REPORT DOCUMENTATION PAGE

Form Approved
OPM No. 0704-0188

AD-A244 360

Public reporting burden for this collection of i... needed, and reviewing the collection of infor... Headquarters Service, Directorate for Inform... Management and Budget, Washington, DC ...viewing instructions, searching existing data sources gathering and maintaining the data ...s collection of information, including suggestions for reducing this burden, to Washington ...ton, VA 22202-4302, and to the Office of Information and Regulatory Affairs, Office of

1. AGENCY USE ONLY (Leave ...

3. REPORT TYPE AND DATES COVERED
Final:14 Mar 1991 to 01 Jun 1993

4. TITLE AND SUBTITLE
SD-SCICON UK Limited, SX Ada MC689000 Version 1.2, Local Area VAX cluster (comprising a VAXserver 3600, 2 MicroVAX II)(Host) to MC68000 processor on an MVME117-3FP MPU VME module using an MC68881(Target), 910314N1.11134

5. FUNDING NUMBERS

6. AUTHOR(S)
AFNOR , Paris, FRANCE

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
AFNOR
Tour Europe, Cedex 7
7-92080 Paris La Defense
France

8. PERFORMING ORGANIZATION REPORT NUMBER
AVF-VSR-90502/73-911128

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

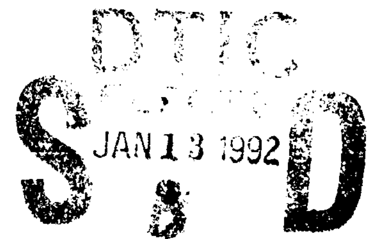
10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

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

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)
SD-SCICON UK Limited, SX Ada MC689000 Version 1.2,Manchester, England, Local Area VAX cluster (comprising a VAXserver 3600, 2 MicroVAX II)(Host) to MC68000 processor on an MVME117-3FP MPU VME module using an MC68881 floating point peripheral (bare machine)(Target), ACVC 1.11



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	

NSN 7540-01-280-550

Standard Form 298, (Rev. 2-89)
Prescribed by ANSI Std. 239-128

AVF Control Number: AVF_VSR_90502/73-911128

Ada COMPILER
VALIDATION SUMMARY REPORT:
Certificate Number: #910314N1.11134
SD-SCICON UK LIMITED
XD Ada MC68000 Version 1.2
Local Area VAX cluster (comprising a VAXserver 3600, 2 MicroVAX 2000's and 1 MicroVAX II) Host and MC68000 processor on an MVME117-3FP MPU VME module using an MC68881 floating point peripheral Target.

Prepared by
Testing Services
The National Computing Centre Limited
Oxford Road
Manchester
M1 7ED
England

VSR Version 90-01-10

92-01046

TESTING
No. 0226S

92 1 10 041

TABLE OF CONTENTS

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

CHAPTER 2
2.1 WITHDRAWN TESTS 1
2.2 INAPPLICABLE TESTS 1
2.3 TEST MODIFICATIONS 3

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

APPENDIX A

APPENDIX B

APPENDIX C

Certificate Information

The following Ada implementation was tested and determined to pass ACVC 1.11. Testing was completed on **14 March 1991.**

Compiler Name and Version:	**XD Ada MC68000 Version 1.2**
Host Computer System:	**Local Area VAX cluster (comprising a VAXserver 3600, 2 MicroVAX 2000's and 1 MicroVAX II) (under VMS 5.4)**
Target Computer System:	**MC68000 processor on an MVME117-3PF MPU VME module using an MC68881 floating point peripheral (bare machine).**

A more detailed description of this Ada implementation is found in section 3.1 of this report.
As a result of this validation effort, Validation Certificate **#910314N1.11134** is awarded to **SD-SCICON UK LIMITED.** This certificate expires on **1 March 1993.**

This report has been reviewed and is approved.

Jon Leigh
Manager, Systems Software Testing
The National Computing Centre Limited
Oxford Road
Manchester
M1 7ED
England

Ada Validation Organisation
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

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

The following declaration of conformance was supplied by the customer.

DECLARATION OF CONFORMANCE

Customer: SD-SCICON UK LIMITED

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

ACVC Version: 1.11

Ada Implementation:

Ada Compiler Name: XD Ada MC68000

Version: Version 1.2

Host Computer System: Local Area VAX cluster (comprising a VAXserver 3600, 2 MicroVAX 2000's and 1 MicroVAX II) (under VMS 5.4)

Target Computer System: MC68000 processor on an MVME117-3PF MPU VME module using an MC68881 floating point peripheral (bare machine).

Customer's Declaration

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

L. Collingbourne

Signature

14th March 1991

Date

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 organisations 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 Organisation
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 behaviour 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 3.2 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

Term	Definition
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 provided 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 Organisation (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.
Conformity	Fulfilment by a product, process or service of all requirements specified.
Customer	An individual or corporate entity who enters into an agreement with an AVF which specifies the terms and conditions for AVF services (of any kind) to be performed.

Declaration of Conformance	A formal statement from a customer assuring that conformity is realized or attainable on the Ada implementation for which validation status is realized.
Host Computer System	A computer system where Ada source programs are transformed into executable form.
Inapplicable test	A test that contains one or more test objectives found to be irrelevant for the given Ada implementation.
ISO	International Organisation for Standardisation.
Operating System	Software that controls the execution of programs and that provides services such as resource allocation, scheduling, input/output control, and data management. Usually, operating systems are predominantly software, but partial or complete hardware implementations are possible.
Target Computer System	A computer system where the executable form of Ada programs are executed.
Validated Ada Compiler	The compiler of a validated Ada implementation.
Validated Ada Implementation	An Ada implementation that has been validated successfully either by AVF testing or by registration [Pro90].
Validation	The process of checking the conformity of an Ada compiler to the Ada programming language and of issuing a certificate for this implementation.
Withdrawn test	A test found to be incorrect and not used in conformity testing. A test may be incorrect because it has an invalid test objective, fails to meet its test objective, or contains erroneous or illegal use of the Ada programming language.

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 **91-02-25.**

E28005C	B28006C	C34006D	C35508I	C35508J	C35508M
C35508N	C35702A	B41308B	C43004A	C45114A	C45346A
C45612A	C45612B	C45612C	C45651A	C46022A	B49008A
A74006A	C74308A	B83022B	B83022H	B83025B	B83025D
B83026B	C83026A	C83041A	B85001L	C86001F	C94021A
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	AD7206A	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 the ISO and 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 **159** tests have floating-point type declarations requiring more digits than SYSTEM.MAX_DIGITS:

C24113O..Y (11 tests)	C35705O..Y (11 tests)
C35706O..Y (11 tests)	C35707O..Y (11 tests)
C35708O..Y (11 tests)	C35802O..Z (12 tests)
C45241O..Y (11 tests)	C45321O..Y (11 tests)
C45421O..Y (11 tests)	C45521O..Z (12 tests)
C45524O..Z (12 tests)	C45621O..Z (12 tests)
C45641O..Y (11 tests)	C46012O..Z (12 tests)

The following 20 tests check for the predefined type LONG_INTEGER:

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

C35713B, C45423B, B86001T, and C86006H check for the predefined type SHORT_FLOAT.

C45423A, C45523A and C45622A check that if MACHINE_OVERFLOWS is TRUE and the results of various floating-point operations lie outside the range of the base type, then the proper exception is raised. For this implementation, MACHINE_OVERFLOWS is FALSE.

C45531M..P (4 tests) and C45532M..P (4 tests) check fixed-point operations for types that require a SYSTEM.MAX_MANTISSA of 47 or greater.

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.

CD2A84A, CD2A84E, CD2A84I..J (2 tests), and CD2A84O use representation clauses specifying non-default sizes for access types.

CD2B15B checks that STORAGE_ERROR is raised when the storage size specified for a collection is too small to hold a single value of the designated type; this implementation allocates more space than was specified by the length clause, as allowed by AI-00558.

The following 264 tests check for sequential, text, and direct access files:

CE2102A..C (3)	CE2102G..H (2)	CE2102K	CE2102N..Y (12)
CE2103C..D (2)	CE2104A..D (4)	CE2105A..B (2)	CE2106A..B (2)
CE2107A..H (8)	CE2107L	CE2108A..H (8)	CE2109A..C (3)
CE2110A..D (4)	CE2111A..I (9)	CE2115A..B (2)	CE2120A..B (2)
CE2201A..C (3)	EE2201D..E (2)	CE2201F..N (9)	CE2203A
CE2204A..D (4)	CE2205A	CE2206A	CE2208B
CE2401A..C (3)	EE2401D	CE2401E..F (2)	EE2401G
CE2401H..L (5)	CE2403A	CE2404A..B (2)	CE2405B
CE2406A	CE2407A..B (2)	CE2408A..B (2)	CE2409A..B (2)
CE2410A..B (2)	CE2411A	CE3102A..C (3)	CE3102F..H (3)
CE3102J..K (2)	CE3103A	CE3104A..C (3)	CE3106A..B (2)
CE3107B	CE3108A..B (2)	CE3109A	CE3110A
CE3111A..B (2)	CE3111D..E (2)	CE3112A..D (4)	CE3114A..B (2)
CE3115A	CE3119A	EE3203A	EE3204A
CE3207A	CE3208A	CE3301A	EE3301B

CE3302A	CE3304A	CE3305A	CE3401A
CE3402A	EE3402B	CE3402C..D (2)	CE3403A..C (3)
CE3403E..F (2)	CE3404B..D (3)	CE3405A	EE3405B
CE3405C..D (2)	CE3406A..D (4)	CE3407A..C (3)	CE3408A..C (3)
CE3409A	CE3409C..E (3)	EE3409F	CE3410A
CE3410C..E (3)	EE3410F	CE3411A	CE3411C
CE3412A	EE3412C	CE3413A..C(3)	CE3414A
CE3602A..D (4)	CE3603A	CE3604A..B (2)	CE3605A..E (5)
CE3606A..B (2)	CE3704A..F (6)	CE3704M..O (3)	CE3705A..E (5)
CE3706D	CE3706F..G (2)	CE3804A..P (16)	CE3805A..B (2)
CE3806A..B (2)	CE3806D..E (2)	CE3806G..H (2)	CE3904A..B (2)
CE3905A..C (3)	CE3905L	CE3906A..C (3)	CE3906E..F (2)

2.3 TEST MODIFICATIONS

Modifications (see section 1.3) were required for 13 tests.

The following test was 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.

B97103E

C45524A..K (11 tests) were graded passed by Test Modification as directed by the AVO. These tests expect that a repeated division will result in zero; but the Ada standard only requires that the result lie in the smallest safe interval. Thus, the tests were modified to check that the result was within the smallest safe interval, by adding the following code after line 141; the modified tests were passed:

ELSIF VAL <= F'SAFE_SMALL THEN COMMENT ("UNDERFLOW SEEMS GRADUAL");

CE3901A was graded passed by Test Modification as directed by the AVO. This test expects that implementations that do not support external files will raise USE_ERROR on the attempt to create a file at line 52; this implementation raises NAME_ERROR, as allowed by AI-00332. The test was modified by inserting '| NAME_ERROR' into the exception choice at line 52, and the modified test was passed.

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:

Tim Magness
SD-Scicon UK Ltd
Pembroke House
Pembroke Broadway
Camberley
Surrey
GU15 3XD

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

Colin Foster
SD-Scicon UK Ltd
Pembroke House
Pembroke Broadway
Camberley
Surrey
GU15 3XD

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.

a)	Total Number of Applicable Tests	3611
b)	Total Number of Withdrawn Tests	92
c)	Processed Inapplicable Tests	467
d)	Non-Processed I/O Tests	0
e)	Non-Processed Floating-Point Precision Tests	0

f)	Total Number of Inapplicable Tests	467	(c+d+e)
g)	Total Number of Tests for ACVC 1.11	4170	(a+b+f)

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 **467** tests were inapplicable to this implementation. All inapplicable tests were processed during validation testing. In addition, the modified tests mentioned in section 2.3 were also processed.

A **magnetic tape** containing the customized test suite (see section 1.3) was taken on-site by the validation team for processing. The contents of the **magnetic tape** were loaded on to a VAX 8600 and then copied across to the host using DECnet.

After the test files were loaded onto the host computer, the full set of tests was processed by the Ada implementation.

The tests were compiled and linked on the host computer system, as appropriate. The executable images were transferred to the target computer system by the communications link described above, and run. The results were captured on the host computer system.

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:

/LIST	used for tests requiring compilation listings
/DEV=DAY_0	in-house compiler option to remove extraneous listing information eg dates and headers.

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.

Annex B

Predefined Language Pragmas

In addition to the standard predefined pragmas, described in Annex B of the *Reference Manual for the Ada Programming Language*, XD Ada supports pragmas CALL_SEQUENCE_FUNCTION, CALL_SEQUENCE_PROCEDURE, LINK_OPTION, and TITLE, which are defined here. This annex also summarizes the definitions given elsewhere of the remaining implementation-defined pragmas.

Definitions

CALL_SEQUENCE_FUNCTION
CALL_SEQUENCE_PROCEDURE

The pragma CALL_SEQUENCE_PROCEDURE is used for describing machine code insertions or exported subprograms. It specifies how parameters are mapped onto registers, and which registers must be preserved, for machine code insertions (see Section 13.8). The pragma CALL_SEQUENCE_FUNCTION is also provided. These pragmas have the form:

```
pragma CALL_SEQUENCE_FUNCTION

       ([  [UNIT =>] internal_name
        [, [RESULT_TYPE =>] type_mark ]
        [, [PARAMETER_TYPES =>] ( parameter_types ) ]
        [, [MECHANISM =>] mechanism ]
        [, [RESULT_MECHANISM =>] mechanism_spec ]
        [, [PRESERVED_REGISTERS => ] ( registers ) ]
       );

pragma CALL_SEQUENCE_PROCEDURE

       ([  [UNIT =>] internal_name
        [, [PARAMETER_TYPES =>] ( parameter_types ) ]
        [, [MECHANISM =>] mechanism ]
        [, [PRESERVED_REGISTERS => ] ( registers ) ]
       );
```

```
parameter_types ::=
    null | type_mark {, type_mark}

mechanism ::=
    mechanism_spec | ( mechanism_spec {, mechanism_spec} )

mechanism_spec ::=
    mechanism_name [ ( [REGISTER => ] register_name ) ]

mechanism_name ::=
    VALUE          |
    REFERENCE      | BIT_REFERENCE     |
    DOPE_VECTOR    | BIT_DOPE_VECTOR

registers ::=
    null | register_name {, register_name }
```

Functions must be identified by their internal names and parameter and result types. The parameter and result types can be omitted only if there is exactly one function of that name in the same declarative part or package specification. Otherwise, both the parameter and result types must be specified.

Procedures must be identified by their internal names and parameter types. The parameter types can be omitted only if there is exactly one procedure of that name in the same declarative part or package specification. Otherwise, the parameter types must be specified.

The parameter types option specifies a series of one or more type marks (type or subtype names), not parameter names. Each type mark is positionally associated with a formal parameter in the subprogram's declaration. The absence of parameters must be indicated by the reserved word **null**.

The result type option is used only for functions; it specifies the type or subtype of the function result.

The mechanism option specifies how the imported subprogram expects its parameters to be passed (for example, by value, by reference or by descriptor). The calling program (namely the XD Ada program) is responsible for ensuring that parameters are passed in the form required by the external routine.

If the first form of the mechanism option is given (a single mechanism name without parentheses), all parameters are passed using that mechanism. If the second form is given (a series of mechanism names in parentheses and separated by commas), each mechanism name determines how the parameter in the same position in the subprogram specification will be passed. With the second form, each parameter name must have an associated mechanism name.

The result mechanism option is used only for functions; it specifies the parameter-passing mechanism for passing the result type.

Mechanism names are described in Section 13.9a.1.1.

The preserved registers option gives a list of hardware registers which are not altered by the procedure or function. If this option is omitted it implies that no registers are preserved; in this case the effect is one of the following:

- If the body of the subprogram is written in Ada, the compiler calculates which registers are preserved
- If the body of the subprogram is a machine code insertion, the pragma has the same effect as pragma IMPORT_PROCEDURE

LINK_OPTION

This pragma is used to associate link option file names with a program. Link option files are used to specify the target and mapping definitions to be used when building the program. In this way, they do not have to be explicitly defined on the XDACS LINK command line. The appropriate external target and mapping definitions (in the form of link option files) are entered into the program library by use of the XDACS command COPY LINK_OPTION/FOREIGN, as described in *Developing XD Ada Programs on VMS Systems for the MC68020*. If a suitable link option file exists in another program library, it can be copied to the current program library with the XDACS command COPY LINK_OPTION. The advantage of using link option files is that the program definition is separate from the program itself, and so can be altered without making the last compile obsolete. The LINK_OPTION pragma therefore removes the need to recompile the whole program. More detail on this topic can be found in Sections 7.9 and 8.10 of *Developing XD Ada Programs on VMS Systems for the MC68020*.

Pragma LINK_OPTION has the form:

```
pragma LINK_OPTION([
    [  [TARGET=>]link-option-file-name]
    [, [MAPPING=>]link-option-file-name]
    ]);
```

This pragma is only allowed in the outermost declarative part of a subprogram that is a library unit; at most one such pragma is allowed in a subprogram. If it occurs in a subprogram other than the main program, this pragma has no effect (see Sections 9.8 and 9.9 (LRM)).

TITLE

Takes a title or a subtitle string, or both, in either order, as arguments. Pragma TITLE has the form:

```
pragma TITLE (titling-option
   [,titling-option]);

titling-option :=
      [TITLE =>] string_literal
    | [SUBTITLE =>] string_literal
```

This pragma is allowed anywhere a pragma is allowed; the given strings supersede the default title or subtitle portions of a compilation listing.

Summary

Pragma	Meaning
EXPORT_EXCEPTION	Takes an internal name denoting an exception, and optionally takes an external designator (the name of an XD Ada Builder global symbol), and a form (ADA) as arguments. This pragma is only allowed at the place of a declarative item, and must apply to an exception declared by an earlier declarative item of the same declarative part or package specification. The pragma permits an Ada exception to be handled by programs written in XD Ada MC68020 assembly language (see Section 13.9a.3.2).
EXPORT_FUNCTION	Takes an internal name denoting a function, and optionally takes an external designator (the name of an XD Ada Builder global symbol), parameter types, and result type as arguments. This pragma is only allowed at the place of a declarative item, and must apply to a function declared by an earlier declarative item of the same declarative part or package specification. In the case of a function declared as a compilation unit, the pragma is only allowed after the function declaration

and before any subsequent compilation unit. This pragma is not allowed for a function declared with a renaming declaration, and is not allowed for a generic function (it can be given for a generic instantiation). This pragma permits an Ada function to be called from a program written in assembly language (see Section 13.9a.1.2).

EXPORT_OBJECT	Takes an internal name denoting an object, and optionally takes an external designator (the name of an XD Ada Builder global symbol), and size designator as arguments. This pragma is only allowed at the place of a declarative item at the outermost level of a library package specification or body, and must apply to a variable declared by an earlier declarative item of the same package specification or body; the variable must be of a type or subtype that has a constant size at compile time. This pragma is not allowed for objects declared with a renaming declaration, and is not allowed in a generic unit. This pragma permits an Ada object to be referred to by a routine written in assembly language (see Section 13.9a.2.2).
EXPORT_PROCEDURE	Takes an internal name denoting a procedure, and optionally takes an external designator (the name of an XD Ada Builder global symbol), and parameter types as arguments. This pragma is only allowed at the place of a declarative item, and must apply to a procedure declared by an earlier declarative item of the same declarative part or package specification. In the case of a procedure declared as a compilation unit, the pragma is only allowed after the procedure declaration and before any subsequent compilation unit. This pragma is not allowed for a procedure declared with a renaming declaration, and is not allowed for a

generic procedure (it may be given for a generic instantiation). This pragma permits an Ada routine to be called from a program written in assembly language (see Section 13.9a.1.2).

IMPORT_EXCEPTION

Takes an internal name denoting an exception, and optionally takes an external designator (the name of an XD Ada Builder global symbol), and a form (ADA) as arguments. This pragma is only allowed at the place of a declarative item, and must apply to an exception declared by an earlier declarative item of the same declarative part or package specification. The pragma is included for compatibility with VAX Ada (see Section 13.9a.3.1).

IMPORT_FUNCTION

Takes an internal name denoting a function, and optionally takes an external designator (the name of an XD Ada Builder global symbol), parameter types, and result type as arguments. Pragma INTERFACE must be used with this pragma (see Section 13.9). This pragma is only allowed at the place of a declarative item, and must apply to a function declared by an earlier declarative item of the same declarative part or package specification. In the case of a function declared as a compilation unit, the pragma is only allowed after the function declaration and before any subsequent compilation unit. This pragma is allowed for a function declared with a renaming declaration; it is not allowed for a generic function or a generic function instantiation. This pragma permits an assembly language routine to be used as an Ada function (see Section 13.9a.1.1).

IMPORT_OBJECT

Takes an internal name denoting an object, and optionally takes an external designator (the name of an XD Ada Builder global symbol), as arguments. This pragma is only allowed at the place of a declarative item at the outermost level of a library package specification or body, and must apply to a variable declared by an earlier declarative item of the same package specification or body; the variable must be of a type or subtype that has a constant size at compile time. This pragma is not allowed for objects declared with a renaming declaration, and is not allowed in a generic unit. This pragma permits storage declared in an assembly language routine to be referred to by an Ada program (see Section 13.9a.2.1).

IMPORT_PROCEDURE

Takes an internal name denoting a procedure, and optionally takes an external designator (the name of an XD Ada Builder global symbol), and parameter types as arguments. Pragma INTERFACE must be used with this pragma (see Section 13.9). This pragma is only allowed at the place of a declarative item, and must apply to a procedure declared by an earlier declarative item of the same declarative part or package specification. In the case of a procedure declared as a compilation unit, the pragma is only allowed after the procedure declaration and before any subsequent compilation unit. This pragma is allowed for a procedure declared with a renaming declaration; it is not allowed for a generic procedure or a generic procedure instantiation. This pragma permits an assembly language routine to be used as an Ada procedure (see Section 13.9a.1.1).

INTERFACE	In XD Ada, pragma INTERFACE is required in combination with pragmas IMPORT_FUNCTION and IMPORT_PROCEDURE (see Section 13.9a.1).
LEVEL	This pragma identifies a task or task type as running at interrupt level. Pragma LEVEL has one argument specifying the level for its interrupts (see Section 13.5.1).
STORAGE_UNIT	In XD Ada, the only argument allowed for this pragma is 8.
SUPPRESS_ALL	This pragma has no argument and is only allowed following a compilation unit. This pragma specifies that all run-time checks in the unit are suppressed (see Section 11.7).
VOLATILE	Takes the simple name of a variable as the single argument. This pragma is only allowed for a variable declared by an object declaration. The variable declaration and the pragma must both occur (in this order) immediately within the same declarative part or package specification. The pragma must appear before any occurrence of the name of the variable other than in an address clause or in one of the XD Ada pragmas IMPORT_OBJECT or EXPORT_OBJECT. The variable cannot be declared by a renaming declaration. The VOLATILE pragma specifies that the variable may be modified asynchronously. This pragma instructs the compiler to obtain the value of a variable from memory each time it is used (see Section 9.11).

Annex B

Predefined Language Pragmas

This chapter supplies details of three pragmas introduced by XD Ada MC68020 Version 1.2, pragma DIRECT_INTERRUPT_ENTRY, pragma IDENT and pragma TIME_SLICE. XD Ada pragmas in addition to those defined in Annex B of the *Reference Manual for the AdaProgramming Language* (CALL_SEQUENCE_FUNCTION, CALL_SEQUENCE_PROCEDURE, LEVEL, LINK_OPTION and TITLE) are described in the *XD Ada MC68020 Supplement to the Ada Language Reference Manual* for Version 1.0.

Definitions

IDENT

Takes a string literal of 31 or fewer characters as the single argument. The pragma IDENT has the following form:

```
pragma IDENT (string_literal);
```

This pragma is allowed only in the outermost declarative part of a compilation unit. The given string is used to identify the object module associated with the compilation unit in which the pragma IDENT occurs.

Summary

Pragma	Meaning
DIRECT_INTERRUPT_ENTRY	Takes the simple name of an interrupt entry, which must have no parameters, as the single argument. This pragma signals to the compiler that the interrupt

entry is to be directly connected to the hardware interrupt (see Section 13.5.1).

TIME_SLICE

Takes a static expression of the predefined fixed point type DURATION (in Package STANDARD) as the single argument. This pragma is only allowed in the outermost declarative part of a library subprogram, and at most one such pragma is allowed in a library subprogram. It has an effect only when the subprogram to which it applies is used as a main program. This pragma specifies the nominal amount of elapsed time permitted for the execution of a task when other tasks of the same priority are also eligible for execution. A positive, nonzero value of the static expression enables scheduling for all tasks in the subprogram; a negative or zero value disables it (see Section 9.8a).

Annex C

Predefined Language Environment

NOTE

The complete Appendix C (specification of the package STANDARD) is reproduced for convenience. The XD Ada additions to this package are the types SHORT_INTEGER, SHORT_SHORT_INTEGER, LONG_FLOAT, and LONG_LONG_FLOAT.

1 This appendix outlines the specification of the package STANDARD containing all predefined identifiers in the language. The corresponding package body is implementation-defined and is not shown.

2 The operators that are predefined for the types declared in the package STANDARD are given in comments since they are implicitly declared. Italics are used for pseudo-names of anonymous types (such as *universal_real*) and for undefined information (such as *implementation_defined* and *any_fixed_point_type*).

3
```
package STANDARD is
```

4
```
   type BOOLEAN is (FALSE, TRUE);
   -- The predefined relational operators for this type are as follows:

   -- function "="   (LEFT, RIGHT : BOOLEAN) return BOOLEAN;
   -- function "/="  (LEFT, RIGHT : BOOLEAN) return BOOLEAN;
   -- function "<"   (LEFT, RIGHT : BOOLEAN) return BOOLEAN;
   -- function "<="  (LEFT, RIGHT : BOOLEAN) return BOOLEAN;
   -- function ">"   (LEFT, RIGHT : BOOLEAN) return BOOLEAN;
   -- function ">="  (LEFT, RIGHT : BOOLEAN) return BOOLEAN;
```

```
-- The predefined logical operators and the predefined logical
-- negation operator are as follows:
-- function "and  (LEFT, RIGHT : BOOLEAN) return BOOLEAN;
-- function "or"  (LEFT, RIGHT : BOOLEAN) return BOOLEAN;
-- function "xor" (LEFT, RIGHT : BOOLEAN) return BOOLEAN;
-- function "not" (RIGHT : BOOLEAN) return BOOLEAN;
```

5

```
-- The universal type universal_integer is predefined.
```

6

```
type INTEGER is implementation_defined;
-- The predefined operators for this type are as follows:
-- function "="   (LEFT, RIGHT : INTEGER) return BOOLEAN;
-- function "/="  (LEFT, RIGHT : INTEGER) return BOOLEAN;
-- function "<"   (LEFT, RIGHT : INTEGER) return BOOLEAN;
-- function "<="  (LEFT, RIGHT : INTEGER) return BOOLEAN;
-- function ">"   (LEFT, RIGHT : INTEGER) return BOOLEAN;
-- function ">="  (LEFT, RIGHT : INTEGER) return BOOLEAN;
-- function "+"   (RIGHT : INTEGER) return INTEGER;
-- function "-"   (RIGHT : INTEGER) return INTEGER;
-- function "abs" (RIGHT : INTEGER) return INTEGER;

-- function "+"   (LEFT, RIGHT : INTEGER) return INTEGER;
-- function "-"   (LEFT, RIGHT : INTEGER) return INTEGER;
-- function "*"   (LEFT, RIGHT : INTEGER) return INTEGER;
-- function "/"   (LEFT, RIGHT : INTEGER) return INTEGER;
-- function "rem" (LEFT, RIGHT : INTEGER) return INTEGER;
-- function "mod" (LEFT, RIGHT : INTEGER) return INTEGER;
-- function "**"  (LEFT  : INTEGER;
                   RIGHT : INTEGER) return INTEGER;
```

7

```
-- An implementation may provide additional predefined integer types.
-- It is recommended that the names of such additional types end
-- with INTEGER as in SHORT_INTEGER or LONG_INTEGER. The specification
-- of each operator for the type universal_integer, or for
-- any additional predefined integer type, is obtained by replacing
-- INTEGER by the name of the type  in  the specification of the
-- corresponding  operator  of the type INTEGER, except for the right
-- operand of the exponentiating operator.
   type SHORT_INTEGER is implementation_defined;
   type SHORT_SHORT_INTEGER is implementation_defined;
```

8

```
-- The universal type universal_real is predefined.
```

9

```
type FLOAT is implementation_defined;
-- The predefined operators for this type are as follows:
-- function "="   (LEFT, RIGHT : FLOAT) return BOOLEAN;
-- function "/="  (LEFT, RIGHT : FLOAT) return BOOLEAN;
-- function "<"   (LEFT, RIGHT : FLOAT) return BOOLEAN;
-- function "<="  (LEFT, RIGHT : FLOAT) return BOOLEAN;
-- function ">"   (LEFT, RIGHT : FLOAT) return BOOLEAN;
-- function ">="  (LEFT, RIGHT : FLOAT) return BOOLEAN;
-- function "+"   (RIGHT : FLOAT) return FLOAT;
-- function "-"   (RIGHT : FLOAT) return FLOAT;
-- function "abs" (RIGHT : FLOAT) return FLOAT;
```

```
-- function "+"   (LEFT, RIGHT : FLOAT) return FLOAT;
-- function "-"   (LEFT, RIGHT : FLOAT) return FLOAT;
-- function "*"   (LEFT, RIGHT : FLOAT) return FLOAT;
-- function "/"   (LEFT, RIGHT : FLOAT) return FLOAT;
-- function "**"  (LEFT : FLOAT; RIGHT : INTEGER) return FLOAT;
```

10

```
-- An implementation may provide additional predefined floating-point
-- types. It is recommended that the names of such additional types
-- end with FLOAT as in SHORT_FLOAT or LONG_FLOAT.  The specification
-- of each operator for the type universal_real, or for any additional
-- predefined  floating-point type, is obtained by replacing FLOAT by
-- the name of the type in the specification of the corresponding
-- operator of the type FLOAT.

   type LONG_FLOAT is implementation_defined;

   type LONG_LONG_FLOAT is implementation_defined;
```

11

```
-- In addition, the following operators are predefined for universal
-- types:

   -- function "*" (LEFT  : universal_integer;
                    RIGHT : universal_real)    return universal_real;
   -- function "*" (LEFT  : universal_real;
                    RIGHT : universal_integer) return universal_real;
   -- function "/" (LEFT  : universal_real;
   -- The type universal_fixed is predefined.  The only operators
   -- declared for this type are
   -- function "*" (LEFT  : any_fixed_point_type;
                    RIGHT : any_fixed_point_type)
        return universal_fixed;
   -- function "/" (LEFT  : any_fixed_point_type;
                    RIGHT : any_fixed_point_type)
        return universal_fixed;
```

12

```
-- The following characters form  the  standard  ASCII  character  set.
-- Character literals corresponding to control characters are not
-- identifiers;  they are indicated in italics in this definition.
```

13

```
   type CHARACTER is

      (nul,  soh,  stx,  etx,     eot,  enq,  ack,  bel,
       bs,   ht,   lf,   vt,      ff,   cr,   so,   si,
       dle,  dc1,  dc2,  dc3,     dc4,  nak,  syn,  etb,
       can,  em,   sub,  esc,     fs,   gs,   rs,   us,
       ' ',  '!',  '"',  '#',     '$',  '%',  '&',  ''',
       '(',  ')',  '*',  '+',     ',',  '-',  '.',  '/',
       '0',  '1',  '2',  '3',     '4',  '5',  '6',  '7',
       '8',  '9',  ':',  ';',     '<',  '=',  '>',  '?',
       '@',  'A',  'B',  'C',     'D',  'E',  'F',  'G',
       'H',  'I',  'J',  'K',     'L',  'M',  'N',  'O',
       'P',  'Q',  'R',  'S',     'T',  'U',  'V',  'W',
       'X',  'Y',  'Z',  '[',     '\',  ']',  '^',  '_',
```

```
          '`',  'a',  'b',  'c',     'd',  'e',  'f',  'g',
          'h',  'i',  'j',  'k',     'l',  'm',  'n',  'o',
          'p',  'q',  'r',  's',     't',  'u',  'v',  'w',
          'x',  'y',  'z',  '{',     '|',  '}',  '~',  del);
      for CHARACTER use  --  128 ASCII character set without holes
            (0, 1, 2, 3, 4, 5, ..., 125, 126, 127);

14    -- The predefined operators for the type CHARACTER are the same
      -- as for any enumeration type.

15    package ASCII is
         --  Control characters:
         NUL       : constant CHARACTER := nul;
         SOH       : constant CHARACTER := soh;
         STX       : constant CHARACTER := stx;
         ETX       : constant CHARACTER := etx;
         EOT       : constant CHARACTER := eot;
         ENQ       : constant CHARACTER := enq;
         ACK       : constant CHARACTER := ack;
         BEL       : constant CHARACTER := bel;
         BS        : constant CHARACTER := bs;
         HT        : constant CHARACTER := ht;
         LF        : constant CHARACTER := lf;
         VT        : constant CHARACTER := vt;
         FF        : constant CHARACTER := ff;
         CR        : constant CHARACTER := cr;
         SO        : constant CHARACTER := so;
         SI        : constant CHARACTER := si;
         DLE       : constant CHARACTER := dle;
         DC1       : constant CHARACTER := dc1;
         DC2       : constant CHARACTER := dc2;
         DC3       : constant CHARACTER := dc3;
         DC4       : constant CHARACTER := dc4;
         NAK       : constant CHARACTER := nak;
         SYN       : constant CHARACTER := syn;
         ETB       : constant CHARACTER := etb;
         CAN       : constant CHARACTER := can;
         EM        : constant CHARACTER := em;
         SUB       : constant CHARACTER := sub;
         ESC       : constant CHARACTER := esc;
         FS        : constant CHARACTER := fs;
         GS        : constant CHARACTER := gs;
         RS        : constant CHARACTER := rs;
         US        : constant CHARACTER := us;
         DEL       : constant CHARACTER := del;
```

```
   -- Other characters:
   EXCLAM      : constant CHARACTER := '!';
   QUOTATION   : constant CHARACTER := '"';
   SHARP       : constant CHARACTER := '#';
   DOLLAR      : constant CHARACTER := '$';
   PERCENT     : constant CHARACTER := '%';
   AMPERSAND   : constant CHARACTER := '&';
   COLON       : constant CHARACTER := ':';
   SEMICOLON   : constant CHARACTER := ';';
   QUERY       : constant CHARACTER := '?';
   AT_SIGN     : constant CHARACTER := '@';
   L_BRACKET   : constant CHARACTER := '[';
   BACK_SLASH  : constant CHARACTER := '\';
   R_BRACKET   : constant CHARACTER := ']';
   CIRCUMFLEX  : constant CHARACTER := '^';
   UNDERLINE   : constant CHARACTER := '_';
   GRAVE       : constant CHARACTER := '`';
   L_BRACE     : constant CHARACTER := '{';
   BAR         : constant CHARACTER := '|';
   R_BRACE     : constant CHARACTER := '}';
   TILDE       : constant CHARACTER := '~';

   -- Lower case letters:
   LC_A : constant CHARACTER := 'a';
   ...
   LC_Z : constant CHARACTER := 'z';
end ASCII;
```

16
```
-- Predefined subtypes:
subtype NATURAL  is INTEGER range 0 .. INTEGER'LAST;
subtype POSITIVE is INTEGER range 1 .. INTEGER'LAST;
```

17
```
-- Predefined string type:
type STRING is array(POSITIVE range <>) of CHARACTER;
pragma PACK(STRING);
```

18
```
-- The predefined operators for this type are as follows:
-- function "="  (LEFT, RIGHT : STRING) return BOOLEAN;
-- function "/=" (LEFT, RIGHT : STRING) return BOOLEAN;
-- function "<"  (LEFT, RIGHT : STRING) return BOOLEAN;
-- function "<=" (LEFT, RIGHT : STRING) return BOOLEAN;
-- function ">"  (LEFT, RIGHT : STRING) return BOOLEAN;
-- function ">=" (LEFT, RIGHT : STRING) return BOOLEAN;

-- function "&" (LEFT  : STRING;
                 RIGHT : STRING)    return STRING;
-- function "&" (LEFT  : CHARACTER;
                 RIGHT : STRING)    return STRING;
-- function "&" (LEFT  : STRING;
                 RIGHT : CHARACTER) return STRING;
-- function "&" (LEFT  : CHARACTER;
                 RIGHT : CHARACTER) return STRING;
```

19

```
    type DURATION is delta implementation_defined
       range implementation_defined;

    -- The predefined operators for the type DURATION are the same as for
    -- any fixed-point type.
```

20

```
 -- The predefined exceptions:

    CONSTRAINT_ERROR : exception;
    NUMERIC_ERROR    : exception;
    PROGRAM_ERROR    : exception;
    STORAGE_ERROR    : exception;
    TASKING_ERROR    : exception;
end STANDARD;
```

21 Certain aspects of the predefined entities cannot be completely described in the language itself. For example, although the enumeration type BOOLEAN can be written showing the two enumeration literals FALSE and TRUE, the short-circuit control forms cannot be expressed in the language.

Note:

22 The language definition predefines the following library units:

- The package CALENDAR (see 9.6)
- The package SYSTEM (see 13.7)
- The package MACHINE_CODE (see 13.8)
- The generic procedure UNCHECKED_DEALLOCATION (see 13.10.1)
- The generic function UNCHECKED_CONVERSION (see 13.10.2)
- The generic package SEQUENTIAL_IO (see 14.2.3)
- The generic package DIRECT_IO (see 14.2.5)
- The package TEXT_IO (see 14.3.10)
- The package IO_EXCEPTIONS (see 14.5)
- The package LOW_LEVEL_IO (see 14.6)

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 valued 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
$MAX_IN_LEN	255
$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
$ACC_SIZE	32
$ALIGNMENT	1
$COUNT_LAST	2147483647
$DEFAULT_MEM_SIZE	16777216
$DEFAULT_STOR_UNIT	8
$DEFAULT_SYS_NAME	MC68000
$DELTA_DOC	2#1.#E-31
$ENTRY_ADDRESS	SYSTEM.TO_ADDRESS (16#68#)
$ENTRY_ADDRESS1	SYSTEM.TO_ADDRESS (16#6C#)
$ENTRY_ADDRESS2	SYSTEM.TO_ADDRESS (16#70#)
$FIELD_LAST	255
$FILE_TERMINATOR	' '
$FIXED_NAME	NO_SUCH_TYPE
$FLOAT_NAME	LONG_LONG_FLOAT
$FORM_STRING	""
$FORM_STRING2	"CANNOT_RESTRICT_FILE_CAPACITY"
$GREATER_THAN_DURATION	131072.0
$GREATER_THAN_DURATION_BASE_LAST	131073.0
$GREATER_THAN_FLOAT_BASE_LAST	3.40283E+38

$GREATER_THAN_FLOAT_SAFE_LARGE	4.255354E+37
$GREATER_THAN_SHORT_FLOAT_SAFE_LARGE	"NO_SUCH_TYPE"
$HIGH_PRIORITY	15
$ILLEGAL_EXTERNAL_FILE_NAME1	ILLEGAL_EXTERNAL_FILE_NAME_1
$ILLEGAL_EXTERNAL_FILE_NAME2	ILLEGAL_EXTERNAL_FILE_NAME_2
$INAPPROPRIATE_LINE_LENGTH	-1
$INAPPROPRIATE_PAGE_LENGTH	-1
$INCLUDE_PRAGMA1	PRAGMA INCLUDE ("A28006D1.TST")
$INCLUDE_PRAGMA2	PRAGMA INCLUDE ("B28006D1.TST")
$INTEGER_FIRST	-2147483648
$INTEGER_LAST	2147483647
$INTEGER_LAST_PLUS_1	2147483648
$INTERFACE_LANGUAGE	ASSEMBLER
$LESS_THAN_DURATION	-131072.0
$LESS_THAN_DURATION_BASE_FIRST	-131073.0
$LINE_TERMINATOR	' '
$LOW_PRIORITY	0
$MACHINE_CODE_STATEMENT	OPERANDLESS_INST' (OPCODE=>NOP);
$MACHINE_CODE_TYPE	OPERANDLESS_INST
$MANTISSA_DOC	31
$MAX_DIGITS	18
$MAX_INT	2147483647
$MAX_INT_PLUS_1	2147483648

$MIN_INT	-2147483648
$NAME	SHORT_SHORT_INTEGER
$NAME_LIST	MC68000
$NAME_SPECIFICATION1	NO_SUCH_NAME
$NAME_SPECIFICATION2	NO_SUCH_NAME
$NAME_SPECIFICATION3	NO_SUCH_NAME
$NEG_BASED_INT	16#FFFF_FFFF#
$NEW_MEM_SIZE	123456
$NEW_STOR_UNIT	8
$NEW_SYS_NAME	MC68000
$PAGE_TERMINATOR	' '
$RECORD_DEFINITION	RECORD OPCODE:OPERANDLESS_OP; END RECORD;
$RECORD_NAME	OPERANDLESS_INST
$TASK_SIZE	32
$TASK_STORAGE_SIZE	2048
$TICK	2#1.0#E-13
$VARIABLE_ADDRESS	SYSTEM.TO_ADDRESS (16#40C#)
$VARIABLE_ADDRESS1	SYSTEM.TO_ADDRESS (16#408#)
$VARIABLE_ADDRESS2	SYSTEM.TO_ADDRESS (16#404#)
$YOUR_PRAGMA	EXPORT_OBJECT EXPORT_EXCEPTION EXPORT_FUNCTION EXPORT_PROCEDURE IMPORT_OBJECT IMPORT_EXCEPTION IMPORT_FUNCTION IMPORT_PROCEDURE

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.

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.

Appendix A

XDACS LINK Command Definition

LINK

Creates an executable image file for the specified units.

Format

LINK *unit-name [file-spec[,...]]*

LINK/NOMAIN *unit-name[,...] file-spec[,...]*

Command Qualifiers	Defaults
/AFTER=time	/AFTER=TODAY
/BATCH_LOG=file-spec	See text.
/BRIEF	See text.
/COMMAND[=file-spec]	See text.
/[NO]DEBUG	/NODEBUG
/ELABORATION=file-spec	See text.
/FULL	See text.
/[NO]IMAGE[=file-spec]	/IMAGE
/[NO]KEEP	/KEEP
/[NO]LOG	/NOLOG
/[NO]MAIN	/MAIN
/[NO]MAP[=file-spec]	/NOMAP
/NAME=job-name	See text.
/[NO]NOTIFY	/NOTIFY
/OUTPUT=file-spec	/OUTPUT=SYS$OUTPUT
/[NO]PRINTER[=queue-name]	/NOPRINTER
/QUEUE=queue-name	/QUEUE=SYS$BATCH
/[NO]SELECTIVE	/SELECTIVE
/SUBMIT	/WAIT
/WAIT	/WAIT

Parameter Qualifiers	Defaults
/LIBRARY	See text.
/MAPPING	See text.
/TARGET	See text.

Prompts

_Unit:
_File:

Command Parameters

unit-name

By default (or if you specify the /MAIN qualifier):

- You can specify only one unit, the source code of which must be written in XD Ada.
- The parameter *unit-name* specifies the XD Ada main program, which must be a procedure or function with no parameters. If the main program is a function, it must return a value of a discrete type; the function value is used as the VMS image exit value.

If you specify the /NOMAIN qualifier:

- You can specify one or more foreign units that are to be included in the executable image. The unit names may include percent signs (%) and asterisks (*) as wildcard characters. (See the *VMS DCL Concepts Manual* for detailed information on wildcard characters.)
- The image transfer address comes from one of the foreign files specified.

file-spec

Specifies a list of object files, object libraries, mapping definition files, and target definition files, that are to be used in linking the program. The default directory is the current default directory. The default file type is .XOB, unless the /LIBRARY, /MAPPING, or /TARGET qualifier is used. No wildcard characters are allowed in a file specification.

If the file is an object library, you must use the /LIBRARY qualifier. The default file type is .XLB.

If the file is a mapping definition file, you must use the /MAPPING qualifier. The default file type is .MPD.

If the file is a target definition file you must use the /TARGET qualifier. The default file type is .TGD.

If you specify the /NOMAIN qualifier, the image transfer address comes from one of the files (not units) specified.

Description

The LINK command performs the following steps:

1. Runs the prebuild phase to generate an elaboration list.
2. Checks if a pragma LINK_OPTION is specified for the main program, and if specified, verifies that the designated link option name is available in the current program library. If available, the copied link option files in the library corresponding to the link option are used, unless overridden by the /TARGET or /MAPPING qualifiers.

   Note that, unlike the CHECK command, the pragma LINK_OPTION association for units other than the main program unit is not checked.

   If no target link option is given for the main program unit or the designated target link option is not found in the library, and the logical name XDADA$TARGET_DEF is not defined, and a /TARGET qualifier is not specified on the LINK command line, an error is issued. If no mapping link option is given for the main program unit or the designated mapping link option is not found in the library, and the logical name XDADA$MAPPING_DEF is not defined, and a /MAPPING qualifier is not specified on the XDACS LINK command line, the default mapping in the target definition file is used.
3. If LINK/NOMAIN is not specified, checks that only one unit is specified and that it is an XD Ada main program.
4. Forms the closure of the main program (LINK/MAIN) or of the specified units (LINK/NOMAIN) and verifies that all units in the closure are present, current and complete. If XDACS detects an error, the operation is terminated at the end of the prebuild phase.
5. Creates a DCL command file for the builder. The command file is deleted after the LINK operation is completed or terminated, unless LINK/COMMAND is specified. If LINK/COMMAND is specified, the command file is retained for future use, and the build phase is not carried out.
6. Unless the /COMMAND qualifier is specified, performs the build phase as follows:
   a. By default (LINK/WAIT), the command file generated in step 5 is executed in a subprocess. You must wait for the build operation to terminate before issuing another command. Note that when you specify the /WAIT qualifier (the default), process logical names are propagated to the subprocess generated to execute the command file.

b. If you specify the /SUBMIT qualifier, the builder command file is submitted as a batch job.

7. If the /DEBUG qualifier is included in the command line the debug symbol table information is placed in the .XXE file.
8. Creates a loadable output file with a default file type of .XXE.

XDACS output originating before the builder is invoked is reported to your terminal by default, or to a file specified with the /OUTPUT qualifier. Diagnostics are reported to your terminal, by default, or to a log file if the LINK command is executed in batch mode (XDACS LINK/SUBMIT).

See *Developing XD Ada Programs on VMS Systems for the MC68020* and the *XD Ada Version 1.2 New Features Manual* for more information on the XD Ada target-specific builder commands.

Command Qualifiers

/AFTER = time

Requests that the batch job be held until after a specific time, when the LINK command is executed in batch mode (LINK/SUBMIT). If the specified time has already passed, the job is queued for immediate processing.

You can specify either an absolute time or a combination of absolute and delta time. See the *VMS DCL Concepts Manual* (or type HELP Specify Date-Time at the DCL prompt) for complete information on specifying time values.

/BATCH_LOG = file-spec

Provides a file specification for the batch log file when the LINK command is executed in batch mode (LINK/SUBMIT).

If you do not give a directory specification with the *file-spec* option, the batch log file is created by default in the current default directory. If you do not give a file specification, the default file name is the job name specified with the /NAME=job-name qualifier. If no job name has been specified, the program library manager creates a file name comprising up to the first 39 characters of the first unit name specified. If you specified LINK/NOMAIN and no job name and there is a wildcard character in the first unit specified, the program library manager uses the default file name XDACS_LINK. The default file type is .LOG.

/BRIEF

Directs the builder to produce a brief image map file. The /BRIEF qualifier is valid only if you also specify the /MAP qualifier with the LINK command. The /BRIEF qualifier is incompatible with the /FULL qualifier.

A brief image map file contains only the following sections:

- Object module information
- Segment mapping information
- Link run statistics

See also the description of the /FULL qualifier.

/COMMAND[=file-spec]

Controls whether the builder is invoked as a result of the LINK command, and determines whether the command file generated to invoke the builder is saved. If you specify the /COMMAND qualifier, XDACS does not invoke the builder, and the generated command file is saved for you to invoke or submit as a batch job.

The *file-spec* option allows you to enter a file specification for the generated command file. The default directory for the command file is the current default directory. By default, XDACS provides a file name comprising up to the first 39 characters of the first unit name specified. If you specified LINK/NOMAIN and you used a wildcard character in the first unit name specified, the program library manager uses the default file name XDACS_LINK. The default file type is .COM. No wildcard characters are allowed in the file specification.

By default, if the /COMMAND qualifier is not specified, XDACS deletes the generated command file when the LINK command completes normally or is terminated.

/DEBUG
/NODEBUG (D)

Controls whether a debugger symbol table is generated in the loadable image file.

By default, no debugger symbol table is created.

/ELABORATION=file-spec

Provides a file specification for the object file generated by the LINK command. The file is retained by XDACS only when the /COMMAND qualifier is used: that is, when the result of the LINK operation is to

produce a builder command file for future use, rather than to invoke the builder immediately.

The generated object file contains the code that directs the elaboration of library packages in the closure of the units specified. Unless you also specify the /NOMAIN qualifier, the object file also contains the image transfer address.

The default directory for the generated text file is the current default directory. The default file type is .ELB. No wildcard characters are allowed in the file specification.

By default, if you do not specify the /ELABORATION qualifier, XDACS provides a file name comprising up to the first 39 characters of the first unit name specified.

By default, if you do not specify the /COMMAND qualifier, XDACS deletes the generated object file when the LINK command completes normally or is terminated.

/FULL

Directs the builder to produce a full image map file, which is the most complete image map. The /FULL qualifier is valid only if you also specify the /MAP qualifier with the LINK command. Also, the /FULL qualifier is incompatible with the /BRIEF qualifier.

A full image map file contains the following sections:

- Object module information
- Segment mapping information
- Symbol address information
- Exception numbers
- Link run statistics

/IMAGE[=file-spec] (D)
/NOIMAGE

Controls whether the LINK command creates a loadable image file and optionally provides a file specification for the file. The default file type is .XXE. No wildcard characters are allowed in the file specification.

By default, an executable image file is created with a file name comprising up to the first 39 characters of the first unit name specified.

/KEEP (D)
/NOKEEP
Controls whether the batch log file generated is deleted after it is printed when the LINK command is executed in batch mode (LINK/SUBMIT).

By default, the log file is not deleted.

/LOG
/NOLOG (D)
Controls whether a list of all the units included in the executable image is displayed. The display shows the units according to the order of elaboration for the program.

By default, a list of all the units included in the executable image is not displayed.

/MAIN (D)
/NOMAIN
Controls where the image transfer address is to be found.

The /MAIN qualifier indicates that the XD Ada unit specified determines the image transfer address, and hence is to be a main program.

The /NOMAIN qualifier indicates that the image transfer address comes from one of the files specified, and not from one of the XD Ada units specified.

By default (/MAIN), only one XD Ada unit can be specified, and that unit must be an XD Ada main program.

/MAP[=file-spec]
/NOMAP (D)
Controls whether the builder creates an image map file and optionally provides a file specification for the file. The default directory for the image map file is the current directory. The default file name comprises up to the first 39 characters of the first unit name specified. The default file type is .MAP. No wildcard characters are allowed in the file specification.

If neither the /BRIEF nor the /FULL qualifier is specified with the /MAP qualifier, /BRIEF is assumed.

By default, no image map file is created.

/NAME=job-name

Specifies a string to be used as the job name and as the file name for the batch log file when the LINK command is executed in batch mode (LINK/SUBMIT). The job name can have from 1 to 39 characters.

By default, if you do not specify the /NAME qualifier, XDACS creates a job name comprising up to the first 39 characters of the first unit name specified. If you specify LINK/NOMAIN but do not specify the /NAME qualifier, and you use a wildcard character in the first unit name specified, the program library manager uses the default file name XDACS_LINK. In these cases, the job name is also the file name of the batch log file.

/NOTIFY (D)
/NONOTIFY

Controls whether a message is broadcast when the LINK command is executed in batch mode (LINK/SUBMIT). The message is broadcast to any terminal at which you are logged in, notifying you that your job has been completed or terminated.

By default, a message is broadcast.

/OUTPUT=file-spec

Requests that any output generated before the builder is invoked be written to the file specified rather than to SYS$OUTPUT. Any diagnostic messages are written to both SYS$OUTPUT and the file.

The default directory is the current default directory. If you specify a file type but omit the file name, the default file name is XDACS. The default file type is .LIS. No wildcard characters are allowed in the file specification.

By default, the LINK command output is written to SYS$OUTPUT.

/PRINTER[=queue-name]
/NOPRINTER (D)

Controls whether the log file is queued for printing when the LINK command is executed in batch mode (LINK/SUBMIT) and the batch job is completed.

The /PRINTER qualifier allows you to specify a particular print queue. The default print queue for the log file is SYS$PRINT.

By default, the log file is not queued for printing. If you specify /NOPRINTER, /KEEP is assumed.

/QUEUE = queue-name

Specifies the batch job queue in which the job is entered when the LINK command is executed in batch mode (LINK/SUBMIT).

By default, if the /QUEUE qualifier is not specified, the job is placed in the default system batch job queue, SYS$BATCH.

/SELECTIVE (D)
/NOSELECTIVE

Specifies whether selective linking is performed.

Performing selective linking ensures that only subprograms that are called will be linked into the program image. Subprograms within the closure of the main program that are not actually called will be omitted from the image file. Selective linking produces a program image that has been optimized according to size.

Non-selective linking ensures that all defined subprograms are linked into the image.

By default, selective linking is performed.

/SUBMIT

Directs XDACS to submit the command file generated for the builder to a batch queue. You can continue to issue commands in your current process without waiting for the batch job to complete. The builder output is written to a batch log file.

By default, the generated command file is executed in a subprocess (LINK/WAIT).

/WAIT

Directs XDACS to execute the command file generated for the builder in a subprocess. Execution of your current process is suspended until the subprocess completes. The builder output is written directly to your terminal. Note that process logical names are propagated to the subprocess generated to execute the command file.

By default, XDACS executes the command file generated for the builder in a subprocess: you must wait for the subprocess to terminate before you can issue another command.

Parameter Qualifiers

/LIBRARY

Indicates that the associated input file is an object module library to be searched for modules to resolve any undefined symbols in the input files. The default file type is .XLB.

By default, if you do not specify the /LIBRARY qualifier, the file is assumed to be an object file with a default file type of .XOB.

/MAPPING

Indicates that the associated input file is a mapping definition file. Mapping definition files control the location of the program on the target system. The default file type is .MPD.

By default, if you do not specify the /MAPPING qualifier, the file is assumed to be an object file with a default file type of .XOB.

/TARGET

Indicates that the associated input file is a target definition file. Target definition files describe the target system's memory. The default file type is .TGD.

By default, if you do not specify the /TARGET qualifier, the file is assumed to be an object file with a default file type of .XOB.

Examples

1.
```
XDACS> LINK CONTROL_LOOP
%ACS-I-CL_LINKING, Invoking the XD Ada Builder
```

The LINK command forms the closure of the unit CONTROL_LOOP, which is an XD Ada main program, creates a builder command file and package elaboration file, then invokes the command file in a spawned subprocess.

2.
```
XDACS> LINK/SUBMIT CONTROL_LOOP LOOP_FUNCTIONS/LIBRARY
%ACS-I-CL_SUBMITTED, Job CONTROL_LOOP (queue ALL_BATCH, entry 134)
    started on FAST_BATCH
```

The LINK command instructs the builder to link the closure of the XD Ada main program CONTROL_LOOP against the library LOOP_FUNCTIONS.XLB. The /SUBMIT qualifier causes XDACS to submit the builder command file as a batch job.

3.
```
XDACS> LINK/NOMAIN FLUID_VOLUME,COUNTER MONITOR.XOB
%ACS-I-CL_LINKING, Invoking the XD Ada Builder
```

The LINK command builds the XD Ada units FLUID_VOLUME and COUNTER with the foreign object file MONITOR.XOB. The /NOMAIN qualifier tells the builder that the image transfer address is in the foreign file.

Appendix B

XDADA Command Definition

XDADA

Invokes the XD Ada compiler to compile one or more source files.

Format

XDADA *file-spec[,...]*

Command Qualifiers	Defaults
/LIBRARY=directory-spec	/LIBRARY=XDADA$LIB

Positional Qualifiers	Defaults
/[NO]ANALYSIS_DATA[=file-spec]	/NOANALYSIS_DATA
/[NO]CHECK	See text.
/[NO]COPY_SOURCE	/COPY_SOURCE
/[NO]DEBUG[=(option[,...])]	/DEBUG=ALL
/[NO]DIAGNOSTICS[=file-spec]	/NODIAGNOSTICS
/[NO]ERROR_LIMIT[=n]	/ERROR_LIMIT=30
/[NO]LIST[=file-spec]	/NOLIST
/[NO]LOAD[=option]	/LOAD=REPLACE
/[NO]MACHINE_CODE[=option]	/NOMACHINE_CODE
/[NO]NOTE_SOURCE	/NOTE_SOURCE
/[NO]OPTIMIZE[=option]	See text.
/[NO]PREDEFINED_UNIT	/NOPREDEFINED_UNIT
/[NO]SHOW[=option]	/SHOW=PORTABILITY
/[NO]SYNTAX_ONLY	/NOSYNTAX_ONLY
/[NO]WARNINGS[=(option[,...])]	See text.

Prompt

_File:

Command Parameters

file-spec

Specifies one or more XD Ada source files to be compiled. If you do not specify a file type, the compiler uses the default file type of .ADA. No wildcard characters are allowed in the file specifications.

If you specify several source files as arguments to the XDADA command, you must separate adjacent file specifications with a comma (,). If you specify more than one input file, you must separate adjacent file specifications with a comma (,). You cannot use a plus sign (+) to separate file specifications.

Description

The XDADA command is one of four commands used to compile compilation units. The other three are the XDACS COMPILE, RECOMPILE and LOAD commands.

The XDADA command can be used at any time to compile one or more source files (.ADA). Source files are compiled in the order they appear on the command line. If a source file contains more than one compilation unit, they are compiled in the order they appear in the source file.

The XDADA command compiles units in the context of the current program library. Whenever a compilation unit is compiled without error, the current program library is updated with the object module and other products of the compilation.

Command Qualifiers

/LIBRARY=directory-spec
Specifies the program library that is to be the current program library for the duration of the compilation. The directory specified must be an already existing XD Ada program library. No wildcard characters are allowed in the directory specification.

By default, the current program library is the program library last specified in an XDACS SET LIBRARY command. The logical name XDADA$LIB is assigned to the program library specified in an XDCAS SET LIBRARY command.

Positional Qualifiers

/ANALYSIS_DATA[=file-spec]
/NOANALYSIS_DATA (D)
Controls whether a data analysis file containing source code cross-reference and static analysis information is created. The data analysis

file is supported only for use with DIGITAL layered products, such as the VAX Source Code Analyzer.

One data analysis file is created for each source file compiled. The default directory for data analysis files is the current default directory. The default file name is the name of the source file being compiled. The default file type is .ANA. No wildcard characters are allowed in the file specification.

By default, no data analysis file is created.

/CHECK
/NOCHECK
Controls whether all run-time checks are suppressed. The /NOCHECK qualifier is equivalent to having all possible SUPPRESS pragmas in the source code.

Explicit use of the /CHECK qualifier overrides any occurrences of the pragmas SUPPRESS and SUPPRESS_ALL in the source code, without the need to edit the source code.

By default, run-time checks are suppressed only in cases where a pragma SUPPRESS or SUPPRESS_ALL appears in the source.

See the *Reference Manual for the Ada Programming Language* for more information on the pragmas SUPPRESS and SUPPRESS_ALL.

/COPY_SOURCE (D)
/NOCOPY_SOURCE
Controls whether a copied source file (.ADC) is created in the current program library when a compilation unit is compiled without error. The RECOMPILE command (and thus the COMPILE command) requires that a copied source file exist in the current program library for any unit that is to be recompiled.

By default, a copied source file is created in the current program library when a unit is compiled without error.

/DEBUG[=(option[,...])] (D)
/NODEBUG
Controls which compiler debugging options are provided. You can debug XD Ada programs with the XD Ada Debugger. You can request the following options:

ALL	Provides both SYMBOLS and TRACEBACK.
NONE	Provides neither SYMBOLS nor TRACEBACK.
[NO]SYMBOLS	Controls whether debugger symbol records are included in the object file.
[NO]TRACEBACK	Controls whether traceback information (a subset of the debugger symbol information) is included in the object file.

By default, both debugger symbol records and traceback information are included in the object file (/DEBUG=ALL, or equivalently: /DEBUG).

/DIAGNOSTICS[=file-spec]
/NODIAGNOSTICS (D)

Controls whether a diagnostics file containing compiler messages and diagnostic information is created. The diagnostics file is supported only for use with DIGITAL layered products, such as the VAX Language-Sensitive Editor.

One diagnostics file is created for each source file compiled. The default directory for diagnostics files is the current default directory. The default file name is the name of the source file being compiled. The default file type is .DIA. No wildcard characters are allowed in the file specification.

By default, no diagnostics file is created.

/ERROR_LIMIT[=n]
/NOERROR_LIMIT

Controls whether execution of the XDADA command for a given compilation unit is terminated upon the occurrence of the nth E-level error within that unit.

Error counts are not accumulated across a sequence of compilation units. If the /ERROR_LIMIT=n option is specified, each compilation unit may have up to n-1 errors without terminating the compilation. When the error limit is reached within a compilation unit, compilation of that unit is terminated, but compilation of subsequent units continues.

The /ERROR_LIMIT=0 option is equivalent to ERROR_LIMIT=1.

By default, execution of the XDADA command is terminated for a given compilation unit upon the occurrence of the 30th E-level error within that unit (equivalent to /ERROR_LIMIT=30).

/LIST[=file-spec]
/NOLIST (D)

Controls whether a listing file is created. One listing file is created for each source file compiled. The default directory for listing files is the current default directory. The default file name is the name of the source file being compiled. The default file type is .LIS. No wildcard characters are allowed in the file specification.

By default, the XDADA command does not create a listing file.

/LOAD[=option] (D)
/NOLOAD

Controls whether the current program library is updated with the successfully processed units contained in the specified source files. Depending on other qualifiers specified (or not specified) with the XDADA command, processing can involve full compilation, syntax checking only, and so on. The /NOLOAD qualifier causes the units in the specified source files to be processed, but prevents the current program library from being updated.

You can specify the following option:

[NO]REPLACE	Controls whether a unit added to the current program library replaces an existing unit with the same name. If you specify the NOREPLACE option, the unit is added to the current program library only if no existing unit has the same name, except if the new unit is the corresponding body of an existing specification or vice versa.

By default, the current program library is updated with the successfully processed units, and a unit added to the current program library replaces an existing unit with the same name.

/MACHINE_CODE[=option]
/NOMACHINE_CODE (D)

Controls whether generated machine code (approximating assembly language notation) is included in the listing file.

You can specify one of the following options:

SYMBOLIC:NONE	Provides machine code listing with no annotation.
SYMBOLIC:NORMAL	Provides machine code in the listing file; where possible, instructions are annotated with simple Ada names.
SYMBOLIC:MAXIMAL	Provides machine code in the listing file; where possible, instructions are annotated with Ada names, in expanded form if necessary.

The /MACHINE_CODE qualifier without options is equivalent to /MACHINE_CODE = SYMBOLIC:NORMAL.

/NOTE_SOURCE (D)
/NONOTE_SOURCE

Controls whether the file specification of the source file is noted in the program library when a unit is compiled without error. The COMPILE command uses this information to locate revised source files.

By default, the file specification of the source file is noted in the program library when a unit is compiled without error.

/OPTIMIZE[=(option[, . . .])
/NOOPTIMIZE

Controls the level of optimization that is applied in producing the compiled code. You can specify one of the following primary options:

TIME	Provides full optimization with time as the primary optimization criterion. Overrides any occurrences of the pragma OPTIMIZE(SPACE) in the source code.
SPACE	Provides full optimization with space as the primary optimization criterion. Overrides any occurrences of the pragma OPTIMIZE(TIME) in the source code.
DEVELOPMENT	Suggested when active development of a program is in progress. Provides some optimization, but development considerations and ease of debugging take preference over optimization. This option overrides pragmas that establish a dependence on a subprogram (the pragma INLINE), and thus reduces the need for recompilations when such bodies are modified.
NONE	Provides no optimization. Suppresses exp[illegible]ions in line of subprograms, including those specified by the pragma INLINE.

The /NOOPTIMIZE qualifier is equivalent to /OPTIMIZE = NONE.

By default, the XDADA command applies full optimization with time as the primary optimization criterion (like /OPTIMIZE=TIME, but observing uses of the pragma OPTIMIZE).

The /OPTIMIZE qualifier also has a set of secondary options that you can use separately or together with the primary options to override the default behavior for inline expansion and code motion.

The INLINE secondary option can have the following values:

INLINE:NONE	Disables subprogram expansion in line. This option overrides any occurrences of the pragma INLINE in the source code, without having to edit the source file. It also disables implicit expansion in line of subprograms. (*Implicit expansion in line* means that the compiler assumes a pragma INLINE for certain subprograms as an optimization.) A call to a subprogram in another unit is not expanded in line, regardless of the /OPTIMIZE options in effect when that unit was compiled.
INLINE:NORMAL	Provides normal subprogram expansion in line. Subprograms to which an explicit pragma INLINE applies are expanded in line under certain conditions. In addition, some subprograms are implicitly expanded in line. The compiler assumes a pragma INLINE for calls to some small local subprograms (subprograms that are declared in the same unit as the unit in which the call occurs).
INLINE:SUBPROGRAMS	Provides maximal subprogram expansion in line. In addition to the normal subprogram expansion in line that occurs when INLINE:NORMAL is specified, this option results in implicit expansion in line of some small subprograms declared in other units. The compiler assumes a pragma INLINE for any subprogram if it improves execution speed and reduces code size. This option may establish a dependence on the body of another unit, as would be the case if a pragma INLINE were specified explicitly in the source code.

INLINE:MAXIMAL	Provides maximal subprogram expansion in line. Maximal subprogram expansion in line occurs as for INLINE:SUBPROGRAMS.
INLINE:GENERICS	Provides normal subprogram inline expansion and maximal generic inline expansion. With this option, subprogram inline expansion occurs in the same manner as for INLINE:NORMAL. The compiler assumes a pragma INLINE_GENERIC for every instantiation in the unit being compiled unless a generic body is not available. This option may establish a dependence on the body of another unit, as would be the case if a pragma INLINE_GENERIC were specified explicitly in the source code.

The MOTION secondary option can have the following values:

MOTION:NONE	Disables code motion optimizations.
MOTION:LOOPS	Permits code motion optimization of loops. Where the compiler detects that a loop body contains invariant processing, it may generate code in which this processing is performed before entry to the loop instead of within the loop.
MOTION:MAXIMAL	Permits all code motion optimizations. In addition to the optimization of loops that occurs when MOTION:LOOPS is specified, this option permits analogous optimization of if and case statements: where the compiler detects that the branches of an if or case statement contain common processing, it may generate code in which this processing is performed before evaluation of the corresponding condition or case expression instead of within the branches.

By default, the /OPTIMIZE qualifier primary options have the following secondary-option values:

/OPTIMIZE=TIME	=(INLINE:NORMAL,MOTION:LOOPS)
/OPTIMIZE=SPACE	=(INLINE:NORMAL,MOTION:MAXIMAL)
/OPTIMIZE=DEVELOPMENT	=(INLINE:NONE,MOTION:NONE)
/OPTIMIZE=NONE	=(INLINE:NONE,MOTION:NONE)

/PREDEFINED_UNIT
/NOPREDEFINED_UNIT (D)

Controls the compilation of package $RUN_TIME_SYSTEM, package $TASKING_SYSTEM, and package MACHINE_CODE. You must specify this qualifier in order to be able to compile these packages.

The qualifier is not required for the compilation of any other source files. See the *XD Ada MC68020 Run-Time Reference Manual* for more information.

By default, /PREDEFINED_UNIT is omitted.

/SHOW[=option] (D)
/NOSHOW
Controls the listing file options included when a listing file is provided. You can specify one of the following options:

ALL	Provides all listing file options.
[NO]PORTABILITY	Controls whether a program portability summary is included in the listing file. By default, the XDADA command provides a portability summary (/SHOW=PORTABILITY). See Appendix E for details of what can be included in a portability summary. See Chapter 5 of Version 2.0 of *Developing Ada Programs on VMS Systems* for more information on program portability.
NONE	Provides none of the listing file options (same as /NOSHOW).

By default, the XDADA command provides a portability summary (/SHOW=PORTABILITY).

/SYNTAX_ONLY
/NOSYNTAX_ONLY (D)
Controls whether the source file is to be checked only for correct syntax. If you specify the /SYNTAX_ONLY qualifier, other compiler checks are not performed (for example, semantic analysis, type checking, and so on).

By default, the compiler performs all checks.

/WARNINGS[=(message-option[,...])]
/NOWARNINGS
Controls which categories of informational (I-level) and warning (W-level) messages are displayed and where those messages are displayed. You can specify any combination of the following message options:

WARNINGS: (*destination*[,...])
NOWARNINGS

WEAK_WARNINGS: (*destination*[,...])

NOWEAK_WARNINGS

SUPPLEMENTAL: (*destination*[,...])
NOSUPPLEMENTAL

COMPILATION_NOTES: (*destination*[,...])
NOCOMPILATION_NOTES

STATUS: (*destination*[,...])
NOSTATUS

The possible values of *destination* are ALL, NONE, or any combination of TERMINAL (terminal device), LISTING (listing file), DIAGNOSTICS (diagnostics file). The message categories are summarized as follows:

WARNINGS	W-level: Indicates a definite problem in a legal program, for example, an unknown pragma.
WEAK_WARNINGS	I-level: Indicates a potential problem in a legal program; for example, a possible CONSTRAINT_ERROR at run time. These are the only kind of I-level messages that are counted in the summary statistics at the end of a compilation.
SUPPLEMENTAL	I-level: Additional information associated with preceding E-level or W-level diagnostics.
COMPILATION_NOTES	I-level: Information about how the compiler translated a program, such as record layout, parameter-passing mechanisms, or decisions made for the pragmas INLINE, INTERFACE, or the import-subprogram pragmas.
STATUS	I-level: End of compilation statistics and other messages.

The defaults are as follows.

```
/WARNINGS=(WARN:ALL,WEAK:ALL,SUPP:ALL,COMP:NONE,STAT:LIST)
```

Note that abbreviations are valid.

If you specify only some of the message categories with the /WARNINGS qualifier, the default values for other categories are used.

Examples

1. ```
   $ XDADA MODEL_INTERFACE_,MODEL_INTERFACE,CONTROL_LOOP
   ```

   The XDADA command compiles the compilation units contained in the three files MODEL_INTERFACE_.ADA, MODEL_INTERFACE.ADA, and CONTROL_LOOP.ADA, in the order given.

2. ```
   $ XDADA/LIST/SHOW=ALL SCREEN_IO_,SCREEN_IO
   ```

   The XDADA command compiles the compilation units contained in the two files SCREEN_IO_.ADA and SCREEN_IO.ADA, in the order given. The /LIST qualifier creates the listing files SCREEN_IO_.LIS and SCREEN_IO.LIS in the current default directory. The /SHOW=ALL qualifier causes all listing file options to be provided in the listing files.

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

type FLOAT is digits 6 range -(2**128 - 2**104) .. 2**128 - 2**104;

type LONG_FLOAT is digits 15 range -(2**1024 - 2**971) .. 2**1024 - 2**971;

type LONG_LONG_FLOAT is digits 18 range -(2**16384 - 2**16320) .. 2**16384 -
2**16320;

type DURATION is delta 1.0E-4 range -131072.0000 .. 131071.9999;

...

end STANDARD;
```

Appendix F

Implementation-Dependent Characteristics

F.3 Specification of Package System

The package SYSTEM for the MC68000 configuration differs from that of the standard MC68020 as follows:

F.3.1 Package SYSTEM for the MC68000 Target

For MC68000, the system description has been redefined as follows:

```
type NAME is (MC68000);

SYSTEM_NAME    : constant NAME := MC68000;
STORAGE_UNIT   : constant := 8;
MEMORY_SIZE    : constant := 2**24;

TICK           : constant := 2#1.0#E-13;

type ADDRESS_INT is range 0 .. MEMORY_SIZE-1;
for ADDRESS_INT'SIZE use 32;
```

F.6 Interpretation of Expressions Appearing in Address Clauses

For MC68020 address clauses on variables, the address expression is interpreted as a Motorola 32-bit address. For MC68000, it is interpreted as a Motorola 24-bit address.

In XD Ada for MC68020, values of type SYSTEM.ADDRESS are interpreted as integers in the range $0 .. 2^{32} - 1$. For XD Ada MC68000, they are interpreted as integers in the range $0 .. 2^{24} - 1$

Appendix F

Implementation-Dependent Characteristics

NOTE

This appendix is not part of the standard definition of the Ada programming language.

This appendix summarizes the following implementation-dependent characteristics of XD Ada:

- Listing the XD Ada pragmas and attributes.
- Giving the specification of the package SYSTEM.
- Presenting the restrictions on representation clauses and unchecked type conversions.
- Giving the conventions for names denoting implementation-dependent components in record representation clauses.
- Giving the interpretation of expressions in address clauses.
- Presenting the implementation-dependent characteristics of the input-output packages.
- Presenting other implementation-dependent characteristics.

References all apply to sections in the *XD Ada MC68020 Supplement to the Ada Language Reference Manual*.

F.1 Implementation-Dependent Pragmas

XD Ada provides the following pragmas, which are defined elsewhere in the text. In addition, XD Ada restricts the predefined language pragmas INLINE and INTERFACE, provides pragma VOLATILE in addition to pragma SHARED, and provides pragma SUPPRESS_ALL in addition to pragma SUPPRESS. See Annex B for a descriptive pragma summary.

- CALL_SEQUENCE_FUNCTION (see Annex B)
- CALL_SEQUENCE_PROCEDURE (see Annex B)
- EXPORT_EXCEPTION (see Section 13.9a.3.2)
- EXPORT_FUNCTION (see Section 13.9a.1.2)
- EXPORT_OBJECT (see Section 13.9a.2.2)
- EXPORT_PROCEDURE (see Section 13.9a.1.2)
- IMPORT_EXCEPTION (see Section 13.9a.3.1)
- IMPORT_FUNCTION (see Section 13.9a.1.1)
- IMPORT_OBJECT (see Section 13.9a.2.1)
- IMPORT_PROCEDURE (see Section 13.9a.1.1)
- LEVEL (see Section 13.5.1)
- LINK_OPTION (see Annex B)
- SUPPRESS_ALL (see Section 11.7)
- TITLE (see Annex B)
- VOLATILE (see Section 9.11)

F.2 Implementation-Dependent Attributes

XD Ada provides the following attributes, which are defined elsewhere in the text. See Appendix A for a descriptive attribute summary.

- BIT (see Section 13.7.2)
- MACHINE_SIZE (see Section 13.7.2)
- TYPE_CLASS (see Section 13.7a.2)

F.3 Specification of the Package System

The package SYSTEM for the MC68020 is as follows:

```
package SYSTEM is

    type NAME is (MC68020);

    SYSTEM_NAME    : constant NAME := MC68020;
    STORAGE_UNIT   : constant := 8;
    MEMORY_SIZE    : constant := 2**31-1;
    MIN_INT        : constant := -(2**31);
    MAX_INT        : constant := 2**31-1;
    MAX_DIGITS     : constant := 18;
    MAX_MANTISSA   : constant := 31;
    FINE_DELTA     : constant := 2.0**(-31);
    TICK           : constant := 162.5E-6;
    subtype PRIORITY is INTEGER range 0 .. 15;

    subtype LEVEL is INTEGER range 0 .. 7;

--  Address type
--
    type ADDRESS is private;

    ADDRESS_ZERO : constant ADDRESS;
    type ADDRESS_INT is range MIN_INT .. MAX_INT;
    function TO_ADDRESS      (X : ADDRESS_INT)            return ADDRESS;
    function TO_ADDRESS      (X : {universal_integer})    return ADDRESS;
    function TO_ADDRESS_INT  (X : ADDRESS)                return ADDRESS_INT;

    function "+" (LEFT : ADDRESS;      RIGHT : ADDRESS_INT)  return ADDRESS;
    function "+" (LEFT : ADDRESS_INT;  RIGHT : ADDRESS)      return ADDRESS;
    function "-" (LEFT : ADDRESS;      RIGHT : ADDRESS)      return ADDRESS_INT;
    function "-" (LEFT : ADDRESS;      RIGHT : ADDRESS_INT)  return ADDRESS;

--  function "="  (LEFT, RIGHT : ADDRESS) return BOOLEAN;
--  function "/=" (LEFT, RIGHT : ADDRESS) return BOOLEAN;
    function "<"  (LEFT, RIGHT : ADDRESS) return BOOLEAN;
    function "<=" (LEFT, RIGHT : ADDRESS) return BOOLEAN;
    function ">"  (LEFT, RIGHT : ADDRESS) return BOOLEAN;
    function ">=" (LEFT, RIGHT : ADDRESS) return BOOLEAN;

--
--  Note that because ADDRESS is a private type
--  the functions "=" and "/=" are already available

-- Generic functions used to access memory
--
    generic
        type TARGET is private;
    function FETCH_FROM_ADDRESS (A : ADDRESS) return TARGET;

    generic
        type TARGET is private;
    procedure ASSIGN_TO_ADDRESS (A : ADDRESS; T : TARGET);
```

```
   type TYPE_CLASS is (TYPE_CLASS_ENUMERATION,
                       TYPE_CLASS_INTEGER,
                       TYPE_CLASS_FIXED_POINT,
                       TYPE_CLASS_FLOATING_POINT,
                       TYPE_CLASS_ARRAY,
                       TYPE_CLASS_RECORD,
                       TYPE_CLASS_ACCESS,
                       TYPE_CLASS_TASK,
                       TYPE_CLASS_ADDRESS);

--
-- XD Ada hardware-oriented types and functions
--
   type     BIT_ARRAY is array (INTEGER range <>) of BOOLEAN;
   pragma   PACK(BIT_ARRAY);
   subtype BIT_ARRAY_8 is BIT_ARRAY  (0 ..  7);
   subtype BIT_ARRAY_16 is BIT_ARRAY (0 .. 15);
   subtype BIT_ARRAY_32 is BIT_ARRAY (0 .. 31);
   subtype BIT_ARRAY_64 is BIT_ARRAY (0 .. 63);
   type UNSIGNED_BYTE      is range 0 .. 255;
   for  UNSIGNED_BYTE'SIZE      use 8;
   function "not" (LEFT         : UNSIGNED_BYTE)   return UNSIGNED_BYTE;
   function "and" (LEFT, RIGHT : UNSIGNED_BYTE)   return UNSIGNED_BYTE;
   function "or"  (LEFT, RIGHT : UNSIGNED_BYTE)   return UNSIGNED_BYTE;
   function "xor" (LEFT, RIGHT : UNSIGNED_BYTE)   return UNSIGNED_BYTE;

   function TO_UNSIGNED_BYTE  (X : BIT_ARRAY_8)   return UNSIGNED_BYTE;
   function TO_BIT_ARRAY_8    (X : UNSIGNED_BYTE) return BIT_ARRAY_8;

   type UNSIGNED_BYTE_ARRAY is array (INTEGER range <>) of UNSIGNED_BYTE;

   type UNSIGNED_WORD      is range 0 .. 65535;
   for  UNSIGNED_WORD'SIZE      use 16;

   function "not" (LEFT         : UNSIGNED_WORD) return UNSIGNED_WORD;
   function "and" (LEFT, RIGHT : UNSIGNED_WORD) return UNSIGNED_WORD;
   function "or"  (LEFT, RIGHT : UNSIGNED_WORD) return UNSIGNED_WORD;
   function "xor" (LEFT, RIGHT : UNSIGNED_WORD) return UNSIGNED_WORD;

   function TO_UNSIGNED_WORD  (X : BIT_ARRAY_16)    return UNSIGNED_WORD;
   function TO_BIT_ARRAY_16   (X : UNSIGNED_WORD)  return BIT_ARRAY_16;

   type UNSIGNED_WORD_ARRAY is array (INTEGER range <>) of UNSIGNED_WORD;

--
   type UNSIGNED_LONGWORD is range MIN_INT .. MAX_INT;

   for UNSIGNED_LONGWORD'SIZE use 32;

   function "not" (LEFT         : UNSIGNED_LONGWORD) return UNSIGNED_LONGWORD;
   function "and" (LEFT, RIGHT : UNSIGNED_LONGWORD) return UNSIGNED_LONGWORD;
   function "or"  (LEFT, RIGHT : UNSIGNED_LONGWORD) return UNSIGNED_LONGWORD;
   function "xor" (LEFT, RIGHT : UNSIGNED_LONGWORD) return UNSIGNED_LONGWORD;

   function TO_UNSIGNED_LONGWORD (X : BIT_ARRAY_32)    return UNSIGNED_LONGWORD;
   function TO_BIT_ARRAY_32          (X : UNSIGNED_WORD)  return BIT_ARRAY_32;

   type UNSIGNED_LONGWORD_ARRAY is array (INTEGER range <>) of UNSIGNED_LONGWORD;
```

```
--   Conventional names for static subtypes of type UNSIGNED_LONGWORD
--
     subtype UNSIGNED_1  is UNSIGNED_LONGWORD range 0 .. 2** 1-1;
     subtype UNSIGNED_2  is UNSIGNED_LONGWORD range 0 .. 2** 2-1;
     subtype UNSIGNED_3  is UNSIGNED_LONGWORD range 0 .. 2** 3-1;
     subtype UNSIGNED_4  is UNSIGNED_LONGWORD range 0 .. 2** 4-1;
     subtype UNSIGNED_5  is UNSIGNED_LONGWORD range 0 .. 2** 5-1;
     subtype UNSIGNED_6  is UNSIGNED_LONGWORD range 0 .. 2** 6-1;
     subtype UNSIGNED_7  is UNSIGNED_LONGWORD range 0 .. 2** 7-1;
     subtype UNSIGNED_8  is UNSIGNED_LONGWORD range 0 .. 2** 8-1;
     subtype UNSIGNED_9  is UNSIGNED_LONGWORD range 0 .. 2** 9-1;
     subtype UNSIGNED_10 is UNSIGNED_LONGWORD range 0 .. 2**10-1;
     subtype UNSIGNED_11 is UNSIGNED_LONGWORD range 0 .. 2**11-1;
     subtype UNSIGNED_12 is UNSIGNED_LONGWORD range 0 .. 2**12-1;
     subtype UNSIGNED_13 is UNSIGNED_LONGWORD range 0 .. 2**13-1;
     subtype UNSIGNED_14 is UNSIGNED_LONGWORD range 0 .. 2**14-1;
     subtype UNSIGNED_15 is UNSIGNED_LONGWORD range 0 .. 2**15-1;
     subtype UNSIGNED_16 is UNSIGNED_LONGWORD range 0 .. 2**16-1;
     subtype UNSIGNED_17 is UNSIGNED_LONGWORD range 0 .. 2**17-1;
     subtype UNSIGNED_18 is UNSIGNED_LONGWORD range 0 .. 2**18-1;
     subtype UNSIGNED_19 is UNSIGNED_LONGWORD range 0 .. 2**19-1;
     subtype UNSIGNED_20 is UNSIGNED_LONGWORD range 0 .. 2**20-1;
     subtype UNSIGNED_21 is UNSIGNED_LONGWORD range 0 .. 2**21-1;
     subtype UNSIGNED_22 is UNSIGNED_LONGWORD range 0 .. 2**22-1;
     subtype UNSIGNED_23 is UNSIGNED_LONGWORD range 0 .. 2**23-1;
     subtype UNSIGNED_24 is UNSIGNED_LONGWORD range 0 .. 2**24-1;
     subtype UNSIGNED_25 is UNSIGNED_LONGWORD range 0 .. 2**25-1;
     subtype UNSIGNED_26 is UNSIGNED_LONGWORD range 0 .. 2**26-1;
     subtype UNSIGNED_27 is UNSIGNED_LONGWORD range 0 .. 2**27-1;
     subtype UNSIGNED_28 is UNSIGNED_LONGWORD range 0 .. 2**28-1;
     subtype UNSIGNED_29 is UNSIGNED_LONGWORD range 0 .. 2**29-1;
     subtype UNSIGNED_30 is UNSIGNED_LONGWORD range 0 .. 2**30-1;
     subtype UNSIGNED_31 is UNSIGNED_LONGWORD range 0 .. 2**31-1;

private
     -- Not shown
end SYSTEM;
```

F.4 Restrictions on Representation Clauses

The representation clauses allowed in XD Ada are length, enumeration, record representation, and address clauses.

In XD Ada, a representation clause for a generic formal type or a type that depends on a generic formal type is not allowed. In addition, a representation clause for a composite type that has a component or subcomponent of a generic formal type or a type derived from a generic formal type is not allowed.

Restrictions on length clauses are specified in Section 13.2; restrictions on enumeration representation clauses are specified in Section 13.3; and restrictions on record representation clauses are specified in Section 13.4.

F.5 Conventions for Implementation-Generated Names Denoting Implementation-Dependent Components in Record Representation Clauses

XD Ada does not allocate implementation-dependent components in records.

F.6 Interpretation of Expressions Appearing in Address Clauses

Expressions appearing in address clauses must be of the type ADDRESS defined in package SYSTEM (see Section 13.7a.1 and Section F.3).

XD Ada allows address clauses for variables (see Section 13.5). For address clauses on variables, the address expression is interpreted as a Motorola full 32-bit address.

XD Ada supports address clauses on task entries to allow interrupts to cause a reschedule directly. For address clauses on task entries, the address expression is interpreted as a Motorola exception vector offset.

In XD Ada for MC68020, values of type SYSTEM.ADDRESS are interpreted as integers in the range $0 .. 2^{32} - 1$. As SYSTEM.ADDRESS is a private type, the only operations allowed on objects of this type are those given in package SYSTEM.

F.7 Restrictions on Unchecked Type Conversions

XD Ada supports the generic function UNCHECKED_CONVERSION with the restrictions given in Section 10.3.2.

F.8 Implementation-Dependent Characteristics of Input-Output Packages

The packages SEQUENTIAL_IO and DIRECT_IO are implemented as null packages that conform to the specification given in the *Reference Manual for the Ada Programming Language*. The packages raise the exceptions specified in Chapter 14 of the *Reference Manual for the Ada Programming Language*. The three possible exceptions that are raised by these packages are given here, in the order in which they are raised.

Exception	When Raised
STATUS_ERROR	Raised by an attempt to operate upon or close a file that is not open (no files can be opened).
NAME_ERROR	Raised if a file name is given with a call of CREATE or OPEN.
USE_ERROR	Raised if exception STATUS_ERROR is not raised.

MODE_ERROR cannot be raised since no file can be opened (therefore it cannot have a current mode).

The predefined package LOW_LEVEL_IO is not provided.

F.8.1 The Package TEXT_IO

The package TEXT_IO conforms to the specification given in the *Reference Manual for the Ada Programming Language*. String input-output is implemented as defined. File input-output is supported to STANDARD_INPUT and STANDARD_OUTPUT only. The possible exceptions that are raised by package TEXT_IO are as follows:

Exception	When Raised
STATUS_ERROR	Raised by an attempt to operate upon or close a file that is not open (no files can be opened).
NAME_ERROR	Raised if a file name is given with a call of CREATE or OPEN.
MODE_ERROR	Raised by an attempt to read from, or test for the end of, STANDARD_OUTPUT, or to write to STANDARD_INPUT.
END_ERROR	Raised by an attempt to read past the end of STANDARD_INPUT.
USE_ERROR	Raised when an unsupported operation is attempted, that would otherwise be legal.

The type COUNT is defined as follows:

```
type COUNT is range 0 .. INTEGER'LAST;
```

The subtype FIELD is defined as follows:

```
type FIELD is INTEGER range 0 .. 132;
```

F.8.2 The Package IO_EXCEPTIONS

The specification of the package IO_EXCEPTIONS is the same as that given in the *Reference Manual for the Ada Programming Language*.

F.9 Other Implementation Characteristics

Implementation characteristics associated with the definition of a main program, various numeric ranges, and implementation limits are summarized in the following sections.

F.9.1 Definition of a Main Program

Any library procedure can be used as a main program provided that it has no formal parameters.

F.9.2 Values of Integer Attributes

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

SHORT_SHORT_INTEGER	-2^7 .. 2^7 -1	(-128 .. 127)
SHORT_INTEGER	-2^{15} .. 2^{15} -1	(-32768 .. 32767)
INTEGER	-2^{31} .. 2^{31} -1	(-2147483648 .. 2147483647)

For the package TEXT_IO, the range of values for types COUNT and FIELD are as follows:

COUNT	0 .. 2^{31} -1	(0 .. 2147483647)
FIELD	0 .. 132	

F.9.3 Values of Floating-Point Attributes

Floating-point types are described in Section 3.5.7. The representation attributes of floating-point types are summarized in the following table:

	FLOAT	LONG_FLOAT	LONG_LONG_FLOAT
DIGITS	6	15	18
SIZE	32	64	96
MANTISSA	21	51	61
EMAX	84	204	244
EPSILON	2^{-20}	2^{-50}	2^{-60}
SMALL	2^{-85}	2^{-205}	2^{-245}
LARGE	$2^{84}-2^{63}$	$2^{204}-2^{153}$	$2^{244}-2^{183}$
SAFE_EMAX	125	1021	16382
SAFE_SMALL	2^{-126}	2^{-1022}	2^{-16383}
SAFE_LARGE	$2^{125}-2^{104}$	$2^{1021}-2^{970}$	$2^{16382}-2^{16321}$
FIRST	$-(2^{128}-2^{104})$	$-(2^{1024}-2^{971})$	$-(2^{16384}-2^{16320})$
LAST	$2^{128}-2^{104}$	$2^{1024}-2^{971}$	$2^{16384}-2^{16320}$
MACHINE_RADIX	2	2	2
MACHINE_MANTISSA	24	53	64
MACHINE_EMAX	128	1024	16384
MACHINE_EMIN	-125	-1021	-16382
MACHINE_ROUNDS	FALSE	FALSE	FALSE
MACHINE_OVERFLOWS	FALSE	FALSE	FALSE

F.9.4 Attributes of Type DURATION

The values of the significant attributes of type DURATION are as follows:

DURATION'DELTA	1.E-4	(10^{-4})
DURATION'SMALL	2#1.0#E-14	(2^{-14})
DURATION'FIRST	-131072.0000	(-2^{17})
DURATION'LAST	131071.9999	(2^{17}-'DELTA)

F.9.5 Implementation Limits

Limit	Description
255	Maximum identifier length (number of characters)
255	Maximum number of characters in a source line
2^{10}	Maximum number of library units and subunits in a compilation closure[1]
2^{12}	Maximum number of library units and subunits in an execution closure[2]
2^{16} -1	Maximum number of enumeration literals in an enumeration type definition
2^{16} -1	Maximum number of lines in a source file
2^{31} -1	Maximum number of bits in any object
2^{16} -1	Maximum number of exceptions

[1] The compilation closure of a given unit is the total set of units that the given unit depends on, directly and indirectly.

[2] The execution closure of a given unit is the compilation closure plus all associated secondary units.

Appendix F

Implementation-Dependent Characteristics

This appendix describes Version 1.2 additions to the range of implementation-dependent pragmas, and enhancements to the functionality of Package TEXT_IO. It supplements information supplied in the Version 1.0 version of this appendix.

F.1 Implementation-Dependent Pragmas

XD Ada MC68020 Version 1.2 supplies three new pragmas, DIRECT_INTERRUPT_ENTRY, IDENT and TIME_SLICE. In the following full list of supported pragmas, references refer to sections in the *XD Ada MC68020 Supplement to the Ada Language Reference Manual*, unless updated by sections supplied in this manual.

- CALL_SEQUENCE_FUNCTION (see Annex B)
- CALL_SEQUENCE_PROCEDURE (see Annex B)
- DIRECT_INTERRUPT_ENTRY (see Section 13.5.1)
- EXPORT_EXCEPTION (see Section 13.9a.3.2)
- EXPORT_FUNCTION (see Section 13.9a.1.2)
- EXPORT_OBJECT (see Section 13.9a.2.2)
- EXPORT_PROCEDURE (see Section 13.9a.1.2)
- IDENT (see Annex B)
- IMPORT_EXCEPTION (see Section 13.9a.3.1)
- IMPORT_FUNCTION (see Section 13.9a.1.1)

- IMPORT_OBJECT (see Section 13.9a.2.1)
- IMPORT_PROCEDURE (see Section 13.9a.1.1)
- LEVEL (see Section 13.5.1)
- LINK_OPTION (see Annex B)
- SUPPRESS_ALL (see Section 11.7)
- TITLE (see Annex B)
- TIME_SLICE (see Section 9.8a)
- VOLATILE (see Section 9.11)

F.8.1 The Package TEXT_IO

The package TEXT_IO conforms to the specification given in the *Reference Manual for the Ada Programming Language*. Package TEXT_IO, as supplied by XD Ada MC68020 Version 1.2, has changed as follows:

- The Run-Time System now supports asynchronous input-output operations, where a TEXT_IO operation will cause only the task that performs the operation to be suspended awaiting its completion, rather than all the tasks in the program. You enable this facility by compiling the file XDADA$TARGET_SOURCE:ASYNC_TERMINAL_IO.ADA into your program library, as described in the *XD Ada MC68020 Version 1.2 New Features Manual*. You disable asynchronous TEXT_IO operations by compiling the file XDADA$TARGET_SOURCE:TERMINAL_IO.ADA into your program library.
- Support is provided for target input-output to be directed to logical input and output streams on the host. This facility is available for both the XDDEBUG and XDRUN commands.

  Input and output are buffered. For input, all characters up to an end of line or end of page are made available to the target program before further characters are read. For output, the buffer is flushed following an end of line or end of page. See the *XD Ada MC68020 Version 1.2 New Features Manual* for details of the TEXT_IO data objects that control this behavior.

  Note that if XDADA$INPUT and XDADA$OUTPUT are defined but opening the file gives an error, for example the file does not exist, the filename is invalid or no read/write permission is assigned, the file is treated as empty.

types can be packed as components of composite types, as well as information on how these types are packed.

A record component that begins a variant is always allocated at the next byte boundary; a variant that begins on other than a byte boundary can be obtained only with a record representation clause.

XD Ada provides no additional representation pragmas.

The following information supplements paragraph 14:

XD Ada does not allow a representation clause for a type that depends on a generic formal type. A type depends on a generic formal type if it has a subcomponent of a generic formal type or a subcomponent that depends on a generic formal type, or if it is derived from a generic formal type or a type that depends on a generic formal type.

13.2 Length Clauses

The following information supplements paragraph 6:

In XD Ada, for a discrete type, the given size must not exceed 32 (bits). The given size becomes the default allocation for all objects and components (in arrays and records) of that type. However, sizes of objects may be increased by the compiler for optimization purposes.

For integer and enumeration types, the given size affects the internal representation as follows: for integer types, high order bits are sign-extended; for enumeration types, the high order bits may be either zero- or sign-extended depending upon the base representation that is selected. For all other types, the given size must equal the size that would apply in the absence of a size specification.

The following information supplements paragraph 8:

The specification of a collection size is interpreted as follows. If the value of the expression is greater than or equal to zero, the specified size (representing the number of bytes in the collection) is rounded up to the longword boundary nearest (4 bytes), and is then used as the initial size of the collection; the collection is not extended should that initial allocation be exhausted. In the absence of a T'STORAGE_SIZE, no storage is initially allocated for the collection; storage is allocated

There are two ways an interrupt entry can be handled, according to whether or not the task has a pragma LEVEL. The *XD Ada MC68020 Run-Time Reference Manual* gives examples of interrupt handlers.

Tasks with interrupt entries but no pragma LEVEL run at interrupt level 0 only while accepting an interrupt in a rendezvous. Other interrupts of the same level or lower levels are inhibited while in the handler. It is, however, possible to lose interrupts with this method.

Tasks with interrupt entries and a pragma LEVEL always run at interrupt level, whether inside or outside a rendezvous. This enables the user to avoid losing interrupts.

An interrupt entry to a task with the pragma LEVEL behaves like an ordinary entry call. An interrupt entry to a task with no pragma LEVEL behaves like a conditional entry call. If there is an accept statement waiting for the interrupt, the body of the accept statement is executed immediately. When the body is complete, the task is inserted in the ready queue and the interrupt completed by a return-from-interrupt instruction. The accept statement can call subprograms and make entry calls, but must not suspend the task before the interrupt is dismissed, otherwise the program repeatedly services the interrupt unsuccessfully.

Writing interrupt handlers in XD Ada requires detailed knowledge of the behavior of the target computer's interrupt system. It is not possible simply to place a use clause on an entry to achieve the desired effect.

Normal Ada interrupt entries cause a tasking reschedule each time an interrupt occurs. This inevitably incurs a performance overhead, and may mean that interrupts are not serviced quickly enough. In order to avoid this problem, XD Ada supplies pragma DIRECT_INTERRUPT_ENTRY, which causes the interrupt entry to be connected directly to the required interrupt vector. This run-time efficiency greatly improves response times. The form of this pragma is as follows:

```
pragma DIRECT_INTERRUPT_ENTRY(interrupt_entry);
```

Pragma DIRECT_INTERRUPT_ENTRY may be used where the program adheres to one of two supported code models. In fact, most applications will naturally adhere to one or other of the models, so the practical restrictions from this requirement are minimal. The use of pragma DIRECT_INTERRUPT_ENTRY must meet certain semantic conditions. These, along with the checks carried out by the compiler and run-time system, are described in full in the *XD Ada MC68020 Run-Time Reference Manual* part of this manual.

Chapter 13

Representation Clauses and Implementation-Dependent Features

The text in this chapter lists the differences between XD Ada MC68000 and XD Ada MC68020, as described in the *XD Ada MC68020 Supplement to the Ada Language Reference Manual.*

13.1 Representation Clauses

The following information replaces the MC68020 supplement to paragraph 13:

Pragma PACK is implemented in XD Ada. As the behaviour of pragma PACK is implementation dependent, users are advised to use representation clauses to ensure a particular representation across targets.

In XD Ada MC68000, all array and record components are aligned according to their types by default; the effect of pragma PACK on a record or array is to cause those components which are packable to be allocated in adjacent bits without regard to byte bondaries. Whether any particular component is packable depends on the rules for its type; the *XD Ada MC68020 Run-Time Reference Manual* gives information on which types can be packed as components of composite types, as well as information on how these types are packed.

Chapter 13

Representation Clauses and Implementation-Dependent Features

Supplementary XD Ada information is provided for Sections 13.1, 13.2, 13.3, 13.4, 13.5, 13.5.1, 13.7, 13.7.1, 13.7.2, 13.7.3, 13.8, 13.9, 13.10.1 and 13.10.2. Two additional sections, Section 13.7a and Section 13.9a, provide XD Ada information on the package SYSTEM and on the XD Ada import and export pragmas.

13.1 Representation Clauses

The following information supplements paragraphs 4 and 8:

In XD Ada, an address clause can only apply to a variable or a single entry; an address clause cannot apply to a constant, subprogram, package, or task unit. See Section 13.5 for further explanation.

The following information supplements paragraph 13:

Pragma PACK is implemented in XD Ada. As the behavior of pragma PACK is implementation dependent, users are advised to use representation clauses to ensure a particular representation across targets.

In XD Ada, all array and record components are aligned on byte boundaries by default; the effect of pragma PACK on a record or array is to cause those components that are packable to be allocated in adjacent bits without regard to byte boundaries. Whether any particular component is packable depends on the rules for its type; the *XD Ada MC68020 Run-Time Reference Manual* gives information on which

from the heap as needed, until all heap memory is exhausted. If the value is less than zero, the exception CONSTRAINT_ERROR is raised.

The following information supplements paragraph 10:

A task storage specification overrides the default task storage size. The specification is interpreted as follows. If the value of the expression is greater than zero, the specified size is rounded up to the nearest longword boundary (4 bytes), and this determines the number of storage units (bytes) to be allocated for an activation of the task of the given type. In the absence of a T'STORAGE_SIZE, a default allocation is used. If the value is less than zero, the exception CONSTRAINT_ERROR is raised.

The following information supplements paragraphs 8 and 10:

NOTE

> The *XD Ada MC68020 Run-Time Reference Manual* discusses task and access type storage and storage allocation in more detail.

The following information supplements paragraph 12:

In XD Ada, arbitrary values of *small* are accepted. The default value of *small* is the largest power of two that is not greater than the given delta (see Section 3.5.9).

If *small* is specified (see Section 3.5.9 (LRM)), the specified value must not exceed the default. For example:

```
for MY_FIXED'SMALL use 0.001;
```

This example is a legal specification for the declaration of MY_FIXED because the value specified for *small* (0.001) is less than the delta (0.1) and that also satisfies the specified range (0.0..1.0).

13.3 Enumeration Representation Clauses

The following information supplements paragraph 4:

In XD Ada, the only specific restriction on enumeration representation clauses is that each expression for an integer code must have a value in the range MIN_INT .. MAX_INT.

13.4 Record Representation Clauses

The following information supplements paragraph 4:

For statically allocated objects and for objects allocated from a collection in XD Ada, the simple expression in an alignment clause must be a power of two. The upper limit is 2^{31}. The alignment then occurs at a location that is a number of bytes times the value of the simple expression: a value of 2 causes word alignment, a value of 4 causes longword alignment, and so on.

Further restrictions apply for objects declared within a subprogram, where XD Ada restricts the alignment to mod 1. In other words, stack-allocated objects can only be byte aligned.

Bit-alignable representation clauses are provided for discrete types, arrays of discrete types, and record types.

See the *XD Ada MC68020 Run-Time Reference Manual* for information on how objects are allocated.

The following information supplements paragraph 5:

A component clause specifies the *storage place* of a component relative to the start of the record. In XD Ada for MC68020 targets, the size of a storage unit (SYSTEM.STORAGE_UNIT) is eight bits (one byte). If the number of bits specified by the range is sufficient for the component subtype, the requested size and placement of the field is observed (and overlaps storage boundaries if necessary); otherwise, the specification is illegal. For a component of a discrete type, the number of bits must not exceed 32; for a component of any other type, the size must not exceed the actual size of the component. See the *XD Ada MC68020 Run-Time Reference Manual* for information about determining the number of bits that are sufficient for any given subtype.

Component values in XD Ada are biased when a component clause requires a very small component storage space; each value stored is the unsigned quantity formed by subtracting COMPONENT_SUBTYPE'FIRST from the original value. See the *XD Ada MC68020 Run-Time Reference Manual* for more detailed information.

Component clauses in XD Ada are restricted as follows. Any component that is not packable must be allocated on a byte boundary. Components that are packable can be allocated without restriction. See the *XD Ada MC68020 Run-Time Reference Manual* for a definition and description of packable components.

The following information supplements paragraph 6:

Components named in a component clause are allocated first; then, unnamed components are allocated in the order in which they are written in the record type declaration. Variants can be overlapped. If pragma PACK is specified, packed allocation rules (see Section 13.1) are used; otherwise, unpacked allocation is used.

The following information supplements paragraph 8:

XD Ada generates no implementation-dependent components or names.

The following information supplements the Notes section:

The example of record representation and address clauses in the *Reference Manual for the Ada Programming Language* is not relevant for XD Ada as it assumes that type ADDRESS is represented in 24 bits, whereas in XD Ada type ADDRESS is represented in 32 bits. The following example is appropriate to XD Ada:

Example:

```
type CONDITION_CODE is (X,N,Z,V,C);

type CONDITION_CODES is array (CONDITION_CODE) of BOOLEAN;
pragma PACK (CONDITION_CODES);

type PROGRAM_STATUS_WORD is
   record
      TRACE_ENABLE     : INTEGER range 0 .. 3;
      SUPERVISOR_STATE : BOOLEAN;
      INTERRUPT_STATE  : BOOLEAN;
      INTERRUPT_MASK   : INTEGER range 0 .. 7;
      CC               : CONDITION_CODES;
   end record;

for PROGRAM_STATUS_WORD use
   record at mod 1;
      TRACE_ENABLE     at 0 range 0  .. 1;
      SUPERVISOR_STATE at 0 range 2  .. 2;
      INTERRUPT_STATE  at 0 range 3  .. 3;
      INTERRUPT_MASK   at 0 range 5  .. 7;
      CC               at 0 range 11 .. 15;
   end record;

for PROGRAM_STATUS_WORD'SIZE use 2 * SYSTEM.STORAGE_UNIT;
```

Note on the example:

The record representation clause defines the record layout. The length clause guarantees that exactly two storage units are used.

Chapter 13

Representation Clauses and Implementation-Dependent Features

This chapter describes XD Ada MC68020 Version 1.2 interrupt handling. In particular, it describes the handling of direct interrupt entries, and use of pragma DIRECT_INTERRUPT_ENTRY.

13.5.1 Interrupts

The following information supplements all of this section:

Unlike VAX Ada, XD Ada supports interrupts. The address in the use clause is the address of the interrupt. The address is interpreted as an offset in bytes from the vector base register. For details of MC68020 exception vector assignments, see Table 6-2 of the *MC68020 32-Bit Microprocessor User's Manual*. Note that when assigning vectors, the offset is a multiple of four of the vector number. In this way, vector number 64 (decimal) would have the hexadecimal offset 100.

In addition to support for normal Ada interrupt entries, XD Ada provides the additional pragmas LEVEL and DIRECT_INTERRUPT_ENTRY. Pragma LEVEL is given for a task type, or single task of anonymous type, and gives the level for its interrupts. Pragma DIRECT_INTERRUPT_ENTRY is used to connect an interrupt entry directly to the required interrupt vector, and is described below.

Component Specification Example:

```
subtype S is INTEGER range 10 .. 13;

 type REC is
     record
        X : S;
        Y : S;
     end record;

 for REC use
     record
        X at 0 range 0 .. 3;      -- legal because 4 bits
                                  -- are sufficient
        Y at 0 range 4 .. 4;      -- illegal because 1 bit is
                                  -- not enough to represent
                                  -- an integer of subtype S
     end record;
```

Notes on the example:

The subtype declaration in this example implies an integer with a minimum size of four bits. However, the components X and Y of subtype S are biased and can be stored in only two bits. The component clause for X is legal because it requires at least the minimum number of bits required for the integer subtype; the component clause for Y is illegal because it does not allow enough bits to represent the integer subtype.

13.5 Address Clauses

The following information supplements paragraph 7:

Like VAX Ada, XD Ada supports address clauses.

In XD Ada, the simple name must be the name of a variable. XD Ada does not allow address clauses that name constants; or subprogram, package, or task units.

An intermediate pointer is created only if the resulting address is not a compile-time constant.

The placement of an address clause in XD Ada must follow the rules given in Section 13.1. In other words, the clause and the variable declaration must both occur immediately within the same declarative part or package specification, and the declaration must occur before the clause. The restrictions for forcing occurrences also apply; with respect to address clauses, any occurrence of the variable name after its declaration is a forcing occurrence.

Address clauses are not allowed in combination with any of the XD Ada pragmas for importing or exporting objects. If used in such cases, the pragma involved is ignored.

The following information supplements the Notes section:

Also, if an address clause is specified for an object of a type that has been declared with an alignment clause, the alignment required for the address is checked against the alignment given for the record type. If the two are incompatible, the exception PROGRAM_ERROR is raised.

The same check applies to a type that contains a component of a type that has been declared with an alignment clause (the alignment of the component forces the alignment of the containing type).

13.5.1 Interrupts

The following information supplements all of this section:

Unlike VAX Ada, XD Ada supports interrupts. The address in the **use** clause is the address of the interrupt. The address is interpreted as an offset from the vector base register.

XD Ada provides the additional pragma LEVEL. This pragma is given for a task type, or single task of anonymous type, and gives the level for its interrupts.

There are two ways an interrupt entry can be handled, according to whether or not the task has a pragma LEVEL. The *XD Ada MC68020 Run-Time Reference Manual* gives examples of interrupt handlers.

Tasks with interrupt entries but no pragma run at interrupt level whilst accepting an interrupt in a rendezvous. Other interrupts of the same level or lower levels are inhibited. It is possible to lose interrupts with this method.

Tasks with interrupt entries and a pragma LEVEL always run at interrupt level, whether inside or outside a rendezvous. This enables the user to avoid losing interrupts.

An interrupt entry to a task with the pragma behaves like an ordinary entry call. An interrupt entry to a task with no pragma behaves like a conditional entry call. If there is an accept statement waiting for the interrupt, the body of the accept statement is executed immediately. When the body is complete, the task is inserted in the ready queue and the interrupt completed by a return-from-interrupt instruction. The

A record component that begins a variant is always allocated on the next byte, word or long-word according to its type: a variant that begins on other than its default can be obtained only with a record representation clause.

XD Ada provides no additional representation pragmas.

13.7 The Package System

The following information replaces the MC68020 supplement to paragraph 5:

In XD Ada MC68000, the enumeration literal for SYSTEM_NAME is MC68000.

The following information replaces the MC68020 supplement to paragraph 9:

In XD Ada MC68000, the number given for MEMORY_SIZE must be 224. Like VAX Ada, XD Ada MC68000 does not provide support for checking or ensuring that the given size is not exceeded.**

13.7a XD Ada Additions to the Package SYSTEM

13.7a.1 Properties of the Type ADDRESS

In XD Ada MC68000, ADDRESS is a private type redefined as follows:

```
type ADDRESS_INT is range 0 .. MEMORY_SIZE - 1;
for ADDRESS_INT'SIZE use 32;
```

13.7.1 System-Dependent Named Numbers

In XD Ada MC68000, the value for the system-dependent named number, TICK, is redefined as follows:

Attribute	MC68000
TICK	1.0×2^{-13}

accept statement can make excursions into other routines, and can even make entry calls, but must not suspend the task before the interrupt is dismissed, otherwise the program repeatedly services the interrupt unsuccessfully.

Writing interrupt handlers in XD Ada requires detailed knowledge of the behavior of the target computer's interrupt system. It is not possible simply to place a **use** clause on an entry to achieve the desired effect.

13.7 The Package System

The following information supplements paragraph 1:

XD Ada additions to the package SYSTEM are described in Section 13.7a.

The following information supplements paragraph 5:

In XD Ada, the enumeration literal for SYSTEM_NAME is MC68020.

The following information supplements paragraph 7:

In XD Ada, the value given for STORAGE_UNIT must be 8 (bits).

The following information supplements paragraph 9:

In XD Ada, the number given for MEMORY_SIZE must be 2**31-1. Like VAX Ada, XD Ada does not provide support for checking or ensuring that the given size is not exceeded.

The following information supplements paragraph 11:

As with VAX Ada, XD Ada imposes no further limitations on these pragmas. To reduce the amount of recompilation required, XD Ada identifies those units that have a real dependence on the values affected by these pragmas; only such units must be recompiled. In particular, predefined XD Ada packages do not depend on the values affected by these pragmas, and none require recompilation if these pragmas are used.

13.7a XD Ada Additions to the Package SYSTEM

In addition to the language-required declarations in package SYSTEM, XD Ada declares the operations, constants, types, and subtypes described in the following sections.

13.7a.1 Properties of the Type ADDRESS

In XD Ada, ADDRESS is a private type for which the following operations are declared:

```
--  Address type
--
   type ADDRESS is private;

   ADDRESS_ZERO : constant ADDRESS;

   type ADDRESS_INT is range MIN_INT .. MAX_INT;

   function TO_ADDRESS      (X : ADDRESS_INT)             return ADDRESS;
   function TO_ADDRESS      (X : {universal_integer})     return ADDRESS;
   function TO_ADDRESS_INT  (X : ADDRESS)                 return ADDRESS_INT;

   function "+" (LEFT : ADDRESS;      RIGHT : ADDRESS_INT) return ADDRESS;
   function "+" (LEFT : ADDRESS_INT;  RIGHT : ADDRESS)     return ADDRESS;
   function "-" (LEFT : ADDRESS;      RIGHT : ADDRESS)     return ADDRESS_INT;
   function "-" (LEFT : ADDRESS;      RIGHT : ADDRESS_INT) return ADDRESS;

--  function "="  (LEFT, RIGHT : ADDRESS) return BOOLEAN;
--  function "/=" (LEFT, RIGHT : ADDRESS) return BOOLEAN;
    function "<"  (LEFT, RIGHT : ADDRESS) return BOOLEAN;
    function "<=" (LEFT, RIGHT : ADDRESS) return BOOLEAN;
    function ">"  (LEFT, RIGHT : ADDRESS) return BOOLEAN;
    function ">=" (LEFT, RIGHT : ADDRESS) return BOOLEAN;
--
--  Note that because ADDRESS is a private type
--  the functions "=" and "/=" are already available

-- Generic functions used to access memory
--
    generic
        type TARGET is private;
    function FETCH_FROM_ADDRESS (A : ADDRESS) return TARGET;

    generic
        type TARGET is private;
    procedure ASSIGN_TO_ADDRESS (A : ADDRESS; T : TARGET);
```

The addition, subtraction, and relational functions provide arithmetic and comparative operations for addresses. The generic subprograms FETCH_FROM_ADDRESS and ASSIGN_TO_ADDRESS provide operations for reading from or writing to a given address interpreted as having any desired type. ADDRESS_ZERO is a deferred constant whose value corresponds to the first (machine) address.

In an instantiation of FETCH_FROM_ADDRESS or ASSIGN_TO_ADDRESS, the actual subtype corresponding to the formal type T must not be an unconstrained array type or an unconstrained type with discriminants. If the actual subtype is a type with discriminants, the value fetched by a call of a function resulting from an instantiation of FETCH_FROM_ADDRESS is checked to ensure that the discriminants satisfy the constraints of the actual subtype. In any other case, no check is made.

Example:

```
X : INTEGER;
A : SYSTEM.ADDRESS := X'ADDRESS;     -- legal

function FETCH  is new FETCH_FROM_ADDRESS(INTEGER);
procedure ASSIGN is new ASSIGN_TO_ADDRESS(INTEGER);

X := FETCH(A);                       -- like "X := A.all;"
ASSIGN(A,X);                         -- like "A.all := X;"
```

13.7a.2 Type Class Enumeration Type

XD Ada declares the following enumeration type for identifying the various Ada type classes:

```
type TYPE_CLASS is (TYPE_CLASS_ENUMERATION,
                    TYPE_CLASS_INTEGER,
                    TYPE_CLASS_FIXED_POINT,
                    TYPE_CLASS_FLOATING_POINT,
                    TYPE_CLASS_ARRAY,
                    TYPE_CLASS_RECORD,
                    TYPE_CLASS_ACCESS,
                    TYPE_CLASS_TASK,
                    TYPE_CLASS_ADDRESS);
```

In addition to the usual operations for discrete types (see Section 3.5.5), XD Ada provides the attribute TYPE_CLASS.

For every type or subtype T:

T'TYPE_CLASS Yields the value of the type class for the full type of T. If T is a generic formal type, then the value is that for the corresponding actual subtype. The value of this attribute is of the type TYPE_CLASS.

This attribute is only allowed if its unit names the predefined package SYSTEM in a **with** clause.

Examples:

Given

```
type MY_INT is range 1..10;
type NEW_INT is new STRING;
package PACK is
   type PRIV is private;
private
   type PRIV is new FLOAT;
end PACK;
```

then

```
-- MY_INT'TYPE_CLASS      equals   TYPE_CLASS_INTEGER
-- NEW_INT'TYPE_CLASS     equals   TYPE_CLASS_ARRAY
-- PRIV'TYPE_CLASS        equals   TYPE_CLASS_FLOATING_POINT
```

13.7a.3 Hardware-Oriented Types and Functions

XD Ada declares the following types, subtypes, and functions for convenience in working with MC68020 hardware-oriented storage:

```
--
-- XD Ada hardware-oriented types and functions
--
   type     BIT_ARRAY is array (INTEGER range <>) of BOOLEAN;
   pragma   PACK(BIT_ARRAY);
   subtype BIT_ARRAY_8 is BIT_ARRAY  (0 ..  7);
   subtype BIT_ARRAY_16 is BIT_ARRAY (0 .. 15);
   subtype BIT_ARRAY_32 is BIT_ARRAY (0 .. 31);
   subtype BIT_ARRAY_64 is BIT_ARRAY (0 .. 63);
   type UNSIGNED_BYTE  is range 0 .. 255;
   for  UNSIGNED_BYTE'SIZE  use 8;
   function "not" (LEFT         : UNSIGNED_BYTE) return UNSIGNED_BYTE;
   function "and" (LEFT, RIGHT : UNSIGNED_BYTE) return UNSIGNED_BYTE;
   function "or"  (LEFT, RIGHT : UNSIGNED_BYTE) return UNSIGNED_BYTE;
   function "xor" (LEFT, RIGHT : UNSIGNED_BYTE) return UNSIGNED_BYTE;

   function TO_UNSIGNED_BYTE (X : BIT_ARRAY_8) return UNSIGNED_BYTE;
   function TO_BIT_ARRAY_8 (X : UNSIGNED_BYTE) return BIT_ARRAY_8;

   type UNSIGNED_BYTE_ARRAY is array (INTEGER range <>) of UNSIGNED_BYTE;
   type UNSIGNED_WORD  is range 0 .. 65535;
   for  UNSIGNED_WORD'SIZE  use 16;
   function "not" (LEFT         : UNSIGNED_WORD) return UNSIGNED_WORD;
   function "and" (LEFT, RIGHT : UNSIGNED_WORD) return UNSIGNED_WORD;
   function "or"  (LEFT, RIGHT : UNSIGNED_WORD) return UNSIGNED_WORD;
   function "xor" (LEFT, RIGHT : UNSIGNED_WORD) return UNSIGNED_WORD;

   function TO_UNSIGNED_WORD (X : BIT_ARRAY_16) return UNSIGNED_WORD;
   function TO_BIT_ARRAY_16 (X : UNSIGNED_WORD) return BIT_ARRAY_16;
```

```
type UNSIGNED_WORD_ARRAY is array (INTEGER range <>) of UNSIGNED_WORD;
--
type UNSIGNED_LONGWORD is range MIN_INT .. MAX_INT;

for UNSIGNED_LONGWORD'SIZE use 32;

function "not" (LEFT        : UNSIGNED_LONGWORD) return UNSIGNED_LONGWORD;
function "and" (LEFT, RIGHT : UNSIGNED_LONGWORD) return UNSIGNED_LONGWORD;
function "or"  (LEFT, RIGHT : UNSIGNED_LONGWORD) return UNSIGNED_LONGWORD;
function "xor" (LEFT, RIGHT : UNSIGNED_LONGWORD) return UNSIGNED_LONGWORD;

function TO_UNSIGNED_LONGWORD (X : BIT_ARRAY_32) return UNSIGNED_LONGWORD;
function TO_BIT_ARRAY_32 (X : UNSIGNED_WORD) return BIT_ARRAY_32;

type UNSIGNED_LONGWORD_ARRAY is array (INTEGER range <>) of UNSIGNED_LONGWORD;
```

13.7a.4 Conventional Names for Unsigned Longwords

The following XD Ada declarations provide conventional names for static subtypes of the predefined type UNSIGNED_LONGWORD:

```
subtype UNSIGNED_1  is UNSIGNED_LONGWORD range 0 .. 2** 1-1;
subtype UNSIGNED_2  is UNSIGNED_LONGWORD range 0 .. 2** 2-1;
subtype UNSIGNED_3  is UNSIGNED_LONGWORD range 0 .. 2** 3-1;
subtype UNSIGNED_4  is UNSIGNED_LONGWORD range 0 .. 2** 4-1;
subtype UNSIGNED_5  is UNSIGNED_LONGWORD range 0 .. 2** 5-1;
subtype UNSIGNED_6  is UNSIGNED_LONGWORD range 0 .. 2** 6-1;
subtype UNSIGNED_7  is UNSIGNED_LONGWORD range 0 .. 2** 7-1;
subtype UNSIGNED_8  is UNSIGNED_LONGWORD range 0 .. 2** 8-1;
subtype UNSIGNED_9  is UNSIGNED_LONGWORD range 0 .. 2** 9-1;
subtype UNSIGNED_10 is UNSIGNED_LONGWORD range 0 .. 2**10-1;
subtype UNSIGNED_11 is UNSIGNED_LONGWORD range 0 .. 2**11-1;
subtype UNSIGNED_12 is UNSIGNED_LONGWORD range 0 .. 2**12-1;
subtype UNSIGNED_13 is UNSIGNED_LONGWORD range 0 .. 2**13-1;
subtype UNSIGNED_14 is UNSIGNED_LONGWORD range 0 .. 2**14-1;
subtype UNSIGNED_15 is UNSIGNED_LONGWORD range 0 .. 2**15-1;
subtype UNSIGNED_16 is UNSIGNED_LONGWORD range 0 .. 2**16-1;
subtype UNSIGNED_17 is UNSIGNED_LONGWORD range 0 .. 2**17-1;
subtype UNSIGNED_18 is UNSIGNED_LONGWORD range 0 .. 2**18-1;
subtype UNSIGNED_19 is UNSIGNED_LONGWORD range 0 .. 2**19-1;
subtype UNSIGNED_20 is UNSIGNED_LONGWORD range 0 .. 2**20-1;
subtype UNSIGNED_21 is UNSIGNED_LONGWORD range 0 .. 2**21-1;
subtype UNSIGNED_22 is UNSIGNED_LONGWORD range 0 .. 2**22-1;
subtype UNSIGNED_23 is UNSIGNED_LONGWORD range 0 .. 2**23-1;
subtype UNSIGNED_24 is UNSIGNED_LONGWORD range 0 .. 2**24-1;
subtype UNSIGNED_25 is UNSIGNED_LONGWORD range 0 .. 2**25-1;
subtype UNSIGNED_26 is UNSIGNED_LONGWORD range 0 .. 2**26-1;
subtype UNSIGNED_27 is UNSIGNED_LONGWORD range 0 .. 2**27-1;
subtype UNSIGNED_28 is UNSIGNED_LONGWORD range 0 .. 2**28-1;
subtype UNSIGNED_29 is UNSIGNED_LONGWORD range 0 .. 2**29-1;
subtype UNSIGNED_30 is UNSIGNED_LONGWORD range 0 .. 2**30-1;
subtype UNSIGNED_31 is UNSIGNED_LONGWORD range 0 .. 2**31-1;
```

13.7.1 System-Dependent Named Numbers

In XD Ada, the values for system-dependent named numbers are as shown in the following table.

Attribute	MC68020
MIN_INT	-2^{31}
MAX_INT	$2^{31}-1$
MAX_DIGITS	18
MAX_MANTISSA	31
FINE_DELTA	2.0^{-31}
TICK	162.5×10^{-6}

13.7.2 Representation Attributes

The following information supplements all of this section:

For any object, program unit, label, or entry X:

X'ADDRESS — Yields the address of the first of the storage elements allocated to X. For a subprogram, package, task unit or label, this value refers to the machine code associated with the corresponding body or statement. For an entry for which an address clause has been given, the value refers to the offset of the interrupt vector from the vector base register. The value of this attribute is of the type ADDRESS defined in the package SYSTEM.

For an object that is a variable, the value is the actual address of the variable (which may be statically or dynamically allocated). This attribute forces a variable to be allocated in memory rather than in a register, and causes the variable to be marked as volatile for the duration of the block statement or body containing use of the attribute. If the location of the variable is not byte-aligned, the value is the address of the lowest byte that contains the variable. For an object that is a constant, the value is the address of the constant value in memory; however, two occurrences of C'ADDRESS, where

C denotes a constant, may or may not yield the same address value. For an object that is a named number, the value is zero (ADDRESS_ZERO).

NOTE

> In the context of these representation attributes, ADDRESS_ZERO means only that no useful interpretation of a nonzero value is currently supported. That is, its use as a result of C'ADDRESS is subject to change.

For an access object, X.all'ADDRESS is the address of the designated object; X.all'ADDRESS is subject to an ACCESS_CHECK for the designated object. For a record component, X.C'ADDRESS is subject to a DISCRIMINANT_CHECK for an object in a variant part. For an array component or slice, X(I)'ADDRESS or X(I1...I2)'ADDRESS is subject to an INDEX_CHECK for the denoted component or slice.

For program units that are task units or package units, the value is zero (ADDRESS_ZERO). For program units that are subprograms, the value is the same as the address that would be exported. (See Section 13.9a.1.4 (LRM) for information on pragmas EXPORT_FUNCTION and EXPORT_PROCEDURE).

For entries, the value is zero (ADDRESS_ZERO).

For labels, the value is the address of the machine code which follows the label.

For any type or subtype X, or for any object X:

X'SIZE — For a type or a subtype, the value is limited to values in the range 0 .. MAX_INT; the exception NUMERIC_ERROR (see Section 11.1) is raised for values outside this range. For an object that is a variable or a constant in XD Ada, the value is its size in bits. For an object that is a named number, the value is zero. For a record component, X.C'SIZE is subject to a DISCRIMINANT_CHECK

for an object in a variant part. For an array component or slice, X(I)'SIZE or X(I1..I2)'SIZE is subject to an INDEX_CHECK for the denoted component or slice.

For any type or subtype X:

X'MACHINE_SIZE Yields the number of machine bits to be allocated for variables of the type or subtype. This value takes into account any padding bits used by XD Ada when allocating a variable on a byte boundary. The value of this attribute is of the type *universal_integer*.

The value is always a multiple of 8 (bits). In particular, for discrete types it is 8, 16, or 32. The value is limited to the range 0..MAX_INT; the exception NUMERIC_ERROR is raised for values outside this range.

For any object X:

X'BIT Yields the bit offset within the storage unit (byte) that contains the first bit of the storage allocated for the object. The value of this attribute is of the type *universal_integer*, and is always in the range 0..7.

For an object that is a variable or a constant allocated in a register, the value is zero. (The use of this attribute does not force the allocation of a variable to memory.) For an object that is a formal parameter, this attribute applies either to the matching actual parameter or to a copy of the matching actual parameter. For an access object, the value is zero (in the absence of CONSTRAINT_ERROR); X.all'BIT is subject to an ACCESS_CHECK for the designated object. For a record component, X.C'BIT is subject to a DISCRIMINANT_CHECK for a component in a variant part. For an array component or slice, X(I)'BIT or X(I1..I2)'BIT is subject to an INDEX_CHECK for the denoted component or slice.

The following information supplements the Notes section:

The attribute X'MACHINE_SIZE gives the size that would be used for a variable of the type or subtype; it does not give the size that may be used for a component of that type or subtype.

The machine size of a type or subtype can be influenced by representation clauses, unlike the size of a type or subtype, which is independent of representation clauses. The machine size of a base type can be less than, equal to, or greater than the size of that same base type. See the *XD Ada MC68020 Run-Time Reference Manual* for examples and additional discussion.

13.7.3 Representation Attributes of Real Types

The following information supplements paragraphs 3 and 4:

For both fixed- and floating-point types:

T'MACHINE_ROUNDS	In XD Ada this value is FALSE
T'MACHINE_OVERFLOWS	In XD Ada this value is FALSE

The XD Ada values of the other representation attributes for floating-point types are dependent on the floating-point type and are listed in Appendix F.

13.8 Machine Code Insertions

The following information supplements paragraph 4:

XD Ada provides the package MACHINE_CODE. Machine code insertions can be expanded in line.

This predefined package and not a user-defined package must be named in a with clause that applies to the compilation unit in which the code statement occurs.

The following is an example of MACHINE_CODE and the **with** clause in use:

```
-- A machine code procedure to evaluate the sine and cosine of
-- parameter X, returning the results in parameters Y and Z
-- respectively; the procedure is to be expanded in line, so
-- it does not require a stack frame of its own:
with MACHINE_CODE;

procedure SINCOS1 ( X: in FLOAT;
                    Y: out FLOAT;
                    Z: out FLOAT ) is
      use MACHINE_CODE;
begin FSINCOS_REG_INST' (
                   OPCODE           => FSINCOS,
                   SOURCE_REGISTER  => FP0,
                   SIN_REGISTER     => FP1,
                   COS_REGISTER     => FP2 );

end SINCOS1;
```

XD Ada provides the pragma CALL_SEQUENCE_PROCEDURE which specifies parameter-passing mechanisms for machine code procedures. The pragma is defined in Appendix B. Examples of machine code insertion are given in Section 6.1 of the *XD Ada MC68020 Run-Time Reference Manual*. For the specification of the package MACHINE_CODE, see Appendix B of the *XD Ada MC68020 Run-Time Reference Manual*.

13.9 Interface to Other Languages

The following information supplements paragraph 4:

As with VAX Ada, use of pragma INTERFACE in XD Ada is interpreted as being equivalent to supplying the body of the named subprogram or subprograms. Therefore, the following rules apply:

- If a subprogram body is given later for a subprogram named with pragma INTERFACE, the body is illegal.
- If pragma INTERFACE names a subprogram body, the pragma is illegal.
- If a duplicate pragma INTERFACE is given, the latter pragma is illegal.

In XD Ada, pragma INTERFACE applies to a renaming only if the renaming occurs in the same declarative part or package specification as the pragma. The renamed subprogram must also occur in that same declarative part or package specification; renamed subprograms that occur outside the declarative part or package specification are ignored (without a warning diagnostic).

In addition, XD Ada interprets the effect of pragma INTERFACE in such a way that it accepts and ignores implicit declarations of subprograms (such as predefined operators, derived subprograms, attribute functions, and so on).

Dependent upon its use in an XD Ada program, pragma INTERFACE is interpreted in combination with one of two XD Ada import subprogram pragmas: IMPORT_FUNCTION or IMPORT_PROCEDURE. These pragmas are described in Section 13.9a.1.

The language name is ignored, and so may be any identifier that suggests the language, source, or nature of the imported subprogram.

If pragma INTERFACE is used without one of these import pragmas, a default interpretation is used, as follows:

- If the subprogram name applies to a single subprogram, then a default import pragma is assumed as follows:

  For a function, the default is as follows:

  ```
  pragma IMPORT_FUNCTION (function_designator);
  ```

  For a procedure, the default is as follows:

  ```
  pragma IMPORT_PROCEDURE (procedure_identifier);
  ```

- If the subprogram name applies to two or more subprograms, the pragma applies to all of them. However, a warning is given if the appropriate XD Ada import pragmas are not given for all of the subprograms.

Whether or not pragma INTERFACE is used with an import pragma, the subprogram name must be an identifier, or a string literal that denotes an operator symbol. In the following example, pragma INTERFACE specifies that the indicated routines SQRT and EXP are to be imported and used as bodies for the XD Ada functions SQRT and EXP in package FORT_LIB:

```
package FORT_LIB is
   function SQRT(X : FLOAT) return FLOAT;
   function EXP(X : FLOAT) return FLOAT;
private
   pragma INTERFACE(FORTRAN, SQRT);
   pragma INTERFACE(FORTRAN, EXP);
end FORT_LIB;
```

The following information supplements paragraph 5:

In XD Ada, the example package FORT_LIB is interpreted as follows: pragma INTERFACE specifies that the indicated routines SQRT and EXP are to be imported and used as bodies for the Ada functions SQRT and EXP in package FORT_LIB.

```
package CHOOSE_R is
   procedure P(X : INTEGER);
   procedure P(X : FLOAT);
private
   procedure R(X : FLOAT) renames P;
   pragma INTERFACE(ASSEMBLER, R);
end CHOOSE_R;
```

In this example, pragma INTERFACE indicates that the body for the second procedure P is to be imported as routine R.

The following information supplements the Notes section:

The meaning of the subprogram name is determined as for any name (see Section 8.3 (LRM)), except that the name can denote more than one subprogram. Thus, in the following declaration the pragma INTERFACE applies to the first two procedures; it does not apply to the third because the declaration is not visible at the place of the pragma.

```
procedure P (B: BOOLEAN);
procedure P (I: INTEGER);
pragma INTERFACE (ASSEMBLER, P);
procedure P (F: FLOAT);
```

This same interpretation is made for pragmas used to import and export subprograms (see Section 13.9a.1).

If pragma INTERFACE and pragma INLINE are used together, the pragma INLINE is ignored regardless of the order in which the two pragmas appear.

Refer to Chapter 3 of the *XD Ada MC68020 Run-Time Reference Manual* for subprogram calling conventions and run-time organisation, while Chapter 6 of the same manual describes low-level interfaces and assembly language modules.

13.9a XD Ada Import and Export Pragmas

XD Ada provides import and export pragmas designed specifically for constructing programs composed of both Ada and non-Ada entities. The import pragmas allow an Ada program to refer to entities written in another language; the export pragmas make Ada entities available to programs written in other languages. The names of the pragmas indicate the kind of entity involved: IMPORT_FUNCTION and EXPORT_FUNCTION apply to nongeneric functions; IMPORT_PROCEDURE and EXPORT_PROCEDURE apply to nongeneric procedures; IMPORT_OBJECT and EXPORT_OBJECT apply to objects; and IMPORT_EXCEPTION and EXPORT_EXCEPTION apply to exceptions. These pragmas are described in this section, summarized in Annex B, and listed in Appendix F.

All the XD Ada import and export pragmas have the following form:

```
pragma import_export_pragma_name
   (internal_name [, external_designator]
      [, pragma_specific_options]);

import_export_pragma_name ::=

     EXPORT_EXCEPTION | EXPORT_FUNCTION
   | EXPORT_OBJECT    | EXPORT_PROCEDURE
   | IMPORT_EXCEPTION | IMPORT_FUNCTION
   | IMPORT_OBJECT    | IMPORT_PROCEDURE

internal_name ::= [INTERNAL =>] simple_name
   | [INTERNAL =>] operator_symbol -- Can be used only for
                                    -- IMPORT_FUNCTION

external_designator ::= [EXTERNAL =>] external_symbol

external_symbol ::= identifier | string_literal
```

The internal name can be an Ada simple name, or, if the declared entity is a function, the internal name can be a string literal that denotes an operator symbol. A subprogram to be imported or exported must be identified by its internal name and parameter types; and, in the case of a function, by the result type (see Section 13.9a.1.1).

The external designator determines a symbol that is referenced or declared in the linker object module. If an identifier is given, the identifier is used. If a string literal is given, the value of the string is used. The value of a string literal must be a symbol that is acceptable to the XD Ada Builder; it need not be valid as an Ada identifier. (For example, the dollar character ($) can be used.) If no external designator is given, the internal name is used as the external designator. If the

external designator (explicit or default) is longer than 12 characters, the import or export pragma is ignored.

Pragma-specific options are described in the individual pragma sections that follow.

The XD Ada import and export pragmas are only allowed at the place of a declarative item, and must apply to an entity declared by an earlier declarative item of the same declarative part or package specification. At most one import or export pragma is allowed for any given entity; in the case of multiple overloaded subprograms, this rule applies to each subprogram independently.

Additional placement and usage rules apply for particular pragmas as described in the following sections.

Note:

Argument associations for XD Ada import and export pragmas can be either positional or named. With positional association, the arguments are interpreted in the order in which they appear in the syntax definition. The rules for the mixing of positional and named association are the same as those that apply to subprograms (see Section 6.4 (LRM)).

A pragma for an entity declared in a package specification must not be given in the package body. (A pragma for an entity given in the visible part of a package specification can, however, be given in either the visible or private part of the specification.)

No checking is provided to ensure that exported symbols do not conflict with each other or with other global symbols; such checking is performed by the XD Ada Builder.

13.9a.1 Importing and Exporting Subprograms

XD Ada provides a series of pragmas that make it possible to call nongeneric subprograms in a mixed-language programming environment. The IMPORT_FUNCTION and IMPORT_PROCEDURE pragmas specify that the body of the subprogram associated with an Ada subprogram specification is to be provided from assembly language. Pragma INTERFACE must precede one of these import pragmas (see Section 13.9). The EXPORT_FUNCTION and EXPORT_PROCEDURE pragmas allow an Ada procedure or function to be called from assembly language. The pragmas support parameter passing by means of registers.

13.9a.1.1 Importing Subprograms

XD Ada provides two pragmas for importing subprograms: IMPORT_FUNCTION and IMPORT_PROCEDURE. These pragmas allow the import of the kind of subprograms indicated.

The pragmas for importing subprograms have the following form:

```
pragma IMPORT_FUNCTION | IMPORT_PROCEDURE

  ([  INTERNAL =>] internal_name
   [, [EXTERNAL =>] external_designator ]
   [, [PARAMETER_TYPES =>] ( parameter_types ) ]
   [, [RESULT_TYPE =>] type_mark ]              -- FUNCTION only
   [, [MECHANISM =>] mechanism ]
   [, [RESULT_MECHANISM =>] mechanism_spec ]  -- FUNCTION only
   [, [PRESERVED_REGISTERS => ] ( registers ) ]
  );

parameter_types ::=
    null | type_mark {, type_mark}

mechanism ::=
    mechanism_spec | ( mechanism_spec {, mechanism_spec} )

mechanism_spec ::=
    mechanism_name [ ( [REGISTER => ] register_name ) ]

mechanism_name ::=
    VALUE        |
    REFERENCE    | BIT_REFERENCE    |
    DOPE_VECTOR  | BIT_DOPE_VECTOR

registers ::=
    null | register_name {, register_name }
```

Functions must be identified by their internal names and parameter and result types. The parameter and result types can be omitted only if there is exactly one function of that name in the same declarative part or package specification. Otherwise, both the parameter and result types must be specified.

Procedures must be identified by their internal names and parameter types. The parameter types can be omitted only if there is exactly one procedure of that name in the same declarative part or package specification. Otherwise, the parameter types must be specified.

The external designator denotes an XD Ada Builder global symbol that is associated with the external subprogram. If no external designator is given, the internal name is used as the global symbol.

The parameter types option specifies a series of one or more type marks (type or subtype names), not parameter names. Each type mark is positionally associated with a formal parameter in the subprogram's declaration. The absence of parameters must be indicated by the reserved word **null**.

The result type option is used only for functions; it specifies the type or subtype of the function result.

The mechanism option specifies how the imported subprogram expects its parameters to be passed (for example, by value, by reference or by descriptor). The calling program (namely the XD Ada program) is responsible for ensuring that parameters are passed in the form required by the external routine.

Mechanism names are described as follows. Within these definitions, the term *bit string* means any one-dimensional array of a discrete type whose components occupy successive single bits. The term *simple record type* means a record type that does not have a variant part and in which any constraint for each component and subcomponent is static. A *simple record subtype* is a simple record type or a static constrained subtype of a record type (with discriminants) in which any constraint for each component and subcomponent of the record type is static.

VALUE	Specifies that the immediate value of the actual parameter is passed. Values of scalars, access types, address types and private types whose full type is either a scalar, an access type or an address type can be passed by VALUE. If the value is a private type, the pragma must occur after the full declaration of the private type. Bit strings can also be passed by VALUE.
REFERENCE	Specifies that the address of the value of the actual parameter is passed. This mechanism can be used for parameters of any type.
DOPE_VECTOR	Specifies that the address of the DOPE_VECTOR is passed, a 32-bit pointer to an object, taking the form described in Section 2.1.4 of the *XD Ada MC68020 Run-Time Reference Manual*.
BIT_DOPE_VECTOR	Specifies that the address of the BIT_DOPE_VECTOR is passed, a 32-bit pointer to an object, taking the form described in Section 2.1.4 of the *XD Ada MC68020 Run-Time Reference Manual*.

If the first form of the mechanism option is given (a single mechanism name without parentheses), all parameters are passed using that mechanism. If the second form is given (a series of mechanism names in parentheses and separated by commas), each mechanism name determines how the parameter in the same position in the subprogram specification will be passed. With the second form, each parameter name must have an associated mechanism name.

The result mechanism option is used only for functions; it specifies the parameter-passing mechanism for passing the result type, and optionally, a specific register used to pass the result.

The preserved registers option gives a list of hardware registers which are not altered by the procedure or function. If this option is omitted it implies that no registers are preserved.

In addition to the rules given in Section 13.9a, the rules for importing subprograms are as follows:

- If an import pragma is given for a subprogram specification, pragma INTERFACE (see Section 13.9) must also be given for the subprogram earlier in the same declarative part or package specification. The use of pragma INTERFACE implies that a corresponding body is not given.
- If a subprogram has been declared as a compilation unit, the pragma is only allowed after the subprogram declaration and before any subsequent compilation unit.
- These pragmas can be used for subprograms declared with a renaming declaration. The internal name must be a simple name, and the renaming declaration must occur in the same declarative part or package specification as the pragma. The renamed subprogram must also occur in that same declarative part or package specification. Renamed subprograms that occur outside the declarative part or package specification are ignored (without a warning diagnostic).
- None of these pragmas can be used for a generic subprogram or a generic subprogram instantiation. In particular, they cannot be used for a subprogram that is declared by a generic instantiation of a predefined subprogram (such as UNCHECKED_CONVERSION).

Examples:

In this example, the pragma INTERFACE identifies SQRT as an external subprogram; the language name argument ASSEMBLER has no effect. The pragma IMPORT_FUNCTION uses positional notation to specify arguments for importing the declared function SQRT. The pragma form indicates that the internal name is SQRT, and the external designator is "MTH$SQRT". The parameter is of type FLOAT, and is passed in FP1; the result is of type FLOAT, and it is returned in FP2.

```
function SQRT (X : FLOAT )  return FLOAT;
pragma   INTERFACE (ASSEMBLER, SQRT);
pragma   IMPORT_FUNCTION
               (SQRT, "MTH$SQRT", (FLOAT),
                FLOAT, (VALUE(FP1)), VALUE(FP2)
               );
```

The next example shows an alternative way of importing the declared function SQRT using named notation. In this case, the parameter is passed in FP5, and the result is returned in FP4; the registers which are preserved by the called function are also specified.

```
function SQRT (X : LONG_FLOAT ) return LONG_FLOAT;
pragma INTERFACE (ASSEMBLER, SQRT);
pragma IMPORT_FUNCTION (INTERNAL            => SQRT,
                        PARAMETER_TYPES     => (LONG_FLOAT),
                        RESULT_TYPE         => LONG_FLOAT,
                        MECHANISM           => (VALUE(FP5)),
                        RESULT_MECHANISM    => VALUE(FP4),
                        EXTERNAL            => "MTH$DSQRT",
                        PRESERVED_REGISTERS =>
                           (D0,  D1,  D2,  D3,  D4,  D5,  D6,  D7,
                            A0,  A1,  A2,  A3,  A4,  A5,
                            FP0, FP1, FP2, FP3,      FP5, FP6));
```

If the previous example is combined with the code in the first example (that is, with only one occurrence of pragma INTERFACE), the result is an overloading of SQRT:

```
function SQRT (X : LONG_FLOAT ) return LONG_FLOAT;
function SQRT (X : FLOAT ) return FLOAT;
pragma INTERFACE (ASSEMBLER, SQRT);
pragma IMPORT_FUNCTION (SQRT,
                        "MTH$SQRT",
                        (FLOAT),
                         FLOAT,
                        (VALUE(FP1)),
                         VALUE(FP2));
pragma IMPORT_FUNCTION (INTERNAL          => SQRT,
                        PARAMETER_TYPES   => (LONG_FLOAT),
                        RESULT_TYPE       => LONG_FLOAT,
                        MECHANISM         => (VALUE(FP5)),
                        RESULT_MECHANISM  => VALUE(FP4),
                        EXTERNAL          => "MTH$DSQRT",
                        PRESERVED_REGISTERS =>
                           (D0,  D1,  D2,  D3,  D4,  D5,  D6,  D7,
                            A0,  A1,  A2,  A3,  A4,  A5,
                            FP0, FP1, FP2, FP3,      FP5, FP6));
```

The next example shows the use of renaming with an imported procedure (it is assumed that these declarations occur in a declarative part or package specification). Note that the renaming causes the imported ASSEMBLER procedure to be used in calls to both procedures CHANGE and EXCHANGE. Also note that because no external designator is specified, the builder global symbol associated with the external subprogram is EXCHANGE, and because no parameter mechanisms are specified, the compiler's defaults will apply in calls to CHANGE or EXCHANGE.

```
procedure CHANGE (X,Y : INTEGER);
procedure EXCHANGE (X,Y : INTEGER) renames CHANGE;
pragma INTERFACE (ASSEMBLER, EXCHANGE);
pragma IMPORT_PROCEDURE (INTERNAL        => EXCHANGE,
                         PARAMETER_TYPES => (INTEGER,INTEGER));
```

13.9a.1.2 Exporting Subprograms

XD Ada provides two pragmas for exporting subprograms: EXPORT_FUNCTION and EXPORT_PROCEDURE. Both export pragmas establish an external name for a subprogram and make the name available to the XD Ada Builder as a global symbol, so that the subprogram can be called by an assembly language module.

The EXPORT_FUNCTION and EXPORT_PROCEDURE pragmas allow the export of the kind of subprograms indicated.

The pragmas for exporting subprograms have the following form:

```
pragma EXPORT_FUNCTION | EXPORT_PROCEDURE

  ( [   INTERNAL =>] internal_name
    [, [EXTERNAL =>] external_designator ]
    [, [PARAMETER_TYPES =>] ( parameter_types ) ]
    [, [RESULT_TYPE =>] type_mark ]          -- FUNCTION only
    [, [MECHANISM =>] mechanism ]
    [, [RESULT_MECHANISM =>] mechanism_spec ]  -- FUNCTION only
  );

parameter_types ::=
    null | type_mark {, type_mark}

mechanism ::=
    mechanism_spec | ( mechanism_spec {, mechanism_spec} )

mechanism_spec ::=
    mechanism_name [ ( [REGISTER => ] register_name ) ]

mechanism_name ::=
    VALUE       |
    REFERENCE   | BIT_REFERENCE   |
    DOPE_VECTOR | BIT_DOPE_VECTOR

registers ::=
    null | register_name {, register_name }

parameter_types ::=
    null | type_mark {, type_mark}
```

Functions must be identified by their internal names and parameter and result types. The parameter and result types can be omitted only if there is exactly one function of that name in the same declarative part or package specification. Otherwise, both the parameter and result types must be specified.

Procedures must be identified by their internal names and parameter types. The parameter types can be omitted only if there is exactly one procedure of that name in the same declarative part or package specification. Otherwise, the parameter types must be specified.

The external designator denotes an XD Ada Builder global symbol that is associated with the external subprogram. If no external name is given, the internal name is used as the global symbol.

The parameter types option specifies a series of one or more type marks (type or subtype names), not parameter names. Each type mark is positionally associated with a formal parameter in the subprogram's declaration. The absence of parameters must be indicated by the reserved word **null**.

The result type option is used only for functions; it specifies the type or subtype of the function result.

The mechanism option specifies how the imported subprogram expects its parameters to be passed (for example, by value, by reference or by descriptor). The calling program (namely the XD Ada program) is responsible for ensuring that parameters are passed in the form required by the external routine. Mechanism options and possible values for mechanism names and class names are described in Section 13.9a.1.1.

If the first form of the mechanism option is given (a single mechanism name without parentheses), all parameters are passed using that mechanism. If the second form is given (a series of mechanism names in parentheses and separated by commas), each mechanism name determines how the parameter in the same position in the subprogram specification will be passed. With the second form, each parameter name must have an associated mechanism name.

The result mechanism option is used only for functions; it specifies the parameter-passing mechanism for passing the result type, and optionally, a specific register used to pass the result.

In addition to the rules given in Section 13.9a, the rules for exporting subprograms are as follows:

- An exported subprogram must be a library unit or be declared in the outermost declarative part of a library package. Thus, pragmas for exporting subprograms are allowed only in the following cases:
  - For a subprogram specification or a subprogram body that is a library unit
  - For a subprogram specification that is declared in the outermost declarations of a package specification or a package body that is a library unit
  - For a subprogram body that is declared in the outermost declarations of a package body that is a library unit

  Consequently, an export pragma for a subprogram body is allowed only if either the body does not have a corresponding specification, or the specification and body occur in the same declarative part.

  This set of rules implies that an EXPORT_FUNCTION or EXPORT_PROCEDURE pragma cannot be given for a generic library subprogram, nor can one be given for a subprogram declared in a generic library package. However, either of these pragmas can be given for a subprogram resulting from the instantiation of

a generic subprogram, provided that the instantiation otherwise satisfies this set of rules.

- In the case of a subprogram declared as a compilation unit, the pragma is only allowed after the subprogram declaration and before any subsequent compilation unit.
- Neither of these pragmas can be used for a subprogram that is declared with a renaming declaration.
- Neither of these pragmas can be used for a subprogram that is declared by a generic instantiation of a built-in library subprogram (such as UNCHECKED_CONVERSION).

Examples:

The following example shows an export pragma that causes the Ada procedure PROC to be exported for use in an assembly language module. The name PROC is declared as an XD Ada Builder global symbol.

```
procedure PROC (Y : INTEGER);
pragma EXPORT_PROCEDURE (PROC);
```

The next example shows an Ada function being called from an assembly language module:

```
function MULTIPLY (Y : in INTEGER) return INTEGER is
begin
        return Y * 10;
end;
pragma EXPORT_FUNCTION ( INTERNAL => MULTIPLY,
                         PARAMETER_TYPES => (INTEGER),
                         RESULT_TYPE     => INTEGER );

pragma CALL_SEQUENCE_FUNCTION (
                         UNIT              => MULTIPLY,
                         PARAMETER_TYPES   => (INTEGER),
                         MECHANISM         => (VALUE(D0)),
                         RESULT_MECHANISM  => VALUE(D0));
------------------------------------------------------------------
    TITLE "MC68020 Calling Ada"
    MODULE "CALL_ADA"

    XDEF CALL_ADA
    XREF MULTIPLY

    DSEG

A   BLKB    4           ! An INTEGER

    PSEG
```

```
CALL_ADA
    MOVE.L  #1,A        ! A := 1;

    MOVE.L  A,D0
    JSR     MULTIPLY.L

    MOVE.L  D0,A        ! A := MULTIPLY( A )
    RTS

    END
```

13.9a.2 Importing and Exporting Objects

XD Ada provides two pragmas for importing and exporting objects: IMPORT_OBJECT and EXPORT_OBJECT. The IMPORT_OBJECT pragma references storage declared in an assembly language module. The EXPORT_OBJECT pragma allows an assembly language module to refer to the storage allocated for an Ada object.

In addition to the rules given in Section 13.9a, the rules for importing and exporting objects are as follows:

- The object to be imported or exported must be a variable declared by an object declaration at the outermost level of a library package specification or body.
- The subtype indication of an object to be imported or exported must denote one of the following:
  - A scalar type or subtype.
  - An array subtype with static index constraints whose component size is static.
  - A record type or subtype that does not have a variant part and in which any constraint for each component and subcomponent is static (a simple record type or subtype).
- Import and export pragmas are not allowed for objects declared with a renaming declaration.
- Import and export pragmas for objects are not allowed in a generic unit.

Notes:

Objects of private or limited private types cannot be imported or exported outside the package that declares the (limited) private type. They can be imported or exported inside the body of the package where the type is declared (that is, where the full type is known).

The XD Ada pragmas for importing or exporting objects can precede or follow a pragma VOLATILE for the same objects (see Section 9.11).

Address clauses are not allowed in combination with any of the XD Ada pragmas for importing or exporting objects. If used in such cases, the pragma involved is ignored (see Section 13.5).

13.9a.2.1 Importing Objects

The XD Ada IMPORT_OBJECT pragma specifies that the storage allocated for the object (when the assembly language module is compiled) be made known to the calling Ada program by an externally-defined XD Ada Builder global symbol.

Pragma IMPORT_OBJECT has the following form:

```
pragma IMPORT_OBJECT
   (internal_name [, external_designator])
```

The internal name is the object identifier. The external designator denotes an XD Ada Builder global symbol that is associated with the external object. If no external designator is given, the internal name is used as the global symbol.

Because it is not created by an Ada elaboration, an imported object cannot have an initial value. Specifically, this restriction means that the object to be imported:

- Cannot be a constant (have an explicit initial value).
- Cannot be an access type (which has a default initial value of null).
- Cannot be a record type that has discriminants (which are always initialized) or components with default initial expressions.
- Cannot be an object of a task type.

Example:

```
PID: INTEGER;
pragma IMPORT_OBJECT (PID, "PROCESS$ID");
```

In this example, the variable PID refers to the externally-defined symbol PROCESS$ID.

Alternatively, this example can be written in named notation as follows:

```
PID : INTEGER;
pragma IMPORT_OBJECT (INTERNAL => PID,
                      EXTERNAL => "PROCESS$ID");
```

13.9a.2.2 Exporting Objects

The XD Ada pragma EXPORT_OBJECT specifies that the storage allocated for the object (when the Ada program is compiled) be made known to assembly language modules by an XD Ada Builder global symbol.

Pragma EXPORT_OBJECT has the following form:

```
pragma EXPORT_OBJECT
    (internal_name [, external_designator])
```

The internal name is the object identifier. The external designator denotes an XD Ada Builder global symbol that is associated with the external object. If no external designator is given, the internal name is used as the global symbol.

Example:

```
PID: INTEGER;
pragma EXPORT_OBJECT (PID, "PROCESS$ID");
```

Alternatively, this example can be written in named notation:

```
PID: INTEGER;
pragma EXPORT_OBJECT (INTERNAL => PID,
                      EXTERNAL => "PROCESS$ID");
```

13.9a.3 Importing and Exporting Exceptions

XD Ada provides the IMPORT_EXCEPTION and EXPORT_EXCEPTION pragmas for importing and exporting exceptions. The pragma IMPORT_EXCEPTION allows non-Ada exceptions to be used in Ada programs; the pragma EXPORT_EXCEPTION allows Ada exceptions to be used by foreign units.

The rules for importing and exporting exceptions are given in Section 13.9a.

Note:

A pragma for an exception that is declared in a package specification is not allowed in the package body.

13.9a.3.1 Importing Exceptions

The XD Ada IMPORT_EXCEPTION pragma is provided for compatibility with VAX Ada. This pragma specifies that the exception associated with an exception declaration in an Ada program be defined externally in non-Ada code.

In XD Ada pragma IMPORT_EXCEPTION has the following form:

```
pragma IMPORT_EXCEPTION
   (internal_name [, external_designator]
      [, [FORM =>] ADA ]);
```

The internal name must be an Ada identifier that denotes a declared exception. The external designator denotes an XD Ada Builder global symbol to be used to refer to the exception. If no external name is given, the internal name is used as the global symbol.

For compatibility with VAX Ada, the form option indicates that an Ada exception is being imported. If omitted, this defaults to ADA.

The external designator refers to an address that identifies the exception.

The VAX Ada version of this pragma supports an alternative form (VMS), and a code option in addition to the XD Ada arguments. If either of these unsupported arguments is specified, the compiler ignores the pragma and issues a warning message.

13.9a.3.2 Exporting Exceptions

The XD Ada EXPORT_EXCEPTION pragma allows Ada exceptions to be visible outside the XD Ada program, so that they can be raised and handled by programs written in XD Ada MC68020 assembly language. This pragma establishes an external name for an Ada exception and makes the name available to the XD Ada Builder as a global symbol. Refer to the *XD Ada MC68020 Run-Time Reference Manual* for further information on exporting exceptions.

Pragma EXPORT_EXCEPTION has the following form:

```
pragma EXPORT_EXCEPTION
    (internal_name [, external_designator]
       [, [FORM =>] ADA ]);
```

The internal name must be an Ada identifier that denotes a declared exception. The external designator denotes an XD Ada Builder global symbol to be used to refer to the exception.

The form option specifies that an Ada exception is being exported.

Example:

```
UNDERFLOW : exception
pragma EXPORT_EXCEPTION (UNDERFLOW, MTH_UNDERFLOW, ADA);
```

In this example, an Ada exception is exported as a global symbol.

13.10 Unchecked Programming

13.10.1 Unchecked Storage Deallocation

The following information supplements the Notes section:

Because UNCHECKED_DEALLOCATION is a predefined generic procedure, XD Ada does not allow the use of the IMPORT_PROCEDURE pragma to substitute an alternative procedure body.

13.10.2 Unchecked Type Conversions

The following information supplements paragraph 2:

XD Ada supports the generic function UNCHECKED_CONVERSION with the following restrictions on the class of types involved:

- The actual subtype corresponding to the formal type TARGET must not be an unconstrained array type.
- The actual subtype corresponding to the formal type TARGET must not be an unconstrained type with discriminants.

Further, when the target type is a type with discriminants, the value resulting from a call of the conversion function resulting from an instantiation of UNCHECKED_CONVERSION is checked to ensure that the discriminants satisfy the constraints of the actual subtype.

The effect with XD Ada is as if the source value is copied one byte in ascending order of address, into the destination, also in ascending order of address. If the destination has fewer bytes than the source value, the high order bytes of the source value are ignored (truncated). If the source value has fewer bytes than the destination, the high order bytes of the destination are set to zero.