSTRAINT_ERROR is raised when 'LENGTH is applied to an array type with SYSTEM.MAX_INT + 2 components if the bounds of the array are not static and if the subprogram declaring the array type contains a local subprogram. (See test C36202B.)

4) A packed BOOLEAN array having a 'LENGTH exceeding INTEGER'LAST raises CONSTRAINT_ERROR when the array type is declared if the bounds of the array are not static and if there are objects of the array type. (See test C52103X.)

5) A packed two-dimensional BOOLEAN array with more than INTEGER'LAST components raises CONSTRAINT_ERROR when the array type is declared if the bounds of the array are not static and if there are objects of the array type. (See test C52104Y.)

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

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

8) 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 if the bounds of the array are not static and if there are objects of the array type. (See test E52103Y.)

f. Discriminated types.

1) In assigning record types with discriminants, the expression is not evaluated in its entirety before CONSTRAINT_ERROR is raised when checking whether the expression's subtype is compatible with the target's subtype. (See test C52013A.)

g. Aggregates.

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

2) In the evaluation of an aggregate containing subaggregates, all choices are evaluated before being checked for identical bounds. (See test E43212B.)

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

h. Pragmas.

1) The pragma INLINE is supported for functions and procedures. (See tests LA3004A..B (2 tests), EA3004C..D (2 tests), and CA3004E..F (2 tests).)

i. Generics.

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

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

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

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

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

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

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

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

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

j. Input and output.

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

2) The package DIRECT_IO can be instantiated with unconstrained array types and record types with discriminants without defaults. However this implementation raises USE_ERROR upon creation of a file for unconstrained array types.(See tests AE2101H, EE2401D, and EE2401G.)

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

CHAPTER 3

TEST INFORMATION

3.1 TEST RESULTS

Version 1.10 of the ACVC comprises 3717 tests. When this compiler was tested, 44 tests had been withdrawn because of test errors. The AVF determined that 481 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 and 238 tests containing file operations not supported by the implementation. Modifications to the code, processing, or grading for 14 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	129	1132	1852	17	16	46	3192
Inapplicable	0	6	463	0	12	0	481
Withdrawn	1	2	35	0	6	0	44
TOTAL	130	1140	2350	17	34	46	3717

3.3 SUMMARY OF TEST RESULTS BY CHAPTER

RESULT	CHAPTER													TOTAL
	2	3	4	5	6	7	8	9	10	11	12	13	14	
Passed	202	591	566	245	172	99	161	331	137	36	252	325	75	3192
N/A	11	58	114	3	0	0	5	1	0	0	0	44	245	481
Wdrn	0	1	0	0	0	0	0	2	0	0	1	35	5	44
TOTAL	213	650	680	248	172	99	166	334	137	36	253	404	325	3717

3.4 WITHDRAWN TESTS

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

E28005C	A39005G	B97102E	C97116A	BC3009B	CD2A62D
CD2A63A	CD2A63B	CD2A63C	CD2A63D	CD2A66A	CD2A66B
CD2A66C	CD2A66D	CD2A73A	CD2A73B	CD2A73C	CD2A73D
CD2A76A	CD2A76B	CD2A76C	CD2A76D	CD2A81G	CD2A83G
CD2A84N	CD2A84M	CD50110	CD2B15C	CD7205C	CD2D11B
CD5007B	ED7004B	ED7005C	ED7005D	ED7006C	ED7006D
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, 481 tests were inapplicable for the reasons indicated:

a. 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)

b. C34007P and C34007S are expected to raise CONSTRAINT_ERROR. This implementation optimizes the code at compile time on lines 205 and 221 respectively, thus avoiding the operation which would raise CONSTRAINT_ERROR and so no exception is raised.

c. C41401A is expected to raise CONSTRAINT_ERROR for the evaluation of certain attributes, however this implementation derives the values from the subtypes of the prefix at compile time as allowed by 11.6 (7) LRM. Therefore elaboration of the prefix is not involved and CONSTRAINT_ERROR is not raised.

d. 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				

e. C45531M..P (4 tests) and C45532M..P (4 tests) are inapplicable because the value of SYSTEM.MAX_MANTISSA is less than 48.

f. C47004A is expected to raise CONSTRAINT_ERROR whilst evaluating the comparison on line 51, but this compmiler evaluates the result without invoking the basic operation qualification (as allowed by 11.6 (7) LRM) which would raise CONSTRAINT_ERROR and so no exception is raised.

g. C86001F is not applicable because, for this implementation, the package TEXT_IO is dependent upon package SYSTEM. These tests recompile package SYSTEM, making package TEXT_IO, and hence package REPORT, obsolete.

h. B86001X, C45231D, and CD7101G are not applicable because this implementation does not support any predefined integer type with a name other than INTEGER, LONG_INTEGER, or SHORT_INTEGER.

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

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

k. C96005B is not applicable because there are no values of type DURATION'BASE that are outside the range of DURATION.

l. CD1009C, CD2A41A, CD2A41B, CD2A41E and CD2A42A..J (10 tests) are not applicable because this implementation imposes restrictions on 'SIZE length clauses for floating point types.

m. CD2A61I and CD2A61J are not applicable because this implementation imposes restrictions on 'SIZE length clauses for array types.

n. CD2A71A..D (4 tests), CD2A72A..D (4 tests), CD2A74A..D (4 tests) and CD2A75A..D (4 tests) are not applicable because this implementation imposes restrictions on 'SIZE length clauses for record types.

o. CD2A84B..I (8 tests), CD2A84K and CD2A84L are not applicable because this implementation imposes restrictions on 'SIZE length clauses for access types.

p. The following 238 tests are inapplicable because sequential, text, and direct access files are not supported:

CE2102A..C (3 tests) CE2102G..H (2 tests)
CE2102K CE2102N..Y (12 tests)
CE2103C..D (2 tests) CE2104A..D (4 tests)
CE2105A..B (2 tests) CE2106A..B (2 tests)
CE2107A..H (8 tests) CE2107L
CE2108A..B (2 tests) CE2108C..H (6 tests)
CE2109A..C (3 tests) CE2110A..D (4 tests)
CE2111A..I (9 tests) CE2115A..B (2 tests)
CE2201A..C (3 tests) CE2201F..N (9 tests)
CE2204A..D (4 tests) CE2205A
CE2208B CE2401A..C (3 tests)
CE2401E..F (2 tests) CE2401H..L (5 tests)
CE2404A..B (2 tests) CE2405B
CE2406A CE2407A..B (2 tests)
CE2408A..B (2 tests) CE2409A..B (2 tests)
CE2410A..B (2 tests) CE2411A
CE3102A..B (2 tests) EE3102C
CE3102F..H (3 tests) CE3102J..K (2 tests)
CE3103A CE3104A..C (3 tests)
CE3107B CE3108A..B (2 tests)
CE3109A CE3110A
CE3111A..B (2 tests) CE3111D..E (2 tests)

CE3112A..D (4 tests)	CE3114A..B (2 tests)
CE3115A	EE3203A
CE3208A	EE3301B
CE3302A	CE3305A
CE3402A	EE3402B
CE3402C..D (2 tests)	CE3403A..C (3 tests)
CE3403E..F (2 tests)	CE3404B..D (3 tests)
CE3405A	EE3405B
CE3405C..D (2 tests)	CE3406A..D (4 tests)
CE3407A..C (3 tests)	CE3408A..C (3 tests)
CE3409A	CE3409C..E (3 tests)
EE3409F	CE3410A
CE3410C..E (3 tests)	EE3410F
CE3411A..B (2 tests)	CE3412A
EE3412C	CE3413A
CE3413C	CE3602A..D (4 tests)
CE3603A	CE3604A..B (2 tests)
CE3605A..E (5 tests)	CE3606A..B (2 tests)
CE3704A..F (6 tests)	CE3704M..O (3 tests)
CE3706D	CE3706F..G (2 tests)
CE3804A..P (16 tests)	CE3805A..B (2 tests)
CE3806A..B (2 tests)	CE3806D..E (2 tests)
CE3806G..H (2 tests)	CE3905A..C (3 tests)
CE3905L	CE3906A..C (3 tests)
CE3906E..F (2 tests)	

These tests were not processed because their inapplicability can be deduced from the result of other tests.

q. Tests CE2103A, CE2103B and CE3107A raise USE_ERROR upon create for Sequential, Direct and Text IO.

r. Tests EE2201D, EE2201E, EE2401D and EE2401G raise USE_ERROR upon create.

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 was not anticipated by the test (such as raising one exception instead of another).

Modifications were required for 14 tests.

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

B22003A	B24009A	B29001A	B38003A	B38009A	B38009B
B51001A	B91001H	BA1101E	BC2001D	BC2001E	BC3204B
BC3205B	BC3205D				

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 SYSTEAM Ada Compiler VAX/VMS x MC68020/BARE Version 1.81 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 SYSTEAM Ada Compiler VAX/VMS x MC68020/BARE Version 1.81 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:	VAX 8530
Host operating system:	VMS Version 4.7
Target computer:	Motorola MC68020
Target operating system:	BARE MACHINE
Compiler:	SYSTEAM Ada Compiler VAX/VMS x MC68020/BARE Version 1.81

The host and target computers were linked via RS-232.

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 included in their modified form on the magnetic tape.

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 and linked on the VAX 8530, then all executable images were

transferred to the target via RS-232 and run. Results were printed from the host computer.

The compiler was tested using command scripts provided by SYSTEAM KG and reviewed by the validation team. Tests were compiled using the command

```
$@ADA:COMPILE <test name> LIST=[]
```

and linked with the command

```
$@ADA:LINK <test name> <test name>.EXE COMPLETE=ON
```

Chapter B tests were compiled with the full listing option. A full description of compiler and linker options is given in Apendix E.

Tests were compiled, linked, and executed (as appropriate) using a single host and target computer. Test output, compilation listings, and job logs were captured on 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 SYSTEAM KG, Karlsruhe and was completed on Friday 25th August 1989.

APPENDIX A

DECLARATION OF CONFORMANCE

SYSTEAM KG has submitted the following Declaration of Conformance concerning the SYSTEAM Ada Compiler VAX/VMS x MC68020/BARE Version 1.81.

Declaration of Conformance

Customer: SYSTEAM KG

Ada Validation Facility: IABG m. b. H., Abt. SZT

ACVC Version: 1.10

Ada Implementation

Ada Compiler Name: SYSTEAM Ada Compiler VAX/VMS x MC68020/BARE

Version: 1.81

Host Computer System: VAX 8530 under VMS, Version 4.7

Target Computer System: Motorola MC68020 on MVME 133XT board with MC68882 floating-point coprocessor (bare machine)

Customer's Declaration

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

Signature ______________________

Date 25.08.1989

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 SYSTEAM Ada Compiler VAX/VMS x MC68020/BARE Version 1.81, as described in this Appendix, are provided by SYSTEAM KG. Unless specifically noted otherwise, references in this appendix are to compiler documentation and not to this report. Implementation-specific portions of the package STANDARD, which are not a part of Appendix F, are:

```
package STANDARD is

   ...

     type INTEGER is range - 2_147_483_648 .. 2_147_483_647;
     type SHORT_INTEGER is range - 32_768 .. 32_767;

     type FLOAT is digits 15 range
        - 16#0.FFFF_FFFF_FFFF_F8#E256 .. 16#0.FFFF_FFFF_FFFF_F8#E256;
     type SHORT_FLOAT is digits 6 range
        - 16#0.FFFF_FF#E32 .. 16#0.FFFF_FF#E32;
     type LONG_FLOAT is digits 18 range
        - 16#0.FFFF_FFFF_FFFF_FFFF#E4096 ..
          16#0.FFFF_FFFF_FFFF_FFFF#E4096;

     type DURATION is delta 2#1.0#E-14 range
        - 131_072.0 .. 131_071.999_938_964_843_75;

   ...

end STANDARD;
```

11 Appendix F

This chapter, together with the Chapters 12 and 13, is the Appendix F required in [Ada], in which all implementation-dependent characteristics of an Ada implementation are described.

11.1 Implementation-Dependent Pragmas

The form, allowed places, and effect of every implementation-dependent pragma is stated in this section.

11.1.1 Predefined Language Pragmas

The form and allowed places of the following pragmas are defined by the language; their effect is (at least partly) implementation-dependent and stated here. All the other pragmas listed in Appendix B of [Ada] are implemented and have the effect described there.

CONTROLLED
   has no effect.

INLINE
   Inline expansion of subprograms is supported with the following restrictions: the subprogram must not contain declarations of other subprograms, tasks, generic units or body stubs. If the subprogram is called recursively only the outer call of this subprogram will be expanded.

INTERFACE
   is supported for assembler.

If a subprogram <ada_name> is implemented by an assembly language program the

```
PRAGMA interface (assembler, <ada_name>)
```

must be used. The actual parameters for the subprogram are written into a parameter block before the call; within the subprogram body, the address of this parameter block is stored at (4,A7). All parameters must be stored in a record object, where the first parameter must be stored at the lowest address of the object; then the subprogram has only one parameter (of the corresponding record type) and the parameter block contains only the address of the record object. This technique guarantees conformity with the calling mechanism of Ada subprograms.

The name of the routine which implements the subprogram <ada_name> should be specified using the pragma `external_name` (see § 10.1.2), otherwise the Compiler will generate an internal name that leads to an unresolved reference during linking.

MEMORY_SIZE
: has no effect.

OPTIMIZE
: has no effect.

PACK
: see §12.1.

PRIORITY
: There are two implementation-defined aspects of this pragma: First, the range of the subtype `priority`, and second, the effect on scheduling (see Chapter 10) of not giving this pragma for a task or main program. The range of subtype `priority` is 0 .. 15, as declared in the predefined library package `system` (see §11.3); and the effect on scheduling of leaving the priority of a task or main program undefined by not giving pragma `priority` for it is the same as if the pragma `priority` 0 had been given (i.e. the task has the lowest priority).

SHARED
: is supported.

STORAGE_UNIT
has no effect.

SUPPRESS
has no effect, but see §11.1.2 for the implementation-defined pragma suppress_all.

SYSTEM_NAME
has no effect.

11.1.2 Implementation-Defined Pragmas

EXTERNAL_NAME (<string>, <ada_name>)
<ada_name> specifies the name of a subprogram or of an object declared in a library package. <string> must be a string literal. It defines the external name of the specified subprogram or object. The Compiler uses a symbol with this name for the entry point of the subprogram or the base address of the object, so that it can be called or accessed by external routines by using this name. The Compiler also uses this name when calling the subprogram or accessing the object. <string> is case given order sensitive, i.e. it must be given in exactly the same manner as it is used by external routines. The subprogram or object declaration of <ada_name> must precede this pragma. If several subprograms or objects with the same name satisfy this requirement the pragma refers to that subprogram or object which immediately precedes it.
For subprograms this pragma is useful in connection with the pragma `interface (assembler)` (see §11.1.1).

RESIDENT (<ada_name>)
this pragma causes the value of the object <ada_name> to be held in storage (it may be held in a register too) and prevents assignments of a value to the object from being eliminated by the optimizer (see §3.2) of the SYSTEAM Ada Compiler. The following code sequence demonstrates the intended usage of the pragma:

```
...
x : integer;
a : system.address;
  ...
BEGIN
  x := 5;
```

```
a := x'address;
do_something (a);   -- let do_something be a non-local
                    -- procedure
                    -- a.ALL will be read in the body
                    -- of do_something
x := 6;
...
```

If this code sequence is compiled by the SYSTEAM Ada Compiler with the option

```
OPTIMIZER=>ON
```

the statement `x := 5;` will be eliminated because from the point of view of the optimizer the value of x is not used before the next assignment to x. Therefore

```
PRAGMA resident (x);
```

should be inserted after the declaration of x.

This pragma can be applied to all those kinds of objects for which the address clause is supported (cf. §8.5).
It will often be used in connection with the pragma `interface (assembler, ...)` (see §11.1.1).

SQUEEZE
see §12.1.

SUPPRESS_ALL
causes all the run_time checks described in [Ada,§11.7] to be suppressed; this pragma is only allowed at the start of a compilation before the first compilation unit; it applies to the whole compilation.

11.2 Implementation-Dependent Attributes

The name, type and implementation-dependent aspects of every implementation-dependent attribute is stated in this chapter.

11.2.1 Language-Defined Attributes

The name and type of all the language-defined attributes are as given in [Ada]. We note here only the implementation-dependent aspects.

ADDRESS

If this attribute is applied to an object for which storage is allocated, it yields the address of the first storage unit that is occupied by the object.
If it is applied to a subprogram or to a task, it yields the address of the entry point of the subprogram or task body.
If it is applied to a task entry for which an address clause is given, it yields the address given in the address clause.
For any other entity this attribute is not supported and will return the value **system.address_zero**.

IMAGE

The image of a character other than a graphic character (cf. [Ada, §3.5.5(11)]) is the string obtained by replacing each italic character in the indication of the character literal (given in [Ada, Annex C(13)]) by the corresponding upper-case character. For example, **character'image**(*nul*) = "NUL".

MACHINE_OVERFLOWS

Yields **true** for each fixed point type or subtype and **false** for each floating point type or subtype.

MACHINE_ROUNDS

Yields **true** for each fixed point type or subtype and **false** for each floating point type or subtype.

STORAGE_SIZE

The value delivered by this attribute applied to an access type is as follows:
If a length specification (STORAGE_SIZE, see §12.2) has been given for that type (static collection), the attribute delivers that specified value.
In case of a dynamic collection, i.e. no length specification by STORAGE_SIZE has been given for the access type, the attribute delivers the number of storage units currently allocated for the collection. Note that dynamic collections are extended if needed.
If the collection manager (cf. §9.3.1) is used for a dynamic collection the attribute delivers the number of storage units currently allocated for the collection. Note

that in this case the number of storage units currently allocated may be decreased by release operations.

The value delivered by this attribute applied to a task type or task object is as follows:
If a length specification (STORAGE_SIZE, see §12.2) has been given for the task type, the attribute delivers that specified value; otherwise, the default value is returned.

11.2.2 Implementation-Defined Attributes

There are no implementation-defined attributes.

11.3 Specification of the Package SYSTEM

The package system required in [Ada,§13.7] is reprinted here with all implementation-dependent characteristics and extensions filled in.

```
PACKAGE system IS

   TYPE designated_by_address IS LIMITED PRIVATE;

   TYPE address IS ACCESS designated_by_address;

   FOR address'size USE 32;

   FOR address'storage_size USE 0;

   address_zero         : CONSTANT address := NULL;

   TYPE name IS (motorola_68020_bare);

   system_name          : CONSTANT name := motorola_68020_bare;
   storage_unit         : CONSTANT := 8;
   memory_size          : CONSTANT := 2 ** 31;
   min_int              : CONSTANT := - 2 ** 31;
   max_int              : CONSTANT := 2 ** 31 - 1;
   max_digits           : CONSTANT := 18;
   max_mantissa         : CONSTANT := 31;
   fine_delta           : CONSTANT := 2.0 ** (- 31);
   tick                 : CONSTANT := 1.0;
```

```
SUBTYPE priority IS integer RANGE 0 .. 15;

FUNCTION  "+" (left  : address; right : integer) RETURN address;

FUNCTION  "+" (left  : integer; right : address) RETURN address;

FUNCTION  "-" (left  : address; right : integer) RETURN address;

FUNCTION  "-" (left  : address; right : address) RETURN integer;

SUBTYPE external_address IS string;

-- External addresses use hexadecimal notation with characters
-- '0'..'9', 'a'..'f' and 'A'..'F'. For instance:
--    "7FFFFFFF"
--    "80000000"
--    "8" represents the same address as "00000008"

TYPE interrupt_number IS RANGE 1 .. 32;

interrupt_vector : ARRAY (interrupt_number) OF address;
-- converts an interrupt_number to an address;

FUNCTION  convert_address (addr : external_address) RETURN address;

   -- CONSTRAINT_ERROR is raised if the external address ADDR
   -- is the empty string, contains characters other than
   -- '0'..'9', 'a'..'f', 'A'..'F' or if the resulting address
   -- value cannot be represented with 32 bits.

FUNCTION  convert_address (addr : address) RETURN external_address;

   -- The resulting external address consists of exactly 8
   -- characters '0'..'9', 'A'..'F'.

non_ada_error       : EXCEPTION;

-- non_ada_error is raised, if some event occurs which does not
-- correspond to any situation covered by Ada, e.g.:
--    illegal instruction encountered
--    error during address translation
--    illegal address

TYPE exception_id IS NEW integer;

no_exception_id     : CONSTANT exception_id := 0;
```

```
-- Coding of the predefined exceptions:

constraint_error_id : CONSTANT exception_id := ... ;
numeric_error_id    : CONSTANT exception_id := ... ;
program_error_id    : CONSTANT exception_id := ... ;
storage_error_id    : CONSTANT exception_id := ... ;
tasking_error_id    : CONSTANT exception_id := ... ;

non_ada_error_id    : CONSTANT exception_id := ... ;

status_error_id     : CONSTANT exception_id := ... ;
mode_error_id       : CONSTANT exception_id := ... ;
name_error_id       : CONSTANT exception_id := ... ;
use_error_id        : CONSTANT exception_id := ... ;
device_error_id     : CONSTANT exception_id := ... ;
end_error_id        : CONSTANT exception_id := ... ;
data_error_id       : CONSTANT exception_id := ... ;
layout_error_id     : CONSTANT exception_id := ... ;

time_error_id       : CONSTANT exception_id := ... ;

no_error_code       : CONSTANT := 0;

TYPE exception_information IS
   RECORD
      excp_id    : exception_id;

         -- Identification of the exception. The codings of
         -- the predefined exceptions are given above.

      code_addr  : address;

         -- Code address where the exception occured. Depending
         -- on the kind of the exception it may be the address of
         -- the instruction which caused the exception, or it
         -- may be the address of the instruction which would
         -- have been executed if the exception had not occured.

      error_code : integer;

   END RECORD;

PROCEDURE get_exception_information
            (excp_info : OUT exception_information);

   -- The subprogram get_exception_information must only be called
   -- from within an exception handler BEFORE ANY OTHER EXCEPTION
```

```
    -- IS RAISED. It then returns the information record about the
    -- actually handled exception.
    -- Otherwise, its result is undefined.

  TYPE exit_code IS NEW integer;

  error                  : CONSTANT exit_code := 1;
  success                : CONSTANT exit_code := 0;

  PROCEDURE set_exit_code (val : exit_code);

    -- Specifies the exit code which is returned to the
    -- operating system if the Ada program terminates normally.
    -- The default exit code is 'success'. If the program is
    -- abandoned because of an exception, the exit code is
    -- 'error'.

PRIVATE

   -- private declarations

END system;
```

11.4 Restrictions on Representation Clauses

See Chapter 12 of this manual.

11.5 Conventions for Implementation-Generated Names

There are implementation generated components but these have no names. (cf. §12.4 of this manual).

11.6 Expressions in Address Clauses

See §12.5 of this manual.

11.7 Restrictions on Unchecked Conversions

The implementation supports unchecked type conversions for all kinds of source and target types with the restriction that the target type must not be an unconstrained array type. The result value of the unchecked conversion is unpredictable if

```
target_type'SIZE > source_type'SIZE
```

11.8 Characteristics of the Input-Output Packages

The implementation-dependent characteristics of the input-output packages as defined in Chapter 14 of [Ada] are reported in Chapter 13 of this manual.

11.9 Requirements for a Main Program

A main program must be a parameterless library procedure. This procedure may be a generic instance; the generic procedure need not be a library unit.

11.10 Unchecked Storage Deallocation

The generic procedure `unchecked_deallocation` is provided, but the only effect of calling an instance of this procedure with an object `X` as actual parameter is

```
X := NULL;
```

i.e. no storage is reclaimed.

However, the implementation does provide an implementation-defined package `collection_manager` to support unchecked storage deallocation (cf. §9.3.1).

11.11 Machine Code Insertions

A package `machine_code` is not provided and machine code insertions are not supported.

11.12 Numeric Error

The predefined exception `numeric_error` is never raised implicitly by any predefined operation; instead the predefined exception `constraint_error` is raised.

12 Appendix F: Representation Clauses

In this chapter we follow the section numbering of Chapter 13 of [Ada] and provide notes for the use of the features described in each section.

12.1 Pragmas

PACK

As stipulated in [Ada,§13.1], this pragma may be given for a record or array type. It causes the Compiler to select a representation for this type such that gaps between the storage areas allocated to consecutive components are minimized. For components whose type is an array or record type the pragma pack has no effect on the mapping of the component type. For all other component types the Compiler will try to choose a more compact representation for the component type. All components of a packed data structure will start at storage unit boundaries and the size of the components will be a multiple of `system.storage_unit`. Thus, the pragma pack does not effect packing down to the bit level (for this see pragma **squeeze**).

SQUEEZE

This is an implementation-defined pragma which takes the same argument as the predefined language pragma pack and is allowed at the same positions. It causes the Compiler to select a representation for the argument type that needs minimal storage space (packing down to the bit level). For components whose type is an array or record type the pragma **squeeze** has no effect on the mapping of the component type. For all other component types the Compiler will try to choose a more compact representation for the component type. The components of a squeezed data structure will not in general start at storage unit boundaries.

12.2 Length Clauses

SIZE

for all integer, fixed point and enumeration types the value must be <= 32;
for short_float types the value must be = 32 (this is the amount of storage which is associated with these types anyway);
for float types the value must be = 64 (this is the amount of storage which is associated with these types anyway).
for long_float types the value must be = 96 (this is the amount of storage which is associated with these types anyway).
for access types the value must be = 32 (this is the amount of storage which is associated with these types anyway).
If any of the above restrictions are violated, the Compiler responds with a RESTRICTION error message in the Compiler listing.

STORAGE_SIZE

Collection size: If no length clause is given, the storage space needed to contain objects designated by values of the access type and by values of other types derived from it is extended dynamically at runtime as needed. If, on the other hand, a length clause is given, the number of storage units stipulated in the length clause is reserved, and no dynamic extension at runtime occurs.

Storage for tasks: The memory space reserved for a task is 10K bytes if no length clause is given (cf. Chapter 10). If the task is to be allotted either more or less space, a length clause must be given for its task type, and then all tasks of this type will be allotted the amount of space stipulated in the length clause (the activation of a small task requires about 1.4K bytes). Whether a length clause is given or not, the space allotted is not extended dynamically at runtime.

SMALL

there is no implementation-dependent restriction. Any specification for SMALL that is allowed by the LRM can be given. In particular those values for SMALL are also supported which are not a power of two.

12.3 Enumeration Representation Clauses

The integer codes specified for the enumeration type have to lie inside the range of the largest integer type which is supported; this is the type integer defined in package standard.

12.4 Record Representation Clauses

Record representation clauses are supported. The value of the expression given in an alignment clause must be 0, 1, 2 or 4. If this restriction is violated, the Compiler responds with a RESTRICTION error message in the Compiler listing. If the value is 0 the objects of the corresponding record type will not be aligned, if it is 1, 2 or 4 the starting address of an object will be a multiple of the specified alignment.

There are implementation-dependent components of record types generated in the following cases:

- If the record type includes variant parts and the difference of the size between the maximum and the minimum variant is greater then 32 bytes, and, in addition, if it has either more than one discriminant or else the only discriminant may hold more than 256 different values, the generated component holds the size of the record object. (If the second condition is not fulfilled, the number of bits allocated for any object of the record type will be the value delivered by the size attribute applied to the record object.)
- If the record type includes array or record components whose sizes depend on discriminants, the generated components hold the offsets of these record components (relative to the corresponding generated component) in the record object.

But there are no implementation-generated names (cf. [Ada,§13.4(8)]) denoting these components. So the mapping of these components cannot be influenced by a representation clause.

12.5 Address Clauses

Address clauses are supported for objects declared by an object declaration and for task entries. If an address clause is given for a subprogram, package or task unit, the Compiler responds with a RESTRICTION error message in the Compiler listing.

If an address clause is given for an object, the storage occupied by the object starts at the given address.

If an address clause is given for a task entry, then this entry can be called by a user written interrupt service routine. Such an entry is called an interrupt entry. While an interrupt entry is identified by its Ada name within the Ada program, it is identified by a number (TYPE `interrupt_number` of PACKAGE `system`) outside the Ada program (e.g. in the interrupt service routine). This number is defined by the address clause for the interrupt entry, as the following example shows, where an interrupt entry `intr_entry` is declared with number 1:

```
ENTRY intr_entry;
FOR intr_entry USE AT system.interrupt_vector (1);
```

The number of an interrupt entry allows an interrupt service routine to call the interrupt entry. Parameters for interrupt entries are not supported.

The connection between a hardware interrupt and an interrupt entry is done within an interrupt service routine. An interrupt service routine is an assembler routine that is automatically activated each time a specific hardware interrupt occurs.

Consider the following example: The user wants to catch the hardware interrupt INTR. (INTR represents the vector number of the hardware interrupt to be caught.) Then he has to write an assembler routine ISR which will be activated each time INTR occurs. This routine must be defined as an interrupt service routine by calling the PROCEDURE define_interrupt_service_routine of PACKAGE privileged_operations. In the Ada program ISR must be declared as an interrupt service routine as follows:

```
-- Declaration of the external routine ISR:
      PROCEDURE isr;
         PRAGMA interface (assembler, isr);
         PRAGMA external_name ("ISR", isr);

-- Definition of ISR as interrupt service routine for interrupt INTR:
      define_interrupt_service_routine (isr'address, INTR);
```

The interrupt service routine ISR must look like this:

```
ISR:    ...                 *-- save all used registers
        ...                 *-- action
        ...
        ...                 *-- restore all used registers
        JMP    ([_IRRETURN])
```

_IRRETURN is defined by the Ada Runtime System. It contains the start address of the Target Kernel routine that carries out the return from interrupt handling. It is very important that when leaving ISR all registers (except the status register) have the same values as they had when entering ISR. ISR is executed in the supervisor state of the processor. So all instructions (including privileged ones) can be used within ISR. The processor's priority depends on the interrupt source.

If the user wants to call the interrupt entry with the number N, then he has to set a bit within the interrupt entry call pending indicator _IRENTRYC by the instruction:

```
        bfset    (_IRRENTRYC).1.{N-1:1}  *-- prepare call of
                                         *-- interrupt entry N
```

This instruction should be placed immediately in front of the last instruction of ISR. ISR need not call the interrupt entry each time it is activated. Instead ISR can, for example, read one character each time it is activated, but call the interrupt entry only when a complete line has been read.

A complete example for interrupt handling follows. For this example the second RS232 serial line of the MVME133XT board is used (available through the P2 connector). The assembler routine ISR_READ is activated each time a character is received on that line. ISR_READ calls interrupt entry char_entry of TASK terminal_in. terminal_in uses TASK terminal_out to output each character read.

```
WITH system,
     privileged_operations,
     text_io;

USE  privileged_operations,
     text_io;

PROCEDURE terminal IS

   PRAGMA priority (2);

   PROCEDURE setup_scc;
      PRAGMA interface (assembler, setup_scc);
      PRAGMA external_name ("SETUP_SCC", setup_scc);

   PROCEDURE isr_read;
      PRAGMA interface (assembler, isr_read);
      PRAGMA external_name ("ISR_READ", isr_read);

   TASK terminal_in IS
      PRAGMA priority (1);

      ENTRY char_entry;
      FOR char_entry USE AT system.interrupt_vector (1);
   END terminal_in;

   TASK terminal_out IS
      PRAGMA priority (0);

      ENTRY put (item : IN character);
   END terminal_out;
```

```
   sccb_rro : CONSTANT system.address :=
                           system.convert_address ("FFFA0000");
   sccb_wro : CONSTANT system.address :=
                           system.convert_address ("FFFA0000");

   sccb_rdr : character;
   FOR sccb_rdr USE AT system.convert_address ("FFFA0001");
   sccb_tdr : character;
   FOR sccb_tdr USE AT system.convert_address ("FFFA0001");

   TASK BODY terminal_in IS
      ch : character;
   BEGIN
      LOOP
         ACCEPT char_entry DO
            ch := sccb_rdr;
            assign_byte (sccb_wro, 16#38#);
         END char_entry;
         put (ch);
         terminal_out.put (ch);
      END LOOP;
   END terminal_in;

   TASK BODY terminal_out IS
   BEGIN
      LOOP
         SELECT
            WHEN bit_value (sccb_rro, 2) =>
               ACCEPT put (item : IN character) DO
                  sccb_tdr := item;
               END put;
         ELSE
            NULL;
         END SELECT;
      END LOOP;
   END terminal_out;

BEGIN
   setup_scc;

   define_interrupt_service_routine
                     (isr_read'address, 16#80#);
END terminal;
```

The following assembler routines also belong to the example:

```
typlang  equ 0
attrrev  equ $8000
*--
        psect    terminal,typlang,attrrev,0,0,0
*--
sccb_rro equ     $FFFA0000
sccb_wro equ     $FFFA0000
sccb_rdr equ     $FFFA0001
sccb_tdr equ     $FFFA0001
*--
SETUP_SCC:
   move.b    #$30,(sccb_wro).l    *-- clear receiver error status
   move.b    #$10,(sccb_wro).l    *-- clear external status interrupts
   move.b    #$09,(sccb_wro).l    *-- WR 9
   move.b    #$40,(sccb_wro).l    *-- reset channel A & B, disable IRs
*--
   move.b    #$0A,(sccb_wro).l    *-- WR 10
   move.b    #$00,(sccb_wro).l    *-- NRZ format
   move.b    #$0E,(sccb_wro).l    *-- WR 14
   move.b    #$82,(sccb_wro).l    *-- source=BR generator, RTXC input
*--                               *-- disable BR generator
   move.b    #$04,(sccb_wro).l    *-- WR 4
   move.b    #$44,(sccb_wro).l    *-- clck mode=x16,1 stop bit,no parity
   move.b    #$03,(sccb_wro).l    *-- WR 3
   move.b    #$C1,(sccb_wro).l    *-- 8 bits, enable receiver
   move.b    #$05,(sccb_wro).l    *-- WR 5
   move.b    #$EA,(sccb_wro).l    *-- DTR&RTS=on,8 bits,enable transmtr
   move.b    #$0C,(sccb_wro).l    *-- WR 12
   move.b    #$02,(sccb_wro).l    *-- lower byte of time const (9600 Bd)
   move.b    #$0D,(sccb_wro).l    *-- WR 13
   move.b    #$00,(sccb_wro).l    *-- upper byte of time const (9600 Bd)
   move.b    #$0B,(sccb_wro).l    *-- WR 11
   move.b    #$56,(sccb_wro).l    *-- RxClock=TxClock=TRxClock=BR output
*--                               *-- TRxC output
   move.b    #$0E,(sccb_wro).l    *-- WR 14
   move.b    #$81,(sccb_wro).l    *-- source=BR generator, RTXC input
*--                               *-- enable BR generator
   move.b    #$01,(sccb_wro).l    *-- WR 1
   move.b    #$11,(sccb_wro).l    *-- interrupt on all received chars or
*--                               *-- special cond, disable external IRs
   move.b    #$0F,(sccb_wro).l    *-- WR 2
   move.b    #$80,(sccb_wro).l    *-- interrupt vector numbers $80
   move.b    #$02,(sccb_wro).l
   move.b    #$80,(sccb_wro).l
   move.b    #$09,(sccb_wro).l    *-- WR 9
   move.b    #$08,(sccb_wro).l    *-- enable interrupts
```

```
   rts
*--
*--
ISR_READ:
*--      bfset    (_IRENTRYC).1{0:1}      *-- call interrupt entry 1
*--    Because of an assembler bug the following construction
*--    is used instead of the BFSET instruction:
         DC.W     $EEF9,$0001
         DC.L     _IRENTRYC
         jmp      ([_IRRETURN])
*--
         ends
```

12.6 Change of Representation

The implementation places no additional restrictions on changes of representation.

13 Appendix F: Input-Output

In this chapter we follow the section numbering of Chapter 14 of [Ada] and provide notes for the use of the features described in each section.

13.1 External Files and File Objects

The implementation only supports the files `standard_input` and `standard_output` of PACKAGE `text_io`. Any attempt to create or open a file raises the exception `use_error`.

13.2 Sequential and Direct Files

Sequential and direct files are not supported.

13.3 Text Input-Output

`standard_input` and `standard_output` are associated with the RS232 serial port of the target. If the Full Target Kernel is used, then all input/output operations are done on the host using the communication line between the host and this port. Both the Debugger and the Starter are prepared to do these I/O operations.

If the Minimal Target Kernel is used, then the same serial port as in the Full Target Kernel is used, but all data of `standard_output` is directly written to this port and all data of `standard_input` is directly read from this port.

For tasking aspects of I/O operations see §10.4.

For further details on the I/O implementation within the Target Kernel see [ST4/89].

13.3.1 Implementation-Defined Types

The implementation-dependent types `count` and `field` defined in the package specification of `text_io` have the following upper bounds:

```
COUNT'LAST = 2_147_483_647 (= INTEGER'LAST)
FIELD'LAST = 512
```

13.4 Exceptions in Input-Output

For each of name_error, use_error, device_error and data_error we list the conditions under which that exception can be raised. The conditions under which the other exceptions declared in the package io_exceptions can be raised are as described in [Ada,§14.4].

NAME_ERROR
: is never raised.

USE_ERROR
: is raised if an attempt is made to create or open a file.

DEVICE_ERROR
: is never raised.

DATA_ERROR
: the conditions under which data_error is raised by text_io are laid down in [Ada].

13.5 Low Level Input-Output

We give here the specification of the package low_level_io:

```
PACKAGE low_level_io IS

   TYPE device_type IS (null_device);

   TYPE data_type IS
      RECORD
         NULL;
      END RECORD;

   PROCEDURE send_control    (device : device_type;
```

```
                                     data   : IN OUT data_type);

   PROCEDURE receive_control (device : device_type;
                              data   : IN OUT data_type);

END low_level_io;
```

Note that the enumeration type `device_type` has only one enumeration value, `null_device`; thus the procedures `send_control` and `receive_control` can be called, but `send_control` will have no effect on any physical device and the value of the actual parameter `data` after a call of `receive_control` will have no physical significance.

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
$ACC_SIZE An integer literal whose value is the number of bits sufficient to hold any value of an access type.	32
$BIG_ID1 An identifier the size of the maximum input line length which is identical to $BIG_ID2 except for the last character.	254 * 'A' & '1'
$BIG_ID2 An identifier the size of the maximum input line length which is identical to $BIG_ID1 except for the last character.	254 * 'A' & '2'
$BIG_ID3 An identifier the size of the maximum input line length which is identical to $BIG_ID4 except for a character near the middle.	127 * 'A' & '3' & 127 * 'A'

Name and Meaning Value

$BIG_ID4 An identifier the size of the maximum input line length which is identical to $BIG_ID3 except for a character near the middle.	127 * 'A' & '4' & 127 * 'A'
$BIG_INT_LIT An integer literal of value 298 with enough leading zeroes so that it is the size of the maximum line length.	252 * '0' & "298"
$BIG_REAL_LIT A universal real literal of value 690.0 with enough leading zeroes to be the size of the maximum line length.	250 * '0' & "690.0"
$BIG_STRING1 A string literal which when catenated with BIG_STRING2 yields the image of BIG_ID1.	'"' & 127 * 'A' & '"'
$BIG_STRING2 A string literal which when catenated to the end of BIG_STRING1 yields the image of BIG_ID1.	'"' & 127 * 'A' & '1' & '"'
$BLANKS A sequence of blanks twenty characters less than the size of the maximum line length.	235 * ' '
$COUNT_LAST A universal integer literal whose value is TEXT_IO.COUNT'LAST.	2147483647
$DEFAULT_MEM_SIZE An integer literal whose value is SYSTEM.MEMORY_SIZE.	2_147_483_648
$DEFAULT_STOR_UNIT An integer literal whose value is SYSTEM.STORAGE_UNIT.	8

Name and Meaning	Value
$DEFAULT_SYS_NAME The value of the constant SYSTEM.SYSTEM_NAME.	MOTOROLA_68020_BARE
$DELTA_DOC A real literal whose value is SYSTEM.FINE_DELTA.	2#1.0#E-31
$FIELD_LAST A universal integer literal whose value is TEXT_IO.FIELD'LAST.	512
$FIXED_NAME The name of a predefined fixed-point type other than DURATION.	NO_SUCH_FIXED_TYPE
$FLOAT_NAME The name of a predefined floating-point type other than FLOAT, SHORT_FLOAT, or LONG_FLOAT.	NO_SUCH_FLOAT_TYPE
$GREATER_THAN_DURATION A universal real literal that lies between DURATION'BASE'LAST and DURATION'LAST or any value in the range of DURATION.	0.0
$GREATER_THAN_DURATION_BASE_LAST A universal real literal that is greater than DURATION'BASE'LAST.	200_000.0
$HIGH_PRIORITY An integer literal whose value is the upper bound of the range for the subtype SYSTEM.PRIORITY.	15
$ILLEGAL_EXTERNAL_FILE_NAME1 An external file name which contains invalid characters.	invalid_1.!@#$%^&*()
$ILLEGAL_EXTERNAL_FILE_NAME2 An external file name which is too long.	invalid_2.!@#$%^&*()

Name and Meaning	Value
$INTEGER_FIRST A universal integer literal whose value is INTEGER'FIRST.	-2147483648
$INTEGER_LAST A universal integer literal whose value is INTEGER'LAST.	2147483647
$INTEGER_LAST_PLUS_1 A universal integer literal whose value is INTEGER'LAST + 1.	2147483648
$LESS_THAN_DURATION A universal real literal that lies between DURATION'BASE'FIRST and DURATION'FIRST or any value in the range of DURATION.	-0.0
$LESS_THAN_DURATION_BASE_FIRST A universal real literal that is less than DURATION'BASE'FIRST.	-200_000.0
$LOW_PRIORITY An integer literal whose value is the lower bound of the range for the subtype SYSTEM.PRIORITY.	0
$MANTISSA_DOC An integer literal whose value is SYSTEM.MAX_MANTISSA.	31
$MAX_DIGITS Maximum digits supported for floating-point types.	18
$MAX_IN_LEN Maximum input line length permitted by the implementation.	255
$MAX_INT A universal integer literal whose value is SYSTEM.MAX_INT.	2147483647
$MAX_INT_PLUS_1 A universal integer literal whose value is SYSTEM.MAX_INT+1.	2_147_483_648

Name and Meaning	Value
$MAX_LEN_INT_BASED_LITERAL A universal integer based literal whose value is 2#11# with enough leading zeroes in the mantissa to be MAX_IN_LEN long.	"2:" & 250 * '0' & "11:"
$MAX_LEN_REAL_BASED_LITERAL A universal real based literal whose value is 16:F.E: with enough leading zeroes in the mantissa to be MAX_IN_LEN long.	"16:" & 248 * '0' & "F.E:"
$MAX_STRING_LITERAL A string literal of size MAX_IN_LEN, including the quote characters.	'"' & 253 * 'A' & '"'
$MIN_INT A universal integer literal whose value is SYSTEM.MIN_INT.	-2147483648
$MIN_TASK_SIZE An integer literal whose value is the number of bits required to hold a task object which has no entries, no declarations, and "NULL;" as the only statement in its body.	32
$NAME A name of a predefined numeric type other than FLOAT, INTEGER, SHORT_FLOAT, SHORT_INTEGER, LONG_FLOAT, or LONG_INTEGER.	NO_SUCH_TYPE
$NAME_LIST A list of enumeration literals in the type SYSTEM.NAME, separated by commas.	MOTOROLA_68020_BARE
$NEG_BASED_INT A based integer literal whose highest order nonzero bit falls in the sign bit position of the representation for SYSTEM.MAX_INT.	16#FFFFFFFE#

Name and Meaning	Value
$NEW_MEM_SIZE An integer literal whose value is a permitted argument for pragma MEMORY_SIZE, other than $DEFAULT_MEM_SIZE. If there is no other value, then use $DEFAULT_MEM_SIZE.	2_147_483_648
$NEW_STOR_UNIT An integer literal whose value is a permitted argument for pragma STORAGE_UNIT, other than $DEFAULT_STOR_UNIT. If there is no other permitted value, then use value of SYSTEM.STORAGE_UNIT.	8
$NEW_SYS_NAME A value of the type SYSTEM.NAME, other than $DEFAULT_SYS_NAME. If there is only one value of that type, then use that value.	MOTOROLA_68020_BARE
$TASK_SIZE An integer literal whose value is the number of bits required to hold a task object which has a single entry with one 'IN OUT' parameter.	32
$TICK A real literal whose value is SYSTEM.TICK.	1.0

APPENDIX D

WITHDRAWN TESTS

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

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

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

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

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

e. 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).

f. 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).

g. 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.

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

i. 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.

j. CD2D11B This test gives a SMALL representation clause for a derived fixed-point type (at line 30) that defines a set of model numbers that are not necessarily represented in the parent type; by Commentary AI-00099, all model numbers of a derived fixed-point type must be representable values of the parent type.

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

l. ED7004B, ED7005C & D, ED7006C & D [5 tests] These tests check various aspects of the use of the three SYSTEM pragmas; the AVO withdraws these tests as being inappropriate for validation.

m. 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).

n. CD7203B, & CD7204B These tests use the 'SIZE length clause and attribute, whose interpretation is considered problematic by the WG9 ARG.

o. 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.

p. 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)

q. CE3111C This test requires certain behavior, when two files are associated with the same external file, that is not required by the Ada standard.

r. CE3301A This test contains several calls to END_OF_LINE & END_OF_PAGE that have no parameter: these calls were intended to specify a file, not to refer to STANDARD_INPUT (lines 103, 107, 113, 132, & 136).

s. CE3411B This test requires that a text file's column number be set to COUNT'LAST in order to check that LAYOUT_ERROR is raised by a subsequent PUT operation. But the former operation will generally raise an exception due to a lack of available disk space, and the test would thus encumber validation testing.

APPENDIX E

COMPILER AND LINKER OPTIONS

This appendix contains information concerning the compilation and linkage commands used within the command scripts for this validation.

3 Compiling and Completing

After a program library has been created, one or more compilation units can be compiled in the context of this library. The compilation units can be placed on different source files or they can all be on the same file.

3.1 Compiling Ada Units

The SYSTEAM Ada Compiler is started by the command:

```
@ADA:COMPILE  <source> [LIBRARY=<directory>] -
                       [OPTIONS="option [,...]"] -
                       [LIST=<filespec>]
```

Option	Default
LIST => ON/OFF	OFF
OPTIMIZER => ON/OFF	ON
INLINE => ON/OFF	ON
COPY_SOURCE => ON/OFF	OFF
SUPPRESS_ALL	
SYMBOLIC_CODE	

The input file for the Compiler is <source>. If the file type of <source> is not specified, <source>.ADA is assumed. The maximum length of lines in <source> is 255; longer lines are cut and an error is reported.

<directory> is the name of the program library; [.ADALIB] is the default of this parameter. The library must exist (see §2.1 for information on program library management).

The listing file is created in the default directory with the file name of <source> and the file type .LIS if no file specification <filespec> is given by the parameter LIST. Otherwise, the directory and file name are determined by the file specification <filespec>. If the file specification given is not a full file specification, missing components are determined as described above (i.e. the default directory is used if no directory is specified, the file name of <source> if no file name is specified and the file type .LIS if the file type is missing).

Options for the Compiler can be specified by using the parameter OPTIONS; they have an effect only for the current compilation. Blanks are allowed following and preceding lexical elements within the OPTIONS parameter. The options accepted by the Compiler are shown in the table above.

The options LIST and SUPPRESS_ALL have the same effect as the corresponding pragmas would have at the beginning of the source (see [Ada,Appendix B] and §11.1.2 of this manual).

No optimizations like constant folding, dead code elimination or structural simplifications are done if OPTIMIZER => OFF is specified.

Inline expansion of subprograms which are specified by a pragma `inline` (cf. §11.1.1) in the Ada source can be suppressed by giving the option INLINE => OFF. The value ON will cause inline expansion of the respective subprograms.

COPY_SOURCE => ON causes the Compiler to copy the source file <source> into the program library so that the Debugger can work on these copies instead of on the original ones.

A symbolic code listing can be produced by specifying the option SYMBOLIC_CODE when calling the Compiler. The code listing is written on a file with file type `.SYM` whose file name and directory are identical with those of the listing file.

The source file may contain a sequence of compilation units, cf. §10.1 of [Ada]. All compilation units in the source file are compiled individually. When a compilation unit is compiled successfully, the program library is updated and the Compiler continues with the compilation of the next unit on the source file. If the compilation unit contained errors, they are reported (see §3.2). In this case, no update operation is performed on the program library and all subsequent compilation units in the compilation are only analyzed without generating code.

The Compiler delivers the status code WARNING on termination (see [VAX/VMS, DCL Dictionary, command EXIT]) if one of the compilation units contained errors. A message corresponding to this code has not been defined; hence %NONAME-W-NOMSG is printed upon notification of a batch job terminated with this status.

3.2 Compiler Listing

A Compiler listing of a compilation unit looks as follows: It starts with the kind and the name of the unit and the library key of the current unit.

Example:

```
* PROCEDURE  MAIN,   Library Index   76
```

By default only source lines referred to by messages of the Compiler are listed. A complete listing can be obtained by using pragma LIST or the Compiler option LIST. The format effectors ASCII.HT, ASCII.VT, ASCII.CR, ASCII.LF and ASCII.FF are

6 Linking

The Linker produces executable code for a given Ada main procedure together with all its Ada units and external units. This process is called *final linking*. The result of a final link is a *code portion* which must be loaded onto the target later on.

Furthermore, the Linker supports linking a program step by step (say in $N \geq 1$ steps 1 ... N). All steps except the last one are called *incremental links*. In an incremental link a *collection* is linked; a collection is a set of Ada units and external units.

Each step $X \in \{2 \ldots N\}$ is based on the result of step X-1. The last step is always a final link, i.e. it links the Ada main program. The result of an incremental link is also a code portion which must be loaded onto the target later on.

So the code of a program may consist of several code portions which are loaded onto the target one by one. This is called *incremental loading*.

The reasons for the introduction of the concept of incremental linking and loading into the Ada Cross System are the following:

- It should be possible that some Ada library units and external units are compiled, linked, and burnt into a ROM that is plugged into the target, and that programs using these units are linked afterwards.
- The loading time during program development should be as short as possible. This is achieved by linking those parts of the program that are not expected to be changed (e.g. some library units and the Ada Runtime System). The resulting code portion is loaded to the target and need not be linked or loaded later on. Instead, only those parts of the program that have been modified or introduced since the first link must be linked, so that the resulting code portion is much smaller in size than the code of the whole program would be. Because typically this code portion is loaded several times during program development the development cycle time is reduced drastically.

The Runtime System (which is always necessary for the execution of Ada programs) is always linked during the first linking step. In particular, this means that also the version of the Runtime System (Debug or Non-Debug) is fixed during the first step.

The Linker gives the user a wide flexibility by allowing him to prescribe the mapping of single Ada units and assembler routines into the memory of the target. This, for example, enables the user to map units that are time critical into the fastest memory parts.

For this purpose the user specifies the regions of the target's memory space that should be used by the Linker, the size of the stack, and the regions that should be used for the stack, for the code, for the data, and for the heap. All these instructions for the Linker are contained in the *linker directive file*, which is a parameter of the Linker.

The main task of the Linker is to map a set of sections into the target's memory regions. Each section belongs either to a compilation unit or to an external (assembler) unit, or is generated by the Linker itself. In mapping sections into regions the Linker has to take into consideration the parameters and directives given by the user.

A *section* is a contiguous sequence of bytes representing code or data. A section is the smallest unit for the Linker.

6.1 Linking Main Programs

Linking a main program is called *final linking*. The Linker determines the compilation units belonging to the Ada program, automatically completes (i.e. compiles by the Completer) all instances of generics units and all packages which do not require a body, determines the elaboration order and finally links the Ada program. Selective linking is done automatically by the Linker. Linking selectively means that only those subprograms of an imported package which are really needed are linked. This can lead to a drastic reduction of the program size, e. g. when only a few subprograms of a package providing mathematical functions are used.

Final linking results in a *program image file*. There is a code portion in this image file which together with the code of the given base (if any) is the code of all Ada units and all external (assembler written) units that belong to the program and that are really needed (selective linking).

If the resulting program is to be a stand-alone program (no communication line between host and target), the Minimal Target Kernel must explicitly be linked to the program during final linking.

The Linker for linking a program (final linking) is started by

```
@ADA:LINK  <ada_name>  <result_file_name>  <directive_file_name> -
                       [LIBRARY=<directory>] -
                       [COMPLETE=ON/OFF] -
                       [OPTIONS=<string>] -
                       [LIST=<filespec>] -
                       [EXTERNAL=<filespec>[....]] -
                       [KERNEL=<filespec>] -
                       [BASE=<filespec>] -
                       [DEBUG=ON/OFF] -
                       [MAP=<filespec>]
```

Option	Default
OPTIMIZER => ON/OFF	ON
INLINE => ON/OFF	ON
SUPPRESS_ALL	
SYMBOLIC_CODE	

<ada_name> is the name of the library procedure which acts as the main program. The main program must be a parameterless library procedure.

<result_file_name> is the name of the file which will contain the result of the final link. The file type .LOD is assumed if none is specified.

<directive_file_name> is the name of the file which contains the linker directives. They describe the target's memory regions and prescribe the mapping of code, data, stack, and heap sections into these regions. The file type .LID is assumed if none is specified. For its format see §6.3. The SYSTEAM Ada System is delivered with a directive file for the MVME133XT. It can be used for those applications that use the whole memory of the MVME133XT and do not require specific units to be linked into specific regions. If the user wants to use this file, he has to specify:

```
ADA:133
```

<directory> is the name of the program library which contains the main program; [.ADALIB] is assumed if this parameter is not specified.

The COMPLETE parameter specifies whether the program is to be completed before it is linked; default is ON. If the Completer is called, the parameters LIBRARY, OPTIONS and LIST are passed to it (cf. §3.2).

The EXTERNAL parameter specifies files which contain the object code of those program units which are written in assembler. If several files are given, they must be separated by commas. Their default file type is .OBJ.

The parameter KERNEL specifies the name of the file that contains the assembled code of the Target Kernel that is to be linked to the program. If the user wants to link the Minimal Target Kernel to his program, then he must specify:

```
KERNEL=ADA:MTK.OBJ
```

If KERNEL is not specified, then no Target Kernel is linked to the program. This is recommended if the Full Target Kernel is already on the target. Note, the Minimal Target Kernel must not be linked to the final program if the Debug Runtime System is used.

The BASE parameter specifies a collection image file (a file containing the result of a previous incremental link). Its default file type is .LOD. If a base is specified, then the final link is done on the base of the given file.

The DEBUG parameter (default: ON) has the following effect: If DEBUG=ON, then the Debug Runtime System is linked to the program and information for the SYSTEAM Debugger is generated. If DEBUG=OFF, then the Non-Debug Runtime System is linked to the program and no information for the SYSTEAM Debugger is generated.
Important: If a base is given (parameter BASE), then no Runtime System is linked because it is already part of the base collection.

The MAP parameter controls the generation of a Linker map. If a map file is specified, then the Linker map will be written into this file. Its default type is .MAP.

The following steps are performed during final linking:

1. First the Completer is called, unless suppressed by COMPLETE=OFF, to complete the bodies of all instances which are used by the main program and all packages which are used by the main program and which do not require a body.
2. Then the Pre-Linker, a component of the Linker for final linking, is executed. The Pre-Linker determines the compilation units that have to be linked together and a valid elaboration order and generates a code sequence to perform the elaboration.
3. Finally, the Linker is called. §6.4 describes the mapping done by the Linker.

The resulting executable program has two different entry points:

- Ada program entry point and
- kernel entry point

The Ada program entry point is the first instruction of the code sequence generated by the Pre-Linker. It is used if the program is started by the SYSTEAM Ada Starter or the SYSTEAM Ada Debugger (see §7.2). This is possible even if the KERNEL parameter is given.

The kernel entry point is the entry point of the startup routine of the Target Kernel specified by the KERNEL parameter. If KERNEL is not given, then the kernel entry point is not defined. It must be used if the program is started as a stand-alone program by any means other than the Starter or the Debugger (see §7.2).

If no errors are detected within the linking process, then the result of a final link is a program image file containing the following in an internal format:

- A code portion that contains the complete code necessary to execute the program on the target, except the code of the base collection.
- Base addresses and lengths of the regions actually occupied by the complete program (including the base collection).
- Checksums of the regions which contain code sections and which are actually occupied by the complete program (including the base collection).
- The kernel entry point (if defined) and the Ada program entry point (if defined). At least one entry point is defined.

- Debugging information if DEBUG=ON was specified.

This program image file serves as input for the Loader or the Debugger in order to load the code portion included in the file onto the target, or for the Starter or the Debugger in order to start the linked program, or for the Format Converter.

The Linker of the SYSTEAM Ada System delivers the status code WARNING on termination (see [VAX/VMS, DCL Dictionary, command EXIT]) if an error was detected during linking. A message corresponding to this code has not been defined; hence %NONAME-W-NOMSG is printed upon notification of a batch job terminated with this status.

6.2 Linking Collections

A set of Ada units and external units which can be linked separately is called a *collection*. Such a collection consists on one hand of all compilation units needed by any of the given library units and on the other hand of all given external units. All compilation units must successfully have been compiled or completed previously.

The code of a linked collection does not contain any unresolved references and can thus be loaded to the target and used by programs linked afterwards without any changes. In particular, this allows the code of a linked collection to be burnt into a ROM. Linking a collection is called *incremental linking.*

Contrary to final linking, incremental linking is not done selectively. Instead all code and data belonging to the collection is linked, because the Linker does not know which programs or collections will be linked on the collection as a base.

Incremental linking results in a *collection image file.* There is a code portion in this image file which, together with the code of the given base (if any), is the code of all Ada units and all external (assembler written) units that belong to the collection.

For linking a collection (incremental linking), the Linker is started by

```
@ADA:LINKINC  <result_file_name>  <directive_file_name> -
                                  [LIBRARY=<directory>] -
                                  [UNITS=<ada_name>[,...]] -
                                  [EXTERNAL=<filespec>[,...]] -
                                  [BASE=<filespec>] -
                                  [DEBUG=ON/OFF] -
                                  [MAP=<filespec>]
```

<result_file_name> is the name of the file which will contain the result of the incremental link. The file type .LOD is assumed if none is specified.

<directive_file_name> is the name of the file which contains the linker directives, which describe the target's memory regions and prescribe the mapping of code, data, stack, and heap sections into these regions. The file type .LID is assumed if none is specified. For its format see §6.3. The SYSTEAM Ada System is delivered with a directive file for the MVME133XT. It can be used for those applications that use the whole memory of the MVME133XT and do not require specific units to be linked into specific regions. If the user wants to use this file, he has to specify:

```
ADA:133
```

<directory> is the name of the program library which contains the main program; [.ADALIB] is assumed if this parameter is not specified.

The UNITS parameter specifies the Ada library units that are to be linked. <ada_name> denotes any library unit within the given program library. All specified library units together with their secondary units and all units needed by them must have been successfully compiled and completed (cf. Chapter 3).

The EXTERNAL parameter specifies files which contain the object code of those program units which are written in assembler. If several files are given, they must be separated by commas. Their default file type is .OBJ.

The BASE parameter specifies a collection image file (a file containing the result of a previous incremental link). Its default file type is .LOD. If a base is specified, then the incremental link is done on the base of the given file.

The DEBUG parameter (default: ON) has an effect only when the parameter BASE is *not* given. If DEBUG=ON, then the complete Debug Runtime System is linked to the collection. If DEBUG=OFF, then the complete Non-Debug Runtime System is linked to the collection.
Important: If a base is given (parameter BASE), then no Runtime System is linked because it is already part of the base collection.

The MAP parameter controls the generation of a Linker map. If a map file is specified, then the Linker map will be written into this file. Its default type is .MAP.

The UNITS and EXTERNAL parameters define a collection C as defined at the beginning of this section. If a base collection is specified (parameter BASE), then C is enlarged by all units belonging to this base collection. The units belonging to the base collection are identified by their names (Ada name of a library unit or name of the external unit) and by their compilation or assembly times. The Linker uses these to check whether a base unit is obsolete or not.

See §6.4 for the mapping process.

If no errors are detected within the linking process, then the result of an incremental link is a collection image file containing the following in an internal format:

- A code portion that contains the complete code of the linked collection, except the code of the base collection.
- Base addresses and lengths of the regions actually occupied by the complete collection (including the base collection).
- Checksums of the regions which contain code sections and which are actually occupied by the complete collection (including the base collection).
- The names of all library units as specified by the user (parameter UNITS) (including those of the base collection).
- The list of all compilation units and external units that belong to the complete linked collection (including the base collection), together with their compilation (resp. assembly) times.
- Information about all sections belonging to the complete linked collection (including the base collection), and about the symbols which they define and refer.

This collection image file serves as input for the Loader or the Debugger in order to load the code portion included in the file onto the target, or for the Linker as a base collection file, or for the Format Converter.

The Linker of the SYSTEAM Ada System delivers the status code WARNING on termination (see [VAX/VMS, DCL Dictionary, command EXIT]) if an error was detected during linking. A message corresponding to this code has not been defined; hence %NONAME-W-NOMSG is printed upon notification of a batch job terminated with this status.

6.3 Linker Directives

The Linker always needs a directive file containing a description of the target's memory regions and directives for the mapping of code, data, stack, and heap sections into these regions. The contents of this file must have the following format:

```
linker_directive_file ::=
            region_description +
            [reset_vector_directive]
            stack_directive
            [location_directive +]
            code_directive
            data_directive
            heap_directive
region_description ::= REGION region_name base_address size
```

```
base_address ::= hex_number
size ::= hex_number
hex_number ::=
          (0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
           A | B | C | D | E | F)+
reset_vector_directive ::= RESET region_name
stack_directive ::= STACK size region_name
location_directive ::=
          UNIT library_unit_name CODE region_name |
          UNIT library_unit_name DATA region_name |
          MODULE object_module_name CODE region_name |
          MODULE object_module_name DATA region_name
code_directive ::= CODE region_list
data_directive ::= DATA region_list
heap_directive ::= HEAP region_list
region_list ::= region_name [(, region_name)+]
```

The syntax is specified in an extended Backus-Naur notation with start symbol linker_directive_file. [X] means that X is optional, X + means that X is repeated several times (but at least once), X | Y means that X or Y is used.

All characters are case insensitive. Hexadecimal numbers must be in the range 0 .. FFFFFFFF. region_name, library_unit_name, and object_module_name can be any sequence of readable characters except comma and blank.

The user has to specify all contiguous memory regions of the target that are to be used for the program or the collection to be linked. Each REGION description defines the name, the base address, and the size in bytes of one region.

The RESET directive specifies a region whose first 8 bytes are to be reserved for the initial program counter and the initial stack pointer. This directive supports the generation of ROMable programs: If a hardware reset occurs, then the processor fetches its reset vector from the start address of the given region. The RESET directive is ignored if KERNEL is not specified.

The STACK directive tells the Linker the size of the main task's stack and the name of the region into which the stack is to be mapped.

It is possible to specify specific regions for the code or the data of Ada library units or of external (assembler written) units in LOCATION directives. If a region for a library unit is specified, this causes this unit and all its secondary units to be mapped into this region. A region for the code or data of a library unit or of an external unit must not be specified more than once.

In the CODE directive a list of regions must be specified to be used for the code and the constants of those units for which no LOCATION CODE directive is given. The specified regions are filled in the given order.

In the DATA directive a list of regions must be specified to be used for the data of those units for which no LOCATION DATA directive is given. The specified regions are filled in the given order.

A list of regions must be given to be used for the heap of the program (HEAP directive).

The following objects are allocated on the heap:

- All Ada collections for which no length clause is specified
- The storage for a task activation (see §10.3)
- All task control blocks (see §10.3)

Enough space for these objects must be allocated; otherwise `storage_error` will be raised when the heap space is exhausted.

The directives for the MVME133XT that is delivered with the Ada System are:

```
REGION main 00010000 003E8000
STACK 40000 main
CODE main
DATA main
HEAP main
```

These directives can be found in the file `ADA:133.LID`. They cause the Linker to use one contiguous region in the address range 16#1_0000# .. 16#3F7_FFFF#. This region contains 4000 kByte. The MVME133XT board has a 4 MByte local memory (4MByte = 4096 kByte). The lowest 64 kByte are left for the data of the target's monitor and the highest 32 kByte for the Full Target Kernel.

256 kByte are allocated for the stack.

The user is free to change these directives. He has to take care that space for the Full Target Kernel is reserved (if it is used).

6.4 Mapping Process

The Linker maps a set S of sections, each belonging to a compilation unit or to an external (assembler) unit or to the code sequence generated by the Pre-Linker, into the target's memory regions. In doing this the Linker has to take into consideration the parameters and directives given by the user.

S is determined as follows:

- Final link: S contains only those sections that are transitively referred by the sections defining the Ada program entry point and the kernel entry point. This is called *selective linking*.
- Incremental link: S contains *all* sections belonging to the collection that is to be linked.

If a base collection is given (parameter BASE), then S is diminished by excluding those base sections that can be reused. A section of the base collection cannot be reused

- if it belongs to a compilation unit that was recompiled since the base collection was linked or
- if it belongs to an external unit that has an assembly time different from that in the base collection or
- if it contains a reference to a section of the base collection that cannot be reused.

If a section of the base collection cannot be reused, then it is extracted from the given Ada library and included in the resulting image file. Nevertheless, the size of the resulting image may be reduced drastically when a base collection is given, even if memory space occupied by the "old" section is not reused.

The Linker automatically takes care that the regions specified by the user in the REGION descriptions of the directive file do not overlap with the regions actually occupied by the given base collection. If the Linker detects an overlap, then it issues a warning and changes the user defined region description so that it does not overlap with a base region any longer.

The mapping of the sections of S into the regions proceeds as follows:

1. Final link only: If there is a RESET directive, then space for the initial program counter and for the initial stack pointer is reserved at the bottom of the given region.
2. Final link only: The stack is mapped into the specified region with the given size. If the given region has not enough space for the stack, then an error message is issued.
3. The LOCATION directives are processed in the order in which they appear in the Linker's directive file. Each directive is treated as follows: If the specified library unit or external unit is not part of the resulting program (e.g. as a consequence of selective linking), then the directive is ignored and a warning is issued. Otherwise all memory sections belonging either to the given library unit or one of its secondary units or to the given external unit, and containing code or data (as given in the directive), are mapped into the given region. If the given region has not enough space left for this mapping, then an error message is issued.
4. Then all sections not yet mapped are processed in an arbitrary order. If a section contains code or constants, then the regions specified in the CODE directive are scanned in the given order and the section is mapped into the first region that

has enough space left. If the section is a data section, then the same is done with the regions specified in the DATA directive. If no region is found that has enough space left, then an error message is issued.

Now each region is filled without any gaps, beginning at its base address. The sections which are mapped into a region are sorted as follows: First the stack, then code sections, then data sections. If there is any space left in a region, then this space is a contiguous byte block at the top of the region.

5. Final link only: The heap is located in those regions that have space of at least a certain minimum size (100 byte) left and are listed within the HEAP directive.
6. Final link only: If there is a RESET directive, then the values of the initial stack pointer and of the kernel entry point are written into the first 8 bytes of the given region.

If the MAP parameter was given, then the result of this mapping is written into the specified map file. The map file is generated even if errors were detected during linking. The information written into the map file has the following structure:

1. Header (including link time and the information whether the linking process was successful or not)
2. Map of the generated code portion
3. List of all sections that could not be mapped (only in case of an unsuccessful link)
4. List of used base sections (only for final link and if BASE was given)
5. Symbol table
6. Ada program entry point and kernel entry point (only for final link)
7. Elaboration order of Ada units (only for final link)