REPORT DOCUMENTATION PAGE

Form Approved
OPM No. 0704-0188

AD-A234 914

REPORT DATE

3. REPORT TYPE AND DATES COVERED
Final: Nov 30 1990 to Mar 1, 1993

4. TITLE AND SUBTITLE
Ada Compiler Validation Summary Report: DDC International A/S, DACS VAX/VMS to 80386 PM Bare Ada Cross Compiler System with Rate Monotonic Scheduling, Version 4.6 (Host) to VAX 8530 running VMS Version 5.3(Target), 901129S1.1178

5. FUNDING NUMBERS

6. AUTHOR(S)
National Institute of Standards and Technology
Gaithersburg, MD
USA

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
National Institute of Standards and Technology
National Computer Systems Laboratory
Bldg. 255, Rm A266
Gaithersburg, MD 20899 USA

8. PERFORMING ORGANIZATION REPORT NUMBER
NIST90DDC500_7_1.11

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

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

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

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*
DDC International A/S, DACS VAX/VMS to 80386 PM Bare Ada Corss Compiler System with Rate Monotonic Scheduling, Version 4.6, Gaithersburh, MD, VAX 8530 (Host) to Bare Board iSBC 386/21 (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	

AVF Control Number: NIST90DDC500_7_1.11
DATE COMPLETED
BEFORE ON-SITE: October 30, 1990
AFTER ON-SITE: November 30, 1990
REVISIONS:

Ada COMPILER
VALIDATION SUMMARY REPORT:
Certificate Number: 901129S1.1178
DDC International A/S
DACS VAX/VMS to 80386 PM Bare Ada Cross Compiler System
with Rate Monotonic Scheduling, Version 4.6
VAX 8530 => Bare Board iSBC 386/21

Prepared By:
Software Standards Validation Group
National Computer Systems Laboratory
National Institute of Standards and Technology
Building 225, Room A266
Gaithersburg, Maryland 20899

AVF Control Number: NIST90DDC500_7_1.11

Certificate Information

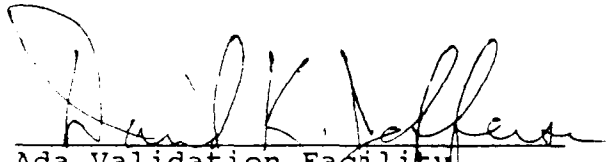
The following Ada implementation was tested and determined to pass ACVC 1.11. Testing was completed on November 29, 1990.

Compiler Name and Version:	DACS VAX/VMS to 80386 PM Bare Ada Cross Compiler System with Rate Monotonic Scheduling, Version 4.6
Host Computer System:	VAX 8530 running VMS Version 5.3
Target Computer System:	Bare Board iSBC 386/21

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 901129S1.1178 is awarded to DDC International A/S. This certificate expires on March 01, 1993.

This report has been reviewed and is approved.



Ada Validation Facility
Dr. David K. Jefferson
Chief, Information Systems
Engineering Division (ISED)
National Computer Systems
Laboratory (NCSL)
National Institute of
Standards and Technology
Building 225, Room A266
Gaithersburg, MD 20899

Ada Validation Facility
Mr. L. Arnold Johnson
Manager, Software Standards
Validation Group
National Computer Systems
Laboratory (NCSL)
National Institute of
Standards and Technology
Building 225, Room A266
Gaithersburg, MD 20899

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

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

DECLARATION OF CONFORMANCE

The following declaration of conformance was supplied by the customer.

DECLARATION OF CONFORMANCE

Customer and Certificate Awardee: DDC International A/S

Ada Validation Facility: National Institute of Standards and Technology
National Computer Systems Laboratory (NCSL)
Software Validation Group
Building 225, Room A266
Gaithersburg, Maryland 20899

ACVC Version: 1.11

Ada Implementation:

Compiler Name and Version: DACS VAX/VMS to 80386 PM Bare Ada Cross Compiler System with Rate Monotonic Scheduling, Version 4.6

Host Computer System: VAX 8530 running VMS Version 5.3

Target Computer System: Bare Board iSBC 386/21

Declaration:

[I/we] the undersigned, declare that [I/we] have no knowledge of deliberate deviations from the Ada Language Standard ANSI/MIL-STD-1815A ISO 8652-1987 in the implementation listed above.

Customer Signature Date
Company
Title

TABLE OF CONTENTS

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

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

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

APPENDIX A A-1
MACRO PARAMETERS A-1

APPENDIX B B-1
COMPILATION SYSTEM OPTIONS B-1
LINKER OPTIONS B-2

APPENDIX C C-1
APPENDIX F OF THE Ada STANDARD C-1

CHAPTER 1

INTRODUCTION

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

1.1 USE OF THIS VALIDATION SUMMARY REPORT

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

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

National Technical Information Service
5285 Port Royal Road
Springfield VA 22161

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

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

1.2 REFERENCES

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

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

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

1.3 ACVC TEST CLASSES

Compliance of Ada implementations is tested by means of the ACVC. The ACVC contains a collection of test programs structured into six test classes: A, B, C, D, E, and L. The first letter of a test name identifies the class to which it belongs. Class A, C, D, and E tests are executable. Class B and class L tests are expected to produce errors at compile time and link time, respectively.

The executable tests are written in a self-checking manner and produce a PASSED, FAILED, or NOT APPLICABLE message indicating the result when they are executed. Three Ada library units, the packages REPORT and SPPRT13, and the procedure CHECK_FILE are used for this purpose. The package REPORT also provides a set of identity functions used to defeat some compiler optimizations allowed by the Ada Standard that would circumvent a test objective. The package SPPRT13 is used by many tests for Chapter 13 of the Ada Standard. The procedure CHECK_FILE is used to check the contents of text files written by some of the Class C tests for Chapter 14 of the Ada Standard. The operation of REPORT and CHECK_FILE is checked by a set of executable tests. If these units are not operating correctly, validation testing is discontinued. Class B tests check that a compiler detects illegal language usage. Class B tests are not executable. Each test in this class is compiled and the resulting compilation listing is examined to verify that all violations of the Ada Standard are detected. Some of the class B tests contain legal Ada code which must not be flagged illegal by the compiler. This behavior is also verified.

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

In some tests of the ACVC, certain macro strings have to be replaced by implementation-specific values -- for example, the largest integer. A list of the values used for this implementation is provided in Appendix A. In addition to these anticipated test modifications, additional changes may be required to remove unforeseen conflicts between the tests and implementation-dependent characteristics. The modifications required for this implementation are described in section 2.3.
For each Ada implementation, a customized test suite is produced by the AVF. This customization consists of making the modifications described in the preceding paragraph, removing withdrawn tests (see section 2.1) and, possibly some inapplicable

tests (see Section 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

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

Some tests are withdrawn by the AVO from the ACVC because they do not conform to the Ada Standard. The following 81 tests had been withdrawn by the Ada Validation Organization (AVO) at the time of validation testing. 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 90-10-12.

E28005C	B28006C	C34006D	B41308B	C43004A	C45114A
C45346A	C45612B	C45651A	C46022A	B49008A	A74006A
C74308A	B83022B	B83022H	B83025B	B83025D	B83026A
B83026B	C83041A	B85001L	C97116A	C98003B	BA2011A
CB7001A	CB7001B	CB7004A	CC1223A	BC1226A	CC1226B
BC3009B	BD1B02B	BD1B06A	AD1B08A	BD2A02A	CD2A21E
CD2A23E	CD2A32A	CD2A41A	CD2A41E	CD2A87A	CD2B15C
BD3006A	BD4008A	CD4022A	CD4022D	CD4024B	CD4024C
CD4024D	CD4031A	CD4051D	CD5111A	CD7004C	ED7005D
CD7005E	AD7006A	CD7006E	AD7201A	AD7201E	CD7204B
BD8002A	BD8004C	CD9005A	CD9005B	CDA201E	CE2107I
CE2117A	CE2117B	CE2119B	CE2205B	CE2405A	CE3111C
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. The inapplicability criteria for some tests are explained in documents issued by ISO and the AJPO known as Ada Issues and commonly referenced in the format AI-dddd. For this implementation, the following tests were inapplicable for the reasons indicated; references to Ada Issues are included as appropriate.

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

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

C45641L..Y (14 tests) C46012L..Z (15 tests)

C24113I..K (3 TESTS) USE A LINE LENGTH IN THE INPUT FILE WHICH EXCEEDS 126 CHARACTERS.

C35702A, C35713B, C45423B, B86001T, AND C86006H CHECK FOR THE PREDEFINED TYPE SHORT_FLOAT.

C35713D AND B86001Z CHECK FOR A PREDEFINED FLOATING-POINT TYPE WITH A NAME OTHER THAN FLOAT, LONG_FLOAT, OR SHORT_FLOAT.

C35404D, C45231D, B86001X, C86006E, AND CD7101G CHECK FOR A PREDEFINED INTEGER TYPE WITH A NAME OTHER THAN INTEGER, LONG_INTEGER, OR SHORT_INTEGER.

C45531M, C45531N, C45531O, C45531P, C45532M, C45532N, C45532O, AND C45532P CHECK FIXED-POINT OPERATIONS FOR TYPES THAT REQUIRE A SYSTEM.MAX_MANTISSA OF 47 OR GREATER.

C45624A CHECKS THAT THE PROPER EXCEPTION IS RAISED IF MACHINE_OVERFLOWS IS FALSE FOR FLOATING POINT TYPES WITH DIGITS 5. FOR THIS IMPLEMENTATION, MACHINE_OVERFLOWS IS TRUE.

C45624B CHECKS THAT THE PROPER EXCEPTION IS RAISED IF MACHINE_OVERFLOWS IS FALSE FOR FLOATING POINT TYPES WITH DIGITS 6. FOR THIS IMPLEMENTATION, MACHINE_OVERFLOWS IS TRUE.

C4A013B CONTAINS THE EVALUATION OF AN EXPRESSION INVOLVING 'MACHINE_RADIX APPLIED TO THE MOST PRECISE FLOATING-POINT TYPE. THIS EXPRESSION WOULD RAISE AN EXCEPTION. SINCE THE EXPRESSION MUST BE STATIC, IT IS REJECTED AT COMPILE TIME.

D56001B USES 65 LEVELS OF BLOCK NESTING WHICH EXCEEDS THE CAPACITY OF THE COMPILER.

C86001F RECOMPILES PACKAGE SYSTEM, MAKING PACKAGE TEXT_IO, AND HENCE PACKAGE REPORT, OBSOLETE. FOR THIS IMPLEMENTATION, THE PACKAGE TEXT_IO IS DEPENDENT UPON PACKAGE SYSTEM.

B86001Y CHECKS FOR A PREDEFINED FIXED-POINT TYPE OTHER THAN DURATION.

C96005B CHECKS FOR VALUES OF TYPE DURATION'BASE THAT ARE OUTSIDE THE RANGE OF DURATION. THERE ARE NO SUCH VALUES FOR THIS IMPLEMENTATION.

CA2009C, CA2009F, BC3204C, AND BC3205D THESE TESTS INSTANTIATE GENERIC UNITS BEFORE THEIR BODIES ARE COMPILED. THIS IMPLEMENTATION CREATES A DEPENDENCE ON GENERIC UNIT AS ALLOWED BY AI-00408 & AI-00530 SUCH THAT A THE COMPILATION OF THE GENERIC UNIT BODIES MAKES THE INSTANTIATING UNITS OBSOLETE.

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.

The following 265 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..D (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	CE3116A	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)	

CE2103A..B and CE3107A EXPECT THAT NAME_ERROR IS RAISED WHEN AN ATTEMPT IS MADE TO CREATE A FILE WITH AN ILLEGAL NAME; THIS IMPLEMENTATION DOES NOT SUPPORT THE CREATION OF EXTERNAL FILES AND SO RAISES USE_ERROR.

2.3 TEST MODIFICATIONS

Modifications (see section 1.3) were required for 67 tests.

The following tests were split into two or more tests because this

CHAPTER 3

PROCESSING INFORMATION

3.1 TESTING ENVIRONMENT

The executable files were prepared on the VAX host computer chapter by chapter. When a chapter was completely processed, the executables were transferred via ethernet to a personal computer (COMPAQ 386 running MS-DOS Version 3.3) acting as a host for an In Circuit Emulation tool (i2ICE). The target was connected via RS232C to second personal computer (COMPAQ 286 running MS-DOS Version 3.3) which acted as a capture device. The second personal computer was connected via ethernet to the VAX.

The DACS VAX/VMS to 80386 PM Bare Ada Cross Compiler System, Version 4.6 was executed on the target board with the following:

Bare Board iSBC 386/21
80387
One internal timer
One serial port
1MB RAM

For each chapter, a command file was generated that loaded and executed every program.

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

Mr. Thorkil B. Rasmussen
DDC International A/S
Gl. Lundtoftevej 1B
DK-2800 Lyngby
DENMARK

Telephone: + 45 42 87 11 44
Telefax: + 45 42 87 22 17

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

In the U.S.A.:

Mr. Mike Turner
DDC-I, Inc.
9630 North 25th Avenue

9630 North 25th Avenue
Suite #118
Phoenix, Arizona 85021

Mailing address:
P.O. Box 37767
Phoenix, Arizona 85069-7767
Telephone: 602-944-1883
Telefax: 602-944-3253

In the rest of the world:

Mr. Palle Andersson
DDC International A/S
Gl. Lundtoftevej 1B
DK-2800 LYNGBY
Denmark

Telephone: + 45 42 87 11 44
Telefax: + 45 42 87 22 17

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	3580	
b)	Total Number of Withdrawn Tests	81	
c)	Processed Inapplicable Tests	509	
d)	Non-Processed I/O Tests	0	
e)	Non-Processed Floating-Point Precision Tests	0	
f)	Total Number of Inapplicable Tests	509	(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 509 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 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 using the communications link described above.

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  /NOSAVE_SOURCE
```

The options invoked by default for validation testing during this test were:

```
/CHECK  /CONFIGURATION_FILE = <default file>
/NOTARGET_DEBUG  /LIBRARY   /NOOPTOMIZE
/NOPROGRESS  /NOXREF
```

Test output, compiler and linker listings, and job logs were captured on magnetic tape and archived at the AVF. Selected listings examined on-site by the validation team were also archived.

APPENDIX A

MACRO PARAMETERS

This appendix contains the macro parameters used for customizing the ACVC. The meaning and purpose of these parameters are explained in [UG89]. The parameter values are presented in two tables. The first table lists the values that are defined in terms of the maximum input-line length, which is 126 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	126
$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') & '"'

The following table contains the values for the remaining macro parameters.

Macro Parameter	Macro Value

```
ACC_SIZE                                : 48
ALIGNMENT                               : 2
COUNT_LAST                              : 2_147_483_647
DEFAULT_MEM_SIZE                        : 16#1_0000_0000#
DEFAULT_STOR_UNIT                       : 16
DEFAULT_SYS_NAME                        : IAPX386_PM
DELTA_DOC                               : 2#1.0#E-31
ENTRY_ADDRESS                           : (140,0)
ENTRY_ADDRESS1                          : (141,0)
ENTRY_ADDRESS2                          : (142,0)
FIELD_LAST                              : 35
FILE_TERMINATOR                         : ASCII.SUB
FIXED_NAME                              : NO_SUCH_FIXED_TYPE
FLOAT_NAME                              : SHORT_SHORT_FLOAT
FORM_STRING                             : ""
FORM_STRING2                            :
   "CANNOT_RESTRICT_FILE_CAPACITY"
GREATER_THAN_DURATION                   : 75_000.0
GREATER_THAN_DURATION_BASE_LAST         : 131_073.0
GREATER_THAN_FLOAT_BASE_LAST            : 16#1.0#E+32
GREATER_THAN_FLOAT_SAFE_LARGE           : 16#5.FFFF_F0#E+31
GREATER_THAN_SHORT_FLOAT_SAFE_LARGE: 1.0E308
HIGH_PRIORITY                           : 31
ILLEGAL_EXTERNAL_FILE_NAME1             : \NODIRECTORY\FILENAME
ILLEGAL_EXTERNAL_FILE_NAME2             :
   THIS-FILE-NAME-IS-TOO-LONG-FOR-MY-SYSTEM
INAPPROPRIATE_LINE_LENGTH               : -1
INAPPROPRIATE_PAGE_LENGTH               : -1
INCLUDE_PRAGMA1                         :
   PRAGMA INCLUDE ("A28006D1.TST")
INCLUDE_PRAGMA2                         :
   PRAGMA INCLUDE ("B28006E1.TST")
INTEGER_FIRST                           : -2147483648
INTEGER_LAST                            :  2147483647
INTEGER_LAST_PLUS_1                     : 2_147_483_648
INTERFACE_LANGUAGE                      : ASM86
LESS_THAN_DURATION                      : -75_000.0
LESS_THAN_DURATION_BASE_FIRST           : -131_073.0
LINE_TERMINATOR                         : ASCII.CR
LOW_PRIORITY                            : 0
MACHINE_CODE_STATEMENT                  :
   MACHINE_INSTRUCTION'(NONE,m_RETN);
MACHINE_CODE_TYPE                       : REGISTER_TYPE
MANTISSA_DOC                            : 31
```

```
MAX_DIGITS                              : 15
MAX_INT                                 : 9223372036854775807
MAX_INT_PLUS_1                          : 9223372036854775808
MIN_INT                                 : -9223372036854775808
NAME                                    : SHORT_SHORT_INTEGER
NAME_LIST                               : IAPX386_PM
NAME_SPECIFICATION1                     :
   DISK$AWC_2:[CROCKETTL.ACVC11.DEVELOPMENT]X2120A.;1
NAME_SPECIFICATION2                     :
   DISK$AWC_2:[CROCKETTL.ACVC11.DEVELOPMENT]X2120B.;1
NAME_SPECIFICATION3                     :
   DISK$AWC_2:[CROCKETTL.ACVC11.DEVELOPMENT]X2120C.;1
NEG_BASED_INT                           : 16#FFFF_FFFF_FFFF_FFFF#
NEW_MEM_SIZE                            : 16#1_0000_0000#
NEW_STOR_UNIT                           : 16
NEW_SYS_NAME                            : IAPX386_PM
PAGE_TERMINATOR                         : ASCII.FF
RECORD_DEFINITION                       : RECORD NULL;END RECORD;
RECORD_NAME                             : NO_SUCH_MACHINE_CODE_TYPE
TASK_SIZE                               : 32
TASK_STORAGE_SIZE                       : 1024
TICK                                    : 0.000_000_062_5
VARIABLE_ADDRESS                        : (16#0#,16#4c#)
VARIABLE_ADDRESS1                       : (16#4#,16#4c#)
VARIABLE_ADDRESS2                       : (16#8#,16#4c#)
YOUR_PRAGMA                             : EXPORT_OBJECT
```

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.

QUALIFIER	DESCRIPTION
/CHECK	Generates run-time constraint checks.
/NOCHECK	
/CONFIGURATION_FILE	Specifies the file used by the compiler.
/DEBUG	Includes symbolic debugging in program library.
/NODEBUG	Does not include symbolic information.
/EXCEPTION_TABLES	Includes/excludes exception handler
/NOEXCEPTION_TABLES	tables from the generated code.
/LIBRARY	Specifies program library used.
/LIST	Writes a source listing on the list file.
/NOLIST	
/OPTIMIZE	Specifies compiler optimization.
/NOOPTIMIZE	
/PROGESS	Displays compiler progress.
/NOPROGRESS	
/SAVE_SOURCE	Copies source to program library.
/NOSAVE_SOURCE	
/TARGET_DEBUG	Includes Intel debug information.
/NOTARGET_DEBUG	Does not include Intel debug information.
/XREF	Creates a cross reference listing.
/NOXREF	
/UNIT	Assigns a specific unit number to the compilation (must be free and in a sublibrary).

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.

QUALIFIER	DESCRIPTION
/OPTIONS	Specifies target link options.
/LIBRARY	The library used in the link.
/LOG	Specifies creation of a log file.
/NOLOG	
/ROOT_EXTRACT	Using non-DDC-I units in the root library.
/NOROOT_EXTRACT	
<unit-name>	Main program to be linked.
[<recompilation-spec>]	Hypothetical recompilation units.
/DEBUG	Links an application for use with
/NODEBUG	the DACS-80x86 Cross Debugger.
/RTS	Includes or excludes the run-time system.
/NORTS	
/NPX	Use of the 80x87 numeric coprocessor.
/NONPX	
/TASK	Maximum number of tasks or non-tasking
/NOTASKS	application.
/PRIORITY	Default task priority.
/TIME_SLICE	Task time slicing.
/NOTIME_SLICE	
/TIMER	Timer resolution.
/RESERVE_STACK	Size of reserve stack.
/NORESERVE_STACK	
/LT_STACK_SIZE	Library task default stack size.
/LT_SEGMENT_SIZE	Library task default segment size.
/MP_STACK_SIZE	Main program stack size.
/MP_SEGMENT_SIZE	Main program segment size.
/SEARCHLIB	Target libraries or object modules to include in target link.
/STOP_BEFORE_LINK	Performs Ada link only.
/TASK_STORAGE_SIZE	Tasks default storage size.
/INTERRUPT_ENTRY_TABLE	Range of interrupt entries.
/ENABLE_TASK_TRACE	Enables trace when a task terminates in unhandled exception.
/SIZE_OPTIMIZE	Forces the linker to remove units that are
/NOSIZE_OPTIMIZE	not used.
/FLEX	Uses the flexible linker to define the
/NOFLEX	target system link environment.

The qualifiers listed below are only recognized when /FLEX is specified: (/FLEX is default; to avoid FLEX-linking, use /NOFLEX)

QUALIFIER	DESCRIPTION
/EXTRACT /NOEXTRACT	Extracts Ada Object modules.
/ELAB /NOELAB	Generates elaboration code.
/UCD /NOUCD	Generates User Configurable Data.
/TEMPLATE	Specifies template file.

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 SHORT_INTEGER is range -32_768 .. 32_767;

type INTEGER is range -2_147_483_648 .. 2_147_483_647;

type LONG_INTEGER is range
       -16#8000_0000_0000_0000# .. 16#7FFF_FFFF_FFFF_FFFF#;

type FLOAT is digits 6
     range -16#0.FFFF_FF#E32 .. 16#0.FFFF_FF#E32;

type LONG_FLOAT is digits 15
     range -16#0.FFFF_FFFF_FFFF_F8#E256 ..
   16#0.FFFF_FFFF_FFFF_F8#E256;
type DURATION is delta 2#1.0#E-14 range -131_072.0 .. 131_071.0;

end STANDARD;
```

This appendix describes the implementation-dependent characteristics of DACS-80X86® as required in Appendix F of the Ada Reference Manual (ANSI/MIL-STD-1815A).

F.1 Implementation-Dependent Pragmas

This section describes all implementation defined pragmas.

F.1.1 Pragma INTERFACE_SPELLING

This pragma allows an Ada program to call a non-Ada program whose name contains characters that would be an invalid Ada subprogram identifier. This pragma must be used in conjunction with pragma INTERFACE, i.e., pragma INTERFACE must be specified for the non-Ada subprogram name prior to using pragma INTERFACE_SPELLING.

The pragma has the format:

```
pragma  INTERFACE_SPELLING  (subprogram name,
                             string literal);
```

where the subprogram name is that of one previously given in pragma INTERFACE and the string literal is the exact spelling of the interfaced subprogram in its native language. This pragma is only required when the subprogram name contains invalid characters for Ada identifiers.

Example:

```
function  RTS_GetDataSegment  return Integer;

pragma  INTERFACE    (ASM86, RTS_GetDataSegment);
pragma  INTERFACE_SPELLING  (RTS_GetDataSegment,
                             "R1SMGS?GetDataSegment");
```

F.1.2 Pragma LT_SEGMENT_SIZE

This pragma sets the size of a library task stack segment. The pragma has the format:

```
pragma LT_SEGMENT_SIZE (T, N);
```

where T denotes either a task object or task type and N designates the size of the library task stack segment in words.

The library task's stack segment defaults to the size of the library task stack. The size of the library task stack is normally specified via the representation clause (note that T must be a task type)

```
for T'STORAGE_SIZE use N;
```

The size of the library task stack segment determines how many tasks can be created which are nested within the library task. All tasks created within a library task will have their stacks allocated from the same segment as the library task stack. Thus, pragma LT_SEGMENT_SIZE must be specified to reserve space within the library task stack segment so that nested tasks' stacks may be allocated.

The following restrictions are places on the use of LT_SEGMENT_SIZE:

1) It must be used only for library tasks.

2) It must be placed immediately after the task object or type name declaration.

3) The library task stack segment size (N) must be greater than or equal to the library task stack size.

F.1.3 Pragma EXTERNAL_NAME

F.1.3.1 Function

The pragma EXTERNAL_NAME is designed to make permanent Ada objects and subprograms externally available using names supplied by the user.

F.1.3.2 Format

The format of the pragma is:

pragma EXTERNAL_NAME(<ada_entity>,<external name>)

where <ada_entity> should be the name of:

- a permanent object, i.e. an object placed in the permanent pool of the compilation unit - such objects originate in package specifications and bodies only,

- a constant object, i.e. an object placed in the constant pool of the compilation unit - please note that scalar constants are embedded in the code, and composite constants are not always placed in the constant pool, because the constant is not considered constant by the compiler,

- a subprogram name, i.e. a name of a subprogram **defined** in this compilation unit - please notice that separate subprogram specifications cannot be used, the code for the subprogram MUST be present in the compilation unit code,

and where the <external name> is a string specifying the external name associated the <ada_entity>. The <external names> should be unique. Specifying identical spellings for different <ada_entities> will generate errors at compile and/or link time, and the responsibility for this is left to the user. Also the user should avoid spellings similar to the spellings generated by the compiler, e.g. E_xxxxx_yyyyy, P_xxxxx, C_xxxxx and other internal identifications. The target debug type information associated with such external names is the null type.

F.1.3.3 Restrictions

Objects that are local variables to subprograms or blocks cannot have external names associated. The entity being made external ("public") MUST be defined in the compilation unit itself. Attempts to name entities from other compilation units will be rejected with a warning.

When an entity is an object the value associated with the symbol will be the relocatable address of the first byte assigned to the object.

F.1.3.4 Example

Consider the following package body fragment:

```
package body example is

    subtype string10 is string(1..10);

    type s is
       record
         len  : integer;
         val  : string10;
       end record;

    global_s  : s;
    const_s   : constant string10 := "1234567890";

    pragma EXTERNAL_NAME(global_s, "GLOBAL_S_OBJECT");
    pragma EXTERNAL_NAME(const_s,  "CONST_S");

    procedure handle(...) is
    ...
    end handle;

    pragma EXTERNAL_NAME(handle, "HANDLE_PROC");

    ...

end example;
```

The objects GLOBAL_S and CONST_S will have associated the names "GLOBAL_S_OBJECT" and "CONST_S". The procedure HANDLE is now also known as "HANDLE_PROC". It is allowable to assign more than one external name to an Ada entity.

F.1.3.5 Object Layouts

Scalar objects are laid out as described in Chapter 9. For arrays the object is described by the address of the first element; the array constraint(s) are NOT passed, and therefore it is recommended only to use arrays with known constraints. Non-discriminated records take a consecutive number of bytes, whereas discriminated records may contain pointers to the heap. Such complex objects should be made externally visible, only if the user has thorough knowledge about the layout.

F.1.3.6 Parameter Passing

The following section describes briefly the fundamentals regarding parameter passing in connection with Ada subprograms. For more detail, refer to Chapter 9.

Scalar objects are always passed by value. For OUT or IN OUT scalars, code is generated to move the modified scalar to its destination. In this case the stack space for parameters is not removed by the procedure itself, but by the caller.

Composite objects are passed by reference. Records are passed via the address of the first byte of the record. Constrained arrays are passed via the address of the first byte (plus a bitoffset when a packed array). Unconstrained arrays are passed as constrained arrays plus a pointer to the constraints for each index in the array. These constraints consist of lower and upper bounds, plus the size in words or bits of each element depending if the value is positive or negative respectively. The user should study an appropriate disassembler listing to thoroughly understand the compiler calling conventions.

A function (which can only have IN parameters) returns its result in register(s). Scalar results are registers/float registers only; composite results leave an address in some registers and the rest, if any, are placed on the stack top. The stack still contains the parameters in this case (since the function result is likely to be on the stack), so the caller must restore the stack pointer to a suitable value, when the function call is dealt with. Again, disassemblies may guide the user to see how a particular function call is to be handled.

F.1.4 Pragma INTERRUPT_HANDLER

This pragma will cause the compiler to generate fast interrupt handler entries instead of the normal task calls for the entries in the task in which it is specified. It has the format:

```
pragma  INTERRUPT_HANDLER;
```

The pragma must appear as the first thing in the specification of the task object. The task must be specified in a package and not a procedure. See Section F.6.2.3 for more details and restrictions on specifying address clauses for task entries.

F.2 Implementation-Dependent Attributes

No implementation-dependent attributes are defined.

F.3 Package SYSTEM

The specifications of package SYSTEM for all DACS-80x86 in Real Address Mode and DACS-80286PM systems are identical except that type Name and constant System_Name vary:

Compiler System	System_Name
DACS-8086	iAPX86
DACS-80186	iAPX186
DACS-80286 Real Mode	iAPX286
DACS-80286 Protected Mode	iAPX286_PM
DACS-80386 Real Mode	iAPX386

Below is package system for DACS-8086.

```
package System is

   type    Word          is new Integer;
   type    DWord         is new Long_integer;

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

   type    byte is range 0..255;
   for     byte'SIZE use 8;

   subtype SegmentId     is UnsignedWord;

   type    Address   is
      record
         offset  : UnsignedWord;
         segment : SegmentId;
      end record;

   subtype Priority  is Integer range 0..31;

   type Name      is (iAPX86);

   SYSTEM_NAME    : constant Name := iAPX86;
```

```
   STORAGE_UNIT   : constant       := 16;
   MEMORY_SIZE    : constant       := 1_048_576;
   MIN_INT        : constant       := -2_147_483_647-1;
   MAX_INT        : constant       := 2_147_483_647;
   MAX_DIGITS     : constant       := 15;
   MAX_MANTISSA   : constant       := 31;
   FINE_DELTA     : constant       := 2#1.0#E-31;
   TICK           : constant       := 0.000_000_125;

   type Interface_language is
                  (ASM86,     PLM86,     C86,     C86_REVERSE,
                   ASM_ACF,   PLM_ACF,   C_ACF,   C_REVERSE_ACF,
                   ASM_NOACF, PLM_NOACF, C_NOACF, C_REVERSE_NOACF);

   type ExceptionId  is record
                           unit_number   : UnsignedWord;
                           unique_number : UnsignedWord;
                        end record;

   type TaskValue      is new Integer;
   type AccTaskValue   is access TaskValue;
   type SemaphoreValue is new Integer;

   type Semaphore    is record
                           counter       : Integer;
                           first         : TaskValue;
                           last          : TaskValue;
                           SQNext        : SemaphoreValue;
                                           -- only used in HDS.
                        end record;

   InitSemaphore : constant Semaphore := Semaphore'(1,0,0,0);

end System;
```

The package SYSTEM specification for DACS-80386PM package system is:

```
package System is

   type     Word          is new Short_Integer;
   type     DWord         is new Integer;
   type     QWord         is new Long_Integer;

   type     UnsignedWord is range 0..65535;
   for      UnsignedWord'SIZE use 16;
   type     UnsignedDWord is range 0..16#FFFF_FFFF#;
   for      UnsignedDWord'SIZE use 32;
   type     Byte is range 0..255,
   for      Byte'SIZE use 8;

   subtype SegmentId     is UnsignedWord;

   type     Address   is
      record
         offset  : UnsignedDWord;
         segment : SegmentId;
      end record;


   for      Address   use
      record
         offset  at 0 range 0..31;
         segment at 2 range 0..15;
      end record;


   subtype Priority  is Integer range 0..31;

   type Name      is (iAPX386_PM);

   SYSTEM_NAME    : constant Name := iAPX386_PM;
   STORAGE_UNIT   : constant      := 16;
   MEMORY_SIZE    : constant      := 16#1_0000_0000#;
   MIN_INT        : constant      := -16#8000_0000_0000_0000#;
   MAX_INT        : constant      := 16#7FFF_FFFF_FFFF_FFFF#;
   MAX_DIGITS     : constant      := 15;
   MAX_MANTISSA   : constant      := 31;

   FINE_DELTA     : constant      := 2#1.0#E-31;
   TICK           : constant      := 0.000_000_062_5;

   type Interface_language is
               (ASM86,       PLM86,       C86,      C86_REVERSE,
                ASM_ACF,     PLM_ACF,     C_ACF,    C_REVERSE_ACF,
                ASM_NOACF,   PLM_NOACF,   C_NOACF,  C_REVERSE_NOACF);
```

```
   type ExceptionId  is record
                          unit_number    : UnsignedDWord;
                          unique_number  : UnsignedDWord;
                       end record;

   type TaskValue       is new Integer;
   type AccTaskValue    is access TaskValue;
   type SemaphoreValue  is new Integer;

   type Semaphore    is record
                          counter        : Integer;
                          first, last    : TaskValue;
                          SQNext         : SemaphoreValue;
                                           -- only used in HDS.
                       end record;

   InitSemaphore : constant Semaphore := Semaphore'(1,0,0,0);

end System;
```

F.4 Representation Clauses

The representation clauses that are accepted are described below. Note that representation specifications can be given on derived types too.

Throughout this subsection, references are made to the size of objects. This number may depend on the compiler variant; in such cases two figures are quoted, ie. 16/32. The first figure refers to all versions of DACS-80x86 except DACS-80386 PM, to which tha last figure refers.

F.4.1 Length Clause

Four kinds of length clauses are accepted.

Size specifications:

The size attribute for a type T is accepted in the following cases:

- If T is a discrete type then the specified size must be greater than or equal to the number of bits needed to represent a value of the type, and less than or equal to 16/32. Note that when the number of bits needed to hold any value of the type is calculated, the range is extended to include 0 if necessary, i.e. the range 3..4 cannot be represented in 1 bit, but needs 3 bits.

- If T is a fixed point type, then the specified size must be greater than or equal to the smallest number of bits needed to hold any value of the fixed point type, and less than 16/32 bits. Note that the Reference Manual permits a representation, where the lower bound and the upper bound is not representable in the type. Thus the type

  ```
  type FIX is delta 1.0 range -1.0 .. 7.0;
  ```

  is representable in 3 bits. As for discrete types, the number of bits needed for a fixed point type is calculated using the range of the fixed point type possibly extended to include 0.0.

- If T is a floating point type, an access type or a task type the specified size must be equal to the number of bits used to represent values of the type (floating points: 32 or 64, access types : 32/48 bits and task types : 16/32 bits).

- If T is a record type the specified size must be greater than or equal to the minimal number of bits used to represent values of the type per default.

- If T is an array type the size of the array must be static, i.e. known at compile time and the specified size must be equal to the minimal number of bits used to represent values of the type per default.

Furthermore, the size attribute has only effect if the type is part of a composite type.

```
type BYTE is range 0..255;
for BYTE'size use 8;
SIXTEEN : BYTE                          -- one word allocated
EIGHT : array(1.4) of BYTE              -- one byte per element
```

Collection size specifications:

Using the STORAGE_SIZE attribute on an access type will set an upper limit on the total size of objects allocated in the collection allocated for the access type. If further allocation is attempted, the exception STORAGE_ERROR is raised. The specified storage size must be less than or equal to INTEGER'LAST.

Task storage size :

When the STORAGE_SIZE attribute is given on a task type, the task stack area will be of the specified size.

Small specifications :

Any value of the SMALL attribute less than the specified delta for the fixed point type can be given.

F.4.2 Enumeration Representation Clauses

Enumeration representation clauses may specify representations in the range of -16#7FFF# .. 16#7FFE#. An enumeration representation clause may be combined with a length clause. If an enumeration representation clause has been given for a type the representational values are considered when the number of bits needed to hold any value of the type is evaluated. Thus the type

```
type ENUM is (A,B,C);
for ENUM use (1,3,5);
```

needs 3 bits not 2 bits to represent any value of the type.

F.4.3 Record Representation Clauses

When component clauses are applied to a record type the following restrictions and interpretations are imposed :

- All values of the component type must be representable within the specified number of bits in the component clause.

- If the component type is either a discrete type a fixed point type, or an array type with a discrete type other than LONG_INTEGER, or a fixed point type as element type, then the component is packed into the specified number of bits (see however the restriction in the paragraph above), and the component may start at any bit boundary.

- If the component type is not one of the types specified in the paragraph above, it must start at a storage unit boundary, a storage unit being 16 bits, and the default size calculated by the compiler must be given as the bit width, i.e. the component must be specified as

  ```
  component at N range 0 .. 16 * M-1
  ```

  where N specifies the relative storage unit number (0,1,...) from the beginning of the record, and M the required number of storage units (1,2,...).

- The maximum bit width for components of scalar types is 16/32.

- A record occupies an integral number of storage units (even though a record may have fields that only define an odd number of bytes)

- A record may take up a maximum of 32 Kbits

- If the component type is an array type with a discrete type other than LONG_INTEGER or a fixed point type as element type, the given bit width must be divisible by the length of the array, i.e. each array element will occupy the same number of bits.

If the record type contains components which are not covered by a component clause, they are allocated consecutively after the component with the value. Allocation of a record component without a component clause is always aligned on a storage unit boundary. Holes created because of component clauses are not otherwise utilized by the compiler.

F.4.3.1 Alignment Clauses

Alignment clauses for records are implemented with the following characteristics:

- If the declaration of the record type is done at the outermost level in a library package, any alignment is accepted.

- If the record declaration is done at a given static lev6~[5~igher than the outermost library level, i.e., the permanent area), only word alignments are accepted.

- Any record object declared at the outermost level in a library package will be aligned according to the alignment clause specified for the type. Record objects declared elsewhere can only be aligned on a word boundary. If the record type is associated with a different alignment, an error message will be issued.

- If a record type with an associated alignment clause is used in a composite type, the alignment is required to be one word; an error message is issued if this is not the case.

F.5 Implementation-Dependent Names for Implementation -Dependent Components

None defined by the compiler.

F.6 Address Clauses

This section describes the implementation of address clauses and what types of entities may have their address specified by the user.

F.6.1 Objects

Address clauses are supported for scalar and composite objects whose size can be determined at compile time. The address value must be static. The given address is the virtual address.

F.6.2 Task Entries

The implementation supports two methods to equate a task entry to a hardware interrupt through an address clause:

1) Direct transfer of control to a task accept statement when an interrupt occurs bypassing the DMS/OS kernel. This form requires the use of pragma INTERRUPT_HANDLER. These handlers are called fast interrupt handlers.

2) Mapping of a signal onto a normal conditional entry call. This form allows the interrupt entry to be called from other tasks (without special actions), as well as being called when a signal occurs.

F.6.2.1 Fast Interrupt Tasks

Directly transferring control to an accept statement when an interrupt occurs requires the implementation dependent pragma INTERRUPT_HANDLER to tell the compiler that the task is an interrupt handler.

F.6.2.2 Features

Fast interrupt tasks provide the following features:

1) Provide the fastest possible response time to an interrupt.

2) Allow entry calls to other tasks during interrupt servicing.

3) Allow procedure and function calls during interrupt servicing.

4) Does not require its own stack to be allocated.

5) Can be coded in packages with other declarations so that desired visiblity to appropriate parts of the program can be achieved.

6) May have multiple accept statements in a single fast interrupt task, each mapped to a different interrupt. If more than one interrupt is to be serviced by a single fast interrupt task, the accept statements should simply be coded consecutively. See example 2 to show how this is done. Note that no code outside the accept statements will ever be executed.

F.6.2.3 Limitations

By using the fast interrupt feature, the user is agreeing to place certain restrictions on the task in order to speed up the software response to the interrupt. Consequently, use of this method to capture interrupts is much faster than the normal method.

The following limitations are placed on a fast interrupt task:

1) It must be a task object, not a task type.

2) The pragma must appear first in the specification of the task object.

3) All entries of the task object must be single entries (no families) with no parameters.

4) The entries must not be called from **any** task.

5) The body of the task must not contain any statements outside the accept statement(s). A loop statement may be used to enclose the accept(s), but this is meaningless because no code outside the accept statements will be executed.

6) The task may make **one** entry call to another task for every handled interrupt, but the call must be single and parameterless and must be made to a **normal** tasks, not another fast interrupt task.

7) The task may only reference **global** variables; no data local to the task may be defined.

8) The task must be declared in a library package, i.e., at the outermost level of some package.

9) Explicit saving of NPX state must be performed by the user within the accept statement if such state saving is required.

F.6.2.4 Making Entry Calls to Other Tasks

Fast interrupt tasks can make entry calls to other normal tasks as long as the entries are single (no indexes) and parameterless.

If such an entry call is made and there is a possibility of the normal task not being ready to accept the call, the entry call can be queued to the normal task's entry queue. This can be forced by using the normal Ada conditional entry call construct shown below:

```
accept E do
    select
        T.E;
    else
        null;
    end select;
end E;
```

Normally, this code sequence means make the call and if the task is not waiting to accept it immediately, cancel the call and continue. In the context of a fast interrupt task, however, the semantics of this construct are modified slightly to force the queuing of the entry call.

If an unconditional entry call is made and the called task is not waiting at the corresponding accept statement, then the interrupt task will **wait** at the entry call. Alternatively, if a timed entry call is made and the called task does not accept the call before the delay expires, then the call will be **dropped.** The conditional entry call is the **preferred** method of making task entry calls from fast interrupt handlers because it allows the interrupt service routine to complete straight through and it guarantees queueing of the entry call if the called task is not waiting.

When using this method, make sure that the interrupt is included in the /INTERRUPT_ENTRY_TABLE specified at link time. See Section 7.2.15 for more details.

F.6.2.5 Implementation of Fast Interrupts

Fast interrupt tasks are not actually implemented as true Ada tasks. Rather, they can be viewed as procedures that consist of code simply waiting to be executed when an interrupt occurs. They do not have a state, priority, or a task control block associated with them, and are not scheduled to "run" by the run-time system.

Since a fast interrupt handler is not really a task, to code it in a loop of somekind is meaningless because the task will never loop; it will simply execute the body of the accept statement whenever the interrupt occurs. However, a loop construct could make the source code more easily understood and has no side effects except for the generation of the executable code to implement to loop construct.

F.6.2.6 Flow of Control

When an interrupt occurs, control of the CPU is transferred directly to the accept statement of the task. This means that the appropriate slot in the interrupt vector table is modified to contain the address of the corresponding fast interrupt accept statement.

Associated with the code for the accept statement is

at the very beginning:
code that saves registers

at the very end:
code that restores registers followed by an IRET instruction.

Note that if the interrupt handler makes an entry call to another task, the interrupt handler is completed through the IRET before the rendezvous is actually completed. After the rendezvous completes, normal Ada task priority rules will be obeyed, and a task context switch may occur.

Normally, the interrupting device must be reenabled by receiving End-Of-Interrupt messages. These can be sent from machine code insertion statements as demonstrated in Example 7.

F.6.2.7 Saving NPX State

If the interrupt handler will perform floating point calculations and the state of the NPX must be saved because other tasks also use the numeric coprocessor, calls to the appropriate

save/restore routines must be made in the statement list of the accept statement. These routines are located in package RTS_Entry_Points and are called RTS_Store_NPX_State and RTS_Restore_NPX_State. See example 6 for more information.

F.6.2.8 Storage Used

This section details the storage requirements of fast interrupt handlers.

F.6.2.9 Stack Space

A fast interrupt handler executes off the stack of the task executing at the time of the interrupt. Since a fast interrupt handler is not a task it does not have its own stack.

Since no local data or parameters are permitted, use of stack space is limited to procedure and function calls from within the interrupt handler.

F.6.2.10 Run-Time System Data

No task control block (TCB) is created for a fast interrupt handler.

If the fast interrupt handler makes a task entry call, an entry in the _CD_INTERRUPT_VECTOR must be made to allocate storage for the queuing mechanism. This table is a run-time system data structure used for queuing interrupts to normal tasks. Each entry is only 10 words for 80386 protected mode compilers and 5 words for all other compiler systems. This table is created by the linker and is constrained by the user through the linker qualifier /INTERRUPT_ENTRY_TABLE. For more information, see Section F.6.2.1 on linking an application with fast interrupts.

If the state of the NPX is saved by user code (see Section F.6.2.7), it is done so in the NPX save area of the TCB of the task executing at the time of the interrupt. This is appropriate because it is that task whose NPX state is being saved.

F.6.3 Building an Application with Fast Interrupt Tasks

This section describes certain steps that must be followed to build an application using one or more fast interrupt handlers.

F.6.3.1 Source Code

The pragma INTERRUPT_HANDLER which indicates that the interrupt handler is the fast form of interrupt handling and not the normal type, must be placed in the task specification as the first statement.

When specifying an address clause for a fast interrupt handler, the offset should be the **interrupt number**, not the offset of the interrupt in the interrupt vector. The segment is not applicable (although a zero value must be specified) as it is not used by the compiler for interrupt addresses. The compiler will place the interrupt vector into the INTERRUPTVECTORTABLE segment. For real address mode programs, the interrupt vector must always be in segment 0 at execution time (see *). For protected mode programs, the user specifies the interrupt vector location at build time.

Calls to RTS_Store_NPX_State and RTS_Restore_NPX_State must be included if the state of the numeric coprocessor must be saved when the fast interrupt occrus. These routines are located in package RTS_EntryPoints in the root library. See example 6 for more information.

F.6.3.2 Compiling the Program

No special compilation options are required.

F.6.3.3 Linking the Program

Since fast interrupt tasks are not real tasks, they do **not** have to be accounted for when using the /TASKS qualifier at link time. In fact, if there are no normal tasks in the application, the program can be linked **without** /TASKS.

This also means that the linker options /LT_STACK_SIZE, /LT_SEGMENT_SIZE, /MP_SEGMENT_SIZE, and /TASK_STORAGE_SIZE do not apply to fast interrupt tasks, except to note that a fast interrupt task will execute off the stack of the task running at the time of the interrupt.

* This placement can be accomplished at locate time by specifying the address to locate the INTERRUPTVECTORTABLE segments with the LOC86 command, or at run time, by having the startup code routine of the UCC copy down the INTERRUPTVECTORTABLE segment to segment 0.

If an entry call is made by a fast interrupt handler the interrupt number must be included in the /INTERRUPT_ENTRY_TABLE qualifier at link time. This qualifier builds a table in the run-time system data segment to handle entry calls of interrupt handlers. The table is indexed by the interrupt number, which is bounded by the low and high interrupt numbers specified at link time.

F.6.3.4 Locating/Building the Program

For real-address mode programs, no special actions need be performed at locate-time; the compiler creates the appropriate entry in the INTERRUPTVECTORTABLE segment. This segment must be at segment 0 before the first interrupt can occur.

For protected mode programs, an interrupt gate must be added and a table entry must be added to the interrupt descriptor table (IDT) for each interrupt serviced by a fast interrupt handler. The value to be put into the build file is the address of the routine that is to be vectored to when the interrupt occurs. This is specified by a label produced by the code generator which can be discerned by disassembling the package specification that contains the interrupt task. For an interrupt handler servicing decimal interrupt 10, the label would be _AIH_00010. "AIH" means Ada Interrupt Handler. An example of creating a fast interrupt handler for a protected mode program is shown in example 5.

F.6.4 Examples

These examples illustrate how to write fast interrupt tasks and then how to build the application using the fast interrupt tasks.

F.6.4.1 Example 1

This example shows how to code a fast interrupt handler that does not make any task entry calls, but simply performs some interrupt handling code in the accept body.

Ada source:

```
with  System;
package  P  is

          <potentially other declarations>

     task  Fast_Interrupt_Handler  is
         pragma INTERRUPT_HANDLER;
         entry  E;
         for E use at (segment => 0, offset => 10);
     end;

          <potentially other declarations>

end  P;

package body  P  is

          <potentially other declarations>

     task body  Fast_Interrupt_Handler  is
     begin
          accept  E  do
              <handle interrupt>
          end  E;
     end;

          <potentially other declarations>

end  P;


with  P;
procedure  Example_1  is
begin
   <main program>
end  Example_1;
```

Compilation and Linking:

```
$ ada       Example_1
$ ada/link  Example_1       !  Note: no other tasks in the
                            !        system in this example
```

F.6.4.2 Example 2

This example shows how to write a fast interrupt handler that services more than one interrupt.

Ada source:

```
with  System;
package  P  is

     task  Fast_Interrupt_Handler  is
         pragma INTERRUPT_HANDLER;

         entry  E1;
         entry  E2;
         entry  E3;

         for E1 use at (segment => 0, offset => 5);
         for E2 use at (segment => 0, offset => 9);
         for E3 use at (segment => 0, offset => 11);

       end;

end  P;

package body  P  is

     task body  Fast_Interrupt_Handler  is
     begin
         accept  E1  do
            <service interrupt 5>
         end  E1;

         accept  E2  do
            <service interrupt 9>
         end  E2;

         accept  E3  do
            <service interrupt 11>
         end  E3;
       end;

 end  P;
```

Compilation and Linking:

```
$ ada  Example_2
$ ada/link/tasks  Example_2    !  assumes application also
                               !  has  normal  tasks  (not
                               ! shown)
```

F.6.4.3 **Example 3**

This example shows how to access global data and make a procedure call from within a fast interrupt handler.

Ada source:

```
with  System;
package  P  is

     A  :  Integer;

     task  Fast_Interrupt_Handler  is
         pragma INTERRUPT_HANDLER;
         entry  E;
         for E use at (segment => 0, offset => 16#127#);
     end;

end  P;

package body  P  is

     B : Integer;

     procedure  P  (X : in out Integer) is
     begin
         X  := X + 1;
     end;

     task body  Fast_Interrupt_Handler  is
     begin
          accept  E  do
              A := A + B;
              P (A);
          end  E;
     end;

 end  P;
```

Compilation and Linking:

```
$ ada       Example_3
$ ada/link  Example_3
```

F.6.4.4 Example 4

This example shows how to make a task entry call and force it to be queued if the called task is not waiting at the accept at the time of the call.

Note that the application is linked with /TASKS=2, where the tasks are T and the main program. Since the fast interrupt handler is making an entry call to T, the techniques used guarantee that it will be queued, if necessary. This is accomplished by using the conditional call construct in the accept body of the fast interrupt handler and by including the interrupt in the /INTERRUPT_ENTRY_TABLE at link time.

Ada source:

```
with  System;
package  P  is

     task  Fast_Interrupt_Handler  is
         pragma INTERRUPT_HANDLER;
         entry  E;
         for E use at (segment => 0, offset => 8);
     end;

     task  T  is
         entry  E;
     end;

end  P;

package body  P  is

     task body  Fast_Interrupt_Handler  is
     begin
          accept  E  do
              select
                  T.E;
              else
                  null;
              end select;
          end  E;
     end;
```

```
        task body  T  is
        begin
            loop
               select
                  accept  E;
               or
                  delay 3.0;
               end select;
            end loop;
        end;

    end  P;
```

Compilation and Linking:

```
$ ada  Example_4
$ ada/link/tasks=2/interrupt_entry_table=(8,8)  Example_4
```

F.6.4.5 Example 5

This example shows how to build an application for 80386 protected mode programs using fast interrupt handlers.

For protected mode programs, special entries must be made in the build file to modify the interrupt vector.

Ada source:

```
with  System;
package  P  is

     task  Fast_Interrupt_Handler  is
         pragma INTERRUPT_HANDLER;
         entry  E;
         for E use at (segment => 0, offset => 17);
     end;

end  P;

package body  P  is

     task body  Fast_Interrupt_Handler  is
     begin
         accept  E  do
            null;
         end  E;
     end;

 end  P;
```

Build File (partial):

```
gate
    D1HWIN?NMIhandler              (interrupt,dpl=0),
    D1HWIN?SingleStepInt           (interrupt,dpl=0),
    D1HWIN?Breakpoint              (interrupt,dpl=0),
    D1HWIN?InvalidOpcode           (interrupt,dpl=0),
    D1HWIN?DevNotAvailable         (interrupt,dpl=0),
    D1HWIN?DoubleFault             (interrupt,dpl=0),
    D1HWIN?SegOverRun              (interrupt,dpl=0),
    D1HWIN?InvalidTSS              (interrupt,dpl=0),
    D1HWIN?SegmentFault            (interrupt,dpl=0),
    D1HWIN?StackFault              (interrupt,dpl=0),
    D1HWIN?ProtFault               (interrupt,dpl=0),
    D1TINT?TimerInterrupt          (interrupt,dpl=0),
    D1IPUT?Transmit                (interrupt,dpl=0),
    D1IGET?Receive                 (interrupt,dpl=0),
    R1EHNE?RaiseNumericError       (interrupt,dpl=0),
    R1EHCE?RaiseConstraintError    (interrupt,dpl=0),
---> _AIH_00017                    (interrupt,dpl=0);


table
  IDT(                                                  -- Vector Id
  entry =  (0:   R1EHNE?RaiseNumericError,              -- 0
            1:   D1HWIN?SingleStepInt,                  -- 1
            2:   D1HWIN?NMIhandler,                     -- 2
            3:   D1HWIN?Breakpoint,                     -- 3
            4:   R1EHNE?RaiseNumericError,              -- 4
            5:   R1EHCE?RaiseConstraintError,           -- 5
            6:   D1HWIN?InvalidOpCode,                  -- 6
            7:   D1HWIN?DevNotAvailable,                -- 7
            8:   D1HWIN?DoubleFault,                    -- 8
            9:   D1HWIN?SegOverRun,                     -- 9
            10:  D1HWIN?InvalidTSS,                     --10
            11:  D1HWIN?SegmentFault,                   --11
            12:  D1HWIN?stackFault,                     --12
            13:  D1HWIN?ProtFault,                      --13
            16:  R1EHNE?RaiseNumericError,              --16
       ---> 17:  _AIH_00017,                            --17
            80h: D1TINT?TimerInterrupt,                 --128
            86h: D1IGET?Receive,                        --134
            87h: D1IPUT?Transmit));                     --135
                 end
```

Compilation, Linking, and Building:

```
$ ada  Example_5
$ ada/link/tasks  Example_5
$ bld386/build=Example.BLD  Example_5.OBJ
```

F.6.4.6 Example 6

This example shows how to save and restore the state of the numeric coprocessor from within a fast interrupt handler. This would be required if other tasks are using the coprocessor to perform floating point calculations and the fast interrupt handler also will use the coprocessor.

Note that the state of the NPX is saved in the task control block of the task executing at the time of the interrupt.

Ada source:

```
with  System;
package  P  is

     task  Fast_Interrupt_Handler  is
         pragma INTERRUPT_HANDLER;
         entry  E;
         for E use at (segment => 0, offset => 25);
     end;

end  P;

with  RTS_EntryPoints;
package body  P  is

     task body  Fast_Interrupt_Handler  is
     begin
         accept  E  do
            RTS_EntryPoints.Store_NPX_State;

            <user code>

            RTS_EntryPoints.Restore_NPX_State;
         end  E;
     end;

 end  P;
```

Compilation and Linking:

```
$ ada  Example_6
$ ada/link/npx/tasks  Example_6
```

F.6.4.7 **Example 7**

This example shows how to send an End-Of-Interrupt message as the last step in servicing the interrupt.

Ada source:

```
with  System;
package  P  is

     task  Fast_Interrupt_Handler  is
         pragma INTERRUPT_HANDLER;
         entry  E;
         for E use at (segment => 0, offset => 5);
     end;

end  P;

with  Machine_Code;   use  Machine_Code;
package body  P  is

  procedure Send_EOI is
  begin
    machine_instruction'
            (register_immediate, m_MOV, AL, 16#66#);
    machine_instruction'
            (immediate_register, m_OUT, 16#0e0#, AL);
  end;
  pragma inline (Send_EOI);

     task body  Fast_Interrupt_Handler  is
     begin
         accept  E  do
            <user code>
            Send_EOI;
         end  E;
     end;

 end  P;
```

Compilation and Linking:

```
$ ada  Example_7
$ ada/link/tasks  Example_7
```

F.6.5 Normal Interrupt Tasks

"Normal" interrupt tasks are the standard method of servicing interrupts. In this case the interrupt causes a conditional entry call to be made to a normal task.

F.6.5.1 Features

Normal interrupt tasks provide the following features:

1) Local data may be defined and used by the interrupt task.

2) May be called by other tasks with no restrictions.

3) Can call other normal tasks with no restrictions.

4) May be declared anywhere in the Ada program where a normal task declaration is allowed.

F.6.5.2 Limitations

Mapping of an interrupt onto a normal conditional entry call puts the following constraints on the involved entries and tasks:

1) The affected entries must be defined in a task object only, not a task type.

2) The entries must be single and parameterless.

F.6.5.3 Implementation of Normal Interrupt Tasks

Normal interrupt tasks are standard Ada tasks. The task is given a priority and runs as any other task, obeying the normal priority rules and any time-slice as configured by the user.

F.6.5.4 Flow of Control

When an interrupt occurs, control of the CPU is transferred to an interrupt service routine generated by the specification of the interrupt task. This routine preserves the registers and calls the run-time system, where the appropriate interrupt task and entry are determined from the information in the _CD_INTERRUPT_VECTOR table and a conditional entry call is made.

If the interrupt task is waiting at the accept statement that corresponds to the interrupt, then the interrupt task is scheduled for execution upon return from the interrupt service routine and the call to the run-time system is completed. The interrupt service routine will execute an IRET, which reenables interrupts, and execution will continue with the interrupt task.

If the interrupt task is not waiting at the accept statement that corresponds to the interrupt, and the interrupt task is not in the body of the accept statement that corresponds to the interrupt, then the entry call is automatically **queued** to the task, and the call to the run-time system is completed.

If the interrupt task is not waiting at the accept statement that corresponds to the interrupt, and the interrupt task is executing in the body of the accept statement that corresponds to the interrupt, then the interrupt service routine will NOT complete until the interrupt task has exited the body of the accept statement. During this period, the interrupt will not be serviced, and execution in the accept body will continue with interrupts **disabled.** Users are cautioned that if from within the body of the accept statement corresponding to an interrupt, an unconditional entry call is made, a delay statement is executed, or some other non-deterministic action is invoked, the result will be erratic and will cause non-deterministic interrupt response.

Example 4 shows how End-Of-Interrupt messages may be sent to the interrupting device.

F.6.5.5 Saving NPX State

Because normal interrupt tasks are standard tasks, the state of the NPX numeric coprocessor is saved automatically by the run-time system when the task executes. Therefore, no special actions are necessary by the user to save the state.

F.6.5.6 Storage Used

This section describes the storage requirements of standard interrupt tasks.

F.6.5.7 Stack Space

A normal interrupt task is allocated its own stack and executes off that stack while servicing an interrupt. See the appropriate sections of this User's Guide on how to set task stack sizes.

F.6.5.8 Run-Time System Data

A task control block is allocated for each normal interrupt task via the /TASKS qualifier at link time.

During task elaboration, an entry is made in the run-time system _CD_INTERRUPT_VECTOR table to "define" the standard interrupt. This mechanism is used by the run-time system to make the conditional entry call when the interrupt occurs. This means that the user is responsible to include **all** interrupts serviced by normal interrupt tasks in the /INTERRUPT_ENTRY_TABLE qualifier at link time.

F.6.6 Building an Application with Normal Interrupt Tasks

This section describes how to build an application that uses standard Ada tasks to service interrupts.

F.6.6.1 Source Code

No special pragmas or other such directives are required to specify that a task is a normal interrupt task. If it contains interrupt entries, then it is a normal interrupt task by default.

When specifying an address clause for a normal interrupt handler, the offset should be the **interrupt number**, not the offset of the interrupt in the interrupt vector. The segment is not applicable (although some value must be specified) because it is not used by the compiler for interrupt addresses. The compiler will place the interrupt vector into the INTERRUPTVECTORTABLE segment. For real address mode programs, the interrupt vector must always be in segment 0 at execution time. This placement can be accomplished by specifying the address to locate the INTERRUPTVECTORTABLE segment with the loc86 command, or at run

time, by having the startup code routine of the UCC copy down the INTERRUPTVECTORTAPLE segment to segment 0 and the compiler will put it there automatically. For protected mode programs, the user specifies the interrupt vector location at build time.

F.6.6.2 Compiling the Program

No special compilation options are required.

F.6.6.3 Linking the Program

The interrupt task must be included in the /TASKS qualifier. The link options /LT_STACK_SIZE, /LT_SEGMENT_SIZE, /MP_SEGMENT_SIZE, and /TASK_STORAGE_SIZE apply to normal interrupt tasks and must be set to appropriate values for your application.

Every normal interrupt task **must** be accounted for in the /INTERRUPT_ENTRY_TABLE qualifier. This qualifier causes a table to be built in the run-time system data segment to handle interrupt entries. In the case of standard interrupt tasks, this table is used to map the interrupt onto a normal conditional entry call to another task.

F.6.6.4 Locating/Building the Program

For real-address mode programs, no special actions need be performed at locate-time; the compiler creates the appropriate entry in the INTERRUPTVECTORTABLE segment. This segment must be located at segment 0 before the occurrence of the first interrupt.

For protected mode programs, an interrupt gate must be added and a table entry must be added to the interrupt descriptor table (IDT) for each interrupt serviced by a fast interrupt handler. The value to be put into the build file is the place in the code that is to be vectored to when the interrupt occurs. This is specified by a label produced by the code generator which can be discerned by disassembling the package specification that contains the interrupt task. For an interrupt handler servicing decimal interrupt 12, the label would be _AIH_00012. "AIH" means Ada Interrupt Handler. An example of creating a normal interrupt handler for a protected mode program is shown in example 3.

F.6.7 Examples

These examples illustrate how to write normal interrupt tasks and then how to build the application using them.

F.6.7.1 Example 1

This example shows how to code a simple normal interrupt handler.

Ada source:

```
with  System;
package  P  is

     task  Normal_Interrupt_Handler  is
         entry  E;
         for E use at (segment => 0, offset => 10);
     end;

end  P;

package body  P  is

     task body  Normal_Interrupt_Handler  is
     begin
          accept  E  do
              <handle interrupt>
          end  E;
     end;

end  P;

with  P;
procedure  Example_1  is
begin
   <main program>
end  Example_1;
```

Compilation and Linking:

```
$ ada  Example_1
$ ada/link/tasks=2/interrupt_entry_table=(10,10) Example_1
```

F.6.7.2 Example 2

This example shows how to write a normal interrupt handler that services more than one interrupt and has other standard task entries.

Ada source:

```
with  System;
package  P  is

     task  Normal_Task  is

         entry  E1;
         entry  E2;          --  standard entry
         entry  E3;

         for E1 use at (segment => 0, offset => 7);
         for E3 use at (segment => 0, offset => 9);

       end;

end  P;

package body  P  is

     task body  Normal_Task  is
     begin
         loop
             select
                 accept  E1  do
                    <service interrupt 7>
                 end  E1;
             or
                 accept  E2  do
                    <standard rendezvous>
                 end  E2;
             or
                 accept  E3  do
                    <service interrupt 9>
                 end  E3;
             end select;
         end loop;
     end  Normal_Task;

 end  P;
```

Compilation and Linking:

```
$ ada  Example_2
$ ada/link/tasks/interrupt_entry_table=(7,9)  Example_2
```

F.6.7.3 **Example 3**

This example shows how to build an application for 80386 protected mode programs using normal interrupt handlers.

For protected mode programs, special entries must be made in the build file to modify the interrupt vector.

Ada source:

```
with  System;
package  P  is

     task  Normal_Interrupt_Handler  is
         entry  E;
         for E use at (segment => 0, offset => 20);
     end;

end  P;

package body  P  is

     task body  Normal_Interrupt_Handler  is
     begin
         accept  E  do
            null;
         end  E;
     end;

 end  P;
```

Build File (partial):

```
gate
    D1HWIN?NMIhandler              (interrupt,dpl=0),
    D1HWIN?SingleStepInt           (interrupt,dpl=0),
    D1HWIN?Breakpoint              (interrupt,dpl=0),
    D1HWIN?InvalidOpcode           (interrupt,dpl=0),
    D1HWIN?DevNotAvailable         (interrupt,dpl=0),
    D1HWIN?DoubleFault             (interrupt,dpl=0),
    D1HWIN?SegOverRun              (interrupt,dpl=0),
    D1HWIN?InvalidTSS              (interrupt,dpl=0),
    D1HWIN?SegmentFault            (interrupt,dpl=0),
    D1HWIN?StackFault              (interrupt,dpl=0),
    D1HWIN?ProtFault               (interrupt,dpl=0),
    D1TINT?TimerInterrupt          (interrupt,dpl=0),
    D1IPUT?Transmit                (interrupt,dpl=0),
    D1IGET?Receive                 (interrupt,dpl=0),
    R1EHNE?RaiseNumericError       (interrupt,dpl=0),
    R1EHCE?RaiseConstraintError    (interrupt,dpl=0),
---> _AIH_00020                    (interrupt,dpl=0);
```

```
table
  IDT(                                                    -- Vector Id
     entry =  (0:   R1EHNE?RaiseNumericError,             -- 0
               1:   D1HWIN?SingleStepInt,                 -- 1
               2:   D1HWIN?NMIhandler,                    -- 2
               3:   D1HWIN?Breakpoint,                    -- 3
               4:   R1EHNE?RaiseNumericError,             -- 4
               5:   R1EHCE?RaiseConstraintError,          -- 5
               6:   D1HWIN?InvalidOpCode,                 -- 6
               7:   D1HWIN?DevNotAvailable,               -- 7
               8:   D1HWIN?DoubleFault,                   -- 8
               9:   D1HWIN?SegOverRun,                    -- 9
               10:  D1HWIN?InvalidTSS,                    --10
               11:  D1HWIN?SegmentFault,                  --11
               12:  D1HWIN?stackFault,                    --12
               13:  D1HWIN?ProtFault,                     --13
               16:  R1EHNE?RaiseNumericError,             --16
      --->     20:  _AIH_00020,                           --20
               80h: D1TINT?TimerInterrupt,                --128
               86h: D1IGET?Receive,                       --134
               87h: D1IPUT?Transmit));                    --135
               end
```

Compilation, Linking, and Building:

```
$ ada  Example_3
$ ada/link/tasks/interrupt_entry_Table=(20,20)  Example_3
$ bld386/build=Example.BLD  Example_3.OBJ
```

F.6.7.4 Example 4

This example shows how an End-Of-Interrupt message may be sent to the interrupting device.

Ada source:

```
with  System;
package  P  is

     task  Normal_Interrupt_Handler  is
         entry  E;
         for E use at (segment => 0, offset => 7);
     end;

end  P;

with  Machine_Code;   use  Machine_Code;
package body  P  is
```


```
procedure Send_EOI is
begin
  machine_instruction'
          (register_immediate, m_MOV, AL, 16#66#);
  machine_instruction'
          (immediate_register, m_OUT, 16#0e0#, AL);
end;
pragma inline (Send_EOI);

  task body  Normal_Interrupt_Handler  is
   begin
       accept  E  do
          <user code>
          Send_EOI;
       end  E;
   end;

end  P;
```

Compilation and Linking:

```
$ ada  Example_4
$ ada/link/tasks/interrupt_entry_table=(7,7)  Example_4
```

F.6.8 Interrupt Queuing

DDC-I provides a useful feature that allows task entry calls made by interrupt handlers (fast and normal variant) to be queued if the called task is not waiting to accept the call, enabling the interrupt handler to complete to the IRET. What may not be clear is that the same interrupt may be queued only once at any given time in DDC-I's implementation. We have made this choice for two reasons:

a) Queuing does not come for free, and queuing an interrupt more than once is considerably more expensive than queuing just one. DDC-I feels that most customers prefer their interrupt handlers to be as fast as possible and that we have chosen an implementation that balances performance with functionality.

b) In most applications, if the servicing of an interrupt is not performed in a relatively short period of time, there is an unacceptable and potentially dangerous situation. Queuing the same interrupt more than once represents this situation.

Note that this note refers to queuing of the **same** interrupt more than once at the **same** time. Different interrupts may be queued at the same time as well as the same interrupt may be queued in a sequential manner as long as there is never a situation where the queuing overlaps in time.

If it is acceptable for your application to queue the same interrupt more than once, it is a relatively simple procedure to implement the mechanism yourself. Simply implement a high priority agent task that is called from the interrupt handler. The agent task accepts calls from the interrupt task and makes the call on behalf of the interrupt handler to the originally called task. By careful design, the agent task can be made to accept all calls from the interrupt task when they are made, but at the very least, must guarantee that at most one will be queued at a time.

F.6.9 Recurrence of Interrupts

DDC-I recommends the following techniques to ensure that an interrupt is completely handled before the same interrupt recurs. There are two cases to consider, i.e. the case of fast interrupt handlers and the case of normal interrupt handlers.

F.6.9.1 Fast Interrupt Handler

If the fast interrupt handler makes an entry call to a normal task, then place the code that reenables the interrupt at the end of the accept body of the **called** task. When this is done, the interrupt will not be reenabled before the rendezvous is actually completed between the fast interrupt handler and the called task even if the call was queued. Note that the interrupt task executes all the way through the IRET **before** the rendezvous is completed if the entry call was queued.

Normally, end-of-interrupt code using Low_Level_IO will be present in the accept body of the **fast interrupt handler**. This implies that the end-of-interrupt code will be executed **before** the rendezvous is completed, possibly allowing the interrupt to come in again before the application is ready to handle it.

If the fast interrupt handler **does not** make an entry call tc another task, then placing the end-of-interrupt code in the accept body of the fast interrupt task will guarantee that the interrupt is completely serviced before another interrupt happens.

F.6.9.2 Normal Interrupt Handler

Place the code that reenables the interrupt at the end of the accept body of the **normal interrupt** task. When this is done, the interrupt will not be reenabled before the rendezvous is actually completed between the normal interrupt handler and the called task even if the call was queued. Even though the interrupt "completes" in the sense that the IRET is executed, the interrupt is not yet reenabled because the rendezvous with the normal task's interrupt entry has not been made.

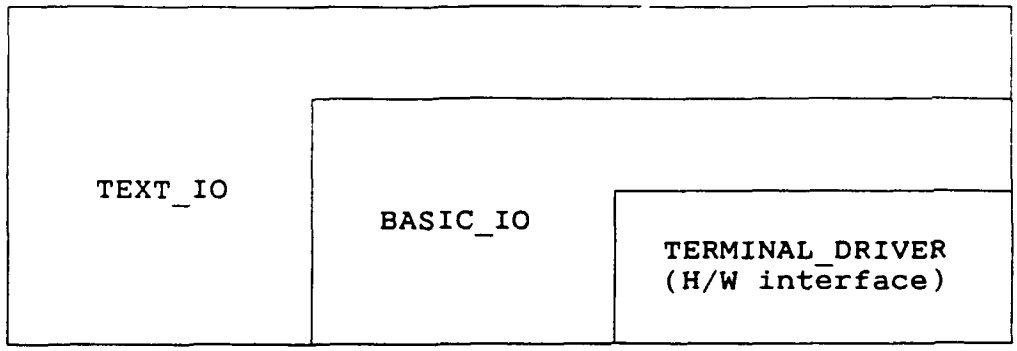
If these techniques are used for **either** variant of interrupt handlers, caution must be taken that other tasks do not call the task entry which reenables interrupts if this can cause adverse side effects.

F.7 Unchecked Conversion

Unchecked conversion is only allowed between objects of the same "size". However, if scalar type has different sizes (packed and unpacked), unchecked conversion between such a type and another type is accepted if either the packed or the unpacked size fits the other type.

F.8 Input/Output Packages

In many embedded systems, there is no need for a traditional I/O system, but in order to support testing and validation, DDC-I has developed a small terminal oriented I/O system. This I/O system consists essentially of TEXT_IO adapted with respect to handling only a terminal and not file I/O (file I/O will cause a USE error to be raised) and a low level package called TERMINAL_DRIVER. A BASIC_IO package has been provided for convenience purposes, forming an interface between TEXT_IO and TERMINAL_DRIVER as illustrated in the following figure.



The TERMINAL_DRIVER package is the only package that is target dependent, i.e., it is the only package that need be changed when changing communications controllers. The actual body of the TERMINAL_DRIVER is written in assembly language, but an Ada interface to this body is provided. A user can also call the terminal driver routines directly, i.e. from an assembly language routine. TEXT_IO and BASIC_IO are written completely in Ada and need not be changed.

BASIC_IO provides a mapping between TEXT_IO control characters and ASCII as follows:

TEXT_IO	ASCII Character
LINE_TERMINATOR	ASCII.CR
PAGE_TERMINATOR	ASCII.FF
FILE_TERMINATOR	ASCII.EM (CTRL/Z)
NEW_LINE	ASCII.LF

The services provided by the terminal driver are:

1) Reading a character from the communications port.

2) Writing a character to the communications port.

.1 Package TEXT_IO

specification of package TEXT_IO:

```
gma page;
ı BASIC_IO;

ı IO_EXCEPTIONS;
tage TEXT_IO is

:ype FILE_TYPE is limited private;

:ype FILE_MODE is (IN_FILE, OUT_FILE);

:ype    COUNT is range 0 .. INTEGER'LAST;
subtype POSITIVE_COUNT is COUNT range 1 .. COUNT'LAST;
INBOUNDED: constant COUNT:= 0; -- line and page length

· max. size of an integer output field 2#....#
subtype FIELD          is INTEGER range 0 .. 35;

ubtype NUMBER_BASE     is INTEGER range 2 .. 16;

:ype TYPE_SET is (LOWER_CASE, UPPER_CASE);

ma PAGE;
- File Management

rocedure CREATE (FILE  : in out FILE_TYPE;
                 MODE  : in     FILE_MODE :=OUT_FILE;
                 NAME  : in     STRING    :="";
                 FORM  : in     STRING    :=""
                );

procedure OPEN  (FILE  : in out FILE_TYPE;
                 MODE  : in     FILE_MODE;
                 NAME  : in     STRING;
                 FORM  : in     STRING    :=""
                );

procedure CLOSE (FILE  : in out FILE_TYPE);
procedure DELETE (FILE : in out FILE_TYPE);
procedure RESET (FILE  : in out FILE_TYPE;
                 MODE  : in FILE_MODE);
procedure RESET (FILE  : in out FILE_TYPE);

function MODE   (FILE  : in FILE_TYPE) return FILE_MODE;
function NAME   (FILE  : in FILE_TYPE) return STRING;
function FORM   (FILE  : in FILE_TYPE) return STRING;

function IS_OPEN(FILE  : in FILE_TYPE return BOOLEAN;
```

```
pragma PAGE;
    -- control of default input and output files

    procedure SET_INPUT  (FILE : in FILE_TYPE);
    procedure SET_OUTPUT (FILE : in FILE_TYPE);

    function STANDARD_INPUT   return FILE_TYPE;
    function STANDARD_OUTPUT  return FILE_TYPE;

    function CURRENT_INPUT    return FILE_TYPE;
    function CURRENT_OUTPUT   return FILE_TYPE;


pragma PAGE;
   -- specification of line and page lengths

   procedure SET_LINE_LENGTH  (FILE : in FILE_TYPE;
                               TO : in COUNT);
   procedure SET_LINE_LENGTH  (TO : in COUNT);

   procedure SET_PAGE_LENGTH  (FILE : in FILE_TYPE;
                               TO : in COUNT);
   procedure SET_PAGE_LENGTH  (TO : in COUNT);

   function  LINE_LENGTH  (FILE : in FILE_TYPE)
                          return CCUNT;
   function  LINE_LENGTH   return COUNT;

   function  PAGE_LENGTH   (FILE : in FILE_TYPE)
                          return COUNT;
   function  PAGE_LENGTH   return COUNT;


pragma PAGE;
   -- Column, Line, and Page Control

   procedure NEW_LINE (FILE : in FILE_TYPE;
                       SPACING : in POSITIVE_COUNT := 1);
   procedure NEW_LINE (SPACING : in POSITIVE_COUNT := 1);

   procedure SKIP_LINE (FILE : in FILE_TYPE;
                        SPACING : in POSITIVE_COUNT := 1);
   procedure SKIP_LINE (SPACING : in POSITIVE_COUNT := 1);

   function  END_OF_LINE (FILE : in FILE_TYPE) return BOOLEAN;
   function  END_OF_LINE                       return BOOLEAN;

   procedure NEW_PAGE    (FILE : in FILE_TYPE);
   procedure NEW_PAGE;

   procedure SKIP_PAGE   (FILE : in FILE_TYPE);
   procedure SKIP_PAGE;
```

```
   function  END_OF_PAGE (FILE : in FILE_TYPE) return BOOLEAN;
   function  END_OF_PAGE                       return BOOLEAN;

   function  END_OF_FILE (FILE : in FILE_TYPE) return BOOLEAN;
   function  END_OF_FILE                       return BOOLEAN;

   procedure SET_COL     (FILE : in FILE_TYPE;
                          TO   : in POSITIVE_COUNT);
   procedure SET_COL     (TO : in POSITIVE_COUNT);

   procedure SET_LINE    (FILE : in FILE_TYPE;
                          TO : in POSITIVE_COUNT);
   procedure SET_LINE    (TO : in POSITIVE_COUNT);

   function  COL         (FILE : in FILE_TYPE)
                         return POSITIVE_COUNT;
   function  COL         return POSITIVE_COUNT;

   function  LINE        (FILE : in FILE_TYPE)
                         return POSITIVE_COUNT;
   function  LINE        return POSITIVE_COUNT;

   function  PAGE        (FILE : in FILE_TYPE)
                         return POSITIVE_COUNT;
   function  PAGE        return POSITIVE_COUNT;

pragma PAGE;
   -- Character Input-Output

   procedure GET  (FILE : in FILE_TYPE; ITEM : out CHARACTER);
   procedure GET  (                     ITEM : out CHARACTER);
   procedure PUT  (FILE : in FILE_TYPE; ITEM :  in CHARACTER);
   procedure PUT  (                     ITEM :  in CHARACTER);

   -- String Input-Output

   procedure GET  (FILE : in FILE_TYPE; ITEM : out CHARACTER);
   procedure GET  (                     ITEM : out CHARACTER);
   procedure PUT  (FILE : in FILE_TYPE; ITEM :  in CHARACTER);
   procedure PUT  (                     ITEM :  in CHARACTER);

   procedure GET_LINE   (FILE : in FILE_TYPE;
                         ITEM : out STRING;
                         LAST : out NATURAL);

   procedure GET_LINE   (ITEM : out STRING;
                         LAST : out NATURAL);

   procedure PUT_LINE   (FILE : in FILE_TYPE;
                         ITEM :  in STRING);
   procedure PUT_LINE   (ITEM :  in STRING);
```

```
pragma PAGE;
   -- Generic Package for Input-Output of Integer Types

  generic
      type NUM is range <>;
  package INTEGER_IO is

     DEFAULT_WIDTH : FIELD        := NUM'WIDTH;
     DEFAULT_BASE  : NUMBER_BASE  :=        10;

     procedure GET  (FILE  : in FILE_TYPE;
                     ITEM  : out NUM;
                     WIDTH : in FIELD := 0);
     procedure GET  (ITEM  : out NUM;
                     WIDTH : in FIELD := 0);

     procedure PUT  (FILE  : in FILE_TYPE;
                     ITEM  :  in NUM;
                     WIDTH :  in FIELD := DEFAULT_WIDTH;
                     BASE  :  in NUMBER_BASE := DEFAULT_BASE);
     procedure PUT  (ITEM  :  in NUM;
                     WIDTH :  in FIELD := DEFAULT_WIDTH;
                     BASE  :  in NUMBER_BASE := DEFAULT_BASE);

     procedure GET  (FROM  : in STRING;
                     ITEM  : out NUM;
                     LAST  : out POSITIVE);

     procedure PUT  (TO    : out STRING;
                     ITEM  :  in NUM;
                     BASE  :  in NUMBER_BASE := DEFAULT_BASE);

  end INTEGER_IO;


pragma PAGE;
```

```
   -- Generic Packages for Input-Output of Real Types

generic
   type NUM is digits <>;
package FLOAT_IO is

   DEFAULT_FORE : FIELD :=               2;
   DEFAULT_AFT  : FIELD := NUM'DIGITS - 1;
   DEFAULT_EXP  : FIELD :=               3;

   procedure GET  (FILE  : in FILE_TYPE;
                   ITEM  : out NUM;
                   WIDTH : in FIELD := 0);
   procedure GET  (ITEM  : out NUM;
                   WIDTH : in FIELD := 0);

   procedure PUT  (FILE : in FILE_TYPE;
                   ITEM : in NUM;
                   FORE : in FIELD := DEFAULT_FORE;
                   AFT  : in FIELD := DEFAULT_AFT;
                   EXP  : in FIELD := DEFAULT_EXP);
   procedure PUT  (ITEM : in NUM;
                   FORE : in FIELD := DEFAULT_FORE;
                   AFT  : in FIELD := DEFAULT_AFT;
                   EXP  : in FIELD := DEFAULT_EXP);

   procedure GET  (FROM : in STRING;
                   ITEM : out NUM;
                   LAST : out POSITIVE);
   procedure PUT  (TO   : out STRING;
                   ITEM :  in NUM;
                   AFT  :  in FIELD := DEFAULT_AFT;
                   EXP  :  in FIELD := DEFAULT_EXP);

end FLOAT_IO;


pragma PAGE;
```

```
generic
   type NUM is delta <>;
package FIXED_IO is

   DEFAULT_FORE : FIELD := NUM'FORE;
   DEFAULT_AFT  : FIELD := NUM'AFT;
   DEFAULT_EXP  : FIELD := 0;

   procedure GET  (FILE  : in FILE_TYPE;
                   ITEM  : out NUM;
                   WIDTH : in FIELD := 0);
   procedure GET  (ITEM  : out NUM;
                   WIDTH : in FIELD := 0);

   procedure PUT  (FILE : in FILE_TYPE;
                   ITEM : in NUM;
                   FORE : in FIELD := DEFAULT_FORE;
                   AFT  : in FIELD := DEFAULT_AFT;
                   EXP  : in FIELD := DEFAULT_EXP);

   procedure PUT  (ITEM : in NUM;
                   FORE : in FIELD := DEFAULT_FORE;
                   AFT  : in FIELD := DEFAULT_AFT;
                   EXP  : in FIELD := DEFAULT_EXP);

   procedure GET  (FROM : in STRING;
                   ITEM : out NUM;
                   LAST : out POSITIVE);

   procedure PUT  (TO   : out STRING;
                   ITEM :  in NUM;
                   AFT  :  in FIELD := DEFAULT_AFT;
                   EXP  :  in FIELD := DEFAULT_EXP);

end FIXED_IO;


pragma PAGE;
```

```
   -- Generic Package for Input-Output of Enumeration Types

generic
   type ENUM is (<>);
package ENUMERATION_IO is

   DEFAULT_WIDTH   : FIELD    := 0;
   DEFAULT_SETTING : TYPE_SET := UPPER_CASE;

   procedure GET  (FILE : in FILE_TYPE; ITEM : out ENUM);
   procedure GET  (                     ITEM : out ENUM);

   procedure PUT  (FILE  : FILE_TYPE;
                   ITEM  : in ENUM;
                   WIDTH : in FIELD       := DEFAULT_WIDTH;
                   SET   : in TYPE_SET    := DEFAULT_SETTING);
   procedure PUT  (ITEM  : in ENUM;
                   WIDTH : in FIELD       := DEFAULT_WIDTH;
                   SET   : in TYPE_SET    := DEFAULT_SETTING);

   procedure GET  (FROM : in  STRING;
                   ITEM : out ENUM;
                   LAST : out POSITIVE);

   procedure PUT  (TO   : out STRING;
                   ITEM : in  ENUM;
                   SET  : in  TYPE_SET := DEFAULT_SETTING);

end ENUMERATION_IO;

pragma PAGE;

   -- Exceptions

   STATUS_ERROR : exception renames IO_EXCEPTIONS.STATUS_ERROR;
   MODE_ERROR   : exception renames IO_EXCEPTIONS.MODE_ERROR;
   NAME_ERROR   : exception renames IO_EXCEPTIONS.NAME_ERROR;
   USE_ERROR    : exception renames IO_EXCEPTIONS.USE_ERROR;
   DEVICE_ERROR : exception renames IO_EXCEPTIONS.DEVICE_ERROR;
   END_ERROR    : exception renames IO_EXCEPTIONS.END_ERROR;
   DATA_ERROR   : exception renames IO_EXCEPTIONS.DATA_ERROR;
   LAYOUT_ERROR : exception renames IO_EXCEPTIONS.LAYOUT_ERROR;

pragma page;
private

   type FILE_TYPE is
       record
          FT : INTEGER := -1;
       end record;

end TEXT_IO;
```

F.8.2 Package IO_EXCEPTIONS

The specification of the package IO_EXCEPTIONS:

```
package IO_EXCEPTIONS is

   STATUS_ERROR : exception;
   MODE_ERROR   : exception;
   NAME_ERROR   : exception;
   USE_ERROR    : exception;
   DEVICE_ERROR : exception;
   END_ERROR    : exception;
   DATA_ERROR   : exception;
   LAYOUT_ERROR : exception;

end IO_EXCEPTIONS;
```

F.8.3 Package BASIC_IO

The specification of package BASIC_IO:

```
with IO_EXCEPTIONS;

package BASIC_IO is

   type count is range 0 .. integer'last;

   subtype positive_count is count range 1 .. count'last;


   function get_integer return string;

   -- Skips any leading blanks, line terminators or page
   -- terminators. Then reads a plus or a minus sign if
   -- present, then reads according to the syntax of an
   -- integer literal, which may be based.  Stores in item
   -- a string containing an optional sign and an integer
   -- literal.
   --
   -- The exception DATA_ERROR is raised if the sequence
   -- of characters does not correspond to the syntax
   -- described above.
   --
   -- The exception END_ERROR is raised if the file terminator
   -- is read.  This means that the starting sequence of an
   -- integer has not been met.
   --
   -- Note that the character terminating the operation must
   -- be available for the next get operation.
   --

   function get_real return string;

   -- Corresponds to get_integer except that it reads according
   -- to the syntax of a real literal, which may be based.


   function get_enumeration return string;

   -- Corresponds to get_integer except that it reads according
   -- to the syntax of an identifier, where upper and lower
   -- case letters are equivalent to a character literal
   -- including the apostrophes.
```

```
function get_item (length : in     integer) return string;

-- Reads a string from the current line and stores it in
-- item.  If the remaining number of characters on the
-- current line is less than length then only these
-- characters are returned.  The line terminator is not
-- skipped.


procedure put_item (item : in     string);

-- If the length of the string is greater than the current
-- maximum line (linelength),  the exception LAYOUT_ERROR
-- is raised.
--
-- If the string does not fit on the current line a line
-- terminator is output, then the item is output.


-- Line and page lengths - ARM 14.3.3.
--

procedure set_line_length (to   : in     count);


procedure set_page_length (to   : in     count);


function line_length return count;


function page_length return count;


-- Operations on columns, lines and pages - ARM 14.3.4.
--

procedure new_line;


procedure skip_line;


function end_of_line return boolean;


procedure new_page;


procedure skip_page;


function end_of_page return boolean;
```

```
   function end_of_file return boolean;

   procedure set_col (to   : in     positive_count);

   procedure set_line (to   : in     positive_count);

   function col return positive_count;

   function line return positive_count;

   function page return positive_count;


   -- Character and string procedures.
   -- Corresponds to the procedures defined in ARM 14.3.6.
   --

   procedure get_character (item :    out character);

   procedure get_string (item :    out string);

   procedure get_line (item :    out string;
                       last :    out natural);

   procedure put_character (item : in     character);

   procedure put_string (item : in     string);

   procedure put_line (item : in     string);


  --  exceptions:

  USE_ERROR    : exception renames IO_EXCEPTIONS.USE_ERROR;
  DEVICE_ERROR : exception renames IO_EXCEPTIONS.DEVICE_ERROR;
  END_ERROR    : exception renames IO_EXCEPTIONS.END_ERROR;
  DATA_ERROR   : exception renames IO_EXCEPTIONS.DATA_ERROR;
  LAYOUT_ERROR : exception renames IO_EXCEPTIONS.LAYOUT_ERROR;

end BASIC_IO;
```

F.8.4 Package LOW_LEVEL_IO

The specification of LOW_LEVEL_IO (16 bits) is:

```
with System;

package LOW_LEVEL_IO is

  subtype port_address is System.UnsignedWord;

  type ll_io_8      is new integer range -128..127;
  type ll_io_16     is new integer;

  procedure send_control(device : in port_address;
                         data   : in System.Byte);
       -- unsigned  8 bit entity

  procedure send_control(device : in port_address;
                         data   : in System.UnsignedWord);
       -- unsigned 16 bit entity

  procedure send_control(device : in port_address;
                         data   : in ll_io_8);
       -- signed  8 bit entity

  procedure send_control(device : in port_address;
                         data   : in ll_io_16);
       -- signed 16 bit entity

  procedure receive_control(device : in port_address;
                            data   : out System.Byte);
       -- unsigned  8 bit entity

  procedure receive_control(device : in port_address;
                            data   : out System.UnsignedWord);
       -- unsigned 16 bit entity

  procedure receive_control(device : in port_address;
                            data   : out ll_io_8);
       -- signed  8 bit entity

  procedure receive_control(device : in port_address;
                            data   : out ll_io_16);
       -- signed 16 bit entity

  private

     pragma inline(send_control, receive_control);

end LOW_LEVEL_IO;
```

The specification of LOW_LEVEL_IO (32 bits) is:

```
with SYSTEM;

package LOW_LEVEL_IO is

  subtype port_address is System.UnsignedWord;

  type ll_io_8      is new short_integer range -128..127;
  type ll_io_16     is new short_integer;
  type ll_io_32     is new integer;

  procedure send_control(device : in port_address;
                         data   : in System.Byte);
       -- unsigned  8 bit entity

  procedure send_control(device : in port_address;
                         data   : in System.UnsignedWord);
       -- unsigned 16 bit entity

  procedure send_control(device : in port_address;
                         data   : in System.UnsignedDWord);
       -- unsigned 32 bit entity

  procedure send_control(device : in port_address;
                         data   : in ll_io_8);
       -- signed  8 bit entity

  procedure send_control(device : in port_address;
                         data   : in ll_io_16);
       -- signed 16 bit entity

  procedure send_control(device : in port_address;
                         data   : in ll_io_32);
       -- signed 32 bit entity

  procedure receive_control(device : in port_address;
                            data   : out System.Byte);
       -- unsigned  8 bit entity

  procedure receive_control(device : in port_address;
                            data   : out System.UnsignedWord);
       -- unsigned 16 bit entity

  procedure receive_control(device : in port_address;
                            data   : out System.UnsignedDWord);
       -- unsigned 32 bit entity

  procedure receive_control(device : in port_address;
                            data   : out ll_io_8);
       -- signed  8 bit entity
```

```
  procedure receive_control(device : in port_address;
                            data   : out ll_io_16);
       -- signed 16 bit entity

  procedure receive_control(device : in port_address;
                            data   : out ll_io_32);
       -- signed 32 bit entity


  private

     pragma inline(send_control, receive_control);

end LOW_LEVEL_IO;
```

F.8.5 Package TERMINAL DRIVER

The specification of package TERMINAL_DRIVER:

```
package TERMINAL_DRIVER is

  procedure put_character (ch :  in  character);

  procedure get_character (ch : out  character);

private

  pragma interface (ASM86, put_character);
  pragma interface_spelling(put_character,"D1IPUT?put_character");

  pragma interface (ASM86, get_character);
  pragma interface_spelling(get_character,"D1IGET?get_character");

end TERMINAL_DRIVER;
```

F.8.6 Package SEQUENTIAL_IO

```
-- Source code for SEQUENTIAL_IO

pragma PAGE;

with IO_EXCEPTIONS;

generic

   type ELEMENT_TYPE is private;

package SEQUENTIAL_IO is

  type FILE_TYPE is limited private;

  type FILE_MODE is (IN_FILE, OUT_FILE);


pragma PAGE;
-- File management

   procedure CREATE(FILE : in out FILE_TYPE;
                    MODE : in     FILE_MODE := OUT_FILE;
                    NAME : in     STRING    := "";
                    FORM : in     STRING    := "");

   procedure OPEN  (FILE : in out FILE_TYPE;
                    MODE : in     FILE_MODE;
                    NAME : in     STRING;
                    FORM : in     STRING := "");

   procedure CLOSE (FILE : in out FILE_TYPE);

   procedure DELETE(FILE : in out FILE_TYPE);

   procedure RESET (FILE : in out FILE_TYPE;
                    MODE : in     FILE_MODE);

   procedure RESET (FILE : in out FILE_TYPE);

   function MODE   (FILE : in FILE_TYPE) return FILE_MODE;

   function NAME   (FILE : in FILE_TYPE) return STRING;

   function FORM   (FILE : in FILE_TYPE) return STRING;

   function IS_OPEN(FILE : in FILE_TYPE) return BOOLEAN;
```

```
pragma PAGE;
-- input and output operations


   procedure READ  (FILE : in     FILE_TYPE;
                    ITEM :    out ELEMENT_TYPE);

   procedure WRITE (FILE : in FILE_TYPE;
                    ITEM : in ELEMENT_TYPE);

   function END_OF_FILE(FILE : in FILE_TYPE) return BOOLEAN;

pragma PAGE;
-- exceptions


   STATUS_ERROR : exception renames IO_EXCEPTIONS.STATUS_ERROR;
   MODE_ERROR   : exception renames IO_EXCEPTIONS.MODE_ERROR;
   NAME_ERROR   : exception renames IO_EXCEPTIONS.NAME_ERROR;
   USE_ERROR    : exception renames IO_EXCEPTIONS.USE_ERROR;
   DEVICE_ERROR : exception renames IO_EXCEPTIONS.DEVICE_ERROR;
   END_ERROR    : exception renames IO_EXCEPTIONS.END_ERROR;
   DATA_ERROR   : exception renames IO_EXCEPTIONS.DATA_ERROR;

pragma PAGE;
private

   type FILE_TYPE is new INTEGER;

end SEQUENTIAL_IO;
```

F.9 Machine Code Insertions

The reader should be familiar with the code generation strategy and the 80x86 instruction set to fully benefit from this section.

As described in chapter 13.8 of the ARM [DoD 83] it is possible to write procedures containing only code statements using the predefined package MACHINE_CODE. The package MACHINE_CODE defines the type MACHINE_INSTRUCTION which, used as a record aggregate, defines a machine code insertion. The following sections list the type MACHINE_INSTRUCTION and types on which it depends, give the restrictions, and show an example of how to use the package MACHINE_CODE.

F.9.1 Predefined Types for Machine Code Insertions

The following types are defined for use when making machine code insertions (their type declarations are given on the following pages):

```
type  opcode_type
type  operand_type
type  register_type
type  segment_register
type  machine_instruction
```

The type REGISTER_TYPE defines registers. The registers STi describe registers on the floating stack. (ST is the top of the floating stack).

The type MACHINE_INSTRUCTION is a discriminant record type with which every kind of instruction can be described. Symbolic names may be used in the form

```
name'ADDRESS
```

Restrictions as to symbolic names can be found in section F.9.2. It should be mentioned that addresses are specified as 80386 addresses. In case of other targets, the scale factor should be set to "scale_1".

```
type opcode_type is (
                -- 8086 instructions:
        m_AAA,      m_AAD,      m_AAM,      m_AAS,
        m_ADC,      m_ADD,      m_AND,
        m_CALL,     m_CALLN,
        m_CBW,      m_CLC,      m_CLD,      m_CLI,
        m_CMC,      m_CMP,      m_CMPS,     m_CWD,
        m_DAA,      m_DAS,
        m_DEC,      m_DIV,      m_HLT,
        m_IDIV,     m_IMUL,     m_IN,       m_INC,
        m_INT,      m_INTO,     m_IRET,
        m_JA,       m_JAE,      m_JB,       m_JBE,
        m_JC,       m_JCXZ,     m_JE,       m_JG,
        m_JGE,      m_JL,       m_JLE,      m_JNA,
        m_JNAE,     m_JNB,      m_JNBE,     m_JNC,
        m_JNE,      m_JNG,      m_JNGE,     m_JNL,
        m_JNLE,     m_JNO,      m_JNP,      m_JNS,
        m_JNZ,      m_JO,       m_JP,       m_JPE,
        m_JPO,      m_JS,       m_JZ,       m_JMP,
        m_LAHF,     m_LDS,      m_LES,      m_LEA,
        m_LOCK,     m_LODS,
        m_LOOP,     m_LOOPE,    m_LOOPNE,   m_LOOPNZ,
        m_LOOPZ,    m_MOV,      m_MOVS,     m_MUL,
        m_NEG,      m_NOP,      m_NOT,      m_OR,
        m_OUT,      m_POP,      m_POPF,     m_PUSH,
        m_PUSHF,
        m_RCL,      m_RCR,      m_ROL,      m_ROR,
        m_REP,      m_REPE,     m_REPNE,
        m_RET,      m_RETP,     m_RETN,     m_RETNP,
        m_SAHF,
        m_SAL,      m_SAR,      m_SHL,      m_SHR,
        m_SBB,      m_SCAS,
        m_STC,      m_STD,      m_STI,      m_STOS,
        m_SUB,      m_TEST,     m_WAIT,     m_XCHG,
        m_XLAT,     m_XOR,
```

```
-- 8087/80187/80287 Floating Point Processor instructions:

m_FABS,      m_FADD,      m_FADDD,     m_FADDP,
m_FBLD,      m_FBSTP,     m_FCHS,      m_FNCLEX,
m_FCOM,      m_FCOMD,     m_FCOMP,     m_FCOMPD,
m_FCOMPP,    m_FDECSTP,   m_FDIV,      m_FDIVD,
m_FDIVP,     m_FDIVR,     m_FDIVRD,    m_FDIVRP,
m_FFREE,     m_FIADD,     m_FIADDD,    m_FICOM,
m_FICOMD,    m_FICOMP,    m_FICOMPD,   m_FIDIV,
m_FIDIVD,    m_FIDIVR,    m_FIDIVRD,
m_FILD,      m_FILDD,     m_FILDL,     m_FIMUL,
m_FIMULD,    m_FINCSTP,   m_FNINIT,    m_FIST,
m_FISTD,     m_FISTP,     m_FISTPD,    m_FISTPL,
m_FISUB,
m_FISUBD,    m_FISUBR,    m_FISUBRD,   m_FLD,
m_FLDD,      m_FLDCW,     m_FLDENV,    m_FLDLG2,
m_FLDLN2,    m_FLDL2E,    m_FLDL2T,    m_FLDPI,
m_FLDZ,      m_FLD1,      m_FMUL,      m_FMULD,
m_FMULP,     m_FNOP,      m_FPATAN,    m_FPREM,
m_FPTAN,     m_FRNDINT,   m_FRSTOR,    m_FSAVE,
m_FSCALE,    m_FSETPM,    m_FSQRT,
m_FST,       m_FSTD,      m_FSTCW,
m_FSTENV,    m_FSTP,      m_FSTPD,     m_FSTSW,
m_FSTSWAX,   m_FSUB,      m_FSUBD,     m_FSUBP,
m_FSUBR,     m_FSUBRD,    m_FSUBPP,    m_FTST,
m_FWAIT,     m_FXAM,      m_FXCH,      m_FXTRACT,
m_FYL2X,     m_FYL2XP1,   m_F2XM1,

-- 80186/80286/80386 instructions:
-- Notice that some immediate versions of the 8086
-- instructions only exist on these targets
-- (shifts,rotates,push,imul,...)

m_BOUND,     m_CLTS,      m_ENTER,     m_INS,
m_LAR,       m_LEAVE,     m_LGDT,      m_LIDT,
m_LSL,       m_OUTS,      m_POPA,      m_PUSHA,
m_SGDT,      m_SIDT,
m_ARPL,      m_LLDT,      m_LMSW,      m_LTR,
-- 16 bit always...

m_SLDT,      m_SMSW,      m_STR,       m_VERR,
m_VERW,
```

```
-- the 80386 specific instructions:
m_SETA,      m_SETAE,        m_SETB,        m_SETBE,
m_SETC,      m_SETE,         m_SETG,        m_SETGE,
m_SETL,      m_SETLE,        m_SETNA,       m_SETNAE,
m_SETNB,     m_SETNBE,       m_SETNC,       m_SETNE,
m_SETNG,     m_SETNGE,       m_SETNL,       m_SETNLE,
m_SETNO,     m_SETNP,        m_SETNS,       m_SETNZ,
m_SETO,      m_SETP,         m_SETPE,       m_SETPO,
m_SETS,      m_SETZ,
m_BSF,       m_BSR,
m_BT,        m_BTC,          m_BTR,         m_BTS,
m_LFS,       m_LGS,          m_LSS,
m_MOVZX,     m_MOVSX,
m_MOVCR,     m_MOVDB,        m_MOVTR,
m_SHLD,                      m_SHRD,

-- the 80387 specific instructions:
m_FUCOM,     m_FUCOMP,       m_FUCOMPP,
m_FPREM1,    m_FSIN,                        m_FCOS,
m_FSINCOS,

-- byte/word/dword variants (to be used, when not
-- deductible from context):
m_ADCB,      m_ADCW,         m_ADCD,
m_ADDB,      m_ADDW,         m_ADDD,
m_ANDB,      m_ANDW,         m_ANDD,
             m_BTW,          m_BTD,
             m_BTCW,         m_BTCD,
             m_BTRW,         m_BTRD,
             m_BTSW,         m_BTSD,
             m_CBWW,         m_CWDE,
             m_CWDW,         m_CDQ,
m_CMPB,      m_CMPW,         m_CMPD,
m_CMPSB,     m_CMPSW,        m_CMPSD,
m_DECB,      m_DECW,         m_DECD,
m_DIVB,      m_DIVW,         m_DIVD,
m_IDIVB,     m_IDIVW,        m_IDIVD,
m_IMULB,     m_IMULW,        m_IMULD,
m_INCB,      m_INCW,         m_INCD,
m_INSB,      m_INSW,         m_INSD,
m_LODSB,     m_LODSW,        m_LODSD,
m_MOVB,      m_MOVW,         m_MOVD,
m_MOVSB,     m_MOVSW,        m_MOVSD,
m_MOVSXB,    m_MOVSXW,
m_MOVZXB,    m_MOVZXW,
m_MULB,      m_MULW,         m_MULD,
m_NEGB,      m_NEGW,         m_NEGD,
m_NOTB,      m_NOTW,         m_NOTD,
m_ORB,       m_ORW,          m_ORD,
m_OUTSB,     m_OUTSW,        m_OUTSD,
             m_POPW,         m_POPD,
             m_PUSHW,        m_PUSHD,
m_   ,       m_RCLW,         m_RCLD,
```

```
        m_RCRB,      m_RCRW,          m_RCRD,
        m_ROLB,      m_ROLW,          m_ROLD,
        m_RORB,      m_RORW,          m_RORD,
        m_SALB,      m_SALW,          m_SALD,
                     m_SARB,          m_SARW,        m_SARD,
        m_SHLB,      m_SHLW,          m_SHLDW,
        m_SHRB,      m_SHRW,          m_SHRDW,
        m_SBBB,      m_SBBW,          m_SBBD,
        m_SCASB,     m_SCASW,         m_SCASD,
        m_STOSB,     m_STOSW,         m_STOSD,
        m_SUBB,      m_SUBW,          m_SUBD,
        m_TESTB,     m_TESTW,         m_TESTD,
        m_XORB,      m_XORW,          m_XORD,
        m_DATAB,     m_DATAW,         m_DATAD,

        -- Special 'instructions':
                     m_label,         m_reset,

        -- 8087 temp real load/store_and_pop:
        m_FLDT,      m_FSTPT);

pragma page;
        type operand_type is ( none, -- no operands

        immediate,                          -- one immediate operand
        register,                           -- one register operand
        address,                            -- one address operand
        system_address,                     -- one 'address operand
        name,                               -- CALL name
        register_immediate,                 -- two operands :
                                            -- destination is
                                            -- register
                                            -- source is immediate
        register_register,                  -- two register operands
        register_address,                   -- two operands :
                                            -- destination is
                                            -- register
                                            -- source is address
        address_register,                   -- two operands :
                                            -- destination is
                                            -- address
                                            -- source is register
        register_system_address,            -- two operands :
                                            -- destination is
                                            -- register
                                            -- source is 'address
        system_address_register,            -- two operands :
                                            -- destination is
                                            -- 'address
                                            -- source is register
```

```
        address_immediate,                  -- two operands :
                                            -- destination is
                                            -- address
                                            -- source is immediate
        system_address_immediate,           -- two operands :
                                            -- destination is
                                            -- 'address
                                            -- source is immediate
        immediate_register,                 -- only allowed for OUT
                                            -- port is immediate
                                            -- source is register
        immediate_immediate,                -- only allowed for
                                            -- ENTER
        register_register_immediate,        -- allowed for IMULimm,
                                            -- SHRDimm, SHLDimm

        register_address_immediate,         -- allowed for IMULimm
        register_system_address_immediate,  -- allowed for IMULimm
        address_register_immediate,         -- allowed for SHRDimm,
                                            -- SHLDimm
        system_address_register_immediate   -- allowed for SHRDimm,
                                            -- SHLDimm
        );


type register_type is (AX, CX, DX, BX, SP, BP, SI, DI, -- word regs
                       AL, CL, DL, BL, AH, CH, DH, BH, -- byte regs
                       EAX,ECX,EDX,EBX,ESP,EBP,ESI,EDI,-- dword regs
                       ES, CS, SS, DS, FS, GS,         -- selectors
   BX_SI, BX_DI, BP_SI, BP_DI,      -- 8086/80186/80286  combinations
   ST,     ST1,    ST2,    ST3,     -- floating registers (stack)
   ST4,    ST5,    ST6,    ST7,
   nil);

   -- the extended registers (EAX .. EDI) plus FS and GS are only
   -- allowed in 80386 targets

type scale_type is (scale_1, scale_2, scale_4, scale_8);

subtype machine_string is string(1..100);

pragma page;
```

```
type machine_instruction (operand_kind : operand_type) is
   record
      opcode   : opcode_type;

      case operand_kind is
         when immediate =>
            immediate1 : integer;        -- immediate

         when register =>
            r_register : register_type; -- source and/or destination

         when address =>
           a_segment  : register_type; -- source and/or destination
           a_address_base      : register_type;
           a_address_index     : register_type;
           a_address_scale     : scale_type;
           a_address_offset    : integer;

         when system_address =>
            sa_address : system.address; -- destination

        when name =>
           n_string   : machine_string;  -- CALL destination

         when register_immediate =>
            r_i_register_to     : register_type;  -- destination
            r_i_immediate       : integer;        -- source

         when register_register =>
            r_r_register_to     : register_type;  -- destination
            r_r_register_from   : register_type;  -- source

         when register_address =>
            r_a_register_to     : register_type;  -- destination
            r_a_segment         : register_type;  -- source
            r_a_address_base    : register_type;
            r_a_address_index   : register_type;
            r_a_address_scale   : scale_type;
            r_a_address_offset  : integer;

         when address_register =>
            a_r_segment         : register_type;  -- destination
            a_r_address_base    : register_type;
            a_r_address_index   : register_type;
            a_r_address_scale   : scale_type;
            a_r_address_offset  : integer;
            a_r_register_from   : register_type;  -- source

         when register_system_address =>
            r_sa_register_to    : register_type;  -- destination
            r_sa_address        : system.address; -- source

         when system_address_register =>
```

```
      sa_r_address          : system.address; -- destination
      sa_r_reg_from         : register_type;  -- source

   when address_immediate =>
      a_i_segment           : register_type;  -- destination
     a_i_address_base       : register_type;
     a_i_address_index      : register_type;
     a_i_address_scale      : scale_type;
     a_i_address_offset     : integer;
      a_i_immediate         : integer;        -- source

   when system_address_immediate =>
      sa_i_address          : system.address; -- destination
      sa_i_immediate        : integer;        -- source

   when immediate_register =>
      i_r_immediate         : integer;        -- destination
      i_r_register          : register_type;  -- source

   when immediate_immediate =>
      i_i_immediate1        : integer;        -- immediate1
      i_i_immediate2        : integer;        -- immediate2

  when register_register_immediate =>
     r_r_i_register1        : register_type;  -- destination
     r_r_i_register2        : register_type;  -- source1
     r_r_i_immediate        : integer;        -- source2

  when register_address_immediate =>
     r_a_i_register         : register_type;  -- destination
     r_a_i_segment          : register_type;  -- source1
     r_a_i_address_base     : register_type;
     r_a_i_address_index    : register_type;
     r_a_i_address_scale    : scale_type;
     r_a_i_address_offset   : integer;
     r_a_i_immediate        : integer;        -- source2

  when register_system_address_immediate =>
     r_sa_i_register        : register_type;  -- destination
     addr10                 : system.address; -- source1
     r_sa_i_immediate       : integer;        -- source2

  when address_register_immediate =>
     a_r_i_segment          : register_type;  -- destination
     a_r_i_address_base     : register_type;
     a_r_i_address_index    : register_type;
     a_r_i_address_scale    : scale_type;
     a_r_i_address_offset   : integer;
     a_r_i_register         : register_type;  -- source1
     a_r_i_immediate        : integer;        -- source2

  when system_address_register_immediate =>
     sa_r_i_address         : system.address; -- destination
```

```
            sa_r_i_register     : register_type;    -- source1
            sa_r_i_immediate    : integer;          -- source2

         when others =>
            null;
      end case;
   end record;

end machine_code;
```

F.9.2 Restrictions

Only procedures, and not functions, may contain machine code insertions.

Symbolic names in the form x'ADDRESS can only be used in the following cases:

1) x is an object of scalar type or access type declared as an object, a formal parameter, or by static renaming.

2) x is an array with static constraints declared as an object (not as a formal parameter or by renaming).

3) x is a record declared as an object (not a formal parameter or by renaming).

The m_CALL can be used with "name" to call (for) a routine.

Two opcodes to handle labels have been defined:

m_label: defines a label. The label number must be in the range 1 <= x <= 25 and is put in the offset field in the first operand of the MACHINE_INSTRUCTION.

m_reset: used to enable use of more than 25 labels. The label number after a m_RESET must be in the range 1<= x <=25. To avoid errors you must make sure that all used labels have been defined before a reset, since the reset operation clears all used labels.

All floating instructions have at most one operand which can be any of the following:

- a memory address
- a register or an immediate value
- an entry in the floating stack

F.9.3 Examples

The following section contains examples of how to use the machine code insertions and lists the generated code.

F.9.4 Example Using Labels

The following assembler code can be described by machine code insertions as shown:

```
     MOV AX,7
     MOV CX,4
     CMP AX,CX
     JG  1
     JE  2
     MOV CX,AX
  1: ADD AX,CX
  2: MOV SS: [BP+DI], AX
```

```
package example_MC is

    procedure test_labels;
    pragma inline (test_labels);

end example_MC;

with MACHINE_CODE; use MACHINE_CODE;
package body example_MC is


procedure test_labels is

begin

  MACHINE_INSTRUCTION'(register_immediate, m_MOV, AX, 7);
  MACHINE_INSTRUCTION'(register_immediate, m_MOV, CX, 4);
  MACHINE_INSTRUCTION'(register_register,  m_CMP, AX, CX);
  MACHINE_INSTRUCTION'(immediate,          m_JG,  1);
  MACHINE_INSTRUCTION'(immediate,          m_JE,  2);
  MACHINE_INSTRUCTION'(register_register,  m_MOV, CX, AX);
  MACHINE_INSTRUCTION'(immediate,          m_label, 1);
  MACHINE_INSTRUCTION'(register_register,  m_ADD, AX, CX);
  MACHINE_INSTRUCTION'(immediate, m_label, 2);
  MACHINE_INSTRUCTION'(address_register,   m_MOV, SS, BP,
                                           DI, scale_1, 0, AX);

end test_labels;

end example_MC;
```

F.9.5 Advanced Topics

This section describes some of the more intricate details of the workings of the machine code insertion facility. Special attention is paid to the way the Ada objects are referenced in the machine code body, and various alternatives are shown.

F.9.5.1 Address Specifications

Package MACHINE_CODE provides two alternative ways of specifying an address for an instruction. The first way is referred to as SYSTEM_ADDRESS and the parameter associated this one must be specified via OBJECT'ADDRESS in the actual MACHINE_CODE insertion. The second way closely relates to the addressing which the 80x86 machines employ: an address has the general form

```
segment:[base+index*scale+offset]
```

The ADDRESS type expects the machine insertion to contain values for ALL these fields. The default value NIL for segment, base, and index may be selected (however, if base is NIL, so should index be). Scale MUST always be specified as scale_1, scale_2, scale_4, or scale_8. For 16 bit targets, scale_1 is the only legal scale choice. The offset value must be in the range of -32768 .. 32767.

F.9.5.2 Referencing Procedure Parameters

The parameters of the procedure that consists of machine code insertions may be referenced by the machine insertions using the SYSTEM_ADDRESS or ADDRESS formats explained above. However, there is a great difference in the way in which they may be specified; whether the procedure is specified as INLINE or not.

INLINE machine insertions can deal with the parameters (and other visible variables) using the SYSTEM_ADDRESS form. This will be dealt with correctly even if the actual values are constants. Using the ADDRESS form in this context will be the user's responsibility since the user obviously attempts to address using register values obtained via other machine insertions. It is in general not possible to load the address of a parameter because an 'address' is a two component structure (selector and offset), and the only instruction to load an immediate address is the LEA, which will only give the offset. If coding requires access to addresses like this, one cannot INLINE expand the machine insertions. Care should be taken with references to objects outside the current block since the code generator in order to calculate

the proper frame value (using the display in each frame) will apply extra registers. The parameter addresses will, however, be calculated at the entry to the INLINE expanded routine to minimize this problem. INLINE expanded routines should NOT employ any RET instructions.

Pure procedure machine insertions need to know the layout of the parameters presented to, in this case, the called procedure. In particular, careful knowledge about the way parameters are passed is required to achieve a succesful machine procedure. Again there are two alternatives:

The first assumes that the user takes over the responsibility for parameter addressing. With this method, the SYSTEM_ADDRESS format does not make sense (since it expects a procedural setup that is not set up in a machine procedure). The user must code the exit from the procedure and is also responsible for taking off parameters if so is required. The rules of Ada procedure calls must be followed. The calling conventions are summarized below.

The second alternative assumes that a specific abstract A-code insertion is present in the beginning and end of the machine procedure. Abstract A-code insertions are not generally available to an Ada user since they require extensive knowledge about the compiler intermediate text called abstract A-code. Thus, they will not be explained further here except for the below use.

These insertions enable the user to setup the procedural frame as expected by Ada and then allow the form SYSTEM_ADDRESS in accesses to parameters and variables. Again it is required to know the calling conventions to some extent; mainly to the extent that the access method for variables is clear. A record is, for example, transferred via its address, so access to record fields must first employ an LES-instruction and then use ADDRESS form using the read registers.

The insertions to apply in the beginning are:

```
pragma abstract_acode_insertions(true);
 aa_instr'(aa_Create_Block,x,y,0,0,0);
 aa_instr'(aa_End_of_declpart,0,0,0,0,0);
pragma abstract_acode_insertions(false);
```

and at the end:

```
pragma abstract_acode_insertions(true);
 aa_instr'(aa_Exit_subprgrm,x,0,x,nil_arg,nil_arg); -- (1)
 aa_instr'(aa_Set_block_level,y-1,0,0,0,0);
pragma abstract_acode_insertions(false);
```

where the x value represents the number of words taken by the parameters, and y is the lexical block level of the machine procedure. However, if the procedure should leave the parameters on the stack (scalar IN OUT or OUT parameters), then the Exit_subprgrm insertion should read:

```
aa_instr'(aa_Exit_subprgrm,0,0,0,nil_arg,nil_arg); -- (2)
```

In this case, the caller moves the updated scalar values from the stack to their destinations after the call.

The NIL_ARG should be defined as :

```
nil_arg : constant := -32768;
```

WARNING: When using the AA_INSTR insertions, great care must be taken to assure that the x and y values are specified correctly. Failure to do this may lead to unpredictable crashes in compiler pass8.

F.9.5.3 Parameter Transfer

It may be a problem to figure out the correct number of words which the parameters take up on the stack (the x value). The following is a short description of the transfer method:

INTEGER types take up at least 1 storage unit. 32 bit integer types take up 2 words, and 64 bit integer types take up 4 words. In 32 bit targets, 16 bit integer types take up 2 words the low word being the value and the high word being an alignment word. TASKs are transferred as INTEGER.

ENUMERATION types take up as 16 bit INTEGER types (see above).

FLOAT types take up 2 words for 32 bit floats and 4 words for 64 bit floats.

ACCESS types are considered scalar values and consist of a 16 bit segment value and a 16 or 32 bit offset value. When 32 bit offset value, the segment value takes up 2 words the high word being the aligment word. The offset word(s) are the lowest, and the segment word(s) are the highest.

RECORD types are always transferred by address. A record is never a scalar value (so no post-procedure action is carried out when the record parameter is OUT or IN OUT). The representation is as for ACCESS types.

ARRAY values are transferred as one or two ACCESS values. If the array is constrained, only the array data address is transferred in the same manner as an ACCESS value. If the array is unconstrained below, the data address will be pushed by the address of the constraint. In this case, the two ACCESS values will NOT have any alignment words in 32 bit targets.

Packed ARRAY values (e.g. STRING types) are transferred as ARRAY values with the addition of an INTEGER bit offset as the highest word(s):

```
+H: BIT_OFFSET
+L: DATA_ADDRESS
+0: CONSTRAINT_ADDRESS      -- may be missing
```

The values L and H depend on the presence/absence of the constraint address and the sizes of constraint and data addresses.

In the two latter cases, the form parameter'address will always yield the address of the data. If access is required to constraint or bit offset, the instructions must use the ADDRESS form.

F.9.5.4 Example

A small example is shown below (16 bit target):

```
procedure unsigned_add
                (op1   : in     integer;
                 op2   : in     integer;
                 res   :    out integer);
```

Notice that machine subprograms cannot be functions.

The parameters take up:

```
op1     : integer       : 1 word
op2     : integer       : 1 word
res     : integer       : 1 word
Total   :                 3 words
```

The body of the procedure might then be the following assuming that the procedure is defined at outermost package level:

```
procedure unsigned_add
                (op1   : in     integer;
                 op2   : in     integer;
                 res   :    out integer) is
begin
```

```
      pragma abstract_acode_insertions(true);
       aa_instr'(aa_Create_Block,3,1,0,0,0);  -- x = 3, y = 1
       aa_instr'(aa_End_of_declpart,0,0,0,0,0);
      pragma abstract_acode_insertions(false);

      machine_instruction'(register_system_address, m_MOV,
                                 AX, op1'address);
      machine_instruction'(register_system_address, m_ADD,
                                 AX, op2'address);
      machine_instruction'(immediate,               m_JNC,  1);
      machine_instruction'(immediate,               m_INT,  5);
      machine_instruction'(immediate,               m_label,1);
      machine_instruction'(system_address_register, m_MOV,
                                 res'address, AX);

      pragma abstract_acode_insertions(true);
       aa_instr'(aa_Exit_subprgrm,0,0,0,nil_arg,nil_arg);-- (2)
       aa_instr'(aa_Set_block_level,0,0,0,0,0);      -- y-1 = 0
      pragma abstract_acode_insertions(false);
   end unsigned_add;
```

A routine of this complexity is a candidate for INLINE expansion. In this case, no changes to the above 'machine_instruction' statements are required. Please notice that there is a difference between addressing record fields when the routine is INLINE and when it is not:

```
type rec is
   record
      low       : integer;
      high      : integer;
   end record;

procedure add_32 is
         (op1   : in     integer;
          op2   : in     integer;
          res   :    out rec);
```

The parameters take up 1 + 1 + 2 words = 4 words. The RES parameter will be addressed directly when INLINE expanded, i.e. it is possible to write:

```
      machine_instruction'(system_address_register, m_MOV,
                                 res'address, AX);
```

This would, in the not INLINED version, be the same as updating that place on the stack where the address of RES is placed. In this case, the insertion must read:

```
      machine_instruction'(register_system_address, m_LES,
                                SI, res'address);
           -- LES SI,[BP+...]
      machine_instruction'(address_register, m_MOV,
```


```
                                        ES, SI, nil, scale_1, 0, AX);
        -- MOV ES:[SI+0],AX
```

As may be seen, great care must be taken to ensure correct machine code insertions. A help could be to first write the routine in Ada, then disassemble to see the involved addressings, and finally write the machine procedure using the collected knowledge.

Please notice that INLINED machine insertions also generate code for the procedure itself. This code will be removed when the /NOCHECK qualifier is applied to the compilation. Also not INLINED procedures using the AA_INSTR insertion, which is explained above, will automatically get a storage_check call (as do all Ada subprograms). On top of that, 8 bytes are set aside in the created frame, which may freely be used by the routine as temporary space. The 8 bytes are lccated just below the display vector of the frame (from SP and up). The storage_check call will not be generated when the compiler is invoked with /NOCHECK.

The user also has the option NOT to create any blocks at all, but then he should be certain that the return from the routine is made in the proper way (use the RETP instruction (return and pop) or the RET). Again it will help first to do an Ada version and see what the compiler expects to be done.

F.10 Package Tasktypes

The TaskTypes packages defines the TaskControlBlock type. This data structure could be useful in debugging a tasking program. The following package Tasktypes is for all DACS-80x86 Real Address Mode compilers, except for DACS-80386PM.

```
with System;

package TaskTypes is

  subtype Offset    is System.UnsignedWord;
  subtype BlockId   is System.UnsignedWord;

  type    TaskEntry     is new System.UnsignedWord;
  type    EntryIndex    is new System.UnsignedWord;
  type    AlternativeId is new System.UnsignedWord;
  type    Ticks         is new System.DWord;
  type    Bool          is new Boolean;
  for     Bool'Size     use 8;
  type UIntg            is new System.UnsignedWord;

  type TaskState is (Initial,
                     -- The task is created, but activation
                     -- has not started yet.

                     Engaged,
                     -- The task has called an entry, and the
                     -- call is now accepted, ie. the rendezvous
                     -- is in progress.

                     Running,
                     -- Covers all other states.

                     Delayed,
                     -- The task awaits a timeout to expire.

                     EntryCallingTimed,
                     -- The task has called an entry which
                     -- is not yet accepted.

                     EntryCallingUnconditional,
                     -- The task has called an entry
                     -- unconditionally,
                     -- which is not yet accepted.

                     SelectingTimed,
                     -- The task is waiting in a select statement
                     -- with an open delay alternative.

                     SelectingUnconditional,
                     -- The task waits in a select statement
                     -- entirely with accept statements.
```

```
                        SelectingTerminable,
                        -- The task waits in a select statement
                        -- with an open terminate alternative.

                        Accepting,
                        -- The task waits in an accept statement.

                        Synchronizing,
                        -- The task waits in an accept statement
                        -- with no statement list.

                        Completed,
                        -- The task has completed the execution of
                        -- its statement list, but not all dependent
                        -- tasks are terminated.

                        Terminated );
                        -- The task and all its descendants
                        -- are terminated.

for TaskState use (Initial => 16#00# ,
                   Engaged => 16#08# ,
                   Running => 16#10# ,
                   Delayed => 16#18# ,
                   EntryCallingTimed => 16#20# ,
                   EntryCallingUnconditional => 16#28# ,
                   SelectingTimed => 16#31# ,
                   SelectingUnconditional => 16#39# ,
                   SelectingTerminable => 16#41# ,
                   Accepting => 16#4A# ,
                   Synchronizing => 16#53# ,
                   Completed => 16#5C# ,
                   Terminated => 16#64#);

for TaskState'size use 8;

type TaskTypeDescriptor is
    record
       priority          : System.Priority;
       entry_count       : UIntg;
       block_id          : BlockId;
       first_own_address : System.Address;
       module_number     : UIntg;
       entry_number      : UIntg;
       code_address      : System.Address;
       stack_size        : System.DWord;
       dummy             : Integer;
       stack_segment_size: UIntg;
    end record;

type AccTaskTypeDescriptor is access TaskTypeDescriptor;

type NPXSaveArea is array(1..48) of System.UnsignedWord;
```

```
type FlagsType is
    record
       NPXFlag          : Bool;
       InterruptFlag    : Bool;
    end record;
pragma pack(FlagsType);

type StatesType is
    record
       state            : TaskState;
       is_abnormal      : Bool;
       is_activated     : Bool;
       failure          : Bool;
    end record;
pragma pack(StatesType);

type ACF_type is
    record
       bp               : Offset;
       addr             : System.Address;
    end record;
pragma pack(ACF_type);

type TaskControlBlock is
    record
       sem              : System.Semaphore;

    -- Delay queue handling

       dnext            : System.TaskValue ;
       dprev            : System.TaskValue ;
       ddelay           : Ticks ;

    -- Saved registers

       SS               : System.UnsignedWord ;
       SP               : Offset ;

    -- Ready queue handling

       next             : System.TaskValue ;

    -- Semaphore handling

       semnext          : System.TaskValue ;

    -- Priority fields

       priority         : System.Priority;
       saved_priority   : System.Priority;

    -- Miscelleanous fields
```

```
   time_slice      : System.UnsignedWord;
   flags           : FlagsType;
   ReadyCount      : System.Word;

-- Stack Specification

   stack_start     : Offset;
   stack_end       : Offset;

-- State fields

   states          : StatesType;

-- Activation handling fields

   activator       : System.TaskValue;
   act_chain       : System.TaskValue;
   next_chain      : System.TaskValue;
   no_not_act      : System.Word;
   act_block       : BlockId;

-- Accept queue fields

   partner         : System.TaskValue;
   next_partner    : System.TaskValue;

-- Entry queue fields

   next_caller     : System.TaskValue;

-- Rendezvous fields

   called_task     : System.TaskValue;
   task_entry      : TaskEntry;
   entry_index     : EntryIndex;
   entry_assoc     : System.Address;
   call_params     : System.Address;
   alt_id          : AlternativeId;
   excp_id         : System.ExceptionId;

-- Dependency fields

   parent_task     : System.TaskValue;
   parent_block    : BlockId;
   child_task      : System.TaskValue;
   next_child      : System.TaskValue;
   first_child     : System.TaskValue;
   prev_child      : System.TaskValue;
   child_act       : System.Word;
   block_act       : System.Word;
   terminated_task: System.TaskValue;

-- Abortion handling fields
```

```
      busy            : System.Word;

   -- Auxiliary fields

      ttd             : AccTaskTypeDescriptor;
      FirstCaller     : System.TaskValue;

   -- Run-Time System fields

      ACF             : ACF_type;          -- cf. section 9.4.2
      SQFirst         : Integer;           -- Only used in HDS
      SemFirst        : Integer;           -- Only used in HDS
      TBlockingTask   : System.TaskValue; -- Only used in HDS
      PBlockingTask   : System.TaskValue; -- Only used in HDS
      collection      : System.Address;
      partition       : Integer;

   -- NPX save area
   --
   -- When the application is linked with /NPX, a special
   -- save area for the NPX is allocated at the very end
   -- of every TCB.
   -- ie:
   --
   --   case NPX_Present is
   --     when TRUE  => NPXsave   : NPXSaveArea;
   --     when FALSE => null;
   --   end case;

   end record;

-- The following is to assure that the TCB has the expected size:

TCB_size     : constant INTEGER := TaskControlBlock'size / 8;

-- subtype TCB_ok_value is INTEGER range 214 .. 214;
-- TCB_ok   : constant TCB_ok_value := TaskControlBlock'size / 8;

end TaskTypes;
```

For DACS-80386PM package TaskTypes is as above except for the below declarations:

```
      subtype Offset    is System.UnsignedDWord;
      subtype BlockId   is System.UnsignedDWord;

      type    TaskEntry     is new System.UnsignedDWord;
      type    EntryIndex    is new System.UnsignedDWord;
      type    AlternativeId is new System.UnsignedDWord;
      type    Ticks         is new System.UnsignedDWord;
      type    UIntg         is new System.UnsignedDword;

      type NPXSaveArea is array(1..54) of System.UnsignedWord;
```

F.11 HDS Tasking (OPTIONAL)

The DACS-80x86 systems may run tasking applications by means of HARD DEADLINE SCHEDULING (HDS). HDS capability is purchased separately from the standard system, so it may not be included in a specific system. Please contact DDC-I for more information regarding HDS and your system.

HDS allows the programmer to guarantee properties of a tasking system, i.e. that some tasks will be scheduled to run within a specific time period. The HDS tasking is selected by specifying /HDS to the Ada link command.