

AD-A191 638

ADA (TRADE NAME) COMPILER VALIDATION SUMMARY REPORT:
TANDEM COMPUTERS INC. (U) AERONAUTICAL SYSTEMS DIV
WRIGHT-PATTERSON AFB OH ADA VALIDITY.. 28 MAR 87

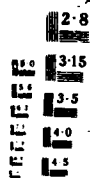
1/1

UNCLASSIFIED

P/C 12/3

NE

END
DATE
FILMED
87



UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

DTIC FILE COPY

2

AD-A191 638

INITIATION PAGE

12 GOVT ACCESSION NO.

3. RECIPIENT'S CATALOG NUMBER

1. TITLE (and Subtitle)

Ada Compiler Validation Summary Report:
Tandem Computers. Tandem Ada Compiler, Release
T9270C00. Tandem NonStop VLX host and target.Ver.C00

5. TYPE OF REPORT & PERIOD COVERED
20 May 1987 to 20 May 1988

6. PERFORMING ORG. REPORT NUMBER

7. AUTHOR(s)

Wright-Patterson AFB

8. CONTRACT OR GRANT NUMBER(s)

9. PERFORMING ORGANIZATION AND ADDRESS

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

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
20 May 1987

13. NUMBER OF PAGES
92

14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)

Wright-Patterson

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 06 1988
S 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)

See Attached

DD FORM

1473

EDITION OF 1 NOV 65 IS OBSOLETE

1 JAN 73

S/N 0102-LF-014-6601

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

EXECUTIVE SUMMARY

This Validation Summary Report (VSR) summarizes the results and conclusions of validation testing performed on the Tandem Ada® Compiler, Release T9270C00, using Version 1.8 of the Ada Compiler Validation Capability (ACVC). The Tandem Ada Compiler is hosted on a Tandem NonStop® VLX operating under GUARDIAN 90, Version C00. Programs processed by this compiler may be executed on a Tandem NonStop VLX operating under GUARDIAN 90, Version C00.

On-site testing was performed 15 May 1987 through 20 May 1987 at Tandem Computers, Cupertino CA, under the direction of the Ada Validation Facility, (AVF), according to Ada Validation Organization (AVO) policies and procedures. The AVF identified 2222 of the 2399 tests in ACVC Version 1.8 to be processed during on-site testing of the compiler. The 19 tests withdrawn at the time of validation testing, as well as the 158 executable tests that make use of floating-point precision exceeding that supported by the implementation, were not processed. After the 2222 tests were processed, results for Class A, C, D, and E tests were examined for correct execution. Compilation listings for Class B tests were analyzed for correct diagnosis of syntax and semantic errors. Compilation and link results of Class L tests were analyzed for correct detection of errors. There were 38 of the processed tests determined to be inapplicable. The remaining 2184 tests were passed.

The results of validation are summarized in the following table:

RESULT	CHAPTER													TOTAL
	2	3	4	5	6	7	8	9	10	11	12	14		
Passed	103	259	339	246	161	97	137	260	124	32	217	210	2184	
Failed	0	0	0	0	0	0	0	0	0	0	0	0	0	
Inapplicable	13	67	81	1	0	0	2	2	6	0	1	23	196	
Withdrawn	0	5	5	0	0	1	1	2	4	0	1	0	19	
TOTAL	116	330	425	247	161	98	140	264	134	32	219	233	2399	

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

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

®NonStop is a trademark of Tandem Computers

AVF Control Number: AVF-VSR-111.0787
86-12-01-TAN

Ada® COMPILER
VALIDATION SUMMARY REPORT:
Tandem Computers
Tandem Ada Compiler, Release T9270C00
Tandem NonStop® VLX host and target

Completion of On-Site Testing:
20 May 1987

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

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

[illegible]

©Ada is a registered trademark of the United States Government
(Ada Joint Program Office).
©NonStop is a trademark of Tandem Computers

+++++
+
+ Place NTIS form here +
+
+++++

Ada[®] Compiler Validation Summary Report:

Compiler Name: Tandem Ada Compiler, Release T9270C00

Host:	Target:
Tandem NonStop [™] VLX under GUARDIAN 90, Version C00	Tandem NonStop VLX under GUARDIAN 90, Version C00

Testing Completed 20 May 1987 Using ACVC 1.8

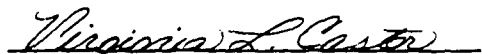
This report has been reviewed and is approved.



Ada Validation Facility
Georgeanne Chitwood
ASD/SCOL
Wright-Patterson AFB OH 45433-6503



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



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

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

[™]NonStop is a trademark of Tandem Computers

EXECUTIVE SUMMARY

This Validation Summary Report (VSR) summarizes the results and conclusions of validation testing performed on the Tandem Ada[®] Compiler, Release T9270C00, using Version 1.8 of the Ada Compiler Validation Capability (ACVC). The Tandem Ada Compiler is hosted on a Tandem NonStop[®] VLX operating under GUARDIAN 90, Version C00. Programs processed by this compiler may be executed on a Tandem NonStop VLX operating under GUARDIAN 90, Version C00.

On-site testing was performed 15 May 1987 through 20 May 1987 at Tandem Computers, Cupertino CA, under the direction of the Ada Validation Facility (AVF), according to Ada Validation Organization (AVO) policies and procedures. The AVF identified 2222 of the 2399 tests in ACVC Version 1.8 to be processed during on-site testing of the compiler. The 19 tests withdrawn at the time of validation testing, as well as the 158 executable tests that make use of floating-point precision exceeding that supported by the implementation, were not processed. After the 2222 tests were processed, results for Class A, C, D, and E tests were examined for correct execution. Compilation listings for Class B tests were analyzed for correct diagnosis of syntax and semantic errors. Compilation and link results of Class L tests were analyzed for correct detection of errors. There were 38 of the processed tests determined to be inapplicable. The remaining 2184 tests were passed.

The results of validation are summarized in the following table:

RESULT	CHAPTER													TOTAL
	2	3	4	5	6	7	8	9	10	11	12	14		
Passed	103	258	339	246	161	97	137	260	124	32	217	210	2184	
Failed	0	0	0	0	0	0	0	0	0	0	0	0	0	
Inapplicable	13	67	81	1	0	0	2	2	6	0	1	23	196	
Withdrawn	0	5	5	0	0	1	1	2	4	0	1	0	19	
TOTAL	116	330	425	247	161	98	140	264	134	32	219	233	2399	

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

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

[®]NonStop is a trademark of Tandem Computers

TABLE OF CONTENTS

CHAPTER 1	INTRODUCTION	
1.1	PURPOSE OF THIS VALIDATION SUMMARY REPORT	1-2
1.2	USE OF THIS VALIDATION SUMMARY REPORT	1-2
1.3	REFERENCES	1-3
1.4	DEFINITION OF TERMS	1-3
1.5	ACVC TEST CLASSES	1-4
CHAPTER 2	CONFIGURATION INFORMATION	
2.1	CONFIGURATION TESTED	2-1
2.2	IMPLEMENTATION CHARACTERISTICS	2-2
CHAPTER 3	TEST INFORMATION	
3.1	TEST RESULTS	3-1
3.2	SUMMARY OF TEST RESULTS BY CLASS	3-1
3.3	SUMMARY OF TEST RESULTS BY CHAPTER	3-2
3.4	WITHDRAWN TESTS	3-2
3.5	INAPPLICABLE TESTS	3-2
3.6	SPLIT TESTS	3-4
3.7	ADDITIONAL TESTING INFORMATION	3-5
3.7.1	Prevalidation	3-5
3.7.2	Test Method	3-5
3.7.3	Test Site	3-5
APPENDIX A	DECLARATION OF CONFORMANCE	
APPENDIX B	APPENDIX F OF THE Ada STANDARD	
APPENDIX C	TEST PARAMETERS	
APPENDIX D	WITHDRAWN TESTS	

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 characteristics of particular operating systems, hardware, or implementation strategies. All of the dependencies observed during the process of testing this compiler are given in this report.

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

INTRODUCTION

1.1 PURPOSE OF THIS VALIDATION SUMMARY REPORT

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

- . To attempt to identify any language constructs supported by the compiler that do not conform to the Ada Standard
- . To attempt to identify any unsupported language constructs required by the Ada Standard
- . To determine that the implementation-dependent behavior is allowed by the Ada Standard

Testing of this compiler was conducted by SofTech, Inc., under the direction of the AVF according to policies and procedures established by the Ada Validation Organization (AVO). On-site testing was conducted from 15 May 1987 through 20 May 1987 at Tandem Computers, Cupertino CA.

1.2 USE OF THIS VALIDATION SUMMARY REPORT

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

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

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

or from:

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

INTRODUCTION

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

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

1.3 REFERENCES

1. Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815A, February 1983.
2. Ada Validation Organization: Procedures and Guidelines, Ada Joint Program Office, 1 January 1987.
3. Ada Compiler Validation Capability Implementers' Guide, SofTech, Inc., December 1984.

1.4 DEFINITION OF TERMS

ACVC	The Ada Compiler Validation Capability. A set of programs that evaluates the conformity of a compiler to the Ada language specification, ANSI/MIL-STD-1815A.
Ada Standard	ANSI/MIL-STD-1815A, February 1983.
Applicant	The agency requesting validation.
AVF	The Ada Validation Facility. In the context of this report, the AVF is responsible for conducting compiler validations according to established policies and procedures.
AVO	The Ada Validation Organization. In the context of this report, the AVO is responsible for setting procedures for compiler validations.
Compiler	A processor for the Ada language. In the context of this report, a compiler is any language processor, including cross-compilers, translators, and interpreters.
Failed test	A test for which the compiler generates a result that demonstrates nonconformity to the Ada Standard.
Host	The computer on which the compiler resides.

INTRODUCTION

Inapplicable test	A 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	A test for which a compiler generates the expected result.
Target	The computer for which a compiler generates code.
Test	A program that checks a compiler's conformity regarding a particular feature or 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	A test found to be incorrect and not used to check conformity to the Ada language specification. A test may be incorrect because it has an invalid test objective, fails to meet its test objective, or contains illegal or erroneous use of the language.

1.5 ACVC TEST CLASSES

Conformity to the Ada Standard is measured using the ACVC. The ACVC contains both legal and illegal Ada programs structured into six test classes: A, B, C, D, E, and L. The first letter of a test name identifies the class to which it belongs. Class A, C, D, and E tests are executable, and special program units are used to report their results during execution. Class B tests are expected to produce compilation errors. Class L tests are expected to produce link errors.

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

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

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

Class D tests check the compilation and execution capacities of a compiler. Since there are no capacity requirements placed on a compiler by the Ada Standard for some parameters--for example, the number of identifiers

INTRODUCTION

permitted in a compilation or the number of units in a library--a compiler may refuse to compile a Class D test and still be a conforming compiler. Therefore, if a Class D test fails to compile because the capacity of the compiler is exceeded, the test is classified as inapplicable. If a Class D test compiles successfully, it is self-checking and produces a PASSED or FAILED message during execution.

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

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

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

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

A compiler must correctly process each of the tests in the suite and demonstrate conformity to the Ada Standard by either meeting the pass criteria given for the test or by showing that the test is inapplicable to the implementation. The applicability of a test to an implementation is considered each time the implementation is validated. A test that is inapplicable for one validation is not necessarily inapplicable for a subsequent validation.

INTRODUCTION

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 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: Tandem Ada Compiler, Release T9270C00

ACVC Version: 1.8

Certificate Number: 870515W1.08089

Host Computer:

Machine:	Tandem NonStop VLX
Operating System:	GUARDIAN 90 Version C00
Memory Size:	8 megabytes

Target Computer:

Machine:	Tandem NonStop VLX
Operating System:	GUARDIAN 90 Version C00
Memory Size:	8 megabytes

CONFIGURATION INFORMATION

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. This compiler is characterized by the following interpretations of the Ada Standard:

- . Capacities.

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

- . Universal integer calculations.

An implementation is allowed to reject universal integer calculations having values that exceed `SYSTEM.MAX_INT`. This implementation does not reject such calculations and processes them correctly. (See tests D4A002A, D4A002B, D4A004A, and D4A004B.)

- . Predefined types.

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

- . Based literals.

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

- . 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`.

CONFIGURATION INFORMATION

A packed BOOLEAN array having a 'LENGTH exceeding INTEGER'LAST raises STORAGE_ERROR when the array objects are declared. (See test C52103X.)

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

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

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

. Discriminated types.

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

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

. Aggregates.

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

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

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

CONFIGURATION INFORMATION

- Functions.

An implementation may allow the declaration of a parameterless function and an enumeration literal having the same profile in the same immediate scope, or it may reject the function declaration. If it accepts the function declaration, the use of the enumeration literal's identifier denotes the function. This implementation rejects the declaration. (See test E66001D.)

- Representation clauses.

The Ada Standard does not require an implementation to support representation clauses. If a representation clause is not supported, then the implementation must reject it. While the operation of representation clauses is not checked by Version 1.8 of the ACVC, they are used in testing other language features. This implementation accepts 'SIZE and 'STORAGE_SIZE for tasks, and 'SMALL clauses; it rejects 'STORAGE_SIZE for collections. Enumeration representation clauses appear not to be supported. (See tests C55B16A, C87B62A, C87B62B, C87B62C, and BC1002A.)

- Pragmas.

The pragma INLINE is supported for procedures and functions. (See tests CA3004E and CA3004F.)

- Input/output.

The package SEQUENTIAL_IO cannot be instantiated with unconstrained array types and record types with discriminants without defaults. The package DIRECT_IO cannot be instantiated with unconstrained array types and record types with discriminants without defaults. (See tests AE2101C, AE2101H, CE2201D, CE2201E, and CE2401D.)

An existing text file can be opened in OUT_FILE mode, but cannot be created in OUT_FILE or IN_FILE mode. (See test EE3102C.)

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

More than one internal file can be associated with each external file for sequential I/O for reading only. (See tests CE2107A..F (6 tests).)

More than one internal file can be associated with each external file for direct I/O for reading only. (See tests CE2107A..F (6 tests).)

CONFIGURATION INFORMATION

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

. Generics.

Separate compilation of generic bodies without specifications is not supported by this implementation. (See tests BA1011C, CA1012A, CA2009C, CA2009F and BC3205D.)

CHAPTER 3

TEST INFORMATION

3.1 TEST RESULTS

Version 1.8 of the ACVC contains 2399 tests. When validation testing of Tandem Ada Compiler was performed, 19 tests had been withdrawn. The remaining 2380 tests were potentially applicable to this validation. The AVF determined that 196 tests were inapplicable to this implementation, and that the 2184 applicable tests were passed by the implementation.

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	67	865	1178	17	13	44	2184
Failed	0	0	0	0	0	0	0
Inapplicable	2	2	190	0	0	2	196
Withdrawn	0	7	12	0	0	0	19
TOTAL	69	874	1380	17	13	46	2399

TEST INFORMATION

3.3 SUMMARY OF TEST RESULTS BY CHAPTER

RESULT	CHAPTER													TOTAL
	2	3	4	5	6	7	8	9	10	11	12	14		
Passed	103	258	339	246	161	97	137	260	124	32	217	210	2184	
Failed	0	0	0	0	0	0	0	0	0	0	0	0	0	
Inapplicable	13	67	81	1	0	0	2	2	6	0	1	23	196	
Withdrawn	0	5	5	0	0	1	1	2	4	0	1	0	19	
TOTAL	116	330	425	247	161	98	140	264	134	32	219	233	2399	

3.4 WITHDRAWN TESTS

The following 19 tests were withdrawn from ACVC Version 1.8 at the time of this validation:

C32114A	C41404A	B74101B
B33203C	B45116A	C87B50A
C34018A	C48008A	C92005A
C35904A	B49006A	C940ACA
B37401A	B4A010C	CA3005A..D (4 tests)
		BC3204C

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 is not necessarily inapplicable for a subsequent attempt. For this validation attempt, 196 tests were inapplicable for the reasons indicated:

- C34001F and C35702A use SHORT_FLOAT which is not supported by this compiler.
- C43212A makes assumptions about the order of operations during the evaluation of an array aggregate. This implementation raises CONSTRAINT_ERROR after evaluating two of the lower bounds and determining that they don't match, while the test considers it an

TEST INFORMATION

error if at least two of the upper bounds aren't also evaluated. The AVO has ruled that this is an acceptable behavior.

- . C55B16A makes use of an enumeration representation clause containing noncontiguous values which is not supported by this compiler.
- . C86001F redefines package SYSTEM, but TEXT_IO is made obsolete by this new definition in this implementation and the test cannot be executed since the package REPORT is dependent on the package TEXT_IO.
- . C87B62B uses a length clause which is not supported by this compiler. The 'STORAGE_SIZE length clause for access types is rejected during compilation.
- . C92005B is inapplicable because in this implementation a task's STORAGE_SIZE attribute yields a value greater than STANDARD.INTEGER'LAST.
- . C96005B checks implementations for which the smallest and largest values in type DURATION are different from the smallest and largest values in DURATION's base type. This is not the case for this implementation.
- . BA1011C, CA1012A, CA2009C, CA2009F, BC3205D, LA5008M, and LA5008N compile generic declarations and bodies in separate compilation units. Separate compilation of generic bodies without specifications is not supported by this implementation.
- . AE2101C, CE2201D, and CE2201E use an instantiation of package SEQUENTIAL_IO with unconstrained array types which is not supported by this compiler.
- . AE2101H and CE2401D use an instantiation of package DIRECT_IO with unconstrained array types which is not supported by this compiler.
- . CE2107B..E (4 tests), CE2110B, CE2111H, CE3111B..E (4 tests), and CE3114B are inapplicable because multiple internal files cannot be associated with the same external file except for reading. The proper exception is raised when multiple access is attempted.
- . CE2102D, CE2102I, CE2105A, and CE2407A create a file with mode IN_FILE which is not supported by this implementation.
- . CE2111D and CE3115A create a file with mode OUT_FILE, reset the file to mode IN_FILE, and then try to open the file again in mode IN_FILE. This implementation retains the file's access rights from when it is opened until the file is closed. When a file is created with mode OUT_FILE, this implementation prohibits multiple access to this file until it is closed. When the test resets the file to mode IN_FILE and attempts to open a second internal file to the same external file, this implementation raises USE_ERROR.

TEST INFORMATION

This behavior has been determined to be acceptable by the AVO.

- CE3605A writes about 300 characters to a line in a text file. This implementation raises USE_ERROR because this is longer than the acceptable line length imposed by the operating system for this file type. This implementation raises USE_ERROR on the subsequent NEW_LINE, which is when they physically try to write the line to the file. The issue will be raised before the LMP (AI-00534), and the AVO has provisionally ruled that this behavior is acceptable. When the form parameter (FORM => "FILE_TYPE=R, RECORDLEN=500") was added to the test, the implementation passes the test objective.
- The following 158 tests require a floating-point accuracy that exceeds the maximum of 16 supported by the implementation:

C24113M..Y (13 tests)	C35705M..Y (13 tests)
C35706M..Y (13 tests)	C35707M..Y (13 tests)
C35708M..Y (13 tests)	C35802M..Y (13 tests)
C45241M..Y (13 tests)	C45321M..Y (13 tests)
C45421M..Y (13 tests)	C45424M..Y (13 tests)
C45521M..Z (14 tests)	C45621M..Z (14 tests)

3.6 SPLIT TESTS

If one or more errors do not appear to have been detected in a Class B test because of compiler error recovery, then the test is split into a set of smaller tests that contain the undetected errors. These splits are then compiled and examined. The splitting process continues until all errors are detected by the compiler or until there is exactly one error per split. Any Class A, Class C, or Class E test that cannot be compiled and executed because of its size is split into a set of smaller subtests that can be processed.

Splits were required for 35 Class B tests:

B22003A	B24005A	B33301A	B51003A
B22004A	B24005B	E35101A	B55A01A
B22004B	B24204A	B36201A	B64001A
B22004C	B24204B	B37201A	B67001A
B23004A	B26002A	B37307B	B67001B
B23004B	B29001A	B38008A	B67001C
B24001A	B2AC03A	B41202A	B67001D
B24001B	B2AC03B	B44001A	B91003B
B24001C	B2AC03C	B45205A	

3.7 ADDITIONAL TESTING INFORMATION

3.7.1 Prevalidation

Prior to validation, a set of test results for ACVC Version 1.8 produced by the Tandem Ada Compiler was submitted to the AVF by the applicant for review. Analysis of these results demonstrated that the compiler successfully passed all applicable tests, and that the compiler exhibited the expected behavior on all inapplicable tests.

3.7.2 Test Method

Testing of the Tandem Ada Compiler using ACVC Version 1.8 was conducted on-site by a validation team from the AVF. The configuration consisted of a Tandem NonStop VLX operating under GUARDIAN 90, Version C00.

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 splits during the prevalidation testing were included in their split form on the magnetic tape.

The contents of the magnetic tape were loaded directly onto the Tandem NonStop VLX, Version B40 computer and read using a special program written by Tandem for reading ANSI tape format. The files were then written to tape using tandem backup format and transferred to the Tandem NonStop VLX, Version C00 computer. After the test files were loaded to disk, the full set of tests was compiled on the Tandem NonStop VLX, and all executable tests were linked and run on the Tandem NonStop VLX using four parallel batch streams. Results were put on tape, using standard Tandem utilities. A special Ada program was used to print hard copies of the results. The results were printed using any system that had available printer resources.

The compiler was tested using command scripts provided by Tandem Computers and reviewed by the validation team.

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

The validation team arrived at Tandem Computers, Cupertino CA on 15 May 1987, and departed after testing was completed on 20 May 1987.

APPENDIX A

DECLARATION OF CONFORMANCE

Tandem Computers has submitted the following
declaration of conformance concerning the Tandem Ada
Compiler.

DECLARATION OF CONFORMANCE

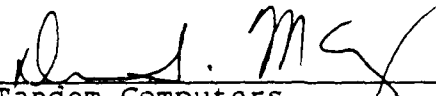
Compiler Implementor: Tandem Computers Incorporated (Tandem Computers)
Ada Validation Facility: ASD/SCOL, Wright-Patterson AFB, OH
Ada Compiler Validation Capability (ACVC) Version: 1.8

Base Configuration

Base Compiler Name: Tandem Ada Compiler Version: Release T9270C00
Host Architecture ISA: Tandem NonStop TM VLX
OS&VER #1: GUARDIAN 90, Version C00
Target Architecture ISA: Tandem NonStop TM VLX
OS&VER #1: GUARDIAN 90, Version C00

Implementor's Declaration

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



Tandem Computers
Dennis McEvoy, Vice President
Software Development

Date: 5-19-87

Owner's Declaration

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



Tandem Computers
Dennis McEvoy, Vice President
Software Development

Date: 5-19-87

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

®NonStop is a trademark of Tandem Computers

APPENDIX B

APPENDIX F OF THE Ada STANDARD

The only allowed implementation dependencies correspond to implementation-dependent pragmas, to certain machine-dependent conventions as mentioned in chapter 13 of MIL-STD-1815A, and to certain allowed restrictions on representation clauses. The implementation-dependent characteristics of the Tandem Ada Compiler, Release T9270C00, are described in the following sections which discuss topics in Appendix F of the Ada Language Reference Manual (ANSI/MIL-STD-1815A). Implementation-specific portions of the package STANDARD are also included in this appendix.

APPENDIX F

IMPLEMENTATION-DEPENDENT CHARACTERISTICS

This appendix contains the Tandem implementation dependencies for Ada programming. It is also Appendix F for the *ANSI Reference Manual for the Ada Programming Language* (ANSI/MIL-STD-1815A, January 1983). That manual does not include this appendix, which appears only in this user's guide.

The implementation dependencies this appendix describes include the following:

- The form, allowed places, and effects of all implementation-dependent pragmas
- The names and types of all implementation-dependent attributes
- The specification of the package `SYSTEM`
- Implementation-defined aspects of the package `STANDARD`
- Implementation-defined predefined packages
- Restrictions on representation clauses
- Semantics of representation attributes
- Restrictions on unchecked conversions
- Information on input and output features
- Information about parameters and function returns for external subprograms
- Rules for compiling generic units
- Implementation-defined limits

IMPLEMENTATION-DEPENDENT CHARACTERISTICS

Pragmas

PRAGMAS

Tandem Ada includes an implementation-defined pragma and has restrictions on several predefined pragmas. This subsection lists the additional pragma and the pragmas that have restrictions, with descriptions.

For information about using pragmas, see sections 3 and 8. For complete information about predefined pragmas for which Tandem Ada has no restrictions, see the *ANSI Reference Manual for the Ada Programming Language*.

Implementation-Defined Pragma

The `EXTERNAL_NAME` pragma, which is implementation-defined, associates a TAL procedure name with the last Ada subprogram declared before the pragma in the source file. `EXTERNAL_NAME` takes two arguments: the simple name of an Ada subprogram and a string. You can use this pragma at the place of a declarative item.

The pragma must apply to the last subprogram declared before the pragma in the same declarative part or package specification. You can not use this pragma for a library unit. The string is the TAL name of the subprogram.

The second argument to `EXTERNAL NAME` should not start with `RSL^` because the compiler reserves all external subprogram names starting with `RSL^` for run-time support routines. If the second argument does start with `RSL^`, the compiler issues a warning message.

Restrictions on Predefined Pragmas

The following list summarizes the restrictions on predefined pragmas for the Tandem Ada compiler:

CONTROLLED This pragma has no effect. Everything is controlled.

INLINE This pragma has the following effects:

It allows the compiler to expand subprogram calls in line.

If the compiler does not expand a subprogram call in line when the latest compilation of the subprogram specification included the INLINE pragma, then the compiler issues one of the following warning messages at each call site:

The call to this Inline subprogram is not expanded because its body is not available.

The call to this Inline subprogram is not expanded because its return type is unconstrained.

The call to this Inline subprogram is not expanded because it is either recursive or mutually recursive.

An in-line expansion of a subprogram call creates a compilation dependency on the body of the called subprogram.

If you do not specify the INLINE pragma for a subprogram, the compiler might expand that subprogram call in line if the expansion does not create any compilation dependencies and if you use the OPTIMIZE switch for the compilation. The effect of the optimize switch is an average of the effects of the SPACE and TIME settings of pragma INLINE. The compiler does not expand calls to derived subprograms.

The compiler can expand a call to a recursive subprogram. However, the compiler does not expand the subprogram's second call to itself.

IMPLEMENTATION-DEPENDENT CHARACTERISTICS
Restrictions on Predefined Pragmas

INTERFACE

The only language you can specify is TAL. An Ada program can call a TAL procedure only if the procedure could have a legal Ada subprogram specification. This means an Ada program cannot call any TAL function that has an out parameter or any TAL procedure that has optional parameters.

You cannot specify the INTERFACE pragma for renamed subprograms or library subprograms.

Without a corresponding pragma EXTERNAL_NAME, the subprogram name in the INTERFACE pragma must be the same as the TAL procedure name.

A TAL procedure for an Ada procedure cannot have a return value, and a TAL procedure for an Ada function must have a return value.

MEMORY_SIZE

This pragma is reserved for development use in compiling the base library.

OPTIMIZE

This pragma does not apply to any block or body nested within the block or body whose declarative part contains the pragma. This gives you greater control because you can specify the OPTIMIZE pragma for individual nested blocks or bodies when you want to. You can specify the OPTIMIZE pragma at more than one place in a declarative part.

If you do not specify the OPTIMIZE switch in the ADA command for a compilation, the compiler does not expand subprogram calls in line unless a pragma INLINE applies to the subprogram. If you do specify the OPTIMIZE switch in the ADA command, the compiler might expand other subprogram calls in line where the expansion does not create additional compilation dependencies. The OPTIMIZE pragma provides additional control over when the compiler performs in line expansion.

This pragma takes one parameter, either SPACE or TIME. If you specify SPACE, then in-line expansion happens only where it does not increase the generated code size. If you specify TIME, in-line expansion might occur more often for subprograms of moderate size.

If you do not specify the OPTIMIZE pragma, the effect is in between the results from SPACE and TIME. If you specify the pragma more than once for a unit of code using SPACE in one instance and TIME in another, the effect is the same as if you did not specify the pragma.

PACK	This pragma has no effect on data layout. If you use this pragma, the compiler issues a warning message.
STORAGE_UNIT	This pragma is reserved for development use in compiling the base library.
SUPPRESS	This pragma has no effect on the suppression or generation of checking code. If you use this pragma, the compiler issues a warning message.
SYSTEM_NAME	This pragma is reserved for development use in compiling the base library.

ATTRIBUTES

Tandem Ada has no implementation-defined attributes. For the semantics of predefined attributes, see "Representation Attributes," later in this appendix.

PREDEFINED PACKAGES

In addition to the packages SYSTEM and STANDARD and the standard input-output packages, Tandem Ada has predefined packages for NonStop systems:

- COMMAND_INTERPRETER_INTERFACE, which reads the startup, assign, and param messages for the Ada process. It also provides procedures and functions that the process can use to get information from these messages.
- SYSTEM_CALLS, which enables Ada programs to set completion codes.

IMPLEMENTATION-DEPENDENT CHARACTERISTICS
Package SYSTEM

Package SYSTEM

The specification of package SYSTEM for the Tandem Ada compiler on NonStop systems follows:

package SYSTEM is

 type ADDRESS is private;

 type NAME is (NONSTOP);

 SYSTEM_NAME : constant NAME := NONSTOP;

 STORAGE_UNIT : constant := 8;

 MEMORY_SIZE : constant := 2 ** 30;

 -- System-dependent named numbers:

 MIN_INT : constant := -9_223_372_036_854_775_808;

 MAX_INT : constant := +9_223_372_036_854_775_807;

 MAX_DIGITS : constant := 16;

 MAX_MANTISSA : constant := 31;

 FINE_DELTA : constant := 2.0 ** (-31);

 TICK : constant := 0.01;

 -- Other system-dependent declaration:

 subtype PRIORITY is INTEGER range 0 .. -1;

private

 .

 .

 .

end SYSTEM;

Package STANDARD

In the predefined package STANDARD, each implementation defines the number of predefined numeric types, their names, and their representation attributes.

Tandem's Ada compiler supports the following predefined numeric types:

```
type SHORT_INTEGER is range -2 ** 7 .. 2 ** 7 - 1;
for SHORT_INTEGER'SIZE use 8;
```

```
type INTEGER is range -2 ** 15 .. 2 ** 15 - 1;
for INTEGER'SIZE use 16;
```

```
type LONG_INTEGER is range -2 ** 31 .. 2 ** 31 - 1;
for LONG_INTEGER'SIZE use 32;
```

```
type LONG_LONG_INTEGER is range -2 ** 63 .. 2 ** 63 - 1;
for LONG_LONG_INTEGER'SIZE use 64;
```

```
type FLOAT is digits 6 range -(2 ** 254 * (1 - 2 ** (-21))) ..
                                2 ** 254 * (1 - 2 ** (-21)));
-- range is -FLOAT'SAFE_LARGE .. FLOAT'SAFE_LARGE
for FLOAT'SIZE use 32;
```

```
type LONG_FLOAT is digits 16
    range -(2 ** 254 * (1 - 2 ** (-55))) ..
           2 ** 254 * (1 - 2 ** (-55));
-- range is -LONG_FLOAT'SAFE_LARGE .. LONG_FLOAT'SAFE_LARGE
for LONG_FLOAT'SIZE use 64;
```

```
type DURATION is delta 1 / 2 ** 14
    range -(2 ** 31 / 2 ** 14) ..
           (2 ** 31 - 1) / 2 ** 14;
for DURATION'SIZE use 64;
```

```
for BOOLEAN'SIZE use 8;
```

```
for CHARACTER'SIZE use 8;
```

IMPLEMENTATION-DEPENDENT CHARACTERISTICS
Package COMMAND_INTERPRETER_INTERFACE

Package COMMAND_INTERPRETER_INTERFACE

The COMMAND_INTERPRETER_INTERFACE package enables an Ada process to communicate with an operating system command interpreter. The specification for this package follows, with comments that describe the information Ada processes can get from the command interpreter.

package COMMAND_INTERPRETER_INTERFACE is

```
-- This package reads the startup, assign, and param
-- messages and returns the specified information from
-- these messages. For details about the interface to
-- the operating system command interpreter, see the
-- GUARDIAN Operating System Programmer's Guide.
--
-- Elaboration code in this package reads the command
-- interpreter messages. To use this package in the
-- elaboration of another compilation unit, the dependent
-- unit should specify pragma ELABORATE for this package.
```

CANT_READ_MESSAGES : exception;

```
-- Raised by all routines if the Ada process could not
-- read the command interpreter messages.
```

FIELD_NOT_PRESENT : exception;

```
-- Raised when a field selection of an assign message is
-- absent.
```

type ASSIGN_MESSAGE_T is private;

NO_ASSIGN : constant ASSIGN_MESSAGE_T;

type FILE_EXCLUSION_T is (SHARED, EXCLUSIVE, PROTECTED);

type FILE_ACCESS_T is (IN_OUT, INPUT, OUTPUT);

subtype LOGICAL_FILENAME_T is STRING (1 .. 31);

type PARAM_MESSAGE_T is private;

NO_PARAM : constant PARAM_MESSAGE_T;

function GET_DEFAULT return STRING;

```
-- Returns the default volume and subvolume specified by
-- the startup message in the form "$VOL.SUBVOL".
```

IMPLEMENTATION-DEPENDENT CHARACTERISTICS
Package COMMAND_INTERPRETER_INTERFACE

```
function GET_INFILE return STRING;
-- Returns the IN file specified by the startup message
-- in the form "$VOL.SUBVOL.DNAME".

function GET_OUTFILE return STRING;
-- Returns the OUT file specified by the startup
-- message in the form "$VOL.SUBVOL.DNAME".

function GET_STARTUP_MESSAGE_PARAM return STRING;
-- Returns the parameter string specified in the RUN
-- command line from the startup message. The returned
-- string does not include any trailing null characters
-- with which the command interpreter padded the string.

procedure ASSIGN_LIST_RESET;
-- Resets the pointer to the first assign message.

function GET_NEXT_ASSIGN return ASSIGN_MESSAGE_T;
-- Returns the next message from the assign message list
-- or, if no message is left, returns NO_ASSIGN.

function SEARCH_ASSIGN (PROG_NAME : in STRING;
                        FILE_NAME : in STRING)
                        return ASSIGN_MESSAGE_T;
-- Searches the list of assign messages for the logical unit
-- specified. A match occurs when both the input program
-- name and file name are identical to those of an assign
-- message. Otherwise, the function returns NO_ASSIGN.

procedure GET_LOGICAL_UNIT_NAMES (ASSIGN : in ASSIGN_MESSAGE_T;
                                  PROG_NAME : out LOGICAL_FILENAME_T;
                                  PROG_NAME_LEN : out INTEGER;
                                  FILE_NAME : out LOGICAL_FILENAME_T;
                                  FILE_NAME_LEN : out INTEGER);
-- Returns the program name and file names of the logical
-- unit for the specified assign message.

function IS_TANDEM_FILENAME_PRESENT
(ASSIGN : ASSIGN_MESSAGE_T) return BOOLEAN;
-- Returns TRUE if the operating system file name is present or
-- FALSE otherwise.

function IS_PRI_EXTENT_PRESENT (ASSIGN : ASSIGN_MESSAGE_T)
                                return BOOLEAN;
-- Returns TRUE if the primary extent is present or FALSE
-- otherwise.
```

IMPLEMENTATION-DEPENDENT CHARACTERISTICS
Package COMMAND_INTERPRETER_INTERFACE

```
function IS_SEC_EXTENT_PRESENT (ASSIGN : ASSIGN_MESSAGE_T)
    return BOOLEAN;
-- Returns TRUE if the secondary extent is present or FALSE
-- otherwise.

function IS_FILECODE_PRESENT (ASSIGN : ASSIGN_MESSAGE_T)
    return BOOLEAN;
-- Returns TRUE if the file code is present or FALSE
-- otherwise.

function IS_EXCLUSION_PRESENT (ASSIGN : ASSIGN_MESSAGE_T)
    return BOOLEAN;
-- Returns TRUE if the exclusion spec is present or FALSE
-- otherwise.

function IS_ACCESS_SPEC_PRESENT (ASSIGN : ASSIGN_MESSAGE_T)
    return BOOLEAN;
-- Returns TRUE if the access spec is present or FALSE
-- otherwise.

function IS_RECORD_SIZE_PRESENT (ASSIGN : ASSIGN_MESSAGE_T)
    return BOOLEAN;
-- Returns TRUE if the record size is present or FALSE
-- otherwise.

function IS_BLOCK_SIZE_PRESENT (ASSIGN : ASSIGN_MESSAGE_T)
    return BOOLEAN;
-- Returns TRUE if the block size is present or FALSE
-- otherwise.

function GET_TANDEM_FILENAME (ASSIGN : ASSIGN_MESSAGE_T)
    return STRING;
-- Returns the operating system file name for the specified
-- assign message. Absence of the field raises
-- FIELD_NOT_PRESENT.

function GET_PRI_EXTENT (ASSIGN : ASSIGN_MESSAGE_T)
    return INTEGER;
-- Returns the primary extent for the specified assign
-- message. Absence of the field raises FIELD_NOT_PRESENT.

function GET_SEC_EXTENT (ASSIGN : ASSIGN_MESSAGE_T)
    return INTEGER;
-- Returns the secondary extent for the specified assign
-- message. Absence of the field raises FIELD_NOT_PRESENT.

function GET_FILECODE (ASSIGN : ASSIGN_MESSAGE_T)
    return INTEGER;
-- Returns the file code for the specified assign message.
-- Absence of the field raises FIELD_NOT_PRESENT.
```

```
function GET_EXCLUSION (ASSIGN : ASSIGN_MESSAGE_T)
    return FILE_EXCLUSION_T;
-- Returns the exclusion spec for the specified assign
-- message. Absence of the field raises FIELD_NOT_PRESENT.

function GET_ACCESS_SPEC (ASSIGN : ASSIGN_MESSAGE_T)
    return FILE_ACCESS_T;
-- Returns the access spec for the specified assign
-- message. Absence of the field raises FIELD_NOT_PRESENT.

function GET_RECORD_SIZE (ASSIGN : ASSIGN_MESSAGE_T)
    return INTEGER;
-- Returns the record size for the specified assign
-- message. Absence of the field raises FIELD_NOT_PRESENT.

function GET_BLOCK_SIZE (ASSIGN : ASSIGN_MESSAGE_T)
    return INTEGER;
-- Returns the block size for the specified assign message.
-- Absence of the field raises FIELD_NOT_PRESENT.

procedure PARAM_LIST RESET;
-- Resets the pointer to the beginning of the param
-- message list.

function GET_NEXT_PARAM return PARAM_MESSAGE_T;
-- Returns the next message from the param message list or,
-- if no message is left, returns NO_PARAM.

function SEARCH_PARAM_LIST (NAME : STRING)
    return PARAM_MESSAGE_T;
-- Searches the param message list for a param with the
-- specified name and returns the message for the param that
-- matches. If none matches, it returns NO_PARAM.

function GET_PARAM_NAME (PARAM : PARAM_MESSAGE_T)
    return STRING;
-- Returns the param name of the specified param message.

function GET_PARAM_VALUE (PARAM : PARAM_MESSAGE_T)
    return STRING;
-- Returns the value of the specified param message.

private
.
.
.
end COMMAND_INTERPRETER_INTERFACE;
```

IMPLEMENTATION-DEPENDENT CHARACTERISTICS
Package SYSTEM_CALLS

Package SYSTEM CALLS

The predefined package SYSTEM_CALLS enables Ada programs to set completion codes. The Ada process can either exit while setting the completion code information or set the completion code information to be used when the process reaches its normal termination.

The package contains six subprogram declarations for operating system procedures, two each for the ABEND, SETCOMPLETIONCODE, and STOP procedures. For each procedure, you can have your Ada program set only the completion code parameter or set the five supplemental information parameters as well as the completion code parameter. For information about the completion code parameter and the supplemental information parameters, see the *System Procedure Calls Reference Manual*.

| If an unhandled exception reaches the main subprogram, an Ada
| process returns the completion code 5, whether or not the Ada
| program set a different code.

| If an Ada process terminates without calling any of the subpro-
| grams in the SYSTEM_CALLS package, the process returns the
| completion code 0 for normal termination of the main subprogram.

The specification for the package SYSTEM_CALLS follows.

```

---+
-----
--  Package Name : SYSTEM_CALLS
--
--  Purpose : Ada interface to operating system procedures
-----
---+
package SYSTEM_CALLS is
--
  type COMPLETION_CODE is new INTEGER;
--
  subtype TANDEM_COMPLETION_CODE is COMPLETION_CODE
    range -32767 .. -1;
  subtype SHARED_COMPLETION_CODE is COMPLETION_CODE
    range 0 .. 999;
  subtype CUSTOMER_COMPLETION_CODE is COMPLETION_CODE
    range 1000 .. 32767;
--
  NORMAL          : constant SHARED_COMPLETION_CODE := 0;
  WARNING         : constant SHARED_COMPLETION_CODE := 1;
  FATAL           : constant SHARED_COMPLETION_CODE := 2;
  PREMATURE       : constant SHARED_COMPLETION_CODE := 3;
  ABNORMAL        : constant SHARED_COMPLETION_CODE := 5;
  WARNING_EXAMINE : constant SHARED_COMPLETION_CODE := 8;
--
---+
-----
--  Subprogram Name:  ABEND
--
--  Purpose:  Ada interface to the operating system
--            procedure ABEND
-----
---+
  procedure ABEND (COMPL_CODE : COMPLETION_CODE);
  pragma INTERFACE (TAL, ABEND);
  pragma EXTERNAL_NAME (ABEND, "Rsl^Abend1");
--
  procedure ABEND (COMPL_CODE      : COMPLETION_CODE;
                   TERMINATION_INFO : INTEGER;
                   SUBSYS_ORG       : STRING;
                   SUBSYS_NUM       : INTEGER;
                   SUBSYS_VER       : STRING;
                   TEXT              : STRING);
  pragma INTERFACE (TAL, ABEND);
  pragma EXTERNAL_NAME (ABEND, "Rsl^Abend2");

```

IMPLEMENTATION-DEPENDENT CHARACTERISTICS
Package SYSTEM_CALLS

```

---+
-----
--  Subprogram Name:  SET_COMPLETION_CODE
--
--  Purpose:  Ada interface to set the completion code to be
--            used when the process stops
-----
---+
procedure SET_COMPLETION_CODE (COMPL_CODE : COMPLETION_CODE);
pragma INTERFACE (TAL, SET_COMPLETION_CODE);
pragma EXTERNAL_NAME (SET_COMPLETION_CODE,
                      "RsI^Set^Completion^Code1");
--
procedure SET_COMPLETION_CODE (COMPL_CODE      : COMPLETION_CODE;
                               TERMINATION_INFO : INTEGER;
                               SUBSYS_ORG       : STRING;
                               SUBSYS_NUM       : INTEGER;
                               SUBSYS_VER       : STRING;
                               TEXT             : STRING);
pragma INTERFACE (TAL, SET_COMPLETION_CODE);
pragma EXTERNAL_NAME (SET_COMPLETION_CODE,
                      "RsI^Set^Completion^Code2");
---+
-----
--  Subprogram Name:  STOP
--
--  Purpose:  Ada interface to the operating system
--            procedure STOP
-----
---+
procedure STOP (COMPL_CODE : COMPLETION_CODE);
pragma INTERFACE (TAL, STOP);
pragma EXTERNAL_NAME (STOP, "RsI^Stop1");
--
procedure STOP (COMPL_CODE      : COMPLETION_CODE;
               TERMINATION_INFO : INTEGER;
               SUBSYS_ORG       : STRING;
               SUBSYS_NUM       : INTEGER;
               SUBSYS_VER       : STRING;
               TEXT             : STRING);
pragma INTERFACE (TAL, STOP);
pragma EXTERNAL_NAME (STOP, "RsI^Stop2");
end SYSTEM_CALLS;

```

Package LOW_LEVEL_IO

The compiler also has the predefined package LOW_LEVEL_IO, as required by the Ada standard. However, a call to a subprogram in LOW_LEVEL_IO does not cause any input or output operation to be performed.

The specification for the predefined package LOW_LEVEL_IO follows:

```
package LOW_LEVEL_IO is
    type DEVICE_TYPE is (NO_DEVICE);
    type DATA_TYPE is (NO_DATA);
    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;
```

A call to either procedure in the package always returns NO_DATA to the formal in out parameter DATA. A call to either procedure has no other effect at run time, except for taking time and memory.

IMPLEMENTATION-DEPENDENT CHARACTERISTICS

Data Representation

DATA REPRESENTATION

This subsection defines how the Tandem Ada compiler represents data. It includes definitions of the default data representation of different types and of how representation clauses and representation pragmas affect data layout.

Default Data Representations

Each type has an associated size and alignment, both expressed in bits. This size and alignment is the default data representation for the type.

The size specifies the size of the container in which the compiler can store an instance of the type. The container size for a scalar is always 8, 16, 32, or 64 bits. The container size for a composite is always an integral number of 8-bit bytes greater than or equal to 0.

The default alignment of a type specifies the alignment that the compiler uses for objects of that type in the absence of a representation clause to the contrary. The alignment is always either 8 or 16 bits.

Scalar values are always right justified in their containers and sign filled. That is, the least significant bit of the scalar value is the least significant bit of the container. Composite values are always left justified in their containers and zero filled. That is, the bit of the composite corresponding to at 0 range 0..0 occupies the most significant bit of the container.

Sizes the Compiler Knows at Compile Time

To understand the rules for record layout, the restrictions on component clauses, the restrictions on `UNCHECKED_CONVERSION`, and some of the information in the data map that the compiler produces, you need to know certain rules about which sizes the Tandem Ada compiler knows at compile time. This subsection explains these rules.

The compiler can always determine the sizes of the following at compile time:

- Scalar types and subtypes
- Access types and subtypes
- Task types and subtypes

For an array subtype, the compiler can determine the size at compile time if it can determine the size of the component at compile time and if all discrete ranges are static.

For a record subtype, to determine the size at compile time, the compiler must be able to determine the sizes of all valid components at compile time. If the subtype is statically constrained, the compiler needs to determine only the sizes of valid components, and it can use discriminant values to figure out the sizes of these components. For an unconstrained record, the compiler must be able to determine the sizes of all components.

If an unconstrained record has any component whose constraints come from a discriminant, the compiler cannot determine the size of the record at compile time.

If an unconstrained record has a variant part, the compiler calculates the size of the largest variant and uses that to compute the size of the record.

For a private type, the compiler can determine the size at compile time only if it can determine the private type's full declaration size at compile time. For a dynamically constrained record, the compiler cannot determine its size at compile time.

IMPLEMENTATION-DEPENDENT CHARACTERISTICS

Scalar Types

Scalar Types

Scalar types obey some common rules:

- Subtypes always have the same size as their base types. If you want to have a different representation, you must introduce a derived type. For example, if you want to select a different size (via "for T'SIZE use n"), you must introduce a derived type.
- Tandem Ada does not perform biased arithmetic. That is, the integer value 3 is always stored as 2#11#, even for a type T for which T'FIRST is 3.
- | • A scalar (derived) type always inherits the size of its parent type.

These rules apply to the following types:

- Integer types
- Enumeration types
- Floating-Point Types
- Fixed-Point Types

The following subsections explain the default data representation for each of these types.

Integer Types

Integer types are represented as two's-complement binary integers. The container size for an integer type is equal to the size of its parent type unless a length clause specifies a different value. The compiler chooses the smallest possible container size for the parent type.

Table F-1 shows the sizes and alignments of the predefined integer types.

Table F-1. Sizes and Alignments of Predefined Integer Types

Type	Size	Alignment
SHORT_INTEGER	8 bits	8 bits
INTEGER	16 bits	16 bits
LONG_INTEGER	32 bits	16 bits
LONG_LONG_INTEGER	64 bits	16 bits

IMPLEMENTATION-DEPENDENT CHARACTERISTICS
Scalar Types

The alignment is 8 bits if the size of the type is 8 bits; otherwise, the alignment is 16 bits. Table F-2 shows examples of this.

Table F-2. Relationship of Alignment to Size

Type	Size	Alignment
type BYTE is range 0 .. 127;	8	8
subtype I is INTEGER range -5 .. 14;	16	16
type J is range -5 .. 14;	8	8
type S is range 0 .. 2 ** 8 - 1;	16	16
type U is range 0 .. 2 ** 16 - 1;	32	16
type Z is INTEGER range VAR1 .. VAR2; -- Var1 and var2 might or might not -- be static integer expressions.	16	16
type X is new INTEGER range 1 .. 10;	16	16
type Y is new LONG_INTEGER range 1 .. 10;	32	16
type W is new LONG_LONG_INTEGER range 1 .. 10;	64	16

Enumeration Types

The value of an enumeration literal is equal to its position represented as an unsigned quantity. For enumeration types with no more than 256 literals, the compiler gives a container size of 8 bits and an alignment of 8 bits and treats the value as an unsigned quantity. For enumeration types with more than 256 but fewer than 32,768 literals, the compiler gives a container size of 16 bits and an alignment of 16 bits and treats the value as a signed quantity. Tandem Ada does not support enumeration types with more than 32,767 literals. Table F-3 shows examples of enumeration types and their sizes and alignments, in bits.

Table F-3. Sizes and Alignments of Enumeration Types

Type	Size	Alignment
type BOOLEAN is (FALSE, TRUE);	8	8
type CHARACTER is (...);	8	8
type Z is (E1,E2,E3,E4,...,E300)	16	16
type ALPHA is new CHARACTER range 'a' .. 'z';	8	8
type NZ is new Z range E1 .. E10	16	16

IMPLEMENTATION-DEPENDENT CHARACTERISTICS
Scalar Types

Floating-Point Types

The definitions for FLOAT and LONG_FLOAT in the package STANDARD and the description of the floating-point hardware in the *Tandem System Description Manual* imply the following values for floating-point types:

FLOAT'DIGITS = 6

FLOAT'MANTISSA = 21

FLOAT'EPSILON = 8#0.4000_000# * 2.0E-21

FLOAT'EMAX = 84

FLOAT'SMALL = 8#0.4000_000# * 2.0E-84

FLOAT'LARGE = 8#0.7777_777# * 2.0E+84

FLOAT'SAFE_EMAX = 254

| FLOAT'SAFE_SMALL = 8#0.4000_000# * 2.0E-254

| FLOAT'SAFE_LARGE = 8#0.7777_777# * 2.0E+254

FLOAT'MACHINE_MANTISSA = 23

LONG_FLOAT'DIGITS = 16

LONG_FLOAT'MANTISSA = 55

LONG_FLOAT'EPSILON = 8#0.4000_0000_0000_0000_000# * 2.0E-55

LONG_FLOAT'EMAX = 220

LONG_FLOAT'SMALL = 8#0.4000_0000_0000_0000_000# * 2.0E-220

LONG_FLOAT'LARGE = 8#0.7777_7777_7777_7777_774# * 2.0E+220

LONG_FLOAT'SAFE_EMAX = 254

| LONG_FLOAT'SAFE_SMALL = 8#0.1000_0000_0000_0000_000# * 2.0E-254

| LONG_FLOAT'SAFE_LARGE = 8#0.7777_7777_7777_7777_774# * 2.0E+254

LONG_FLOAT'MACHINE_MANTISSA = 55

IMPLEMENTATION-DEPENDENT CHARACTERISTICS
Scalar Types

The following values apply to both predefined floating-point types:

'MACHINE_EMAX = 256
'MACHINE_EMIN = -255
'MACHINE_RADIX = 2
'MACHINE_ROUNDS = TRUE
'MACHINE_OVERFLOWS = TRUE

The representation sizes for floating-point types are 32 and 64 bits for FLOAT and LONG FLOAT, respectively. The alignment is 16 for both types. FLOAT is the base type of a user-defined floating-point type if the user-defined type has the following:

- No more than 6 digits
- If specified, a range within the range of the safe numbers of FLOAT

The format for floating-point data items is the format that the Tandem System Description Manual defines for floating-point numbers.

IMPLEMENTATION-DEPENDENT CHARACTERISTICS

Scalar Types

Fixed-Point Types

Fixed-point types are represented as signed integers equal to the value of $\text{sign} * \text{mantissa} * \text{small}$, which Section 3.5.9 of the ANSI Reference Manual for the Ada Programming Language defines.

When you give a fixed-point type specification and a 'SMALL specification, the compiler chooses a fixed-point base type with that value of 'SMALL and with a mantissa of 31 bits. When you do not specify 'SMALL, the compiler chooses a fixed-point base type with a value of 'SMALL that is the largest power of 2 smaller than or equal to 'DELTA of the type specification and with a mantissa of 31 bits. In either case, the base type is for the fixed-point subtype being defined by the fixed-point type definition.

The size for fixed-point types is always 64 bits, although MAX MANTISSA is 31. 'SMALL for fixed-point types must be in the following range:

$2^{*(-256)}$ to 2^{*255}

The range of fixed-type values follows:

$-2^{*31} * 2^{*255} .. -2^{*(-256)}, 0, 2^{*(-256)} .. (2^{*31}-1) * 2^{*255}$

If the definition type T is delta DEL range LB .. UB is in effect, then all of the following are true:

- $T'SMALL = T'BASE'SMALL = T'SAFE_SMALL =$ largest power of 2 less than or equal to DEL
- $T'BASE'MANTISSA = 31$
- $T'DELTA = DEL$
- $T'BASE'DELTA = T'SMALL$
- $T'BASE'LARGE = T'SAFE_LARGE = (2 ** 31 - 1) * T'SMALL$
- $T'MACHINE_OVERFLOWS = TRUE$
- $T'MACHINE_ROUNDS = TRUE$

The range of T'BASE is $-(2 ** 31) * T'SMALL .. 2 ** 31 * T'SMALL$.

Ada fixed-point types do not correspond directly to any type in Tandem's Transaction Application Language (TAL). Also, the values of certain fixed-point types cannot be represented as floating-point values because the bounds of a fixed-point type can be greater than the bounds of the largest floating-point type. So, when an Ada program passes a parameter of a fixed-

point type to a TAL procedure, the TAL procedure must recreate the fixed-point value, as explained under "Scalar, Access, and Constrained Composite Parameters" in Section 8.

Array Types

The size and alignment of an array depends on the size and alignment of its components.

The compiler arranges arrays so that all components have the same size and alignment as the component type, and the entire array has the same alignment. For example, if an array has 1-byte components aligned to 1-byte boundaries, then the array also has 1-byte alignment. The stride of an array is the size of each component of the array--the difference between `A(I)'ADDRESS` and `A(T'SUCC(I))'ADDRESS`. The compiler stores multidimensional arrays in row-major order.

The algorithm for calculating the size and alignment of an array follows:

1. Set the alignment of the array to the alignment of the component.
2. Set the stride to the component size.
3. Set the size of the array to the product of the stride and the number of elements. For unconstrained array types, the number of elements is undefined; however, the size is equal to the maximum possible size that a constrained subtype could have.

IMPLEMENTATION-DEPENDENT CHARACTERISTICS

Record Types

Record Types

The size and alignment of a record depends on the size and alignment of its components and any record representation clauses and length specifications for the record. The following subsections describe the default data representation for simple and complex records, including static, nonvariant record types and dynamic and variant record types.

Simple Records

A record that has no discriminants, no dynamically constrained composite components, and no unconstrained components is a *simple record*. The compiler arranges a simple record so that each component has its own default size and alignment, relative to the base of the record.

The algorithm for calculating the size and alignment of a record follows:

1. For each component, align at the default alignment for the component type and allocate space for the component's size.
2. Set component locations as follows:
 - a) Sort components by alignment, largest first, and then by order of declaration.
 - b) Assign each component the location that (1) is closest to the base of the record, (2) is not yet assigned to another component, and (3) meets the default size and alignment requirements of the component type.
3. Set the alignment of the record to the largest alignment of its components. That alignment is the most restrictive and can only be 8 or 16 bits.
4. Set the size of the record to the sum of the offset of the component that has the largest offset ('POSITION attribute) and the size of that component rounded up to the next multiple of the record type's alignment.

Complex Records

A complex record has at least one of the following:

- Discriminants (one or more)
- Dynamically constrained composite components
- Unconstrained components

For example, consider these declarations:

```
N : INTEGER := F(10); -- F is a function.
type DYNAMIC_ARRAY is array (1..N) of INTEGER;
type R( D : NATURAL ) is
  record
    A : NATURAL := 0;
    B : STRING(1..D);
    C1 : DYNAMIC_ARRAY;
    C2 : DYNAMIC_ARRAY;
  end record;
```

In the example, R is a complex record because it has a discriminant and also because the values of B'LAST, C1'LAST, and C2'LAST are determined at run time.

Layout of Record Components. The compiler lays the record components out in the following order:

1. The discriminants, if any
2. The variant index field, if the record type has variants, as if the variant index were another discriminant
3. Everything the compiler knows the size of declared in the invariant part
4. For each variant, everything the compiler knows the size of declared in the variant part
5. Everything the compiler does not know the size of declared in the invariant part
6. For each variant, everything the compiler does not know the size of declared in the variant part

IMPLEMENTATION-DEPENDENT CHARACTERISTICS

Record Types

The compiler always determines the ordering of all components at compile time. It cannot determine the position values for any component the compiler does not know the size of at compile time or for any component constrained by a discriminant, except for the first such component.

Components With Unknown Sizes or Discriminant Constraints. The compiler always stores components it does not know the size of and components constrained by discriminants at the end of a record. If the record has more than one of these components per variant, each of the components except the last one causes the compiler to create an extra component and assign it a location within the record. This extra component holds the byte-relative offset of the corresponding component.

The extra component is a 16-bit field aligned on a 16-bit boundary. The field is located in the static invariant part for a corresponding component in the dynamic invariant part or in the static variant part for a corresponding component in the dynamic variant part.

If a record subtype is constrained and has no dynamic part, the compiler allocates space only for the active variant parts. If a constrained record subtype has a dynamic part, the compiler allocates space for all variants in the static part to insure that the dynamic part of all subtypes start at the same offset.

Multidimensional Subtypes. When a record component is a multidimensional subtype in which more than one bound depends on a discriminant, the size the compiler allocates for the component can be the minimum space the component needs or larger. For example, the following declaration can cause the compiler to allocate more than the minimum space for a component:

```
subtype INT is INTEGER range 1 .. 20;
type ARR is array (INT range <>, INT range <>) of INT;
type R (D : INT := 5) is
  record
    A : ARR (1 .. D, D .. 20);
  end record;
```

When D is 10, the number of components in A is 110, the maximum size that A can be. However, the compiler plugs in the upper-and lower bounds for the discriminant type in the appropriate parts of the subtype range for A and calculates the number of elements in A to be 400.

Descriptor Components. For each component in the record that has a constraint which depends on a discriminant of the record, the compiler also generates an additional descriptor component. The size of the descriptor component varies with the type of the component, as described under "Run-Time Descriptors," later in this appendix. However, the alignment is always 16 bits.

NOTE

The compiler does not round up the size of a descriptor component to a multiple of two bytes.

The compiler, at compile time, sizes the components it generates and allocates space for them in the appropriate static areas. If a descriptor's corresponding component is in the dynamic invariant part, then the descriptor component goes in the static invariant part. If a descriptor's corresponding component is in the dynamic variant part, then the descriptor component goes in the static variant part.

For a variant record, the compiler also generates a 16-bit variant index field. The compiler uses the index field to simplify component selection checks. The alignment of the index field is 16 bits.

Representation of Complex Records. The algorithm for determining the representation of complex records follows:

1. Divide the components, including compiler-generated components, into three groups:
 - Discriminants
 - Invariants
 - Variant part
2. Apply the algorithm for laying out simple record components to the set of discriminants.
3. If the record type has any variants, generate a variant index field. Place the index field at the location closest to the base of the record that (1) is not assigned to a discriminant and (2) has a size of 2 bytes and an alignment of 2 bytes.

IMPLEMENTATION-DEPENDENT CHARACTERISTICS
Record Types

4. Apply a slightly modified version of the algorithm for laying out simple records to the set of everything the compiler knows the size of in the invariant part. The only modification is that, for each component, the compiler allocates the unoccupied location closest to the last location occupied by a discriminant, or by the variant index field if present, instead of the unoccupied location closest to the base of the record.
5. For each variant case, lay out everything the compiler knows the size of, using a slightly modified version of the algorithm for laying out simple records. The only modification is that, for each component, the compiler allocates the location closest to the last location occupied by a component in the static invariant part, instead of the unoccupied location closest to the base of the record. The compiler computes each variant case independently of any of the others, so components in one ordinarily overlay components in another.
6. Lay out the subvariants, if any, within each variant in the same manner as the variants.
7. Finally, at run time, lay out components the compiler does not know the size of and components that are constrained by discriminants. The size of an unconstrained component is the maximum possible size for that component.

The algorithm for laying out the components in the final step follows:

- a) For each invariant component, apply a slightly modified version of the algorithm for laying out simple records. The only modification is that each component is allocated the location closest to the last location occupied by a component in all of the static variant parts, instead of the unoccupied location closest to the base of the record.
- b) For each variant case, lay out the components the compiler does not know the size of and the components that are constrained by discriminants, using a slightly modified version of the algorithm for laying out simple records. The only modification is that each component is allocated the location closest to the last location occupied by a component in the dynamic variant part, instead of the unoccupied location closest to the base of the record. Each variant case is computed independently of any of the others, so components in one ordinarily overlay components in another.

The compiler can always determine the ordering of all components at compile time, but the position values for components the compiler does not know the size of and components constrained by discriminants must be computed at run time.

For the record R, we might arrive at the following layout for a declaration of $R(D \Rightarrow 5)$ and $N := 5$:

Component	Position
D	0
A	2
B's Indirect Pointer	4
B's Descriptor	6
C2's Indirect Pointer	14
C1	16
C2	26
B	36

There is no indirect pointer for the dynamic component C1 because it is the first dynamic component in the record. The compiler determines this component's offset within the record at compile time.

Figure F-1 illustrates the layout of the record R.

IMPLEMENTATION-DEPENDENT CHARACTERISTICS
Record Types

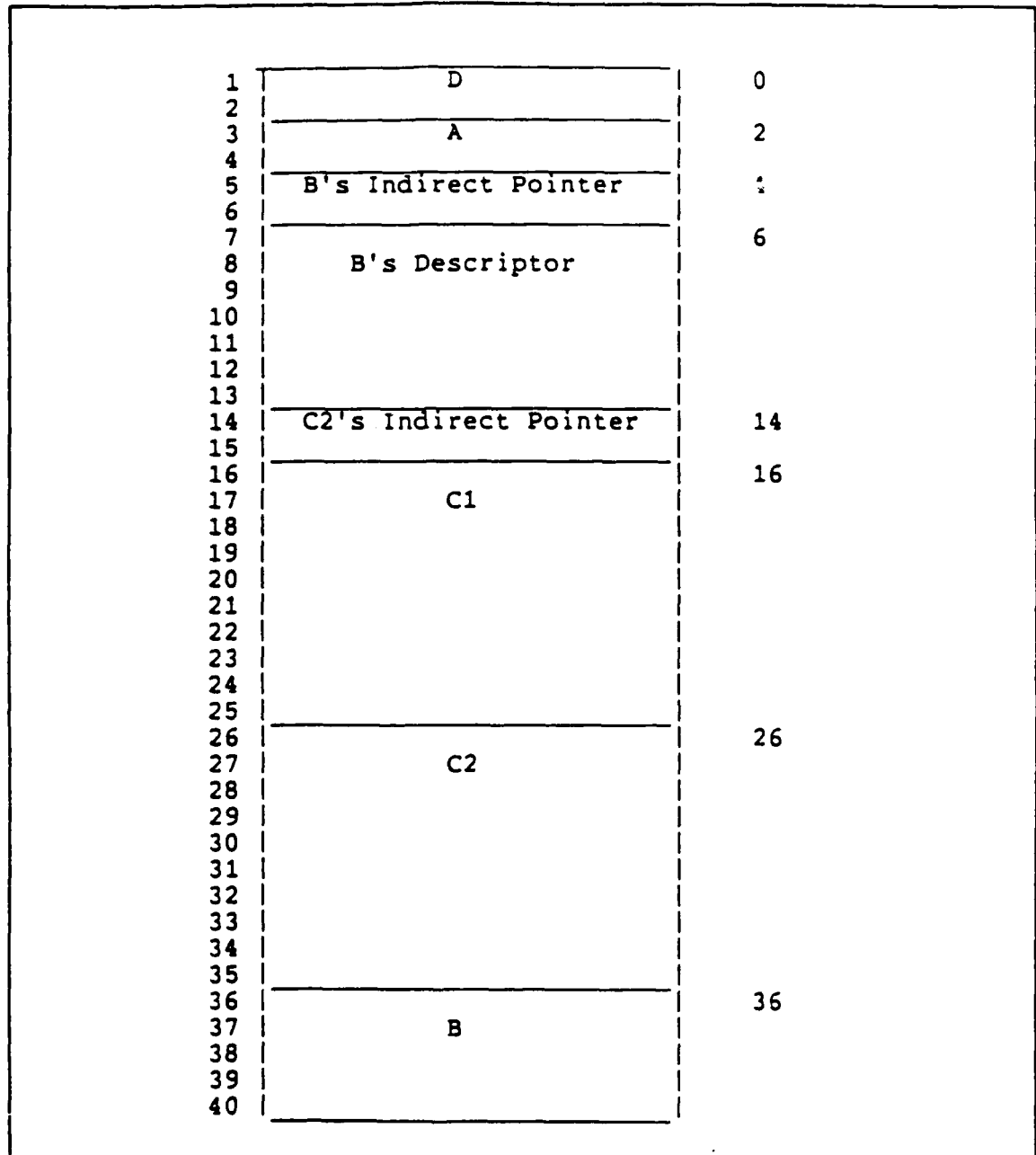


Figure F-1. Layout of a Complex Record

Access Types

The size of an access type depends on whether the accessed type is constrained and on whether the accessed type is completed in the same compilation with its incomplete type specification. The compiler allocates 64 bits for an access type to an unconstrained record or array subtype. The compiler also allocates 64 bits for an access type to a type not completed in the same compilation, if the corresponding full type declaration for the accessed type does not occur before a forcing occurrence of the access type. (Section 13.1 of the *ANSI Reference Manual for the Ada Programming Language* defines forcing occurrence.) For any other access type, the compiler allocates 32 bits.

Subtypes and derived types of access types always have the same sizes as their base or parent types.

Task Types

The size of a task type is 32 bits. The 32 bits contain a task identifier.

Component Clauses

A record representation clause can give one or more component clauses for a record. The representation clause is legal if the compiler can assign all the specified locations and sizes without incorrectly overlapping any fields.

For any field mentioned with a component clause, the compiler assigns precisely the specified location and size. After it locates all such fields, the compiler applies the algorithm described in the preceding text to locate the remaining fields. In doing so, it avoids incorrectly overlapping those fields that it explicitly located. In any case, the alignment of a record type that has a component clause is equal to the largest component alignment.

For any component, the size, as implied by the range, must be the same as the size of the component type. The alignment for any component relative to the base of the record must be a multiple of the alignment of the component's type. A record representation clause cannot specify the insertion of a component from a component list or the discriminant part of a type declaration into another component list or the discriminant part of the same type declaration. (For an explanation of

IMPLEMENTATION-DEPENDENT CHARACTERISTICS
Component Clauses

component lists, see sections 3.7 and 3.7.3 of the *ANSI Reference Manual for the Ada Programming Language*.)

For example, the following record representation clause is illegal because it tries to place a variant component in the list of invariant components.

```
type R (D : INTEGER) is
  record;
    A,B : INTEGER;
    case D is
      when 0 =>
        C : INTEGER;
      when others =>
        null;
    end case;
  end record;

for R use
  record
    -- D is at 0 range 0..15.
    -- Variant index is at 4 range 0..15.

    A at 6  range 0..15;
    B at 10 range 0..15;
    C at 8  range 0..15;

  end record;
```

| To make this representation clause legal, the position of
| component C should follow the position of component B.

In record representation clauses, you must leave space in the component list for the following:

- Variant index field, which the compiler puts in the discriminant part
- Descriptors for dependent subtypes, described in "Descriptor Components" under "Complex Records," earlier in this appendix
- Indirect pointers to components, described in "Components With Unknown Sizes or Discriminant Restraints" under "Complex Records," earlier in this appendix

A representation clause can leave space for the variant index anywhere in the discriminant part, before the first component, A. The compiler considers the variant index to be part of the component list of discriminants. For example, the following representation clause is illegal:

```
type R (D : INTEGER) is
  record
    I : INTEGER;
    case D is
      when 1 => J : INTEGER;
      when others => null;
    end case;
  end record;

for R use
  record

    D at 0 range 0..15;
    I at 2 range 0..15;

  end record
```

The compiler cannot honor the representation clause for component I because it already placed the variant index at 2 range 0..15. The same applies to components the compiler does not know the size of and components constrained by discriminants.

IMPLEMENTATION-DEPENDENT CHARACTERISTICS
Alignment Clauses

Alignment Clauses

An alignment clause in a record representation clause affects only the alignment of the record. It does not affect the layout of the components of the record.

The following examples illustrate the use of an alignment clause:

```
--      Sample record type:

type      S is range 0..1000;

type      Q is range 0..20;

type      T is
          record
            A : CHARACTER;
            B : LONG_INTEGER; -- 32 bits
            C : S;
            D : Q;
          end record;

--      Default representation:

for        T use
          record at mod 2 -- SYSTEM.STORAGE_UNIT = 8 bits
            B at 0 range 0..31;
            C at 4 range 0..15;
            A at 6 range 0..7;
            D at 7 range 0..7;
          end record;
for        T'SIZE use 64; -- bits
```

Restrictions on Representation Clauses

In addition to the rules for representation clauses and representation pragmas in the *ANSI Reference Manual for the Ada Programming Language*, Tandem Ada has restrictions on the following:

- Size specifications in length clauses
- Record representation clauses
- Address clauses
- Enumeration representation clauses
- Specification of 'SMALL for fixed-point types
- Specification of 'STORAGE_SIZE

Size Specifications in Length Clauses

For records and arrays, the value of the static expression in a length clause for T'SIZE must be a multiple of 8 and of the alignment. Also, the value must be at least as large as the default size the compiler calculates for T.

Table F-4 shows the possible values of N for different data types in the clause for T'SIZE use N. For scalar types, just a few values are valid. You cannot use the length clause for access types.

IMPLEMENTATION-DEPENDENT CHARACTERISTICS
Restrictions on Representation Clauses

Table F-4. Size Specifications for Different Types

Type	Possible Values of T'SIZE
Integer	8, 16, 32, and 64
Enumeration	8 and 16
Fixed point	64
Floating point	32 and 64
Task	32
Composite	Multiples of 8 and of the alignment
Access	Not Supported

If a representation clause increases the size of a type, then the compiler creates some filler space. For scalar types, which the compiler always stores right justified within a container, the filler space is sign extended. For composite types, which the compiler always stores left justified within a container, the filler space is zero filled.

A subtype of a type typically has the same size as the type; however, a constrained record subtype can be smaller than the record type. If you specify a length clause for the record type, the compiler uses the length in the clause to allocate space for the type. But when allocating space for an object of a constrained subtype, the compiler ignores the length clause for the type and chooses a size for the subtype.

Record Representation Clauses

The following restrictions apply to component clauses (at N range L .. R):

- The compiler must be able to determine the size of the component subtype at compile time, as explained under "Sizes the Compiler Knows at Compile Time," earlier in this appendix.
- The size of the range you specify (R-L+1) must equal the size of the component type.
- The value for L must be a multiple of 8. The component must begin on a byte boundary.
- All values supplied for a record component offset must be nonnegative ($N * 8 + L \geq 0$).
- Components from a variant part must follow components from the fixed part in the record layout.

The compiler's layout algorithm implies some additional restrictions. For a description of the algorithm, see "Complex Records" under "Record Types," earlier in this appendix.

The following restrictions apply to alignment clauses (at mod N):

- The value of N must be at least as large as the default alignment the compiler chose for the record.
- The value of N must be either 1 or 2 bytes.

The Tandem Ada compiler does not support record representation clauses for records that contain generic formals.

Address Clauses

The Tandem Ada compiler does not support address clauses.

Enumeration Representation Clauses

The Tandem Ada compiler does not support enumeration representation clauses.

IMPLEMENTATION-DEPENDENT CHARACTERISTICS
Restrictions on Representation Clauses

| Specification of 'SMALL for Fixed-Point Types

| The value of the static expression in a length clause for 'SMALL must be a power of 2 in the following range:

2 ** (-256) to 2 ** 255

| The value specified for 'SMALL must be in the range precisely represented by the positive range of the predefined types FLOAT and LONG_FLOAT.

The *ANSI Reference Manual for the Ada Programming Language* also implies that the value must satisfy the following relation:

max (ceiling (log2 (abs (LB) / small)),
ceiling (log2 (abs (UB) / small))) <= SYSTEM.MAX_MANTISSA

In other words, the number of binary digits in the mantissa of the model numbers for fixed-point type must be less than or equal to the maximum number of binary digits, SYSTEM.MAX_MANTISSA, which is 31.

Specification of 'STORAGE_SIZE

For tasks, the value you specify for 'STORAGE_SIZE, must be greater than 0 but less than 2 ** 27 bytes. The default is 2 ** 18 bytes. For a description of how tasks use memory, see "Memory Usage on NonStop Systems," later in this appendix.

Tandem Ada does not support 'STORAGE_SIZE for access types.

Representation Attributes

The Tandem Ada compiler supports all representation attributes. However, the following attributes might not have meaningful values:

- 'ADDRESS

'ADDRESS returns the 32-bit extended address of an object. (For a task object, it returns the address of the variable that contains the task identifier.)

Meaningful addresses exist for some, but not all, objects. You can use the 'ADDRESS attribute to determine whether an object has a meaningful address.

For example, to determine whether an object of type T has a meaningful address, first declare an access type--call it A--whose designated subtype is T, and instantiate UNCHECKED_CONVERSION for types SYSTEM.ADDRESS and A. Then, if an object of type T does not have a meaningful address, the result of using the 'ADDRESS attribute followed by unchecked conversion to type A is the value null. If objects do have meaningful addresses, the result of the conversion is a valid, non-null access value, which you can use to gain access to the corresponding object of type T.

NOTE

Unchecked conversion between SYSTEM.ADDRESS and an access type is possible only for access types represented in 32 bits. If T is an unconstrained type or is not completed in the same compilation, an access value to type T might require 64 bits, as explained earlier in this appendix under "Access Types." In such cases, use the corresponding full type declaration of T, or a constrained subtype of T, for the designated subtype of A.

- 'STORAGE_SIZE

For access types or subtypes, this attribute does not return a meaningful value.

For tasks, if the program gives a storage-size representation clause, then 'STORAGE_SIZE returns that value. Otherwise, 'STORAGE_SIZE returns the default task storage size, which is 2 ** 18.

IMPLEMENTATION-DEPENDENT CHARACTERISTICS

Run-Time Descriptors

Run-Time Descriptors

This section describes record and array subtype descriptors. When the calling program passes an actual parameter corresponding to an unconstrained formal parameter, it also passes a pointer to a subtype descriptor. The program pushes the pointer to the descriptor onto the stack immediately after the pointer to the data.

The compiler also uses a run-time descriptor when record components depend on discriminants.

Record Subtype Descriptors

A record subtype descriptor describes a record subtype. Each record descriptor is at least four bytes. All record descriptors have a size field and a field that tells whether the subtype is constrained. The second field occupies two bytes.

If a record subtype is constrained, its descriptor has additional fields containing a copy of the discriminant values of the subtype. The compiler lays out the discriminant values in the record descriptor in the same way as it lays them out in the actual record, beginning where the discriminant part begins in the record layout. For example, if a discriminant is at offset X in the actual record, and the smallest offset of any discriminant in the actual record is Y, then that same discriminant is at offset X-Y+4 in the record descriptor.

-- The following declaration describes a descriptor for a
-- record subtype:

```
type RECORD_DESCRIPTOR is
  record
    SIZE                : INTEGER; -- at 0 range 0 .. 15
    IS_CONSTRAINED      : INTEGER; -- at 2 range 0 .. 15
    RECORD_DISCRIMINANTS : DISCRIM_RECORD;
    --If the record is constrained and has
    --discriminants, then this field is a copy
    --of the discriminant constraints of the
    --record subtype.
  end record;
```

NOTE

The compiler does not round up the size of a descriptor component to a multiple of two bytes.

Array Subtype Descriptors

An array subtype descriptor describes an array subtype. Each descriptor specifies an array of records with two numeric fields and an array of array strides. Each of these arrays has from 1 to 7 elements, depending upon the number of dimensions of the array being described. The compiler knows this number at compile time. The last field in the descriptor is the array size.

Examples of array subtype descriptors follow:

```

MAX_INDICES : constant INTEGER := 7;
subtype INDEX_COUNT is INTEGER range 1 .. MAX_INDICES;
subtype STRIDE is INTEGER;

type STRIDE_ARRAY is array (INDEX_COUNT range <>)
    of STRIDE;

type AIT is
    record
        LOWER_BOUND : INTEGER;
        UPPER_BOUND : INTEGER;
    end record;
type AIT_ARRAY is array (INDEX_COUNT range <>) of AIT;

type LONG_AIT is
    record
        LOWER_BOUND : LONG_INTEGER;
        UPPER_BOUND : LONG_INTEGER;
    end record;
type LONG_AIT_ARRAY is array (INDEX_COUNT range <>)
    of LONG_AIT;

type LONG_LONG_AIT is
    record
        LOWER_BOUND : LONG_LONG_INTEGER;
        UPPER_BOUND : LONG_LONG_INTEGER;
    end record;
type LONG_LONG_AIT_ARRAY is array (INDEX_COUNT range <>)
    of LONG_LONG_AIT;

```

IMPLEMENTATION-DEPENDENT CHARACTERISTICS
Run-Time Descriptors

--The following types describe arrays that have index --subtypes less than or equal to 16 bits.

```
type ARRAY_DESCRIPTOR_1 is --for a 1-dimensional array
  record
    AIT      : AIT_ARRAY (1 .. 1);
    STRIDE   : STRIDE_ARRAY (1 .. 1);
    SIZE     : INTEGER;
  end record;
```

```
type ARRAY_DESCRIPTOR_2 is --for a 2-dimensional array
  record
    AIT      : AIT_ARRAY (1 .. 2);
    STRIDE   : STRIDE_ARRAY (1 .. 2);
    SIZE     : INTEGER;
  end record;
```

```
type ARRAY_DESCRIPTOR_3 is --for a 3-dimensional array
  record
    AIT      : AIT_ARRAY (1 .. 3);
    STRIDE   : STRIDE_ARRAY (1 .. 3);
    SIZE     : INTEGER;
  end record;
```

```
type ARRAY_DESCRIPTOR_4 is --for a 4-dimensional array
  record
    AIT      : AIT_ARRAY (1 .. 4);
    STRIDE   : STRIDE_ARRAY (1 .. 4);
    SIZE     : INTEGER;
  end record;
```

```
type ARRAY_DESCRIPTOR_5 is --for a 5-dimensional array
  record
    AIT      : AIT_ARRAY (1 .. 5);
    STRIDE   : STRIDE_ARRAY (1 .. 5);
    SIZE     : INTEGER;
  end record;
```

```
type ARRAY_DESCRIPTOR_6 is --for a 6-dimensional array
  record
    AIT      : AIT_ARRAY (1 .. 6);
    STRIDE   : STRIDE_ARRAY (1 .. 6);
    SIZE     : INTEGER;
  end record;
```

```
type ARRAY_DESCRIPTOR_7 is --for a 7-dimensional array
  record
    AIT      : AIT_ARRAY (1 .. 7);
    STRIDE   : STRIDE_ARRAY (1 .. 7);
    SIZE     : INTEGER;
  end record;
```

--The following types describe arrays that have at least
--one index subtype greater than 16 bits and no index
--subtype greater than 32 bits.

```
type LONG_ARRAY_DESCRIPTOR_1 is --for a 1-dimensional array
  record
    AIT      : LONG AIT ARRAY (1 .. 1);
    STRIDE   : STRIDE ARRAY (1 .. 1);
    SIZE     : INTEGER;
  end record;
```

```
type LONG_ARRAY_DESCRIPTOR_2 is --for a 2-dimensional array
  record
    AIT      : LONG AIT ARRAY (1 .. 2);
    STRIDE   : STRIDE ARRAY (1 .. 2);
    SIZE     : INTEGER;
  end record;
```

```
type LONG_ARRAY_DESCRIPTOR_3 is --for a 3-dimensional array
  record
    AIT      : LONG AIT ARRAY (1 .. 3);
    STRIDE   : STRIDE ARRAY (1 .. 3);
    SIZE     : INTEGER;
  end record;
```

```
type LONG_ARRAY_DESCRIPTOR_4 is --for a 4-dimensional array
  record
    AIT      : LONG AIT ARRAY (1 .. 4);
    STRIDE   : STRIDE ARRAY (1 .. 4);
    SIZE     : INTEGER;
  end record;
```

```
type LONG_ARRAY_DESCRIPTOR_5 is --for a 5-dimensional array
  record
    AIT      : LONG AIT ARRAY (1 .. 5);
    STRIDE   : STRIDE ARRAY (1 .. 5);
    SIZE     : INTEGER;
  end record;
```

```
type LONG_ARRAY_DESCRIPTOR_6 is --for a 6-dimensional array
  record
    AIT      : LONG AIT ARRAY (1 .. 6);
    STRIDE   : STRIDE ARRAY (1 .. 6);
    SIZE     : INTEGER;
  end record;
```

```
type LONG_ARRAY_DESCRIPTOR_7 is --for a 7-dimensional array
  record
    AIT      : LONG AIT ARRAY (1 .. 7);
    STRIDE   : STRIDE ARRAY (1 .. 7);
    SIZE     : INTEGER;
  end record;
```

IMPLEMENTATION-DEPENDENT CHARACTERISTICS
Run-Time Descriptors

--The following types describe arrays that have at least one
--index subtype greater than 32 bits and no index subtype
--greater than 64 bits.

```
type LONG_LONG_ARRAY_DESCRIPTOR_1 is --for a 1-dimensional array
  record
    AIT      : LONG LONG AIT ARRAY (1 .. 1);
    STRIDE   : STRIDE ARRAY (1 .. 1);
    SIZE     : INTEGER;
  end record;
```

```
type LONG_LONG_ARRAY_DESCRIPTOR_2 is --for a 2-dimensional array
  record
    AIT      : LONG LONG AIT ARRAY (1 .. 2);
    STRIDE   : STRIDE ARRAY (1 .. 2);
    SIZE     : INTEGER;
  end record;
```

```
type LONG_LONG_ARRAY_DESCRIPTOR_3 is --for a 3-dimensional array
  record
    AIT      : LONG LONG AIT ARRAY (1 .. 3);
    STRIDE   : STRIDE ARRAY (1 .. 3);
    SIZE     : INTEGER;
  end record;
```

```
type LONG_LONG_ARRAY_DESCRIPTOR_4 is --for a 4-dimensional array
  record
    AIT      : LONG LONG AIT ARRAY (1 .. 4);
    STRIDE   : STRIDE ARRAY (1 .. 4);
    SIZE     : INTEGER;
  end record;
```

```
type LONG_LONG_ARRAY_DESCRIPTOR_5 is --for a 5-dimensional array
  record
    AIT      : LONG LONG AIT ARRAY (1 .. 5);
    STRIDE   : STRIDE ARRAY (1 .. 5);
    SIZE     : INTEGER;
  end record;
```

```
type LONG_LONG_ARRAY_DESCRIPTOR_6 is --for a 6-dimensional array
  record
    AIT      : LONG LONG AIT ARRAY (1 .. 6);
    STRIDE   : STRIDE ARRAY (1 .. 6);
    SIZE     : INTEGER;
  end record;
```

```
type LONG_LONG_ARRAY_DESCRIPTOR_7 is --for a 7-dimensional array
  record
    AIT      : LONG LONG AIT ARRAY (1 .. 7);
    STRIDE   : STRIDE ARRAY (1 .. 7);
    SIZE     : INTEGER;
  end record;
```

UNCHECKED PROGRAMMING

This subsection explains when you can use instances of the predefined generic subprograms for unchecked programming and what they do. For complete descriptions of `UNCHECKED DEALLOCATION` and `UNCHECKED CONVERSION`, see Section 13.10 of the *ANSI Reference Manual for the Ada Programming Language*.

Unchecked Storage Deallocation

The generic procedure `UNCHECKED DEALLOCATION` resets an access value to null but does not reclaim the memory space used by the allocated object.

The compiler reclaims any space it allocates for temporary variables it creates for execution of a subprogram. However, the compiler does not automatically do garbage collection of data that a program creates.

Unchecked Type Conversions

The generic function `UNCHECKED_CONVERSION` has the following restrictions:

- The source and target types must have the same size, and the compiler must be able to determine the size at compile time. For information about sizes the compiler can determine at compile time, see "Sizes the Compiler Knows at Compile Time," earlier in this appendix.
- The source and target types must not be unconstrained records or arrays.

IMPLEMENTATION-DEPENDENT CHARACTERISTICS

Input-Output Packages

INPUT-OUTPUT PACKAGES

Tandem's Ada compiler supports SEQUENTIAL_IO, DIRECT_IO, and TEXT_IO. I/O routines recognize only disk files, terminals, and spoolers. If an output file is a process other than a terminal, the I/O routines treat the file like a spooler.

Calling subprograms in LOW_LEVEL_IO does not cause any I/O operations to be performed.

FORM Parameter of CREATE and OPEN Procedures

The FORM parameter of the CREATE and OPEN procedures in the TEXT_IO, DIRECT_IO, and SEQUENTIAL_IO packages has the following syntax:

```
form-string ::=
    create-open-spec {, create-open-spec}

create-open-spec ::= create-spec | open-spec | null
    -- Create-spec is only for the CREATE
    -- procedure. Open-spec is for both the
    -- CREATE and OPEN procedures.

create-spec ::=
    DATA_BLOCKLEN = block-length -- This data-block length
    -- is for structured files.
    -- The default is 1024. You
    -- can specify any integer in
    -- the range 1..4096. If you
    -- do not specify 512, 1024,
    -- 2048, or 4096, the compiler
    -- rounds the block length up
    -- to the next highest of
    -- these values.
    | FILE_CODE = code-number -- The code number is the
    -- operating system file code,
    -- any integer in the range
    -- 0..65535. When a program
    -- uses TEXT_IO with an edit-
    -- format file, the file code
    -- is always 101, and if you
    -- try to specify it, the
    -- program raises USE_ERROR.
    -- For TEXT_IO with other file
    -- types and for DIRECT_IO and
    -- SEQUENTIAL_IO, the default
    -- file code is 0.
```

IMPLEMENTATION-DEPENDENT CHARACTERISTICS
FORM Parameter of CREATE and OPEN Procedures

```

| PRIMARY_EXTENT_SIZE = extent-size -- The extent size is
-- any integer in the range
-- 0..65535. The default
-- primary extent size is 4.
| SECONDARY_EXTENT_SIZE = extent-size -- The extent size
-- any integer in the range
-- 0..65535. The default
-- secondary extent size is 16.
| FILE_TYPE = file-type
| RECORDLEN = record-length -- The record length is any
-- integer in the range
-- 1..1320. This option
-- specifies the maximum record
-- length, which applies only
-- when a program creates a
-- relative or entry-sequenced
-- file through TEXT_IO. The
-- default record length is
-- 132 for relative files and
-- 1320 for entry-sequenced
-- files.
| ODDUNSTR -- This option works like the
-- file-system CREATE proce-
-- dure's ODDUNSTR parameter.
-- For a description of the
-- ODDUNSTR parameter, see
-- the System Procedure Calls
-- Reference Manual.
file-type ::= U | R | E | D -- U is for unstructured,
-- R for relative, E for entry
-- sequenced, and D for edit
-- format. For TEXT_IO, the
-- default file type is E.
-- For DIRECT_IO, the default
-- type is R, and an attempt
-- to specify anything else
-- raises USE_ERROR. For
-- SEQUENTIAL_IO, the default
-- type is E, and an attempt
-- to specify anything else
-- raises USE_ERROR.
open-spec ::= SHARED | EXCLUSIVE | PROTECTED --For opening
-- any type of file with mode
-- in, the default is SHARED.
-- For creating a SEQUENTIAL_IO
-- or TEXT_IO file, the default
-- is EXCLUSIVE. For creating
-- a DIRECT_IO file or opening
-- a DIRECT_IO file with mode
-- in out or out, the default
-- is EXCLUSIVE, and no other
-- open-spec is allowed.

```

IMPLEMENTATION-DEPENDENT CHARACTERISTICS
FORM Parameter of CREATE and OPEN Procedures

null ::=

-- A null create-open-spec is
-- zero or more blanks.

Order of Options

In the FORM parameter, you can specify the options in any order. Also, you can specify each option only once.

IN and OUT Files for an Ada Process

The TEXT_IO package opens the IN file and the OUT file for the Ada process using the file names supplied by the COMMAND_INTERPRETER_INTERFACE package for the default IN and OUT files. You can specify the default IN file and OUT file for the process in the RUN command. If you do not, the command interpreter passes the names of its current default IN and OUT files in the startup message for the new process.

If the file named as the OUT file does not exist, TEXT_IO creates a new file, using all the default values of the CREATE procedure (that is, it creates an edit-format file). If the file named as the OUT file already exists, TEXT_IO deletes the file and recreates it with the same characteristics it had.

Startup, Assign, and Param Messages

The TEXT_IO package names the COMMAND_INTERPRETER_INTERFACE package in a with clause. As a result, any program that uses the TEXT_IO package automatically reads the startup, assign, and param messages on \$RECEIVE. To use any of the information in the messages, the program must also name the COMMAND_INTERPRETER_PACKAGE in a with clause.

File Open Mode

An open-spec in the FORM parameter specifies the operating system open mode for the file. The value of the open-spec depends on the kind of file and, for disk files, the purpose for which the program opens the file. You can specify only EXCLUSIVE for the following:

- Creation of a disk file
- Opening a DIRECT_IO file with mode in out or mode out

The value is always SHARED for terminals and spoolers.

Resetting any kind of file to mode in does not affect the open-spec previously specified for the file. However, if an Ada program resets a DIRECT_IO file to mode in out or mode out, the run-time routines close and reopen the file with an open-spec of EXCLUSIVE, even if it was not EXCLUSIVE before. If an Ada program opens or resets a SEQUENTIAL_IO or TEXT_IO file with mode in out, the run-time routines recreate the file, and the default open-spec is EXCLUSIVE; you can override the open-spec only if the program is opening the file.

Record Length Specification

You can specify the RECORDLEN option in a create-spec only when the program uses TEXT_IO with a relative or entry-sequenced file. For either type of file, an attempt to specify a record length larger than 1320 raises USE_ERROR. If you do not specify the record length, TEXT_IO uses 132 for a relative file or 1320 for an entry-sequenced file.

An attempt to specify RECORDLEN raises USE_ERROR when the program uses DIRECT_IO, SEQUENTIAL_IO, or TEXT_IO with an edit-format file, or TEXT_IO with an unstructured file.

IMPLEMENTATION-DEPENDENT CHARACTERISTICS

File Types

File Types

TEXT_IO supports four Tandem disk file types:

- Unstructured
- Relative
- Entry sequenced
- Edit format

The default file type for TEXT_IO is edit format.

TEXT_IO also supports terminals and spoolers. The first time a program opens a spooler file, the file uses level 3 protocols. However, subsequent spooler files do not use level 3 protocols.

The range for TEXT_IO.COUNT and DIRECT_IO.COUNT is
0 .. LONG_INTEGER'LAST. The range for TEXT_IO.FIELD is
0 .. INTEGER'LAST.

Pages

For all disk file types except unstructured, a form feed (ASCII.FF) in a record by itself marks the end of a page. For unstructured files, a form feed followed by a line feed (ASCII.LF) marks the end of a page.

Lines

The maximum line length for TEXT_IO is 1320.

No character marks the end of a line in a relative, entry-sequenced, or edit-format file.

For an odd unstructured file (ODDUNSTR parameter set), ASCII.LF (line feed) marks the end of a line. For an even unstructured file (ODDUNSTR parameter not set), ASCII.LF marks the end of a line that ends at an odd byte, and two ASCII.LF characters mark the end of a line that ends at an even byte.

USE_ERROR Exception

SEQUENTIAL_IO supports only entry-sequenced files, and DIRECT_IO supports only relative files. An attempt to use SEQUENTIAL_IO or DIRECT_IO to create any other type of file raises the exception USE_ERROR.

The compiler cannot instantiate SEQUENTIAL_IO or DIRECT_IO for an unconstrained type, except for a record that has discriminants with default expressions, in which case the compiler chooses the record length needed for the largest possible object of the type. You cannot specify the record length option for a file that you use through the facilities of SEQUENTIAL_IO or DIRECT_IO.

The following conditions also raise USE_ERROR:

- An attempt to create a new file with the same name as an existing file
- An attempt to create a file for input
- An attempt to create or open a DIRECT_IO or SEQUENTIAL_IO file with record size 0
- Any error in the FORM parameter of the OPEN or CREATE procedure
- In TEXT_IO, attempting to open an existing entry-sequenced or relative file with a maximum record length larger than 1320
- In TEXT_IO, attempting to specify any file characteristics when creating an edit-format file

An attempt to use DIRECT_IO to read a nonexistent record raises DATA_ERROR.

File Closing

If a program terminates normally, the run-time routines automatically close STANDARD OUTPUT. The run-time routines never close other files automatically, so we recommend that you always explicitly close any files that you open. For example, if you did not close an edit-format file, it would be left in an inconsistent state, and the last buffer would be missing from the first spooler file that you wrote.

IMPLEMENTATION-DEPENDENT CHARACTERISTICS

Operating System File Names

Output Files

When a process opens an existing SEQUENTIAL_IO or TEXT_IO file for output, the run-time routines delete the file and recreate it with the same characteristics. The FORM parameter of the OPEN procedure provides the open-spec value. The default value for open-spec is EXCLUSIVE for a disk file or SHARED for any other kind of file.

When a process resets a file for output, the run-time routines delete and recreate the file with the same characteristics. The exclusion mode is EXCLUSIVE for a disk file and SHARED for any other kind of file.

Operating System File Names

Each external file name in an Ada program must be identical to an operating system file name. Operating system file names have an external and internal form, and Ada programs use only the external form. These file names can also be logical file names, if you use the DEFMODE ON run option when you execute the Ada program.

If a parameter of a TAL procedure uses the internal form of a file name, and an Ada program calls the procedure directly, the Ada run-time routines convert the external name to the internal form. If the TAL procedure passes an internal name back to an Ada program, the Ada run-time routines convert the name to the external form.

If your Ada programs have TAL procedures that call other TAL procedures, you might need to convert file names. To convert the external form to the internal form, you can use the FNAMEEXPAND system procedure. To convert the internal form to the external form, you can use the FNAMECOLLAPSE system procedure.

Disk file names have the following format:

`[\system.]file-name`

File-name has the following format:

`[$volume.][subvolume.]disk-file-name`

If file-name does not include volume or subvolume, the compiler uses the default volume or subvolume name provided by the COMMAND_INTERPRETER_INTERFACE package. TEXT_IO does not add the system name if the program does not provide it.

IMPLEMENTATION-DEPENDENT CHARACTERISTICS
Operating System File Names

The file names for processes and devices have the following format:

```
[\system.]{ $process }[. #name1[.name2]]  
          { $device }
```

A logical file name, called a DEFINE, has the following format:

=name

To create a DEFINE and assign a file name to it, you can use the command interpreter ADD DEFINE command, in the following format:

ADD DEFINE =name, FILE file-name

To create a DEFINE from a program, use the ADDDEFINE system procedure instead of the ADD DEFINE command.

For more information on operating system file names and DEFINES, see the *GUARDIAN Operating System User's Guide* and the *Labeled-Tape Support Manual*. For information about the FNAMEEXPAND, FNAMECOLLAPSE, and ADDDEFINE procedures, see the *System Procedure Calls Reference Manual*.

IMPLEMENTATION-DEPENDENT CHARACTERISTICS
Calling Sequences for External Subprograms

CALLING SEQUENCES FOR EXTERNAL SUBPROGRAMS

The order in which formal parameters appear in a subprogram declaration is the order for pushing the corresponding actual parameters onto the data stack. Parameter passing is either by value or by reference. Some parameters also require passing a descriptor. The type mark and the mode of the formal parameter determines the method for passing values.

Function subprograms have return values. Functions return results either by value or by reference. A result that a function returns by reference can be either of the following:

- 32-bit extended address
- 64-bit extended address pair

Parameter Types

The following subsections describe parameter-passing conventions for different data types.

Scalar Types

The calling program passes scalar-type parameters of mode in by value in 16, 32, or 64 bits. For objects less than 16 bits in length, such as short enumeration types and BOOLEAN, CHARACTER, and SHORT_INTEGER types, the program passes the values in 16-bit containers, right justified.

| The calling program passes scalar-type parameters of mode out and
| mode in out by reference to a temporary variable. This is also
| known as call by value result. The compiler creates a temporary
| variable for each actual parameter. The calling program passes
| the temporary variable addresses to the called subprogram. For
| objects that are 8 bits in length, the program passes 32-bit
| extended addresses. For all other objects, the program passes
| 16-bit word addresses.

Array and Record Types

For array and record types, the calling program passes parameters of all modes by reference. If an actual parameter specifies a type conversion, the program passes the address of a temporary variable. Otherwise, the program passes the address of the actual parameter itself.

If the formal type mark of the actual parameter is constrained or unconstrained, the calling program passes a 32-bit extended address for the actual parameter. If the type mark is unconstrained, the program also passes an additional 32-bit extended address of the descriptor of the actual parameter. The descriptor pointer comes immediately after the pointer to the object itself.

Access Types

For access types, the calling program passes parameters of mode in by value in either 32 or 64 bits. If the formal type mark is constrained, the program passes a 32-bit extended address of the object. If the type mark is unconstrained, the program passes a 64-bit value, consisting of a 32-bit extended address of the object followed by a 32-bit extended address of the object descriptor. If the access type denotes an accessed type that is not completed in the same compilation with its incomplete type declaration, the size is 64 bits.

The calling program passes access-type parameters of mode out and mode in out by reference to a temporary variable, using 16-bit word addresses, just like for scalar types. The compiler creates a temporary variable for each actual parameter. The calling program passes the temporary variable addresses to the called subprogram. Each address points to a 32-bit or 64-bit object, depending on whether the accessed type is constrained or unconstrained and on whether the accessed type is completed in the same compilation, as described in the preceding paragraph.

Static Link for Nested Subprograms

A *nested subprogram* is a subprogram to which the stack frame of some lexically enclosing subprogram, block, or task is visible. Every nested subprogram has a static link that the subprogram uses to refer to objects allocated in enclosing stack frames.

IMPLEMENTATION-DEPENDENT CHARACTERISTICS

Function Returns

The static link is a 16-bit quantity that contains the address of the stack mark's L-register component of the immediate lexically enclosing stack frame. The calling program passes the static link value to a nested subprogram by value as the last actual parameter. This affects TAL procedures that have nested Ada specifications. Such procedures should expect an extra 16-bit value passed on the stack.

Function Returns

The following subsections describe how functions return results of different data types.

Scalar Types

For scalar types, functions return results by value on the register stack. Results are 16, 32, or 64 bits long. Functions return scalar results shorter than 16 bits in 16-bit containers, right justified, with sign extension for signed values and zero extension for unsigned values.

Composite Types

For array types and record types, functions return results by reference as if the functions were access types. If the type mark of the function return value is constrained, then the function returns the 32-bit extended address of the result. If the type mark is unconstrained, then a 64-bit value is returned, including a 32-bit extended address of the array or record followed by a 32-bit extended address of the descriptor of the result.

Access Types

For access types, functions return results by value in either 32 or 64 bits, as for scalar types.

GENERIC INSTANTIATIONS

The compiler expands generic units at the point of instantiation. A generic body must be in the same compilation as its specification and must occur after the specification. Any subunits of a generic unit must also be part of the same compilation that the parent is in and must follow the parent in the compilation.

MEMORY USAGE ON NONSTOP SYSTEMS

A program running under the GUARDIAN 90 operating system stores data either in its 128K-byte data segment or in extended memory. A program's extended memory can consist of several extended segments, each as large as 128 megabytes. An Ada program uses only one extended segment, whose initial size can grow to 128 megabytes, as needed.

Main Task and Active Task

An Ada program contains a main task and possibly other tasks. At any time only one task is active. Whether active or not, a task has a stack of frames in which it stores objects.

For the main task, the stack contains objects declared within the main program, objects declared within subprograms called from the main program, and so on. When the program creates a new task, the frames visible then are logically part of the new task's stack, which diverges after that.

Memory Stack and Data Segment

The first half (64K bytes) of the data segment is called the memory stack. It contains the stack frames visible to the active task. Composite objects are not stored in these frames.

When a different task becomes active, frames no longer visible to the new task are swapped out of the memory stack. Frames that are visible to the new task but were not visible to the previously active task are swapped in.

The second half (64K bytes) of the data segment contains global package data for the various packages in the program, except for composite objects.

IMPLEMENTATION-DEPENDENT CHARACTERISTICS

Extended Stacks

Extended Stacks

Each task, including the main task, also has an extended stack. Frames in the extended stack correspond either to frames in the memory stack or to frames swapped out of it. Extended stack frames store composite objects.

You can specify the size of the extended stack for the main task in the EXTENDED_STACK_SIZE switch of the ADABIND command. The default size is 2^{18} bytes, and you can specify a size greater than 0 and less than 2^{27} bytes.

The size of extended stacks for tasks other than the main task is 256K bytes, by default. You can specify a different value in a STORAGE_SIZE representation clause for the corresponding task type.

Extended Data Segment

The extended data segment contains the extended stacks, composite global package data, frames swapped out of the memory stack, objects created by allocators, and other program data. The compiler reclaims the space from any temporary variables it creates for execution of a subprogram. There is no way to reuse the space from objects created by allocators.

If the extended data segment needs to be larger than 128 megabytes for the program to run, the program raises the exception STORAGE_ERROR. It also raises STORAGE_ERROR if any task needs more space than that allocated for its extended stack, if the memory stack needs more than 64K bytes, or if the noncomposite global package data needs more than 64K bytes.

COMPLETION CODES FOR COMPILER AND ADABIND PROCESSES

Each compiler or ADABIND process supplies a completion code value upon exit to the operating system. Table F-5 shows the possible completion code values and their meanings.

Table F-5. Completion Codes

Value	Meaning
0	The process terminated normally with no errors.
1	Warning messages were issued.
2	The compiler or ADABIND process detected one or more fatal errors, including source code errors or the inability to open specified files.
5	The process terminated abnormally due to an error in the compiler or ADABIND.

IMPLEMENTATION LIMITS

Table F-6 lists some Tandem Ada limits on the use of language features.

IMPLEMENTATION-DEPENDENT CHARACTERISTICS
Implementation Limits

Table F-6. Implementation Limits

Language Feature	Maximum Number
Characters in an identifier	200
Characters in a line	200
Discriminants in a constraint	256
Associations in a record aggregate	256
Fields in a record aggregate	256
Formal parameters in a generic unit	256
Nested contexts	250
Bytes for an object	32766
Words of object code for a subprogram	32767
Library units in a program	500
Compilation units and subprograms in a program (The compiler reserves approximately 1,000 entries for run-time routines.)	~15000
Units named in a compilation unit's with clauses	255
Dynamic components in a record	256
Array dimensions	7
Control statement nesting level	256
Literals for an enumeration type	32767
Tasks for a program	32767
Entries for a task	32767
Subprogram nesting level in a calling sequence (For example, f(f(f(x))) has 3 nesting levels.)	256
Unique strings and identifiers for a compilation unit	4096

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.

<u>Name and Meaning</u>	<u>Value</u>
\$BIG_ID1 Identifier the size of the maximum input line length with varying last character.	(1..199 => 'A', 200 => '1')
\$BIG_ID2 Identifier the size of the maximum input line length with varying last character.	(1..199 => 'A', 200 => '2')
\$BIG_ID3 Identifier the size of the maximum input line length with varying middle character.	(1..99 => 'A', 100 => '3', 101..200 => 'A')
\$BIG_ID4 Identifier the size of the maximum input line length with varying middle character.	(1..99 => 'A', 100 => '4', 101..200 => '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.	(1..197 => '0', 198..200 => "298")

TEST PARAMETERS

<u>Name and Meaning</u>	<u>Value</u>
\$BIG_REAL_LIT A real literal that can be either of floating- or fixed-point type, has value 690.0, and has enough leading zeroes to be the size of the maximum line length.	(1..194 => '0', 195..200 => "(.0E1)")
\$BLANKS A sequence of blanks twenty characters fewer than the size of the maximum line length.	(1..180 => ' ')
\$COUNT_LAST A universal integer literal whose value is TEXT_IO.COUNT'LAST.	2_147_483_647
\$EXTENDED_ASCII_CHARS A string literal containing all the ASCII characters with printable graphics that are not in the basic 55 Ada character set.	"abcdefghijklmnopqrstuvwxyz!\$%?@[\]^_`{ }~"
\$FIELD_LAST A universal integer literal whose value is TEXT_IO.FIELD'LAST.	32767
\$FILE_NAME_WITH_BAD_CHARS An illegal external file name that either contains invalid characters, or is too long if no invalid characters exist.	x}}!@
\$FILE_NAME_WITH_WILD_CARD_CHAR An external file name that either contains a wild card character, or is too long if no wild card character exists.	XYZ*
\$GREATER_THAN_DURATION A universal real value that lies between DURATION'BASE'LAST and DURATION'LAST if any, otherwise any value in the range of DURATION.	100_000.0
\$GREATER_THAN_DURATION_BASE_LAST The universal real value that is greater than DURATION'BASE'LAST, if such a value exists.	100_000_000.0

TEST PARAMETERS

<u>Name and Meaning</u>	<u>Value</u>
\$ILLEGAL_EXTERNAL_FILE_NAME1 An illegal external file name.	bad-character**
\$ILLEGAL_EXTERNAL_FILE_NAME2 An illegal external file name that is different from \$ILLEGAL_EXTERNAL_FILE_NAME1.	muchtoolongname
\$INTEGER_FIRST The universal integer literal expression whose value is INTEGER'FIRST.	-32768
\$INTEGER_LAST The universal integer literal expression whose value is INTEGER'LAST.	32767
\$LESS_THAN_DURATION A universal real value that lies between DURATION'BASE'FIRST and DURATION'FIRST if any, otherwise any value in the range of DURATION.	-100_000.0
\$LESS_THAN_DURATION_BASE_FIRST The universal real value that is less than DURATION'BASE'FIRST, if such a value exists.	-100_000_000.0
\$MAX_DIGITS The universal integer literal whose value is the maximum digits supported for floating-point types.	16
\$MAX_IN_LEN The universal integer literal whose value is the maximum input line length permitted by the implementation.	200
\$MAX_INT The universal integer literal whose value is SYSTEM.MAX_INT.	9_223_372_036_854_775_807

TEST PARAMETERS

<u>Name and Meaning</u>	<u>Value</u>
\$NAME A name of a predefined numeric type other than FLOAT, INTEGER, SHORT_FLOAT, SHORT_INTEGER, LONG_FLOAT, or LONG_INTEGER if one exists, otherwise any undefined name.	LONG_LONG_INTEGER
\$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#FFFF_FFFF_FFFF_FFFE#
\$NON_ASCII_CHAR_TYPE An enumerated type definition for a character type whose literals are the identifier NON_NULL and all non-ASCII characters with printable graphics.	(NON_NULL)

APPENDIX D

WITHDRAWN TESTS

Some tests are withdrawn from the ACVC because they do not conform to the Ada Standard. The following 19 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.

- . C32114A: An unterminated string literal occurs at line 62.
- . B33203C: The reserved word "IS" is misspelled at line 45.
- . C34018A: The call of function G at line 114 is ambiguous in the presence of implicit conversions.
- . C35904A: The elaboration of subtype declarations SFX3 and SFX4 may raise NUMERIC_ERROR instead of CONSTRAINT_ERROR as expected in the test.
- . B37401A: The object declarations at lines 126 through 135 follow subprogram bodies declared in the same declarative part.
- . C41404A: The values of 'LAST and 'LENGTH are incorrect in the if statements from line 74 to the end of the test.
- . B45116A: ARRPRIBL1 and ARRPRIBL2 are initialized with a value of the wrong type--PRIBOOL_TYPE instead of ARRPRIBOOL_TYPE--at line 41.
- . C48008A: The assumption that evaluation of default initial values occurs when an exception is raised by an allocator is incorrect according to AI-00397.
- . B49006A: Object declarations at lines 41 and 50 are terminated incorrectly with colons, and end case; is missing from line 42.
- . B4A010C: The object declaration in line 18 follows a subprogram body of the same declarative part.

WITHDRAWN TESTS

- . B74101B: The begin at line 9 causes a declarative part to be treated as a sequence of statements.
- . C87B50A: The call of "/"= at line 31 requires a use clause for package A.
- . C92005A: The "/"= for type PACK.BIG_INT at line 40 is not visible without a use clause for the package PACK.
- . C940ACA: The assumption that allocated task TT1 will run prior to the main program, and thus assign SPYNUMB the value checked for by the main program, is erroneous.
- . CA3005A..D (4 tests): No valid elaboration order exists for these tests.
- . BC3204C: The body of BC3204C0 is missing.

FILMED
ON 8