DOCUMENTATION PAGE

Form Approved
OPM No. 0704-0188

(2)

AD-A240 502

1. AGE...	DATE	3. REPORT TYPE AND DATES COVERED
		Final 01 Aug 1991 to 01 Jun 1993

4. TITLE AND SUBTITLE
Wang Laboratories, Inc., Wang VS Ada Version 5.00.00, Wang VS 8480 under Wang VSOS 7.30.02 (Host & Target), 901129W1.11093

5. FUNDING NUMBERS

6. AUTHOR(S)
Wright-Patterson AFB, Dayton, OH
USA

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Ada Validation Facility, Language Control Facility ASD/SCEL
Bldg. 676, Rm 135
Wright-Patterson AFB
Dayton, OH 45433

8. PERFORMING ORGANIZATION REPORT NUMBER
AVF-VSR-418-0891

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Ada Joint Program Office
United States Department of Defense
Pentagon, Rm 3E114
Washington, D.C. 20301-3081

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*
Wang Laboratories, Inc., Wang VS Ada Version 5.00.00, Wright-Patterson AFB, Wang VS 8480 under Wabng VSOS 7.30.02 (Hos & Target), ACVC 1.11.

91-10994

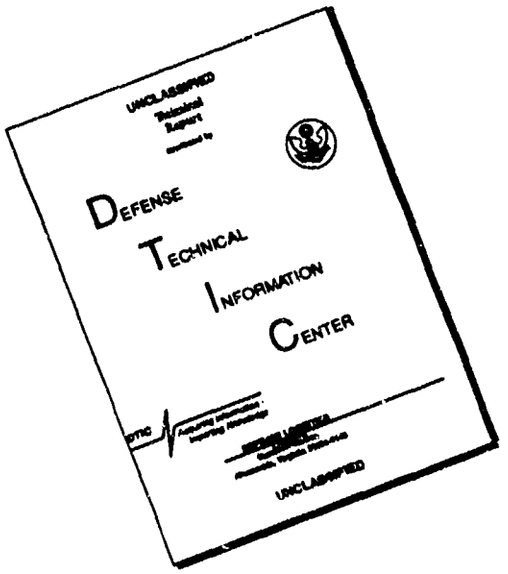
14. SUBJECT TERMS
Ada programming language, Ada Compiler Val. Summary Report, Ada Compiler Val. Capability, Val. Testing, Ada Val. Office, Ada Val. Facility, ANSI/MIL-STD-1815A, AJPO.

15. NUMBER OF PAGES

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFED	UNCLASSIFIED	

DISCLAIMER NOTICE



THIS DOCUMENT IS BEST QUALITY AVAILABLE. THE COPY FURNISHED TO DTIC CONTAINED A SIGNIFICANT NUMBER OF PAGES WHICH DO NOT REPRODUCE LEGIBLY.

AVF Control Number: AVF-VSR-418-0891
1 August 1991
90-07-30-WAN

Ada COMPILER
VALIDATION SUMMARY REPORT:
Certificate Number: 901129W1.11093
Wang Laboratories, Inc.
Wang VS Ada Version 5.00.00
Wang VS 8480 under Wang VSOS 7.30.02 =>
Wang VS 8480 under Wang VSOS 7.30.02

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

Certificate Information

The following Ada implementation was tested and determined to pass ACVC 1.11. Testing was completed on 29 December 1990.

Compiler Name and Version: Wang VS Ada Version 5.00.00

Host Computer System: Wang VS 8480 under Wang VSOS 7.30.02

Target Computer System: Wang VS 8480 under Wang VSOS 7.30.02

Customer Agreement Number: 90-07-30-WAN

See Section 3.1 for any additional information about the testing environment.

As a result of this validation effort, Validation Certificate 901129W1.11093 is awarded to Wang Laboratories, Inc. This certificate expires on 1 March 1993.

This report has been reviewed and is approved.

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

Ada Validation Organization
Director, Computer & Software Engineering Division
Institute for Defense Analyses
Alexandria VA 22311

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

DECLARATION OF CONFORMANCE

Compiler Implementor: Wang Laboratories, Inc.

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

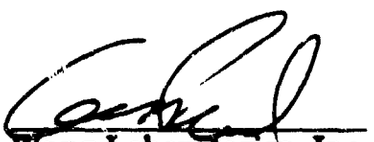
Ada Compiler Validation Capability (ACVC) Version: 1.11

Base Configuration

Base Compiler Name:	Wang VS Ada	Version: 5.00.00
Host Architecture ISA:	Wang VS 8480	OS&VER #: Wang VSOS 7.30.02
Target Architecture ISA:	Wang VS 8480	OS&VER #: Wang VSOS 7.30.02

Implementor's Declaration

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



Wang Laboratories, Inc.
Gerald Paul, Vice President,
Systems/Communications Development

10/5/90
Date

TABLE OF CONTENTS

CHAPTER 1 INTRODUCTION

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

CHAPTER 2 IMPLEMENTATION DEPENDENCIES

2.1 WITHDRAWN TESTS 2-1
2.2 INAPPLICABLE TESTS 2-1
2.3 TEST MODIFICATIONS 2-4

CHAPTER 3 PROCESSING INFORMATION

3.1 TESTING ENVIRONMENT 3-1
3.2 SUMMARY OF TEST RESULTS 3-1
3.3 TEST EXECUTION 3-2

APPENDIX A MACRO PARAMETERS

APPENDIX B COMPILATION SYSTEM OPTIONS

APPENDIX C APPENDIX F OF THE Ada STANDARD

CHAPTER 1

INTRODUCTION

The Ada implementation described above was tested according to the Ada Validation Procedures [Pro90] against the Ada Standard [Ada83] using the current Ada Compiler Validation Capability (ACVC). This Validation Summary Report (VSR) gives an account of the testing of this Ada implementation. For any technical terms used in this report, the reader is referred to [Pro90]. A detailed description of the ACVC may be found in the current ACVC User's Guide [UG89].

1.1 USE OF THIS VALIDATION SUMMARY REPORT

Consistent with the national laws of the originating country, the Ada Certification Body 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 implementation has no nonconformities to the Ada Standard other than those presented. Copies of this report are available to the public from the AVF which performed this validation or from:

National Technical Information Service
5285 Port Royal Road
Springfield VA 22161

Questions regarding this report or the validation test results should be directed to the AVF which performed this validation or to:

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

1.2 REFERENCES

[Ada83] Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815A, February 1983 and ISO 8652-1987.

[Pro90] Ada Compiler Validation Procedures, Version 2.1, Ada Joint Program Office, August 1990.

[UG89] Ada Compiler Validation Capability User's Guide, 21 June 1989.

1.3 ACVC TEST CLASSES

Compliance of Ada implementations is tested by means of the ACVC. The ACVC contains a collection of test 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. Class B and class L tests are expected to produce errors at compile time and link time, respectively.

The executable tests are written in a self-checking manner and produce a PASSED, FAILED, or NOT APPLICABLE message indicating the result when they are executed. Three Ada library units, the packages REPORT and SPPRT13, and the procedure CHECK_FILE are used for this purpose. The package REPORT 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 package SPPRT13 is used by many tests for Chapter 13 of the Ada Standard. The procedure CHECK_FILE is used to check the contents of text files written by some of the Class C tests for Chapter 14 of the Ada Standard. The operation of REPORT and CHECK_FILE is checked by a set of executable tests. If these units are not operating correctly, validation testing is discontinued.

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 all violations of the Ada Standard are detected. Some of the class B tests contain legal Ada code which must not be flagged illegal by the compiler. This behavior is also verified.

Class L tests check that an Ada implementation correctly detects violation of the Ada Standard involving multiple, separately compiled units. Errors are expected at link time, and execution is attempted.

In some tests of the ACVC, certain macro strings have to be replaced by implementation-specific values -- for example, the largest integer. A list of the values used for this implementation is provided in Appendix A. In addition to these anticipated test modifications, additional changes may be required to remove unforeseen conflicts between the tests and implementation-dependent characteristics. The modifications required for this implementation are described in section 2.3.

For each Ada implementation, a customized test suite is produced by the AVF. This customization consists of making the modifications described in the preceding paragraph, removing withdrawn tests (see section 2.1) and, possibly some inapplicable tests (see Section 2.2 and [UG89]).

In order to pass an ACVC an Ada implementation must process each test of the customized test suite according to the Ada Standard.

1.4 DEFINITION OF TERMS

Ada Compiler	The software and any needed hardware that have to be added to a given host and target computer system to allow transformation of Ada programs into executable form and execution thereof.
Ada Compiler Validation Capability (ACVC)	The means for testing compliance of Ada implementations, consisting of the test suite, the support programs, the ACVC user's guide and the template for the validation summary report.
Ada Implementation	An Ada compiler with its host computer system and its target computer system.
Ada Joint Program Office (AJPO)	The part of the certification body which provides policy and guidance for the Ada certification system.
Ada Validation Facility (AVF)	The part of the certification body which carries out the procedures required to establish the compliance of an Ada implementation.
Ada Validation Organization (AVO)	The part of the certification body that provides technical guidance for operations of the Ada certification system.
Compliance of an Ada Implementation	The ability of the implementation to pass an ACVC version.
Computer System	A functional unit, consisting of one or more computers and associated software, that uses common storage for all or part of a program and also for all or part of the data necessary for the execution of the program; executes user-written or user-designated programs; performs user-designated data manipulation, including arithmetic operations and logic operations; and that can execute programs that modify themselves during execution. A computer system may be a stand-alone unit or may consist of several inter-connected units.

CHAPTER 2

IMPLEMENTATION DEPENDENCIES

2.1 WITHDRAWN TESTS

The following tests have been withdrawn by the AVO. The rationale for withdrawing each test is available from either the AVO or the AVF. The publication date for this list of withdrawn tests is 11 November 1990.

E28005C	B28006C	C34006D	C35702A	B41308B	C43004A
C45114A	C45346A	C45612B	C45651A	C46022A	B49008A
A74006A	C74308A	B83022B	B83022H	B83025B	B83025D
B83026B	C83026A	C83041A	B85001L	C97116A	C98003B
BA2011A	CB7001A	CB7001B	CB7004A	CC1223A	BC1226A
CC1226B	BC3009B	BD1B02B	BD1B06A	AD1B08A	BD2A02A
CD2A21E	CD2A23E	CD2A32A	CD2A41A	CD2A41E	CD2A87A
CD2B15C	BD3006A	BD4008A	CD4022A	CD4022D	CD4024B
CD4024C	CD4024D	CD4031A	CD4051D	CD5111A	CD7004C
ED7005D	CD7005E	AD7006A	CD7006E	AD7201A	AD7201E
CD7204B	BD8002A	BD8004C	CD9005A	CD9005B	CDA201E
CE2107I	CE2117A	CE2117B	CE2119B	CE2205B	CE2405A
CE3111C	CE3116A	CE3118A	CE3411B	CE3412B	CE3607B
CE3607C	CE3607D	CE3812A	CE3814A	CE3902B	

2.2 INAPPLICABLE TESTS

A test is inapplicable if it contains test objectives which are irrelevant for a given Ada implementation. Reasons for a test's inapplicability may be supported by documents issued by ISO and the AJPO known as Ada Commentaries and commonly referenced in the format AI-ddddd. For this implementation, the following tests were determined to be inapplicable for the reasons indicated; references to Ada Commentaries are included as appropriate.

The following 201 tests have floating-point type declarations requiring more digits than SYSTEM.MAX_DIGITS:

C24113L..Y (14 tests)	C35705L..Y (14 tests)
C35706L..Y (14 tests)	C35707L..Y (14 tests)
C35708L..Y (14 tests)	C35802L..Z (15 tests)
C45241L..Y (14 tests)	C45321L..Y (14 tests)
C45421L..Y (14 tests)	C45521L..Z (15 tests)
C45524L..Z (15 tests)	C45621L..Z (15 tests)
C45641L..Y (14 tests)	C46012L..Z (15 tests)

The following 21 tests check for the predefined type LONG_INTEGER:

C35404C	C45231C	C45304C	C45411C	C45412C
C45502C	C45503C	C45504C	C45504F	C45611C
C45612C	C45613C	C45614C	C45631C	C45632C
B52004D	C55B07A	B55B09C	B86001W	C86006C
CD7101F				

A35801E checks that FLOAT'FIRST..FLOAT'LAST may be used as a range constraint in a floating-point type declaration; for this implementation, that range exceeds the range of safe numbers of the largest predefined floating-point type and must be rejected. (See section 2.3.)

C35702B, C35713C, B86001U, and C86006G check for the predefined type LONG_FLOAT.

C35713D and B86001Z check for a predefined floating-point type with a name other than FLOAT, LONG_FLOAT, or SHORT_FLOAT.

C45531M..P and C45532M..P (8 tests) check fixed-point operations for types that require a SYSTEM.MAX_MANTISSA of 47 or greater; for this implementation, MAX_MANTISSA is less than 47.

C45536A, C46013B, C46031B, C46033B, and C46034B contain 'SMALL representation clauses which are not powers of two or ten.

C45624A checks that the proper exception is raised if MACHINE_OVERFLOWS is FALSE for floating point types with digits 5. For this implementation, MACHINE_OVERFLOWS is TRUE.

C45624B checks that the proper exception is raised if MACHINE_OVERFLOWS is FALSE for floating point types with digits 6. For this implementation, MACHINE_OVERFLOWS is TRUE.

C86001F recompiles package SYSTEM, making package TEXT_IO, and hence package REPORT, obsolete. For this implementation, the package TEXT_IO is dependent upon package SYSTEM.

B86001Y checks for a predefined fixed-point type other than DURATION.

C96005B checks for values of type DURATION'BASE that are outside the range of DURATION. There are no such values for this implementation.

CD1009C uses a representation clause specifying a non-default size for a floating-point type.

CD2A53A checks operations of a fixed-point type for which a length clause specifies a power-of-ten TYPE'SMALL; this implementation does not support decimal 'SMALLs. (See section 2.3.)

CD2A84A, CD2A84E, CD2A84I..J (2 tests), and CD2A840 use representation clauses specifying non-default sizes for access types.

BD8001A, BD8003A, BD8004A..B (2 tests), and AD8011A use machine code insertions.

EE2401D uses instantiations of package DIRECT_IO with unconstrained array types. This implementation raises USE_ERROR on the attempt to create a file of such type.

The tests listed in the following table check that USE_ERROR is raised if the given file operations are not supported for the given combination of mode and access method; this implementation supports these operations.

Test	File Operation	Mode	File Access Method
CE2102D	CREATE	IN_FILE	SEQUENTIAL_IO
CE2102E	CREATE	OUT_FILE	SEQUENTIAL_IO
CE2102F	CREATE	INOUT_FILE	DIRECT_IO
CE2102I	CREATE	IN_FILE	DIRECT_IO
CE2102J	CREATE	OUT_FILE	DIRECT_IO
CE2102N	OPEN	IN_FILE	SEQUENTIAL_IO
CE2102O	RESET	IN_FILE	SEQUENTIAL_IO
CE2102P	OPEN	OUT_FILE	SEQUENTIAL_IO
CE2102Q	RESET	OUT_FILE	SEQUENTIAL_IO
CE2102R	OPEN	INOUT_FILE	DIRECT_IO
CE2102S	RESET	INOUT_FILE	DIRECT_IO
CE2102T	OPEN	IN_FILE	DIRECT_IO
CE2102U	RESET	IN_FILE	DIRECT_IO
CE2102V	OPEN	OUT_FILE	DIRECT_IO
CE2102W	RESET	OUT_FILE	DIRECT_IO
CE3102E	CREATE	IN_FILE	TEXT_IO
CE3102F	RESET	Any Mode	TEXT_IO
CE3102G	DELETE	--------	TEXT_IO
CE3102I	CREATE	OUT_FILE	TEXT_IO
CE3102J	OPEN	IN_FILE	TEXT_IO
CE3102K	OPEN	OUT_FILE	TEXT_IO

The following 16 tests check operations on sequential, direct, and text files when multiple internal files are associated with the same external file and one or more are open for writing; USE_ERROR is raised when this association is attempted.

CE2107B..E	CE2107G..H	CE2107L	CD2110B	CE2110D
CE2111D	CE2111H	CE3111B	CE3111D..E	CE3114B
CE3115A				

CE2203A checks that WRITE raises USE_ERROR if the capacity of the external file is exceeded for SEQUENTIAL_IO. This implementation does not restrict file capacity.

CE2403A checks that WRITE raises USE_ERROR if the capacity of the external file is exceeded for DIRECT_IO. This implementation does not restrict file capacity.

CE3111B, CE3111D..E (2 tests), CE3114B, and CE3115A attempt to associate multiple internal files with the same external file when one or more files is writing for text files. The proper exception is raised when multiple access is attempted.

CE3304A checks that USE_ERROR is raised if a call to SET_LINE_LENGTH or SET_PAGE_LENGTH specifies a value that is inappropriate for the external file. This implementation does not have inappropriate values for either line length or page length.

CE3305A checks the SET_LINE_LENGTH procedure, including the particular call SET_LINE_LENGTH (UNBOUNDED); this implementation does not support unbounded lines.

CE3413B checks that PAGE raises LAYOUT_ERROR when the value of the page number exceeds COUNT'LAST. For this implementation, the value of COUNT'LAST is greater than 150000 making the checking of this objective impractical.

2.3 TEST MODIFICATIONS

Modifications (see section 1.3) were required for 22 tests.

The following tests were split into two or more tests because this implementation did not report the violations of the Ada Standard in the way expected by the original tests.

B23004A	B24007A	B24009A	B28003A	B28003C	B32202A
B32202B	B32202C	B33001A	B37004A	B45102A	B61012A
B62001B	B91004A	B95069A	B95069B	BA1101B1	BC2001D
BC3009C					

A35801E was graded inapplicable by Evaluation Modification as directed by the AVO. The compiler rejects the use of the range FLOAT'FIRST..FLOAT'LAST as the range constraint of a floating-point type declaration because the bounds lie outside of the range of safe numbers (cf. LRM 3.5.7:12). b. CD2A53A was graded inapplicable by Evaluation Modification as directed by the AVO. The test contains a specification of a power-of-10 value as 'SMALL for a fixed-point type. The AVO ruled that, under ACVC 1.11, support of decimal 'SMALLs may be omitted.

AD9001B was graded passed by Test Modification as directed by the AVO. This test checks that no bodies are required for interfaced subprograms; among the procedures that it uses is one with a parameter of mode OUT (line 36). This implementation does not support pragma INTERFACE for procedures with parameters of mode OUT. The test was modified by commenting out line 36; the modified test passed. b. EA3004D was graded passed by Evaluation and Processing Modification as directed by the AVO. The test requires that either pragma INLINE is obeyed for a function call in each of three contexts and that thus three library units are made obsolete by the re-compilation of the inlined function's body, or else the pragma is ignored completely. This implementation obeys the pragma except when the call is within the package specification. When the test's files are processed in the given order, only two units are made obsolete; thus, the expected error at line 27 of file EA3004D6M is not valid and is not flagged. To confirm that indeed the pragma is not obeyed in this one case, the test was also processed with the files re-ordered so that the re-compilation follows only the package declaration (and thus the other library units will not be made obsolete, as they are compiled later); a "NOT APPLICABLE" result was produced, as expected. The revised order of files was 0-1-4-5-2-3-6.

BA2001E was graded passed by Evaluation Modification as directed by the AVO. The test expects that duplicate names of subunits with a common ancestor will be detected as compilation errors; this implementation detects the errors at link time, and the AVO ruled that this behavior is acceptable.

CHAPTER 3

PROCESSING INFORMATION

3.1 TESTING ENVIRONMENT

The Ada implementation tested in this validation effort is described adequately by the information given in the initial pages of this report.

For a point of contact for technical information about this Ada implementation system, see:

Fred Rozakis
One Industrial Avenue
Lowell MA 01851

For a point of contact for sales information about this Ada implementation system, see:

Fred Rozakis
One Industrial Avenue
Lowell MA 01851

Testing of this Ada implementation was conducted at the customer's site by a validation team from the AVF.

3.2 SUMMARY OF TEST RESULTS

An Ada Implementation passes a given ACVC version if it processes each test of the customized test suite in accordance with the Ada Programming Language Standard, whether the test is applicable or inapplicable; otherwise, the Ada Implementation fails the ACVC [Pro90].

For all processed tests (inapplicable and applicable), a result was obtained that conforms to the Ada Programming Language Standard.

Total Number of Applicable Tests	3786
Total Number of Withdrawn Tests	83
Processed Inapplicable Tests	100
Non-Processed I/O Tests	0
Non-Processed Floating-Point Precision Tests	201
Total Number of Inapplicable Tests	301
Total Number of Tests for ACVC 1.11	4170

All I/O tests of the test suite were processed because this implementation supports a file system. The above number of floating-point tests were not processed because they used floating-point precision exceeding that supported by the implementation. When this compiler was tested, the tests listed in section 2.1 had been withdrawn because of test errors.

3.3 TEST EXECUTION

Version 1.11 of the ACVC comprises 4170 tests. When this compiler was tested, the tests listed in section 2.1 had been withdrawn because of test errors. The AVF determined that 303 tests were inapplicable to this implementation. All inapplicable tests were processed during validation testing except for 201 executable tests that use floating-point precision exceeding that supported by the implementation. In addition, the modified tests mentioned in section 2.3 were also processed.

MS-DOS diskettes containing the customized test suite (see section 1.3) were taken on-site by the validation team for processing. The contents of the diskettes were transferred to a PC. A batch file was used to upload the files to the host computer. The uploading was done using PCVS, a PC-to-VS file transfer program which uses a Wang 928 communications link.

After the test files were loaded onto the host computer, the full set of tests was processed by the Ada implementation. The test load was shared between two identical Wang VS 8480 configurations.

Testing was performed using command scripts provided by the customer and reviewed by the validation team. See Appendix B for a complete listing of the processing options for this implementation. It also indicates the default options. The options invoked explicitly for validation testing during this test were:

Option/Switch	Effect
Compiler:	
Debug=No	Do not generate symbolic debugger info

Level=Object	Generate object code
Generics=Yes	Inline code for generics
Errors=999	Permit 999 compilation errors
Memory=256	Reserve 256 kilobytes in memory for program library data
Annotate=No	Do not save comments in program library
Checks=All	Enable full run-time checking
Inline=Yes	Inline code
Reduct=No	Turn off high-level optimization
Express=No	Turn off low-level optimization
Peephole=Yes	Turn on peephole optimization
Source=Yes/No	Include source text in listing for E tests, .TST tests, and for tests which yield compilation errors. Do not include source text for the remainder.
Dmap=No	Do not include data map in listing
Pmap=No	Do not include program map in listing
Warning=No	Do not include warning messages in listing
Detail=No	Do not include extra info detail in listing
Linelen=79	Use 79 for width of listing file
Banner=Yes	Include banners in listing
Stack=1024	Allocate 1024 for stack objects
Global=1024	Allocate 1024 for global objects
Unnested=16	Allocate 16 for unnested objects

Binder:

Debug=No	Do not generate symbolic debugger info
Level=Link	Link after binding
Trace=No	Do not generate trace info for unhandled exceptions
Data=No	Do not include program data in listing
Modules=No	Do not link other modules with the Ada program module
Uncalled=Yes	Remove uncalled subprograms
Warning=Yes	Include warnings in bind listing
Stack=64	Reserve 64 kilobytes for program stack
Task=16	Reserve 16 kilobytes for each task stack
Heap=256	Reserve 256 kilobytes for initial program heap
Moreheap=4	Add 4 kilobytes to heap when it is full
Cancel=No	Do not intercept program cancels
Ioerrors=No	Do not display IO errors on workstation

Linker:

Map=Yes	Create a link map
Symbolic=Yes	Retain symbolic data
Linkage=Yes	Retain linkage data
Staticsl=No	Do not create a static subroutine library
Sharedsl=No	Do not create a shared subroutine library
Entnames=No	Do not select subroutine library entry names
Alias=No	Do not assign aliases to SSL symbols
Reorder=No	Do not reorder output program section
Resolve=Yes	Resolve undefined symbols interactively

Dupsect=No	Resolve duplicate section names interactively
Inprogrs=No	Do not display in-program screen
Objform=1	Use object file format '1'
Warnings=Yes	Print small link map of errors or warnings
Datesel=No	Do not select duplicate sections by date

Test output, compiler and linker 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.

APPENDIX A

MACRO PARAMETERS

This appendix contains the macro parameters used for customizing the ACVC. The meaning and purpose of these parameters are explained in [UG89]. The parameter values are presented in two tables. The first table lists the values that are defined in terms of the maximum input-line length, which is the value for $MAX_IN_LEN--also listed here. These values are expressed here as Ada string aggregates, where "V" represents the maximum input-line length.

Macro Parameter	Macro Value
$BIG_ID1	(1..V-1 => 'A', V => '1')
$BIG_ID2	(1..V-1 => 'A', V => '2')
$BIG_ID3	(1..V/2 => 'A') & '3' & (1..V-1-V/2 => 'A')
$BIG_ID4	(1..V/2 => 'A') & '4' & (1..V-1-V/2 => 'A')
$BIG_INT_LIT	(1..V-3 => '0') & "298"
$BIG_REAL_LIT	(1..V-5 => '0') & "690.0"
$BIG_STRING1	'"' & (1..V/2 => 'A') & '"'
$BIG_STRING2	'"' & (1..V-1-V/2 => 'A') & '1' & '"'
$BLANKS	(1..V-20 => ' ')
$MAX_LEN_INT_BASED_LITERAL	"2:" & (1..V-5 => '0') & "11:"
$MAX_LEN_REAL_BASED_LITERAL	"16:" & (1..V-7 => '0') & "F.E:"
$MAX_STRING_LITERAL	'"' & (1..V-2 => 'A') & '"'

MACRO PARAMETERS

The following table lists all of the other macro parameters and their respective values.

Macro Parameter	Macro Value
$MAX_IN_LEN	255
$ACC_SIZE	32
$ALIGNMENT	4
$COUNT_LAST	2147483647
$DEFAULT_MEM_SIZE	16777216
$DEFAULT_STOR_UNIT	8
$DEFAULT_SYS_NAME	WANG_VS
$DELTA_DOC	2#1.0#E-31
$ENTRY_ADDRESS	SYSTEM.NULL_ADDRESS
$ENTRY_ADDRESS1	SYSTEM.NULL_ADDRESS
$ENTRY_ADDRESS2	SYSTEM.NULL_ADDRESS
$FIELD_LAST	255
$FILE_TERMINATOR	' '
$FIXED_NAME	NO_SUCH_FIXED_TYPE
$FLOAT_NAME	NO_SUCH_TYPE
$FORM_STRING	""
$FORM_STRING2	"CANNOT_RESTRICT_FILE_CAPACITY"
$GREATER_THAN_DURATION	100000.0
$GREATER_THAN_DURATION_BASE_LAST	10000000.0
$GREATER_THAN_FLOAT_BASE_LAST	7.23700557733226E+75
$GREATER_THAN_FLOAT_SAFE_LARGE	16#0.FFFFFFFFFFFFF9#E+63

```
$GREATER_THAN_SHORT_FLOAT_SAFE_LARGE
                        7.23701E+75

$HIGH_PRIORITY          1

$ILLEGAL_EXTERNAL_FILE_NAME1
                        BAD-FNAM

$ILLEGAL_EXTERNAL_FILE_NAME2
                        THIS-FILE-NAME-IS-TOO-LONG-FOR-MY-SYSTEM

$INAPPROPRIATE_LINE_LENGTH
                        -1

$INAPPROPRIATE_PAGE_LENGTH
                        -1

$INCLUDE_PRAGMA1        PRAGMA INCLUDE ("A28006D1.TST")

$INCLUDE_PRAGMA2        PRAGMA INCLUDE ("B28006F1.TST")

$INTEGER_FIRST          -2147483648

$INTEGER_LAST           214748647

$INTEGER_LAST_PLUS_1    2147483648

$INTERFACE_LANGUAGE     C

$LESS_THAN_DURATION     -100000.0

$LESS_THAN_DURATION_BASE_FIRST
                        -10000000.0

$LINE_TERMINATOR        ' '

$LOW_PRIORITY           1

$MACHINE_CODE_STATEMENT
                        NULL;

$MACHINE_CODE_TYPE      NO_SUCH_TYPE

$MANTISSA_DOC           31

$MAX_DIGITS             15

$MAX_INT                2147483647

$MAX_INT_PLUS_1         2147483648

$MIN_INT                2147483647
```

MACRO PARAMETERS

$NAME	SHORT_SHORT_INTEGER
$NAME_LIST	WANG_VS
$NAME_SPECIFICATION1	ACVC.ACVCDATA.X2120A
$NAME_SPECIFICATION2	ACVC.ACVCDATA.X2120B
$NAME_SPECIFICATION3	ACVC.ACVCDATA.X3119A
$NEG_BASED_INT	16#FFFFFFFF#
$NEW_MEM_SIZE	0
$NEW_STOR_UNIT	0
$NEW_SYS_NAME	WANG_VS
$PAGE_TERMINATOR	' '
$RECORD_DEFINITION	NEW INTEGER;
$RECORD_NAME	NO_SUCH_MACHINE_CODE_TYPE
$TASK_SIZE	32
$TASK_STORAGE_SIZE	16384
$TICK	0.01
$VARIABLE_ADDRESS	V_ADDRESS
$VARIABLE_ADDRESS1	V_ADDRESS1
$VARIABLE_ADDRESS2	V_ADDRESS2
$YOUR_PRAGMA	PACK

APPENDIX B

COMPILATION SYSTEM OPTIONS

The compiler options of this Ada implementation, as described in this Appendix, are provided by the customer. Unless specifically noted otherwise, references in this appendix are to compiler documentation and not to this report.

COMPILATION SYSTEM OPTIONS

Option/Switch	Effect
Debug=No	Do not generate symbolic debugger info
Level=Object	Generate object code
Generics=Yes	Inline code for generics
Errors=50	Permit 50 compilation errors
Memory=500	Reserve 500 kilobytes in memory for program library data
Annotate=No	Do not save comments in program library
Checks=All	Enable full run-time checking
Inline=No	Do not Inline code
Reduct=No	Turn off high-level optimization
Express=No	Turn off low-level optimization
Peephole=Yes	Turn on peephole optimization
Source=No	Do not include source text in listing
Dmap=No	Do not include data map in listing
Pmap=No	Do not include program map in listing
Warning=Yes	Include warning messages in listing
Detail=Yes	Include extra info detail in listing
Linelen=132	Use 132 for width of listing file
Banner=Yes	Include banners in listing
Stack=1024	Allocate 1024 for stack objects
Global=1024	Allocate 1024 for global objects
Unnested=16	Allocate 16 for unnested objects

LINKER OPTIONS

The linker options of this Ada implementation, as described in this Appendix, are provided by the customer. Unless specifically noted otherwise, references in this appendix are to linker documentation and not to this report.

COMPILATION SYSTEM OPTIONS

Option/Switch	Effect
Binder:	
Debug=No	Do not generate symbolic debugger info
Level=Link	Link after binding
Trace=No	Do not generate trace info for unhandled exceptions
Data=No	Do not include program data in listing
Modules=No	Do not link other modules with the Ada program module
Uncalled=Yes	Remove uncalled subprograms
Warning=Yes	Include warnings in bind listing
Stack=64	Reserve 64 kilobytes for program stack
Task=16	Reserve 16 kilobytes for each task stack
Heap=256	Reserve 256 kilobytes for initial program heap
Moreheap=4	Add 4 kilobytes to heap when it is full
Cancel=No	Do not intercept program cancels
Ioerrors=No	Do not display I0 errors on workstation
Linker:	
Map=Yes	Create a link map
Symbolic=Yes	Retain symbolic data
Linkage=Yes	Retain linkage data
Staticsl=No	Do not create a static subroutine library
Sharedsl=No	Do not create a shared subroutine library
Entnames=No	Do not select subroutine library entry names
Alias=No	Do not assign aliases to SSL symbols
Reorder=No	Do not reorder output program section
Resolve=Yes	Resolve undefined symbols interactively
Dupsect=No	Resolve duplicate section names interactively
Inprogrs=No	Do not display in-program screen
Objform=1	Use object file format '1'
Warnings=Yes	Print small link map of errors or warnings
Datesel=No	Do not select duplicate sections by date

APPENDIX C

APPENDIX F OF THE Ada STANDARD

The only allowed implementation dependencies correspond to implementation-dependent pragmas, to certain machine-dependent conventions as mentioned in Chapter 13 of the Ada Standard, and to certain allowed restrictions on representation clauses. The implementation-dependent characteristics of this Ada implementation, as described in this Appendix, are provided by the customer. Unless specifically noted otherwise, references in this Appendix are to compiler documentation and not to this report. Implementation-specific portions of the package STANDARD, which are not a part of Appendix F, are:

```
package STANDARD is

    ...

    type INTEGER is range -2147483648 .. 2147483647;

    type FLOAT is digits 15
         range -16#0.FFFFFFFFFFFFFF#E+63 .. 16#0.FFFFFFFFFFFFFF#E+63;

    type DURATION is delta 2#0.000001# range -131072.0 .. 131071.0;

    type SHORT_INTEGER is range -32768 .. 32767;

    type SHORT_FLOAT is digits 6 range
                        -16#0.FFFFFF#E+63 .. 16#0.FFFFFF#E+63;

    type SHORT_SHORT_INTEGER is range -128 .. 127;

    ...


end STANDARD;
```

APPENDIX F
IMPLEMENTATION-DEPENDENT CHARACTERISTICS

This appendix describes the implementation-dependent characteristics of the VS Ada compiler. Appendix F is a required part of the LRM.

Appendix F contains the following sections:

F.1 Implementation-dependent pragmas

F.2 Implementation-dependent attributes

F.3 Specification of the package SYSTEM

F.4 Restrictions on representation clauses

F.5 Conventions for implementation-generated names

F.6 Address clauses

F.7 Restrictions on unchecked conversions

F.8 Implementation-dependent characteristics of the input-output packages

F.9 Characteristics of numeric types

F.10 Other implementation-dependent characteristics

Throughout this appendix, citations in square brackets refer to the relevant sections of the *Reference Manual for the Ada Program Language* (LRM).

F.1 IMPLEMENTATION-DEPENDENT PRAGMAS

F.1.1 Pragma INLINE

Pragma INLINE is fully supported by VS Ada with one exception: a function that is called in a declarative part cannot be expanded inline.

F.1.2 Pragma INTERFACE

Ada programs can interface to subprograms written in assembler or other languages through the predefined pragma INTERFACE [13.9] and the VS Ada-defined pragma INTERFACE_NAME. Pragma INTERFACE is described in this section. For a description of pragma INTERFACE_NAME, see Section F.1.3.

Pragma INTERFACE specifies the name of an interfaced subprogram and the name of the programming language for which calling and parameter-passing conventions are generated. The pragma takes the form specified in the LRM:

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

where

language_name is the name of the language whose calling and parameter-passing conventions are to be used;

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

Two language names are currently accepted by pragma INTERFACE: CLE and C. The language name CLE refers to the standard VS Common Language Environment calling and parameter-passing conventions. You can use the language name CLE to interface Ada subprograms with subroutines written in any language that follows the standard VS calling conventions: BASIC, COBOL, FORTRAN, PL1, RPGII, and some C functions. You can also call Assembler programs with CLE, although there is nothing inherent in the Assembler language to expect or support this calling convention, as there is in the other languages listed.

You must use the language name C when calling VS C programs that expect the normal C calling convention. You should, however, use pragma INTERFACE CLE to call C functions that are declared with the option cle.

Calling Conventions

For both CLE and C pragma INTERFACE language names, the machine state is saved as part of the process of calling the interfaced subprogram. This is accomplished by the JSCI machine instruction, which saves the register state on the VS system stack prior to jumping to the subroutine. The Ada environment is properly restored from this save area by a normal return from the subprogram. From an Assembler subprogram, this return must be performed by the RT machine instruction.

The Ada runtime system treats any program interruption occurring during execution of the body of the interfaced subprogram as an exception being raised at the point of call of the subprogram. See Section 5.2 for a discussion of how non-Ada errors are mapped onto Ada exceptions.

Interfaced subprograms called with pragma INTERFACE may not issue any form of PCEXIT SVC. Interfaced subprograms may also be restricted from issuing any form of CEXIT SVC.

Parameter-Passing Conventions

The following conventions apply to both of the language names, CLE and C, currently accepted by pragma INTERFACE:

- Register 1 contains the address of the parameter area on entry to the subprogram.
- No consistency checking is performed between the subprogram parameters declared in Ada and the corresponding parameters of the interfaced subprogram. It is your responsibility to ensure correct access to the parameters.
- Formal parameters of mode out are not allowed.

For pragma INTERFACE CLE, register 1 contains the address of a parameter address list. Each word in this list is an address corresponding to a parameter. The most significant (sign) bit of the last word in the list is set to indicate the end of the list.

If the subprogram is a function, the result is returned in an area allocated by the Ada program and addressed by the first address in the parameter address list.

For formal parameters of mode in out, the address passed is that of the actual parameter. This provides a true pass by reference from Ada to the CLE interfaced subprogram for parameters of all types: scalar, access, record, and array. For a record type, the address in the parameter list is that of the first component of the record. For an array type, the address in the parameter list is that of the first element of the array.

Formal parameters of mode in are also all allowed for CLE interfaced subprograms. For parameters of scalar and access type, the address passed is that of a copy of the value of the actual parameter. The actual parameters are thereby protected from modification within the interfaced subprogram. For all other parameters, the address passed is the address of the actual parameter. Consequently, nonscalar and nonaccess parameters to interfaced subprograms cannot be protected from modification by the called subprogram, even when they are formally declared to be of mode in. It is your responsibility to ensure that the semantics of the Ada parameter modes are honored in these cases.

For pragma INTERFACE C, register 1 again serves to address the parameter list. These parameters are passed by value, however, in accordance with the VS C parameter-passing conventions. For all parameter types, the values of the actual parameters are passed in the parameter area.

Formal parameters of mode in are allowed for all types of actual parameters: scalar, access, record, and array. For a scalar or access parameter, the value of the actual parameter is stored in the parameter area. For a record parameter, a copy of the entire record is stored in the parameter area. For an array parameter, the address of the array is stored in the parameter area, since the value of an array in a C expression is simply the address of the first element of the first dimension of the array.

Because array parameters to C interfaced subprograms are passed by address, they cannot be protected from modification by the called subprogram, even when they are formally declared to be of mode in. It is your responsibility to ensure that the semantics of the Ada parameter mode are honored in this case.

A formal parameter of mode in out is not allowed for any scalar, access or record type but is allowed for an array type since an array parameter is passed by its address.

If the C subprogram is declared and called as a function, register 0 is used to return the result. Scalar and access values are returned in general register 0. Floating point values are returned in floating point register 0. Record values are returned by address in general register 0. Array values are not supported, since VS C does not allow a function to return an array result.

Pragma INTERFACE C lets you declare and call a C subprogram as a procedure. It is your responsibility to do so only where appropriate; specifically, when calling a C function that is declared in C to be of type VOID.

Parameter Representations

This section describes the representation of values of the types that can be passed as parameters to an interfaced subprogram. The discussion assumes that you have not used representation clauses to change the default representations of the types involved. The effect of representation clauses on the representation of values is described in Section F.4.

The following types can be passed as parameters to an interfaced subprogram:

Integer types [3.5.4] -- Ada integer types are represented in two's complement form and occupy 8 (SHORT_SHORT_INTEGER), 16 (SHORT_INTEGER), or 32 (INTEGER) bits.

Boolean types [3.5.3]-- Ada Boolean types are represented as 8-bit values: FALSE is represented by the value 0; TRUE, by the value 1.

Enumeration types [3.5.1]-- Ada enumeration types are represented internally as unsigned values corresponding to their positions in the list of enumeration literals that defines the type. The first literal in the list has the value 0.

An enumeration type can include a maximum of 2**31 values. Enumeration types with 256 elements or fewer are represented in 8 bits; those with 256 to 65536 (2**16) elements are represented 16 bits; all others are represented in 32 bits. Accordingly, th Ada predefined type CHARACTER [3.5.2] is represented in 8 bits, using the standard ASCII codes [C].

Floating point types [3.5.7, 3.5.8] -- Ada floating point types occupy 32 (SHORT_FLOAT) or 64 (FLOAT) bits and are held in VS (short or long) format.

Fixed point types [3.5.9, 3.5.10] -- Ada fixed point types are managed by the compiler as the product of a signed mantissa and a constant small. The mantissa is implemented as a 16- or 32-bit integer value. Small is a compile-time quantity that is the power of two equal or immediately inferior to the delta specified in the declaration of the type.

The attribute MANTISSA is defined as the smallest number such that

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

The size of a fixed point type is

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

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

Access types [3.8] -- Values of Ada access types are represented internally by the 31-bit address of the designated object held in a 32-bit word. You should not alter any bits of this word, even those that are ignored by the architecture on which the program is running. The value 0 is used to represent null.

Array types [3.6] -- The elements of an array are allocated by row. When an array is passed as a parameter to an interfaced subprogram, the usual consistency checking between the array bounds declared in the calling program and the subprogram is not enforced. Therefore, it is your responsibility to ensure that the subprogram does not violate the bounds of the array.

Values of the predefined type STRING [3.6.3] are arrays and are passed in the same way as the values of any other array type. Elements of a string are represented in 8 bits, using the standard ASCII codes.

Record types [3.7] -- Components of a record are aligned on their natural boundaries (e.g., INTEGER is aligned on a word boundary), and the compiler may reorder the components to minimize the total size of objects of the record type. If a record contains discriminants or components having a dynamic size, implicit components may be added to the record. Thus, the default layout of the internal structure of the record cannot be inferred directly from its Ada declaration. It is therefore recommended that you use a representation clause to control the layout of any record type whose values are to be passed to interfaced subprograms.

Restrictions on Interfaced Subprograms

Refer to Chapter 5 for a description of the restrictions on interfaced subprograms.

F.1.3 Pragma INTERFACE_NAME

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

Pragma INTERFACE_NAME takes the form

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

where

subprogram_name is the name used within the Ada program to refer to the interfaced subprogram;

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

The use of INTERFACE_NAME is optional; you need not use it if a subprogram has the same name in Ada as it has in its language of origin. INTERFACE_NAME is useful if, for example, the name of the subprogram in its original language contains characters that are not permitted in Ada identifiers. Although Ada identifiers can contain letters, digits, and underscores, the VS Linker limits external names to the letters A-Z; the digits 0-9; and the symbols $, @, and #. Thus, only these characters are permitted in the string_literal argument of the pragma INTERFACE_NAME.

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

In order to conform to the naming conventions of the VS Linker, the link-time name of an interfaced subprogram is truncated to 32 characters and converted to upper case.

Example

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

F.1.4 Other Pragmas

Pragmas IMPROVE and PACK -- These pragmas are discussed in detail in section F.4.

Pragma PRIORITY -- Pragma PRIORITY is accepted with the range of priorities running from 1 to 1 (see the definition of the predefined package SYSTEM, Section F.3). The undefined priority (no pragma PRIORITY) is treated as less than any defined priority value (see Appendix B).

Pragma SUPPRESS -- You can substitute the compiler option CHECKS for the pragma SUPPRESS to suppress all checks in a compilation.

The following pragmas have no effect:

- CONTROLLED
- MEMORY_SIZE
- OPTIMIZE
- STORAGE_UNIT
- SYSTEM_NAME

Note that all access types are implemented by default as controlled collections, as described in [4.8] (see Section F.10.1).

F.2 IMPLEMENTATION-DEPENDENT ATTRIBUTES

In addition to the representation attributes described in the LRM [13.7.2 and 13.7.3], VS Ada provides the following attributes:

T'VARIANT_INDEX, C'ARRAY_DESCRIPTOR, T'RECORD_SIZE, C'RECORD_DESCRIPTOR -- These attributes, which are used in record representation clauses, are described in Section F.5.

T'DESCRIPTOR_SIZE , T'IS_ARRAY -- These attributes are described in Section F.2.1.

VS Ada imposes certain limitations on the use of the attribute ADDRESS. These limitations are described in Section F.2.2.

F.2.1 The Attributes T'DESCRIPTOR_SIZE and T'IS_ARRAY

The attributes T'DESCRIPTOR_SIZE and T'IS_ARRAY are described as follows:

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

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

F.2.2 Limitations on the Use of the Attribute ADDRESS

The attribute ADDRESS is implemented for all prefixes that have meaningful addresses. The following entities do not have meaningful addresses; each causes a compilation warning if used as a prefix to ADDRESS:

- A constant that is implemented as an immediate value; that is, one for which no space is allocated
- A package specification that is not a library unit
- A package body that is not a library unit or subunit

F.3 SPECIFICATION OF THE PACKAGE SYSTEM

```
package SYSTEM is

  type NAME is (WANG_VS);

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

  type ADDRESS is range MIN_INT .. MAX_INT;

  STORATE_UNIT : constant := 8;
  MAX_DIGITS   : constant := 15;
  MAX_MANTISSA : constant := 31;
  FINE_DELTA   : constant := 2#1.0#e-31;
  TICK         : constant := 0.01;
  NULL_ADDRESS : constant ADDRESS := 0;

  subtype PRIORITY is INTEGER range 1 .. 1;

  PROGRAM_CANCEL : exception;

end SYSTEM;
```

F.4 RESTRICTIONS ON REPRESENTATION CLAUSES

This section explains how objects are represented and allocated by the VS Ada compiler and how you can use representation clauses to control the representation and allocation of objects.

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

With the exception of array and record types, the description of each type is independent of the others. To understand the representation of an array type or of a record type, you must first understand the representation of its components.

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

- A size specification can be used to control the size of any object.
- The predefined pragma PACK can be used to control the size of an array, an array component, a record, or a record component.
- A record representation clause can be used to control the size of a record or a record component.

The sections that follow describe the effect of size specifications on each class of type; Sections F.4.7 and F.4.8 describe the interaction among size specifications, packing, and record representation clauses on array and record types.

Representation clauses on derived record types or derived task types are not supported, and size representation clauses are not supported on types derived from private types when they are declared outside the private part of the defining package.

F.4.1 Enumeration Types

Internal codes of enumeration literals

As described in the LRM [13.3], you can use an enumeration representation clause to specify the value of the internal code associated with an enumeration literal. Enumeration representation clauses are fully implemented.

Since internal codes must be machine integers, the internal codes specified in an enumeration representation clause must be in the range $-2^{31} .. 2^{31}-1$.

If you do not specify an enumeration representation clause, the internal code associated with an enumeration literal is the position number of the enumeration literal. Thus, for an enumeration type with n elements, the internal codes are the integers 0, 1, 2, .. n-1.

Encoding of enumeration values

In the program generated by the compiler, an enumeration value is always represented by its internal code.

Minimum size of an enumeration subtype

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

If a static subtype has a null range, its minimum size is 1. Otherwise, if m and M are the values of the internal codes associated with the first and last enumeration values of the subtype, then its minimum size L is determined as follows:

For m >= 0, L is the smallest positive integer such that $M <= 2^{L}-1$.
For m < 0, L is the smallest positive integer such that $-2^{L-1} <= m$ and $M <= 2^{L-1}-1$.

For example:

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

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

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

Size of an enumeration subtype

You can specify the size of an enumeration type and each of its subtypes in the length clause of a size specification. The length clause also determines the size of a first-named subcype. You must of course specify a value greater than or equal to the minimum size of the type or subtype.

For example:

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

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

The compiler fully implements size specifications. As enumeration values are represented as integers, however, the length you specify cannot be greater than 32 bits, the size of the largest predefined integer on the VS.

If you do not use a size specification for an enumeration type or first-named subtype, the objects of that type or subtype are represented as signed integers if the internal code associated with the first enumeration value is negative and as unsigned integers otherwise. VS Ada provides 8-, 16- and 32-bit integers; the compiler automatically selects the smallest machine integer that can hold each of the internal codes of the enumeration type or subtype. Thus, the size of the enumeration type and any of its subtypes is 8, 16 or 32 bits.

Size of the objects of an enumeration subtype

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

Alignment of an enumeration subtype

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

Address of an object of an enumeration subtype

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

F.4.2 Integer Types

Predefined integer types

The VS provides three predefined integer types:

```
type SHORT_SHORT_INTEGER is range -2**07 .. 2**07-1;
type SHORT_INTEGER       is range -2**15 .. 2**15-1;
type INTEGER             is range -2**31 .. 2**31-1;
```

Selection of the parent of an integer type

An integer type declared as

```
type T is range L .. R;
```

is implicitly derived from either the SHORT_INTEGER or INTEGER predefined integer type. The compiler automatically selects the predefined integer type with the shortest range containing the values L to R inclusive. Note that the the compiler never automatically selects the SHORT_SHORT_INTEGER representation.

Encoding of integer values

In the program generated by the compiler, integer values are represented in binary code in a conventional two's complement form.

Minimum size of an integer subtype

The minimum size of an integer subtype is the minimum number of bits necessary to represent the internal codes of the subtype values in normal binary form (that is, in an unbiased form that includes a sign bit only if the range of the subtype includes negative values).

If a static subtype has a null range, its minimum size is 1. Otherwise, if m and M are the lower and upper bounds of the subtype, then its minimum size L is determined as follows:

For m >= 0, L is the smallest positive integer such that M <= $2^{L}-1$; for m < 0, L is the smallest positive integer such that -2^{L-1} <= m and M <= $2^{L-1}-1$.

For example:

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

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

Size of an integer subtype

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

You can specify the size of an integer type and each of its subtypes in the length clause of a size specification. The length clause also determines the size of a first-named subtype. You must of course specify a value greater than or equal to the minimum size of the type or subtype.

For example:

```
type S is range 80 .. 100;
for S'SIZE use 32;
-- S is derived from SHORT_INTEGER, but its size is 32 bits
-- because of the size specification.

type J is range 0 .. 255;
for J'SIZE use 8;
-- J is derived from SHORT_INTEGER, but its size is 8 bits because
-- of the size specification.

type N is new J range 80 .. 100;
-- N is indirectly derived from SHORT_INTEGER, but its size is 8 bits
-- because N inherits the size specification of J.
```

The compiler implements size specifications. As integers are implemented using machine integers, however, the length specified cannot be greater than 32 bits.

If you do not use a size specification for an integer type or its first-named subtype (if any), the size of the integer and any of its subtypes is the size of the predefined type from which it derives, directly or indirectly.

For example:

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

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

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

Size of the objects of an integer subtype

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

Alignment of an integer subtype

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

Address of an object of an integer subtype

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

F.4.3 Floating Point Types

Predefined floating point types

VS Ada provides two predefined floating point types:

```
type SHORT_FLOAT is
     digits 6 range -2.0**252 *(1.0-2.0**-24) .. 2.0**252* (1.0-2.0**-24);

type FLOAT is
     digits 15 range -2.0**252 *(1.0-2.0**-56) ..2.0**252* (1.0-2.0**-56);
```

Selection of the parent of a floating point type

A floating point type declared as

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

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

Encoding of floating point values

In the program generated by the compiler, floating point values are represented using VS data formats for single-precision or double-precision floating point values, as appropriate.

Values of the predefined type SHORT_FLOAT are represented using the single precision format; values of the predefined type FLOAT are represented using the double precision format. The values of any other floating point type are represented in the same format as the values of the predefined type from which it derives, directly or indirectly.

Minimum size of a floating point subtype

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

Size of a floating point subtype

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

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

The only size you can specify for a floating point type or first-named subtype in a size specification is its usual size (32 or 64 bits).

Size of the objects of a floating point subtype

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

Alignment of a floating point subtype

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

Address of an object of a floating point subtype

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

F.4.4 Fixed Point Types

Small of a fixed point type

You can specify the value of small in the length clause of a size specification. The value you specify must be a power of two.

If you do not use a size specification to specify small of a fixed point type, then the value of small is determined by the value of delta, as defined by the LRM [3.5.9].

Predefined fixed point types

VS Ada provides a set of anonymous predefined fixed point types of the form

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

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

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

Selection of the parent of a fixed point type

A fixed point type declared as

```
type T is delta D range L .. R;
```

optionally, with a small specification

```
for T'SMALL use S;
```

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

Encoding of fixed point values

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

```
V / F'BASE'SMALL
```

Minimum size of a fixed point subtype

The minimum size of a fixed point subtype is the minimum number of binary digits necessary to represent the values of the range of the subtype using the small of the base type (that is, in an unbiased form that includes a sign bit only if the range of the subtype includes negative values).

If a static subtype has a null range, its minimum size is 1. Otherwise, if s and S are the bounds of the subtype, and if i and I are the integer representations of m and M (the smallest and greatest model numbers of the base type such that s < m and M < S), then the minimum size L is determined as follows:

For i >= 0, L is the smallest positive integer such that $I <= 2^{L}-1$; for i < 0, L is the smallest positive integer such that $-2^{L-1} <= i$ and $I <= 2^{L-1}-1$.

For example:

```
type F is delta 2.0 range 0.0 .. 500.0;
-- The minimum size of F is 8 bits.

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

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

Size of a fixed point subtype

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

You can specify the size of a fixed point type and each of its subtypes in the length clause of a size specification. The length clause also determines the size of a first-named subtype. You must of course specify a value greater than or equal to the minimum size of the type or subtype.

For example:

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

type L is delta 0.01 range 0.0 .. 300.0;
for F'SIZE use 16;
-- F is derived from a 32 bit predefined fixed type, but its size is
-- 16 bits because of the size specification.  The size
-- specification is legal since the range contains no negative values
-- and therefore no sign bit is required.

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

The VS Ada compiler implements size specifications. As fixed point objects are represented using machine integers, however, the length specified cannot be greater than 32 bits.

If you do not use a size specification for a fixed point type or its first-named subtype, the size of the fixed point type and any of its subtypes is the size of the predefined type from which it derives, directly or indirectly.

For example:

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

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

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

Size of the objects of a fixed point subtype

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

Alignment of a fixed point subtype

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

Address of an object of a fixed point subtype

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

F.4.5 Access Types

Collection Size

As described in the LRM [13.2], you can use the length clause of a size specification to indicate the amount of storage space to be reserved for the collection of an access type. The compiler fully implements this kind of specification.

If you do not specify collection size for an access type, no storage space is reserved for its collection; the value of the attribute STORAGE_SIZE is then 0.

Minimum size of an access subtype

The minimum size of an access subtype is 32 bits.

Size of an access subtype

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

The only size you can specify for an access type in a size specification is its usual size (32 bits).

Size of the objects of an access subtype

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

Alignment of an access subtype

An access subtype is always word-aligned.

Address of an object of an access subtype

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

F.4.6 Task Types

Storage for a task activation

As described in the LRM [13.2], you can use the length clause of a size specification to indicate the amount of storage space to be reserved for the activation of every task of a given type.

If you do not specify storage space in a length clause, the amount of storage space you indicate at bind time is allocated.

You cannot use a length clause with a derived type. VS Ada reserves the same amount of storage space for the activation of a task of a derived type as for the activation of a task of the parent type.

Encoding of task values

Task values are machine addresses.

Minimum size of a task subtype

The minimum size of a task subtype is 32 bits.

Size of a task subtype

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

The only size you can specify for a task type in a size specification is its usual size (32 bits).

Size of the objects of a task subtype

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

Alignment of a task subtype

A task subtype is always word-aligned.

Address of an object of a task subtype

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

F.4.7 Array Types

Layout of an array

Every array is allocated in a contiguous area of storage units. All the components of an array are the same size. A gap may exist between two consecutive components and after the last component. All gaps are the same size.

Components

If the array is not packed, its components are the same size as the subtype of the components.

For example:

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

type DECIMAL_DIGIT is range 0 .. 9;
for DECIMAL_DIGIT'SIZE use 4;
type BINARY_CODED_DECIMAL is
  array (INTEGER range <>) of DECIMAL_DIGIT;
-- The size of the type DECIMAL_DIGIT is 4 bits.  Thus, in an array
-- of type BINARY_CODED_DECIMAL, each component will be represented
-- in 4 bits as in the usual BCD representation.
```

If the array is packed and its components are neither records nor arrays, the size of the components is the minimum size of the subtype of the components.

For example:

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

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

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

Gaps

Provided an array is not packed, its components are records or arrays, and no size specification applies to the subtype of the components, the compiler may choose a representation with a gap after each component. By inserting such gaps, the compiler optimizes access to the array components and their subcomponents. The size of the gaps is such that the relative displacement of consecutive components is a multiple of the alignment of the subtype of the components. This strategy gives each component and subcomponent an address consistent with the alignment of its subtype

For example:

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

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

If the array is packed, or if a size specification does apply to the subtype of the components, no gaps are inserted.

For example:

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

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

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

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

Size of an array subtype

The size of an array subtype is obtained by multiplying the number of its components by the sum of the size of the components and the size of any gaps. If the subtype is unconstrained, the maximum number of components is used to determine the size.

The size of an array subtype cannot be computed at compile time in the following cases:

- If the array subtype has nonstatic constraints or if it is an unconstrained array type with nonstatic index subtypes (because the number of components can then only be determined at run time).

- If the components are records or arrays and their constraints or the constraints of their subcomponents (if any) are not static (because the size of the components and the size of the gaps can then only be determined at run time).

As indicated previously, the effect of a pragma PACK on an array type is to suppress the gaps and reduce the size of the components. Thus, packing an array type reduces its size.

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

The only size you can specify for an array type or first-named subtype in the length clause of a size specification is its usual size. Nevertheless, a length clause can be useful to verify that the layout of an array is the layout expected by the application.

Size of the objects of an array subtype

An object of an array subtype is always the same size as the subtype of the object.

Alignment of an array subtype

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

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

Address of an object of an array subtype

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

F.4.8 Record Types

Layout of a record

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

As described in the LRM [13.4], you can use a record representation clause to control the positions and sizes of a record's components. VS Ada imposes no restrictions on a component's position. Bits within a storage unit are numbered from 0 to 7, with the most-significant bit numbered 0. The range of bits specified in a component clause may extend into following storage units. If a component is not a record or an array, its size can be any size from the minimum size to the size of its subtype. If a component is a record or an array, its size must be the size of its subtype:

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

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

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

type INTERRUPT is (IO, CLOCK, MACHINE_CHECK);
type INTERRUPT_MASK_TYPE is array (INTERRUPT) of BOOLEAN;

type INTERRUPT_CODE is range 0 .. 7;

type PROC_LEVEL is range 0 .. 7;
```

```
type PROGRAM_CONTROL_WORD is
  record
      INTERRUPT_CAUSE          : INTERRUPT_CODE;
      CURRENT_INST_ADDRESS     : ADDRESS;
      WAIT_STATE               : BOOLEAN;
      CONTROL_MODE             : BOOLEAN;
      PROTECTION_TRAP          : BOOLEAN:
      VIRTUAL_MACHINE          : BOOLEAN;
      INTERRUPT_MASK           : INTERRUPT_MASK_TYPE:
      DEBUG_CONTROL            : BOOLEAN;
      CONDITION_CODE           : CONDITIONS;
      PROGRAM_MASK             : PROG_MASK;
      PROCESS_LEVEL            : PROC_LEVEL;
  end record;

-- This type can be used to map the program control word of the VS.

for PROGRAM_CONTROL_WORD use
  record at mod 4;
      INTERRUPT_CAUSE          at 0    range 0 .. 7;
      CURRENT_INST_ADDRESS     at 1    range 8 .. 31;
      WAIT_STATE               at 4    range 0 .. 0;
      CONTROL_MODE             at 4    range 1 .. 1;
      PROTECTION_TRAP          at 4    range 2 .. 2;
      VIRTUAL_MACHINE          at 4    range 3 .. 3;
      INTERRUPT_MASK           at 4    range 5 .. 7;-- bit 4 unused
      DEBUG_CONTROL            at 5    range 0 .. 0;-- bits 1..7 unused
      CONDITION_CODE           at 6    range 0 .. 1;
      PROGRAM_MASK             at 6    range 2 .. 5;-- bits 6,7 unused
      PROCESS_LEVEL            at 7    range 5 .. 7;-- bits 0..4 unused
  end record;
```

You need not specify the size and position of every component of a record in a record representation clause. If you do not specify size, the component is the same size as its subtype. If you do not specify position, the compiler chooses the position that optimizes access to the components of the record: the offset of the component is a multiple of the alignment of the component subtype. The compiler also chooses a position that reduces the number of gaps and thus the size of the record objects.

Because of these optimizations, there is no connection between the order of components in a record type declaration and the positions of the components in a record object.

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

Indirect components

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

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

For example:

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

type COLOR is (GREEN, RED, BLUE);

type SERIES is array (POSITIVE range <>) of INTEGER;

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

Q : POSITIVE;

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

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

Thus, the only indirect components are dynamic components. But not all dynamic components are necessarily indirect: If a component list that is not followed by a variant part contains dynamic components, exactly one dynamic component of this list is a direct component because its offset can be computed at compilation time.

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

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

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

Implicit components

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

An implicit component may contain information that is used when the record object or several of its components are accessed. In this case, the component is included in any record object; that is, the implicit component is considered to be declared before any variant part in the record type declaration. There are two components of this kind: RECORD_SIZE and VARIANT_INDEX.

On the other hand, an implicit component may be used to access a given record component. In this case the implicit component exists whenever the record component exists; that is, the implicit component is considered to be declared at the same place as the record component. There are two components of this kind: ARRAY_DESCRIPTORS and RECORD_DESCRIPTORS.

The implicit components RECORD_SIZE, VARIANT_INDEX, ARRAY_DESCRIPTORS, and RECORD_DESCRIPTORS are described as follows:

RECORD_SIZE - The compiler creates the implicit component RECORD_SIZE when the record type has a variant part and its discriminants are defaulted. RECORD_SIZE contains the size of the storage space needed to store the current value of the record object. (Note that the storage actually allocated for the record object may be larger.)

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

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

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

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

Component lists that do not contain a variant part are numbered. These numbers are the possible values of the implicit component VARIANT_INDEX.

For example:

```
type VEHICLE is (AIRCRAFT, ROCKET, BOAT, CAR);

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

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

Variant Index	Set
1	{KIND, SPEED, WHEELS, WINGSPAN}
2	{KIND, SPEED, WHEELS}
3	{KIND, SPEED, STEAM}
4	{KIND, SPEED, STAGES}

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

Component	Interval
KIND	--
SPEED	--
WHEELS	1 .. 2
WINGSPAN	1 .. 1
STEAM	3 .. 3
STAGES	4 .. 4

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

If R is the name of the record type, VARIANT_INDEX can be specified in a component clause by the implementation-generated name R'VARIANT_INDEX.

ARRAY_DESCRIPTOR -- The compiler associates the implicit component ARRAY_DESCRIPTOR with each record component whose subtype is an anonymous array subtype that depends on a discriminant of the record. ARRAY_DESCRIPTOR contains information about the component subtype.

The structure of the implicit component ARRAY_DESCRIPTOR is not described in this documentation. However, if you wish to specify the location of an ARRAY_DESCRIPTOR component in a component clause, you can obtain the size of the component by specifying DMAP = YES when you run the compiler.

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

RECORD_DESCRIPTOR -- The compiler associates the implicit component RECORD_DESCRIPTOR with each record component whose subtype is an anonymous record subtype that depends on a discriminant of the record. RECORD_DESCRIPTOR contains information about the component subtype.

The structure of the implicit component RECORD_DESCRIPTOR is not described in this documentation. However, if you wish to specify the location of a RECORD_DESCRIPTOR component in a component clause, you can obtain the size of the component by specifying DMAP = YES when you run the compiler.

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

Suppressing implicit components

The VS Ada-defined pragma IMPROVE enables you to suppress the implicit components RECORD_SIZE and/or VARIANT_INDEX from a record type. The syntax of pragma IMPROVE is as follows:

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

The first argument specifies whether TIME or SPACE is the primary criterion for the choice of the representation of the record type, which is denoted by the second argument.

If you specify TIME, the compiler inserts implicit components as described above. If you specify SPACE, the compiler inserts a VARIANT_INDEX or a RECORD_SIZE component only if that component appears in a record representation clause that applies to the record type. Thus, you can use a record representation clause to keep one implicit component while suppressing the other.

A pragma IMPROVE that applies to a given record type can occur anywhere a representation clause is allowed for that type.

Size of a record subtype

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

The size of a constrained record subtype is obtained by adding the sizes of its components and the sizes of any gaps. The size is not computed at compile time if the record subtype has nonstatic constraints or if a component is an array or a record and its size is not computed at compile time.

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

A size specification applied to a record type or first-named subtype has no effect: The only size you can specify is the default size of the record type or first-named subtype. Nevertheless, a length clause can be useful to verify that the layout of a record is the layout expected by the application.

Size of the objects of a record subtype

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

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

Alignment of a record subtype

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

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

When a record representation clause that contains an alignment clause applies to its base type, a record subtype has the alignment specified by the alignment clause.

Address of an object of a record subtype

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

F.5 Conventions for Implementation-Generated Names

The compiler introduces special record components for certain record type definitions. Such record components are implementation-dependent; they are used by the compiler to improve the quality of the generated code for certain operations on the record types.

The compiler issues an error message if you refer to an implementation-dependent component that does not exist. If the implementation-dependent component does exist, the compiler checks that the storage location specified in the component clause is compatible with the treatment of the component and the storage locations of other components, issuing an error message if this check fails.

Four attributes are defined to refer to these implementation-dependent components in record representation clauses:

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

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

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

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

F.6 ADDRESS CLAUSES

F.6.1 Address Clauses for Objects

As described in the LRM [13.5], you can use an address clause to specify an address for an object. When you do use an address clause, no storage is allocated for the object in the program generated by the compiler. Instead, the program uses the address specified in the clause to access the object.

You cannot use an address clause for task objects or for unconstrained records whose maximum possible size is greater than 8 Kbytes.

F.6.2 Address Clauses for Program Units

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

F.6.3 Address Clauses for Entries

Address clauses for entries are not implemented in the current version of VS Ada.

F.7 Restrictions on Unchecked Conversions

Unconstrained arrays are not allowed as target types, nor are unconstrained record types without defaulted discriminants allowed as target types.

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

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

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

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

F.8 IMPLEMENTATION-DEPENDENT CHARACTERISTICS OF THE INPUT-OUTPUT PACKAGES

The package LOW_LEVEL_IO [14.6], which is concerned with low-level machine-dependent input-output, is not implemented in VS Ada. The predefined input-output packages SEQUENTIAL_IO [14.2.3], DIRECT_IO [14.2.5], TEXT_IO [14.3.10] and IO_EXCEPTIONS [14.5] are implemented as described in the Language Reference Manual.

This section describes those characteristics of the input-out packages that are specific to Wang VS Ada.

F.8.1 Unbounded Line Lengths

VS Ada does not support unbounded line lengths.

F.8.2 The FORM Parameter

The FORM parameter is passed to the Ada CREATE and OPEN procedures to specify VS file attributes.

FORM is an optional parameter. If you omit it or pass it as a null string, the file assumes VS default attributes. If you do include the FORM parameter when creating or opening a file, you need not specify a value for every attribute. In some cases, the attribute may not apply (e.g., a disk file cannot take a tape label type). In other cases, you may choose to accept the default value for the attribute.

The syntax of the FORM string is:

```
form_parameter      ::=  [ attribute [ {, attribute} ] ]

attribute           ::=  key_word [ => value ]
```

Incorrect syntax causes a USE_ERROR exception to be raised.

The following section contains a description of each attribute supported by the FORM parameter.

Attributes of the FORM Parameter

This section lists and describes the VS file attributes supported by the FORM parameter. The attributes are listed alphabetically, and each entry includes the following information:

- Description - A brief description of what the attribute does
- Values - A list of the possible values for the attribute, followed by explanations where necessary
- Default - The default value for the attribute
- Restrictions - Where applicable, a description of any restrictions that apply to the attribute

The *VS Data Management System Reference* contains detailed information about these attributes.

APPEND

Description: The APPEND attribute specifies whether output is appended to the file or overwrites the existing file.

Values: APPEND => [YES | NO]

If APPEND => YES, the record pointer is positioned at end-of-file and output is appended to the file.

If APPEND => NO, the record pointer is positioned at the beginning of the file and output overwrites the existing file.

Default: The default is NO.

Restrictions: Failure to meet the following condition causes a USE_ERROR to be raised:

APPEND => YES can be used only on existing files opened for OUT_FILE or INOUT_FILE mode.

BLOCK_SIZE

Description: The BLOCK_SIZE attribute specifies the size in bytes of the DMS block for the file.

Values: BLOCK_SIZE => [2048 ... 32768]

Default: The default (and only legal value) for DISK, PRINTER, and WS devices is 2048.

The default for TAPE devices is 2048.

BUF_SIZE

Description: The BUF_SIZE attribute specifies the size in bytes of the DMS buffer for the file.

Values: BUF_SIZE => [2048 | 4096 | 6144 | 8192 | 10240 | 12288 | 14336 | 16384 | 18432]

Default: The default is 2048.

Restrictions: Failure to meet the following condition causes a USE_ERROR to be raised:

BUF_SIZE must be a multiple of 2048 bytes (2K), in the range 2048 (2K) to 18432 (18K).

COMPRESS

Description: The COMPRESS attribute specifies whether or not DMS automatically compresses records.

Values: COMPRESS => [YES | NO]

If COMPRESS => YES, DMS compresses records.

If COMPRESS => NO, DMS does not compress records.

Default: The default is NO.

CONFIRM

Description: The CONFIRM attribute specifies whether or not DMS automatically scratches a file when you call the CREATE procedure with the file name (NAME) of an existing external file.

Values: CONFIRM => [YES | NO]

If CONFIRM => YES, DMS issues a GETPARM asking you to confirm the deletion of the existing file.

If CONFIRM => NO, DMS automatically deletes the file.

Default: The default is NO.

DENSITY

Description: The DENSITY attribute applies to tape files only (DEVICE => TAPE). It specifies the density of a tape in bits per inch (BPI).

Values: DENSITY => [526 | 800 | 1600 | 6250]

Default: The default is 1600.

DEVICE

Description: The DEVICE attribute specifies the device that is to be associated with the file. The device can be a disk, a magnetic tape, a printer, or a workstation.

Values: DEVICE => [DISK | TAPE | PRINTER | WS]

Default: The default is DISK.

Restrictions: Failure to meet any of the following conditions causes a USE_ERROR to be raised:

If DEVICE => WS, then the file organization must be consecutive (ORGANIZATION => CONSECUTIVE).

If DEVICE => PRINTER, then the file organization must be print (ORGANIZATION => PRINT) and the open access mode must be OUT_FILE.

If DEVICE => TAPE, the file cannot be opened for DIRECT_IO, as DMS allows neither random access to or updating of tape records.

If DEVICE => TAPE, an existing file cannot be opened in OUT_FILE mode, as DMS does not allow updating of tape records.

If DEVICE => DISK, the block size must be 2048 (BLOCK_SIZE => 2048).

DISMOUNT

Description: The DISMOUNT attribute applies to tape files only (DEVICE => TAPE). It specifies whether or not DMS logically dismounts a tape volume when it closes the tape file.

Values: DISMOUNT => [YES | NO]

If DISMOUNT => YES, DMS dismounts the tape.

If DISMOUNT => NO, DMS does not dismount the tape.

Default: The default is NO.

DISPLAY

Description: The DISPLAY attribute specifies whether or not a GETPARM screen is displayed at the workstation at runtime. You can respecify file attributes on the GETPARM screen before opening the file.

Values: DISPLAY => [YES | NO]

If DISPLAY => YES, a GETPARM screen is displayed.

If DISPLAY => NO, a GETPARM screen is not displayed.

Default: The default is NO.

EOF_STRING =>

Description: The EOF_STRING attribute applies to workstation files only (DEVICE = WS). It specifies an end-of-file (eof) string. If a line equal to the eof string is typed in, subsequent calls to the END_OF_FILE function return TRUE; subsequent attempts to read from the workstation raise the END ERROR exception.

Values: EOF_STRING => [/* | sequence_of_characters]

Note: *sequence_of_characters cannot contain commas or spaces.*

Default: The default is /*.

FILE_CLASS

Description: The FILE_CLASS attribute applies to disk files only (DEVICE => DISK). It specifies the file's VS file protection class.

Values: FILE_CLASS => [A ... Z, #, $, @, (blank)]

For information on VS file protection classes, see the *VS System User's Introduction*.

Default: The default is derived from your usage constants.

For information on how to set usage constants, see the *VS System User's Introduction*.

FILE_SEQ

Description: The FILE_SEQ attribute applies to tape files only (DEVICE => TAPE). It specifies the file sequence number of a tape file.

Values: FILE_SEQ => [1 ... 9999]

Default: The default is 1.

FORCE_EOR

Description: The FORCE_EOR attribute applies to tape files only (DEVICE => TAPE). It specifies whether or not DMS forces an end-of-reel when it closes a tape file that spans multiple volumes.

Values: FORCE_EOR => [YES | NO]

If FORCE_EOR => YES, DMS forces end-of-reel.

If FORCE_EOR => NO, DMS does not force end-of-reel.

Default: The default is NO.

LABEL

Description: The LABEL attribute applies to tape files only (DEVICE => TAPE). It specifies the label type for a tape.

Values: LABEL => [NONE | ANY | ANSI | IBM]

If LABEL => NONE, the tape contains no labels, or it contains labels that correspond to neither the ANSI nor the IBM standards. DMS treats such as labels as if they were the first data block(s) of the file.

If LABEL => ANY, the existing label on the tape is used.

If LABEL => ANSI, the tape contains ANSI-standard labels, which are written in ASCII.

If LABEL => IBM, the tape contains IBM-standard labels, which are written in EBCDIC.

Default: The default is ANSI.

NOT_SHARED
SHARED

Description: The NOT_SHARED/SHARED attribute specifies whether or not several internal files within a single program can share one external file.

Values:
```
NOT_SHARED
SHARED => [READERS | SINGLE_WRITER | ANY]
```

When NOT_SHARED, the external file cannot be shared.

When SHARED => READERS, several internal files can read, but not update, the same external file.

When SHARED => SINGLE_WRITER, one internal file can update an external file while several other internal files read the same file.

When SHARED => ANY, several internal files can both read and update one external file simultaneously.

Default: The sharing mode is taken from the "brother" files if there are any. In the absence of brother files, if DEVICE => WS, then SHARED => ANY; if MODE => IN_FILE, then SHARED => READERS; otherwise, the sharing mode is NOT_SHARED.

N_RECS

Description: The N_RECS attribute applies to disk files only (DEVICE => DISK). It specifies your estimate of the number of records the file will contain. DMS uses this estimate to calculate the number of disk blocks to allocate for the primary extent.

Values:
```
N_RECS => [1 ... 16777215]
```

Default: The default is 500.

ORGANIZATION

Description: The ORGANIZATION attribute specifies the file organization type. For further information about file organization, see the *VS Data Management System Reference*.

Values: ORGANIZATION => [CONSECUTIVE | PROGRAM | PRINT]

If ORGANIZATION => CONSECUTIVE, the file is a data file consisting of consecutively written records - that is, records stored in the order in which they are created.

If ORGANIZATION => PROGRAM, the file is a consecutive file of 1024-byte records in VS program format.

If ORGANIZATION => PRINT, the file is a consecutive file containing program output to be sent to a printer.

Default: The default is CONSECUTIVE when DEVICE => DISK, TAPE, and WS. When DEVICE => PRINTER, ORGANIZATION => PRINT.

Restrictions: Failure to meet any of the following conditions causes a USE_ERROR to be raised:

If ORGANIZATION => PROGRAM, then the record size must be 1024 bytes (REC_SIZE => 1024); the record format must be fixed (REC_FORMAT => F); and the file must be opened for either SEQUENTIAL_IO or DIRECT_IO.

If ORGANIZATION => PRINT, then the record format must be variable (REC_FORMAT => V), the records must be compressed (COMPRESS => YES), and the file must be opened for TEXT_IO.

If ORGANIZATION => PRINT, then output must overwrite the existing file (APPEND => NO); since DMS does not allow print files to be opened in I/O mode, output cannot be appended to a print file.

PAD_CHAR

Description: The PAD_CHAR attribute specifies how records are padded.

Values: PAD_CHAR => [NUL | BLANK | any_displayable_ASCII_character]

If PAD_CHAR => NUL, records are padded with nulls (hex 00)

If PAD_CHAR => BLANK, records are padded with blanks (hex 20).

If PAD_CHAR => any_displayable_ASCII_character, records are padded with that character.

Default: The default for sequential and direct files is NUL.

The default for text files is BLANK.

PARITY

Description: The PARITY attribute applies to tape files only (DEVICE => TAPE). It specifies parity for a 7-track tape. (You need not specify parity for 9-track tapes.)

Values: PARITY => [ODD | EVEN]

Default: The default is ODD.

PRINT_CLASS

Description: The PRINT_CLASS attribute applies to print files only (DEVICE = PRINTER). It specifies the file's print class.

Values: PRINT_CLASS => [A ... Z]

For information about print classes, see the *VS System User's Introduction*.

Default: The default is extracted from your usage constants.

For information on how to set usage constants, see the *VS System User's Introduction*.

PRINT_FORM

Description: The PRINT_FORM attribute applies to print files only (DEVICE = PRINTER). It specifies the type of paper to be mounted in the printer. Printing is then inhibited if the wrong paper is mounted.

For further information about form numbers, see the *VS System Operator's Guide*.

Values: PRINT_FORM => [0 ... 255]

Default: The default is extracted from your usage constants.

For information on how to set usage constants, see the *VS System User's Introduction*.

PR_NAME

Description: The PR_NAME attribute specifies a parameter reference name (prname) for the file. The prname is used by DMS for GETPARM and PUTPARM processing.

Values: PR_NAME => [1_to_6_character_string]

Default: The default is a blank string.

REC_FORMAT

Description: The REC_FORMAT attribute specifies whether the file contains fixed-length or variable-length records.

Values: REC_FORMAT => [F | V]

IF REC_FORMAT => F, the file contains fixed-length records.

If REC_FORMAT => V, the file contains variable-length records.

Default: The default for sequential and direct files is F.

The default for text files is V.

REC_SIZE

Description: The REC_SIZE attribute specifies the size in bytes of the file's records. For variable length records, REC_SIZE specifies the largest size record that can be written to the file.

Values: REC_SIZE => [1 ... 2048]

Default: The default for constrained element types is the size of the instantiated element type.

The defaults for unconstrained element types are

2048 for fixed-length records (REC_FORMAT => F);
2024 for variable-length records (REC_FORMAT => V).

The default (and only legal value) for program files (ORGANIZATION => PROGRAM) is 1024.

Restrictions: Failure to meet any of the following conditions causes a USE_ERROR to be raised:

If the record format is fixed (REC_FORMAT => F), then REC_SIZE cannot exceed 2048 bytes.

If the record format is variable (REC_FORMAT => V), then REC_SIZE cannot exceed 2024 bytes.

The record size of files with constrained types that are opened for SEQUENTIAL_IO or DIRECT_IO must be equal to
(ELEMENT_TYPE'SIZE - 1) / SYSTEM.STORAGE_UNIT + 1.
(This is the minimum number of bytes needed to represent an object of ELEMENT_TYPE.)

RELEASE

Description: The RELEASE attribute applies to disk files only (DEVICE => DISK). It specifies whether or not DMS returns unused primary extent blocks to the operating system when the file is closed.

Values: RELEASE => [YES | NO]

If RELEASE => YES, DMS returns unused blocks.

If RELEASE => NO, DMS does not return unused blocks.

Default: The default is NO.

RETENTION

Description: The RETENTION attribute applies to disk files only (DEVICE => DISK). It specifies the number of days beyond the date of its creation that a file is retained.

Values: RETENTION => [0 ... 999]

Default: The default is 0 (which specifies that the files's expiration date is the same as its creation date).

REWIND

Description: The REWIND attribute applies to tape files only (DEVICE => TAPE). It specifies whether or not DMS rewinds the tape volume when it close[illegible] e tape file.

Values: REWIND => [YES | NO]

If REWIND => YES, DMS rewinds the tape.

If REWIND => NO, DMS does not rewind the tape.

Default: The default is NO.

TRACKS

Description: [illegible]e TRACKS attribute applies to tape files only [illegible]'ICE => TAPE). It specifies the num[illegible] of [illegible]s on a tape.

Values: [illegible]S => [7 | 9]

Default: [illegible]fault is 9.

TRUNCATE

Description: T[illegible] UNCATE attribute specifies whether or not re[illegible]ds in the file are truncated by eliminating trailing blanks. The TRUNCATE attribute applies only to files opened for TEXT_IO in IN_FILE mode.

Values: TRUNCATE => [YES | NO]

If TRUNCATE => YES, records are truncated.

If TRUNCATE => NO, records are not truncated.

Default: The default is NO.

VOL_NUM

Description: The VOL_NUM attribute applies to tape files only (DEVICE => TAPE). It specifies the volume sequence number of a tape. Volume sequence numbers are used when a file spans several tape volumes.

Values: VOL_NUM => [1 ... 9999]

Default: The default is 1.

Examples of FORM Parameter Usage

The following examples illustrate the FORM parameter as it appears in calls to the CREATE or OPEN procedures:

Example 1

```
CREATE( FILE => FD,
        MODE => OUT_FILE,
        NAME => "FORT77.JM1#SRCE.TEST",
        FORM => "DEVICE => DISK, ORGANIZATION => PROGRAM,
                 REC_SIZE => 1024, REC_FORMAT => F" );
```

This example creates a program file. Note that the DEVICE, REC_SIZE, and REC_FORMAT attributes could be omitted, since the values supplied for those attributes match their default values.

Example 2

```
OPEN(   FILE => FD,
        MODE => IN_FILE,
        NAME => "TAPE.JM1#SRCE.TEST",
        FORM => "DEVICE => TAPE, REC_SIZE => 80, DENSITY => 6250,
                 LABEL => NONE" );
```

This example opens a 6250-bpi, nonlabeled, nine-track tape file in read-only mode. The tape contains 80-byte, fixed-length, consecutive records.

Example 3

```
CREATE( FILE => FD,
        MODE => OUT_FILE,
        NAME => "FORT77.#JM1PRT.PRTFILE",
        FORM => "ORGANIZATION => PRINT, REC_SIZE => 134,
                 COMPRESS => YES, BUF_SIZE => 18432" );
```

This example creates a print file containing 134-byte print records. The BUF_SIZE attribute specifies an 18K DMS buffer to maximize performance.

F.9 CHARACTERISTICS OF NUMERIC TYPES

F.9.1 Integer Types

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

```
SHORT_SHORT_INTEGER    -128 .. 127 -- -2**7 .. 2**7 - 1

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

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

The ranges of values for types COUNT and POSITIVE_COUNT declared in packages DIRECT_IO and TEXT_IO are as follows:

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

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

The range of values for the type FIELD declared in package TEXT_IO is as follows:

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

F.9.2 Floating Point Type Attributes

SHORT_FLOAT

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

FLOAT

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

F.9.3 Attributes of Type DURATION

DURATION'DELTA	2.0 ** -6
DURATION'SMALL	2.0 ** -6
DURATION'LARGE	131072.0
DURATION'FIRST	-131072.0
DURATION'LAST	131071.0

F.10 OTHER IMPLEMENTATION-DEPENDENT CHARACTERISTICS

F.10.1 Characteristics of the Heap

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

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

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

Space for the heap is initially claimed from the system at program start-up, and additional space may be claimed as required when the initial allocation is exhausted. You can use the binder options HEAP and MOREHEAP to control both the size of the initial allocation and the size of the individual increments. You can also use ADAPATCH to set these values.

F.10.2 Characteristics of Tasks

The default initial task stack size is 16 Kbytes. Using either the binder option TASK or the ADAPATCH program, you can set all task stacks in a program to any size from 4 Kbytes to 16 Mbytes.

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

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

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

Abnormal completion of an aborted task takes place immediately unless the abnormal task is the caller of an entry that is engaged in a rendezvous. In this case, abnormal completion of the task takes place as soon as the rendezvous is completed.

A global deadlock situation arises when every task, including the main program, is waiting for another task. When this happens, the program is aborted and the state of all tasks is displayed.

F.10.3 Definition of a Main Program

A main program must be a nongeneric library procedure with no parameters.

F.10.4 Ordering of Compilation Units

The VS Ada compiler imposes no ordering constraints on compilations beyond those required by the language.

F.10.5 Implementation-Defined Packages

The following packages are defined by the VS Ada implementation.

Package SYSTEM_ENVIRONMENT

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

The specification of package SYSTEM_ENVIRONMENT is as follows:

```
package SYSTEM_ENVIRONMENT is

   subtype EXIT_STATUS is INTEGER;

   procedure SET_EXIT_STATUS (STATUS : in EXIT_STATUS);

   procedure ABORT_PROGRAM (STATUS : in EXIT_STATUS);

   procedure CLEAR_SCREEN;

end SYSTEM_ENVIRONMENT;
```

SET_EXIT_STATUS

You can set the exit status of the program (returned in register 0 on exit) by calling SET_EXIT_STATUS. Subsequent calls of SET_EXIT_STATUS modify the exit status; the status finally returned is the status specified by the last executed call to SET_EXIT_STATUS. If SET_EXIT_STATUS is not called, the value 0 is returned.

ABORT_PROGRAM

You can cause the program to be aborted, returning the specified exit code, by calling the ABORT_PROGRAM procedure.

CLEAR_SCREEN

You can erase the entire contents of the workstation screen by calling the CLEAR_SCREEN procedure.

Package STRINGS

The VS Ada-defined package STRINGS is a utility package that provides many commonly required string manipulation facilities.

The specification of package STRINGS is as follows:

```
with UNCHECKED_DEALL  TION;
package STRINGS  s

   --                          **********
   --                          * TYPES *
   --                          **********

   type ACCESS_STRING is access STRING;
   procedure DEALLOCATE_STRING is new UNCHECKED_DEALLOCATION (STRING,
                                        ACCESS_STRING);
```

```
    --          **************
    --          * UTILITIES *
    --          **************

    function  UPPER (C : in CHARACTER) return CHARACTER;
    function  UPPER (S : in STRING) return STRING;
    procedure UPPER (S : in out STRING);

    function  LOWER (     in CHARACTER) return CHARACTER;
    function  LOWER (S       STRING) return STRING;
    procedure LOWER (S :     out STRING);

    function  CAPITAL (S    :  STRING) return STRING;
    procedure CAPITAL (S : i.      STRING);

    function  REMOVE_LEADING_BLANKS   (S : in STRING) return STRING;
    function  REMOVE_TRAILING_BLANKS  (S : in STRING) return STRING;
    function  TRIM (S : in STRING) return STRING;

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

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

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

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

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

end STRINGS;
```

ACCESS_STRING

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

DEALLOCATE_STRING

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

UPPER

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

LOWER

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

CAPITAL

The CAPITAL subprograms "capitalize" their parameters. That is, they convert the first character of the string to upper case and all subsequent characters to lower case. The procedure is more efficient than the corresponding function, as it does not use the program heap.

REMOVE_LEADING_BLANKS

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

REMOVE_TRAILING_BLANKS

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

TRIM

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

INDEX

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

NOT_INDEX

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

IS_AN_ABBREV

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

The parameter IGNORE_CASE controls whether or not case is significant.

MATCH_PATTERN

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

For example, the string "ABCDEFG" matches the pattern "A*G" and the pattern "ABCD*EFG*".

The parameter IGNORE_CASE controls whether or not case is significant.

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