

F

AD-A279 757

ION PAGE

Form Approved
OPM No.



Public reporting
and maintaining
suggestions for
22202-4302, or



per response, including the time for reviewing instructions, searching existing data sources gathering
omments regarding this burden estimate or any other aspect of this collection of information, including
to for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA
ment and Budget, Washington, DC 20503.

1. AGENCY

3. REPORT TYPE AND DATES

4. TITLE AND

940325S1.11343, AVF: 94ddc500 1A
DDC-I, DACS Sun SPARC/Solaris to 80186 Bare Ada Cross Compiler
System, Version 4.6.4

5. FUNDING

6. AUTHORS:

National Institute of Standards and Technology
Gaithersburg, Maryland

7. PERFORMING ORGANIZATION NAME(S) AND

National Institute of Standards and Technology
Building 255, Room A266
Gaithersburg, Maryland 20899
USA

8. PERFORMING
ORGANIZATION

9. SPONSORING/MONITORING AGENCY NAME(S) AND

Ada Joint Program Office
The Pentagon, Rm 3E118
Washington, DC 20301-3080

DTIC
ELECTE
MAY 26 1994
S G D

1188 94-15731

11. SUPPLEMENTARY

12a. DISTRIBUTION/AVAILABILITY

Approved for Public Release; distribution unlimited

12b. DISTRIBUTION

13. (Maximum 200)

Host: Sun SPARCclassic (under Solaris, Release 2.1)
Target: Intel iSBC 186/100 (bare machine)

14. SUBJECT

Ada programming language, Ada Compiler Validation Summary Report, Ada
Compiler Val. Capability Val. Testing, Ada Val. Office, Ada Val. Facility
ANSP/MIL-STD-1815A, ADPO

15. NUMBER OF

16. PRICE

17. SECURITY
CLASSIFICATION
UNCLASSIFIED

18. SECURITY
CLASSIFICATION
UNCLASSIFIED

19. SECURITY
CLASSIFICATION
UNCLASSIFIED

20. LIMITATION OF
UNCLASSIFIED

NNN

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

94 5 25 015

DTIC QUALITY INSPECTED 1

AVF Control Number: NIST94DDC500_1B_1.11

DATE COMPLETED

BEFORE ON-SITE: 94-03-18

AFTER ON-SITE: 94-03-28

REVISIONS: 94-04-11

Ada COMPILER

VALIDATION SUMMARY REPORT:

Certificate Number: 940325S1.11343

DDC-I

DACS Sun SPARC/Solaris to 80186 Bare Ada

Cross Compiler System, Version 4.6.4

Sun SPARCclassic => Intel iSBC 186/100 (Bare Machine)

Prepared By:

Software Standards Validation Group

Computer Systems Laboratory

National Institute of Standards and Technology

Building 225, Room A266

Gaithersburg, Maryland 20899

U.S.A.

Accession For	
NTIS CRA&I	<input checked="" type="checkbox"/>
DTIC TAB	<input type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification	
By	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A-1	

AVF Control Number: NIST94DDC500_1B_1.11

Certificate Information

The following Ada implementation was tested and determined to pass ACVC 1.11. Testing was completed on March 25, 1994.

Compiler Name and Version: DACS Sun SPARC/Solaris to 80186 Bare Ada Cross Compiler System, Version 4.6.4

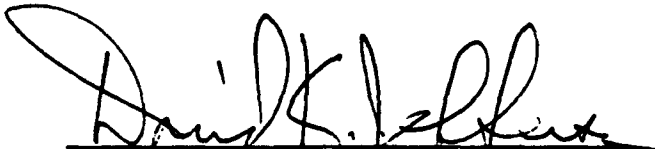
Host Computer System: Sun SPARCclassic running under Solaris, Release 2.1

Target Computer System: Intel iSBC 186/100 (Bare Machine)

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

As a result of this validation effort, Validation Certificate 940325S1.11343 is awarded to DDC-I. This certificate expires 2 years after ANSI/MIL-STD-1815B is approved by ANSI.

This report has been reviewed and is approved.

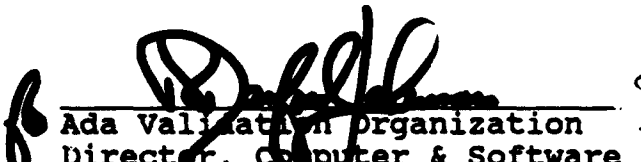


Ada Validation Facility
Dr. David K. Jefferson
Chief, Information Systems
Engineering Division (ISED)



Ada Validation Facility
Mr. L. Arnold Johnson
Manager, Software Standards
Validation Group

Computer Systems Laboratory (CSL)
National Institute of Standards and Technology
Building 225, Room A266
Gaithersburg, Maryland 20899
U.S.A.



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



Ada Joint Program Office
David R. Basel
Deputy Director,
Ada Joint Program Office
Defense Information Systems Agency,
Center for Information Management
Washington DC 20301

U.S.A.

NIST94DDC500_1B_1.11

DECLARATION OF CONFORMANCE

The following declaration of conformance was supplied by the customer.

Customer: DDC-I

Certificate Awardee: DDC-I

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

ACVC Version: 1.11

Ada Implementation:

Compiler Name and Version: DACS Sun SPARC/Solaris to 80186 Bare Ada Cross
Compiler System, Version 4.6.4

Host Computer System: Sun SPARCclassic running under Solaris, Release 2.1

Target Computer System: Intel iSBC 186/100 (Bare Machine)

Declaration:

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

James H. H. H.
Customer Signature
Company DDC-I
Title

94-03-25
Date

James H. H. H.
Certificate Awardee Signature
Company DDC-I
Title

94-03-25
Date

TABLE OF CONTENTS

CHAPTER 1.....	1-1
INTRODUCTION.....	1-1
1.1 USE OF THIS VALIDATION SUMMARY REPORT.....	1-1
1.2 REFERENCES.....	1-2
1.3 ACVC TEST CLASSES.....	1-2
1.4 DEFINITION OF TERMS.....	1-3
CHAPTER 2.....	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.....	3-1
3.3 TEST EXECUTION.....	3-2
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 [Pro92] 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 [Pro92]. 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, Virginia 22161
U.S.A.

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

Ada Validation Organization
Computer and Software Engineering Division
Institute for Defense Analyses
1801 North Beauregard Street
Alexandria, Virginia 22311-1772
U.S.A.

1.2 REFERENCES

- [Ada83] Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815A, February 1983 and ISO 8652-1987.
- [Pro92] Ada Compiler Validation Procedures, Version 3.1, Ada Joint Program Office, August 1992.
- [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 SPRT13, 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 SPRT13 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 Joint Program Office (AJPO)	The part of the certification body which provides policy and guidance for the Ada certification Office 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.
ISO	International Organization for Standardization.
LRM	The Ada standard, or Language Reference Manual, published as ANSI/MIL-STD-1815A -1983 and ISO 8652-1987. Citations from the LRM take the form "<section>.<subsection>:<paragraph>."
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 [Pro92].

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 104 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 93-11-22.

B27005A	E28005C	B28006C	C32203A	C34006D	C35507K
C35507L	C35507N	C35507O	C35507P	C35508I	C35508J
C35508M	C35508N	C35702A	C35702B	C37310A	B41308B
C43004A	C45114A	C45346A	C45612A	C45612B	C45612C
C45651A	C46022A	B49008A	B49008B	A54B02A	C55B06A
A74006A	C74308A	B83022B	B83022H	B83025B	B83025D
B83026B	C83026A	C83041A	B85001L	C86001F	C94021A
C97116A	C98003B	BA2011A	CB7001A	CB7001B	CB7004A
CC1223A	BC1226A	CC1226B	BC3009B	BD1B02B	BD1B06A
AD1B08A	BD2A02A	CD2A21E	CD2A23E	CD2A32A	CD2A41A
CD2A41E	CD2A87A	CD2B15C	BD3006A	BD4008A	CD4022A
CD4022D	CD4024B	CD4024C	CD4024D	CD4031A	CD4051D
CD5111A	CD7004C	ED7005D	CD7005E	AD7006A	CD7006E
AD7201A	AD7201E	CD7204B	AD7206A	BD8002A	BD8004C
CD9005A	CD9005B	CDA201E	CE2107I	CE2117A	CE2117B
CE2119B	CE2205B	CE2405A	CE3111C	CE3116A	CE3118A
CE3411B	CE3412B	CE3607B	CE3607C	CE3607D	CE3812A
CE3814A	CE3902B				

2.2 INAPPLICABLE TESTS

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

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

C24113L..Y (14 tests)	C35705L..Y (14 tests)
C35706L..Y (14 tests)	C35707L..Y (14 tests)

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

C24113I..K (3 tests) use a line length in the input file which exceeds 126 characters.

C35404D, C45231D, B86001X, C86006E, and CD7101G check for a predefined integer type with a name other than INTEGER, LONG_INTEGER, or SHORT_INTEGER; for this implementation, there is no such type.

C35713B, C45423B, B86001T, and C86006H check for the predefined type SHORT_FLOAT; for this implementation, there is no such type.

C35713D and B86001Z check for a predefined floating-point type with a name other than FLOAT, LONG_FLOAT, or SHORT_FLOAT; for this implementation, there is no such type.

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

C45624A..B (2 tests) check that the proper exception is raised if MACHINE_OVERFLOW is FALSE for floating point types and the results of various floating-point operations lie outside the range of the base type; for this implementation, MACHINE_OVERFLOW is TRUE.

C4A013B contains a static universal real expression that exceeds the range of this implementation's largest floating-point type; this expression is rejected by the compiler.

D56001B uses 65 levels of block nesting; this level of block nesting exceeds the capacity of the compiler.

B86001Y uses the name of a predefined fixed-point type other than type DURATION; for this implementation, there is no such type.

C96005B uses values of type DURATION's base type that are outside the range of type DURATION; for this implementation, the ranges are the same.

CA2009C and CA2009F check whether a generic unit can be instantiated before its body (and any of its subunits) is compiled; this implementation creates a dependence on generic units as allowed by AI-00408 and AI-00506 such that the compilation of the generic unit bodies makes the instantiating units obsolete. (See section 2.3.)

CD1009C checks whether a length clause can specify a non-default size for a floating-point type; this implementation does not support such sizes.

CD2A84A, CD2A84E, CD2A84I..J (2 tests), and CD2A84O use length clauses to specify non-default sizes for access types; this implementation does not support such sizes.

The following 264 tests check operations on sequential, text, and direct access files; this implementation does not support external files:

CE2102A..C (3)	CE2102G..H (2)	CE2102K	CE2102N..Y (12)
CE2103C..D (2)	CE2104A..D (4)	CE2105A..B (2)	CE2106A..B (2)
CE2107A..H (8)	CE2107L	CE2108A..H (8)	CE2109A..C (3)
CE2110A..D (4)	CE2111A..I (9)	CE2115A..B (2)	CE2120A..B (2)
CE2201A..C (3)	EE2201D..E (2)	CE2201F..N (9)	CE2203A
CE2204A..D (4)	CE2205A	CE2206A	CE2208B
CE2401A..C (3)	EE2401D	CE2401E..F (2)	EE2401G
CE2401H..L (5)	CE2403A	CE2404A..B (2)	CE2405B
CE2406A	CE2407A..B (2)	CE2408A..B (2)	CE2409A..B (2)
CE2410A..B (2)	CE2411A	CE3102A..C (3)	CE3102F..H (3)
CE3102J..K (2)	CE3103A	CE3104A..C (3)	CE3106A..B (2)
CE3107B	CE3108A..B (2)	CE3109A	CE3110A
CE3111A..B (2)	CE3111D..E (2)	CE3112A..D (4)	CE3114A..B (2)
CE3115A	CE3119A	EE3203A	EE3204A
CE3207A	CE3208A	CE3301A	EE3301B
CE3302A	CE3304A	CE3305A	CE3401A
CE3402A	EE3402B	CE3402C..D (2)	CE3403A..C (3)
CE3403E..F (2)	CE3404B..D (3)	CE3405A	EE3405B
CE3405C..D (2)	CE3406A..D (4)	CE3407A..C (3)	CE3408A..C (3)
CE3409A	CE3409C..E (3)	EE3409F	CE3410A
CE3410C..E (3)	EE3410F	CE3411A	CE3411C
CE3412A	EE3412C	CE3413A..C (3)	CE3414A
CE3602A..D (4)	CE3603A	CE3604A..B (2)	CE3605A..E (5)
CE3606A..B (2)	CE3704A..F (6)	CE3704M..O (3)	CE3705A..E (5)
CE3706D	CE3706F..G (2)	CE3804A..P (16)	CE3805A..B (2)
CE3806A..B (2)	CE3806D..E (2)	CE3806G..H (2)	CE3904A..B (2)
CE3905A..C (3)	CE3905L	CE3906A..C (3)	CE3906E..F (2)

CE2103A, CE2103B, and CE3107A use an illegal file name in an attempt to create a file and expect `NAME_ERROR` to be raised; this implementation does not support external files and so raises `USE_ERROR`. (See section 2.3.)

2.3 TEST MODIFICATIONS

Modifications (see section 1.3) were required for 71 tests.

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

the way expected by the original tests.

B22003A	B26001A	B26002A	B26005A	B28003A	B29001A	B33301B
B35101A	B37106A	B37301B	B37302A	B38003A	B38003B	B38009A
B38009B	B55A01A	B61001C	B61001F	B61001H	B61001I	B61001M
B61001R	B61001W	B67001H	B83A07A	B83A07B	B83A07C	B83E01C
B83E01D	B83E01E	B85001D	B85008D	B91001A	B91002A	B91002B
B91002C	B91002D	B91002E	B91002F	B91002G	B91002H	B91002I
B91002J	B91002K	B91002L	B95030A	B95061A	B95061F	B95061G
B95077A	B97103E	B97104G	BA1001A	BA1101B	BC1109A	BC1109C
BC1109D	BC1202A	BC1202F	BC1202G	BE2210A	BE2413A	

C83030C and C86007A were graded passed by Test Modification as directed by the AVO. These tests were modified by inserting "PRAGMA ELABORATE (REPORT);" before the package declarations at lines 13 and 11, respectively. Without the pragma, the packages may be elaborated prior to package Report's body, and thus the packages' calls to function REPORT.IDENT_INT at lines 14 and 13, respectively, will raise PROGRAM_ERROR.

CA2009C and CA2009F were graded inapplicable by Evaluation Modification as directed by the AVO. These tests contain instantiations of a generic unit prior to the compilation of that unit's body; as allowed by AI-00408 and AI-00506, the compilation of the generic unit bodies makes the compilation unit that contains the instantiations obsolete.

BC3204C and BC3205D were graded passed by Processing Modification as directed by the AVO. These tests check that instantiations of generic units with unconstrained types as generic actual parameters are illegal if the generic bodies contain uses of the types that require a constraint. However, the generic bodies are compiled after the units that contain the instantiations, and this implementation creates a dependence of the instantiating units on the generic units as allowed by AI-00408 and AI-00506 such that the compilation of the generic bodies makes the instantiating units obsolete--no errors are detected. The processing of these tests was modified by re-compiling the obsolete units; all intended errors were then detected by the compiler.

CE2103A, CE2103B, and CE3107A were graded inapplicable by Evaluation Modification as directed by the AVO. The tests abort with an unhandled exception when USE_ERROR is raised on the attempt to create an external file. This is acceptable behavior because this implementation does not support external files (cf. AI-00332).

CHAPTER 3

PROCESSING INFORMATION

3.1 TESTING ENVIRONMENT

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

For technical information about this Ada implementation, contact:

Forrest Holemon
410 North 44th Street, Suite 320
Phoenix, Arizona 85008 (U.S.A.)
Telephone: 602-275-7172
Telefax: 602-275-7502

For sales information about this Ada implementation, contact:

Mike Halpin
410 North 44th Street, Suite 320
Phoenix, Arizona 85008 (U.S.A.)
Telephone: 602-275-7172
Telefax: 602-275-7502

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 [Pro92].

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

The list of items below gives the number of ACVC tests in various categories. All tests were processed, except those that were withdrawn because of test errors (item b; see section 2.1), those that require a floating-point precision that exceeds the implementation's maximum precision (item e; see section 2.2), and those that depend on the support of a file system--if none is supported (item d). All tests passed, except those that are listed in sections 2.1 and 2.2 (counted in items b and f, below).

a) Total Number of Applicable Tests	3562	
b) Total Number of Withdrawn Tests	104	
c) Processed Inapplicable Tests	504	
d) Non-Processed I/O Tests	0	
e) Non-Processed Floating-Point Precision Tests	0	
f) Total Number of Inapplicable Tests	504	(c+d+e)
g) Total Number of Tests for ACVC 1.11	4170	(a+b+f)

3.3 TEST EXECUTION

A magnetic tape containing the customized test suite (see section 1.3) was taken on-site by the validation team for processing. The contents of the magnetic tape were loaded directly onto the host computer.

After the test files were loaded onto the host computer, the full set of tests was processed by the Ada implementation. The DDC-I Ada downloader runs on the Sun SPARCclassic and is used for downloading the executable images to the target Intel iSBC 186/100 (Bare Machine). The DDC-I Debug Monitor runs on the target Intel iSBC 186/100 (Bare Machine) and provides communication interface between the host downloader and the executing target Intel iSBC 186/100 (Bare Machine). The two processes communicate via ethernet.

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

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

-list

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

APPENDIX A

MACRO PARAMETERS

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

Macro Parameter	Macro Value
\$MAX_IN_LEN	126 -- Value of V
\$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	: 32
ALIGNMENT	: 1
COUNT_LAST	: 32767
DEFAULT_MEM_SIZE	: 1_048_576
DEFAULT_STOR_UNIT	: 16
DEFAULT_SYS_NAME	: IAPX186
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.ADA")
INCLUDE_PRAGMA2	:
	: PRAGMA INCLUDE ("B28006E1.ADA")
INTEGER_FIRST	: -32768
INTEGER_LAST	: 32767
INTEGER_LAST_PLUS_1	: 32768
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_NOP);
MACHINE_CODE_TYPE	: REGISTER_TYPE
MANTISSA_DOC	: 31

```

MAX_DIGITS           : 15
MAX_INT              : 2147483647
MAX_INT_PLUS_1      : 2147483648
MIN_INT              : -2147483648
NAME                 : SHORT_SHORT_INTEGER
NAME_LIST            : IAPX186
NAME_SPECIFICATION1 :
    DISK$AWC_2:[CROCKETTL.ACVC11.DEVELOPMENT]X2120A
NAME_SPECIFICATION2 :
    DISK$AWC_2:[CROCKETTL.ACVC11.DEVELOPMENT]X2120B
NAME_SPECIFICATION3 :
    DISK$AWC_2:[CROCKETTL.ACVC11.DEVELOPMENT]X3119A
NEG_BASED_INT        : 16#FFFFFFFF#
NEW_MEM_SIZE         : 1_048_576
NEW_STOR_UNIT        : 16
NEW_SYS_NAME         : IAPX186
PAGE_TERMINATOR      : ASCII.FF
RECORD_DEFINITION    : RECORD NULL;END RECORD;
RECORD_NAME          : NO_SUCH_MACHINE_CODE_TYPE
TASK_SIZE            : 16
TASK_STORAGE_SIZE    : 1024
TICK                 : 0.000_000_125
VARIABLE_ADDRESS     : (16#0#,16#1FF9#)
VARIABLE_ADDRESS1    : (16#4#,16#1FF9#)
VARIABLE_ADDRESS2    : (16#8#,16#1FF9#)
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.

5 THE ADA COMPILER

The Ada Compiler compiles all program units within the specified source file and inserts the generated objects into the current program library. Compiler options are provided to allow the user control of optimization, run-time checks, and compiler input and output options such as list files, configuration files, the program library used, etc.

The input to the compiler consists of the source file, the configuration file (which controls the format of the list file), and the compiler options. Section 5.1 provides a list of all compiler options, and Section 5.2 describes the source and configuration files.

If any diagnostic messages are produced during the compilation, they are output on the diagnostic file and on the current output file. The diagnostic file and the diagnostic messages are described in Section 5.3.2.

Output consists of an object placed in the program library, diagnostic messages, and optional listings. The configuration file and the compiler options specify the format and contents of the list information. Output is described in Section 5.3.

The compiler uses a program library during the compilation. The compilation unit may refer to units from the program library, and an internal representation of the compilation unit will be included in the program library as a result of a successful compilation. The program library is described in Chapter 3. Section 5.4 briefly describes how the Ada compiler uses the library.

5.1 Invoking the Ada Compiler

Invoke the Ada compiler with the following command to the SunOS shell:

```
$ ada {<option>} <source-file-name>
```

where the options and parameters are:

DACS-80x86 User's Guide
Ada Compiler

OPTION	DESCRIPTION	REFERENCE
<code>-[no]auto_inline</code>	Specifies whether local subprograms should be inline expanded.	5.1.1
<code>-check</code>	Controls run-time checks.	5.1.2
<code>-configuration_file</code>	Specifies the configuration file used by the compiler.	5.1.3
<code>-[no]debug</code>	Includes symbolic debugging information in program Library. Does not include symbolic information.	5.1.4
<code>-[no]fixpoint_rounding</code>	Generates fixed point rounding code. Avoids fixed point rounding code.	5.1.5
<code>-[no]float_allowed</code>	Flags generation of float instructions as error if selected.	5.1.6
<code>-[no]library</code>	Specifies program library used.	5.1.7
<code>-[no]list</code>	Writes a source listing on the list file.	5.1.8
<code>-[no]optimize</code>	Specifies compiler optimization.	5.1.9
<code>-[no]progress</code>	Displays compiler progress.	5.1.10
<code>-[no]xref</code>	Creates a cross reference listing.	5.1.11
<code>-[no]save_source</code>	Copies source to program library.	5.1.12
<code>-[no]target_debug</code>	Includes Intel debug information. Does not include Intel debug information.	5.1.13
<code>-unit</code>	Assigns a specific unit number to the compilation (must be free and in a sublibrary).	5.1.14
<code>-recompile</code>	Interpret the file name as a compilation unit body that must be recompiled from library.	5.1.15
<code>-specification</code>	With <code>-recompile</code> interpret file name as a compilation unit specification rather than body.	5.1.16

Examples:

```
$ ada -list testprog
```

This example compiles the source file `testprog.ada` and generates a list file with the name `testprog.lis`.

```
$ ada -library my_library test
```

This example compiles the source file `test.ada` into the library `my_library`.

Default values exist for most options as indicated in the following sections. Option names may be abbreviated (characters omitted from the right) as long as no ambiguity arises.

<source-file-name>

The Ada compiler has one mandatory parameter that should specify the Ada source file. This parameter specifies the text file containing the source text to be compiled. If the file type is omitted in the source file specification, the file type ".ada" is assumed by default.

The allowed format of the source text is described in Section 5.2.1.

Below follows a description of each of the available options to the invocation of the Ada compiler.

5.1.1 -[no]auto_inline

-auto_inline local | global
-noauto_inline (default)

This option specifies whether subprograms should be inline expanded. The inline expansion only occurs if the subprogram has less than 4 object declarations and less than 6 statements, and if the subprogram fulfills the requirements defined for pragma **INLINE** (see Section C.2.3). **LOCAL** specifies that only inline expansion of locally defined subprograms should be done, while **GLOBAL** will cause inline expansion of all subprograms, including subprograms from other units.

5.1.2 -check

-check [<keyword> = ON | OFF { ,<keyword> = ON | OFF }]
-check ALL=ON (default)

-check specifies which run-time checks should be performed. Setting a run-time check to **ON** enables the check, while setting it to **OFF** disables the check. All run-time checks are enabled by default. The following explicit checks will be disabled/enabled by using the name as **<keyword>**:

ACCESS	Check for access values being non NULL.
ALL	All checks.
DISCRIMINANT	Checks for discriminated fields.
ELABORATION	Checks for subprograms being elaborated.
INDEX	Index check.
LENGTH	Array length check.
OVERFLOW	Explicit overflow checks.
RANGE	Checks for values being in range.
STORAGE	Checks for sufficient storage available.

5.1.3 -configuration_file

-configuration_file <file-spec>
-configuration_file config (default)

This option specifies the configuration file to be used by the compiler in the current compilation. The configuration file allows the user to format compiler listings, set error limits, etc. If the option is omitted the configuration file config located in the same directory as the Ada compiler is used by default. Section 5.2.2 contains a description of the configuration file.

5.1.4 -[no]debug

-debug
-nodebug (default)

Generate debug information for the compilation and store the information in the program library. This is necessary if the unit is to be debugged with the DDC-I Ada Symbolic Cross Debugger. Note that the program must also be linked with the **-debug** option, if the program is to be debugged with the DDC-I Ada Symbolic Cross Debugger. See Section 6.5.11.

5.1.5 -[no]fixpoint_rounding

-fixpoint_rounding (default)
-nofixpoint_rounding

Normally all inline generated code for fixed point MULTIPLY and DIVIDE is rounded, but this may be avoided with **-nofixpoint_rounding**. Inline code is generated for all 16 bit fixed point types and for 32 bit fixed point types, when the target is 80386PM or 80486PM.

5.1.6 -[no]float_allowed

-float_allowed (default)
-nofloat_allowed

Float instruction generation may be flagged as errors, if **-nofloat** is selected. This is for use in systems, where no floating point processor (nor emulator) is available. Notice that TEXT_IO uses floats in connection with FLOAT_IO and FIXED_IO.

5.1.7 -library

-library <file-spec>
-library Sada_library (default)

This option specifies the current sublibrary that will be used in the compilation and will receive the object when the compilation is complete. By specifying a current sublibrary, the current program library (current sublibrary and ancestors up to root) is also implicitly specified.

If this option is omitted, the sublibrary designated by the environmental variable `ada_library` is used as the current sublibrary. Section 5.4 describes how the Ada compiler uses the library.

5.1.8 -(no)list

-list
-nolist (default)

-list specifies that a source listing will be produced. The source listing is written to the list file, which has the name of the source file with the extension `.lis`. Section 5.3.1.1 contains a description of the source listing.

If **-nolist** is active, no source listing is produced, regardless of `LIST` pragmas in the program or diagnostic messages produced.

5.1.9 -optimize

-optimize [<keyword> = on | off { ,<keyword> = on | off }]
-optimize all=off

This option specifies which optimizations will be performed during code generation. The possible keywords are: (casing is irrelevant)

all	All possible optimizations are invoked.
check	Eliminates superfluous checks.
cse	Performs common subexpression elimination including common address expressions.
fcn2proc	Change function calls returning objects of constrained array types or objects of record types to procedure calls.
reordering	Transforms named aggregates to positional aggregates and named parameter associations to positional associations.
stack_height	Performs stack height reductions (also called Aho Ullman reordering).
block	Optimize block and call frames.

Setting an optimization to `on` enables the optimization, while setting an optimization to `off` disables the optimization. All optimizations are disabled by default. In addition to the optional optimizations, the compiler always performs the following optimizations: constant folding, dead code elimination, and selection of optimal jumps.

5.1.10 `-[no]progress`

`-progress`
`-noprogess` (default)

When this option is given, the compiler will output data about which pass the compiler is currently running.

5.1.11 `-[no]xref`

`-xref`
`-noxref` (default)

A cross-reference listing can be requested by the user by means of the option `-xref`. If the `-xref` option is given and no severe or fatal errors are found during the compilation, the cross-reference listing is written to the list file. The cross-reference listing is described in Section 7.

5.1.12 `-[no]save_source`

`-save_source` (default)
`-nosave_source`

When `-save_source` is specified, a copy of the compiled source code is placed in the program library. If `-nosave_source` is used, source code will not be retained in the program library.

Using `-nosave_source`, while helping to keep library sizes smaller, does affect the operation of the recomplier, see Chapter 7 for more details. Also, it will not be possible to do symbolic debugging at the Ada source code level with the DACS-80x86 Symbolic Ada Debugger, if the source code is not saved in the library.

5.1.13 `-[no]target_debug`

`-target_debug`
`-notarget_debug` (default)

Specifies whether symbolic debug information on standard OMF is included in the object file. Currently the linker does not support the OMF debug information.

This option may be used when debugging with standard OMF tools (i.e., PICE).

5.1.14 -unit

-unit = <unit_number>

The specified unit number will be assigned to the compilation unit if it is free and it is a legal unit number for the library.

5.1.15 -recompile

-recompile

The file name (source) is interpreted as a compilation unit name which has its source saved from a previous compilation. If **-specification** is not specified, it is assumed to be body which must be recompiled.

5.1.16 -specification

-specification

Works only together with **-recompile**, see Section 5.1.15.

5.2 Compiler Input

Input to the compiler consists of the command line options, a source text file and, optionally, a configuration file.

5.2.1 Source Text

The user submits one file containing a source text in each compilation. The source text may consist of one or more compilation units (see ARM Section 10.1).

The format of the source text must be in ISO-FORMAT ASCII. This format requires that the source text is a sequence of ISO characters (ISO standard 646), where each line is terminated by either one of the following termination sequences (CR means carriage return, VT means vertical tabulation, LF means line feed, and FF means form feed):

- A sequence of one or more CRs, where the sequence is neither immediately preceded nor immediately followed by any of the characters VT, LF, or FF.
- Any of the characters VT, LF, or FF, immediately preceded and followed by a sequence of zero or more CRs.

In general, ISO control characters are not permitted in the source text with the following exceptions:

- The horizontal tabulation (HT) character may be used as a separator between lexical units.
- LF, VT, FF, and CR may be used to terminate lines, as described above.

The maximum number of characters in an input line is determined by the contents of the configuration file (see section 5.1.3). The control characters CR, VT, LF, and FF are not considered a part of the line. Lines containing more than the maximum number of characters are truncated and an error message is issued.

5.2.2 Configuration File

Certain processing characteristics of the compiler, such as format of input and output, and error limit, may be modified by the user. These characteristics are passed to the compiler by means of a configuration file, which is a standard SPARC/SunOS text file. The contents of the configuration file must be an Ada positional aggregate, written on one line, of the type `CONFIGURATION_RECORD`, which is described below.

The configuration file (`config`) is not accepted by the compiler in the following cases:

- The syntax does not conform with the syntax for positional Ada aggregates.
- A value is outside the ranges specified.
- A value is not specified as a literal.
- `LINES_PER_PAGE` is not greater than `TOP_MARGIN + BOTTOM_MARGIN`.
- The aggregate occupies more than one line.

If the compiler is unable to accept the configuration file, an error message is written on the current output file and the compilation is terminated.

This is the record whose values must appear in aggregate form within the configuration file. The record declaration makes use of some other types (given below) for the sake of clarity.

DACS-80x86 User's Guide
Ada Compiler

```
type CONFIGURATION_RECORD is
  record
    IN_FORMAT:    INFORMATTING;
    OUT_FORMAT:   OUTFORMATTING;
    ERROR_LIMIT:  INTEGER;
  end record;

type INPUT_FORMATS is (ASCII);

type INFORMATTING is
  record
    INPUT_FORMAT:    INPUT_FORMATS;
    INPUT_LINELENGTH: INTEGER range 70..250;
  end record;

type OUTFORMATTING is
  record
    LINES_PER_PAGE   : INTEGER range 30..100;
    TOP_MARGIN       : INTEGER range 4.. 90;
    BOTTOM_MARGIN     : INTEGER range 0.. 90;
    OUT_LINELENGTH    : INTEGER range 80..132;
    SUPPRESS_ERRORNO : BOOLEAN;
  end record;
```

The outformatting parameters have the following meaning:

- 1) **LINES_PER_PAGE**: specifies the maximum number of lines written on each page (including top and bottom margin).
- 2) **TOP_MARGIN**: specifies the number of lines on top of each page used for a standard heading and blank lines. The heading is placed in the middle lines of the top margin.
- 3) **BOTTOM_MARGIN**: specifies the minimum number of lines left blank in the bottom of the page. The number of lines available for the listing of the program is **LINES_PER_PAGE - TOP_MARGIN - BOTTOM_MARGIN**.
- 4) **OUT_LINELENGTH**: specifies the maximum number of characters written on each line. Lines longer than **OUT_LINELENGTH** are separated into two lines.
- 5) **SUPPRESS_ERRORNO**: specifies the format of error messages (see Section 5.3.5.1).

The name of a user-supplied configuration file can be passed to the compiler through the **configuration_file** option. DDC-I supplies a default configuration file (**config**) with the following content:

((ASCII, 126), (48,5,3,100,FALSE), 200)

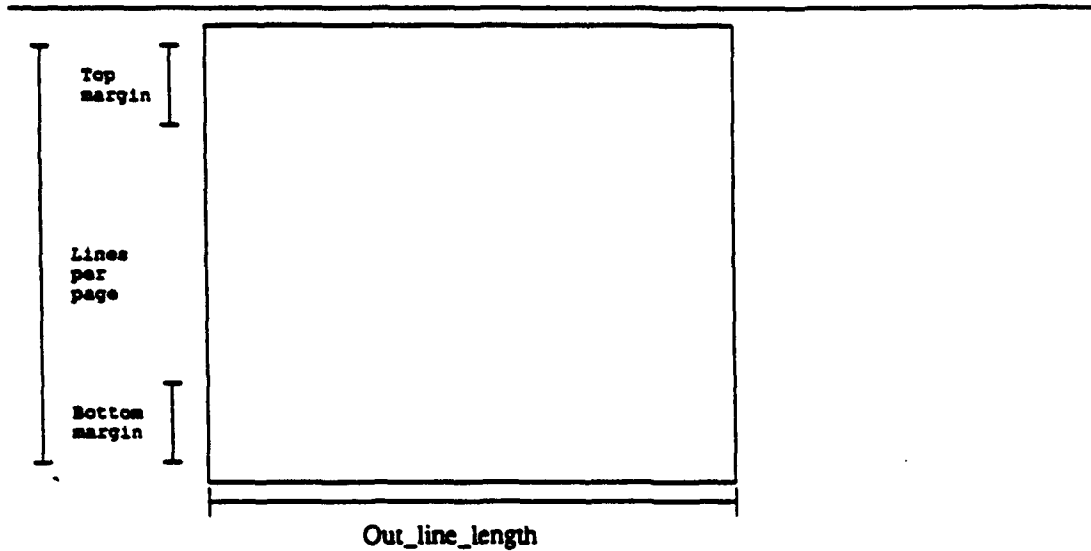


Figure 5-1. Page Layout

5.3 Compiler Output

The compiler may produce output in the list file, the diagnostic file, and the current output file. It also updates the program library if the compilation is successful. The present section describes the text output in the three files mentioned above. The updating of the program library is described in Section 5.4.

The compiler may produce the following text output:

- 1) A listing of the source text with embedded diagnostic messages is written on the list file, if the option `-list` is active.
- 2) A compilation summary is written on the list file, if `-list` is active.
- 3) A cross-reference listing is written on the list file, if `-xref` is active and no severe or fatal errors have been detected during the compilation.
- 4) If there are any diagnostic messages, a diagnostic file containing the diagnostic messages is written.
- 5) Diagnostic messages other than warnings are written on the current output file.

5.3.1 The List File

The name of the list file is identical to the name of the source file except that it has the file type ".lis". The file is located in the current (default) directory. If any such file exists prior to the compilation, the newest version of the file is deleted. If the user requests any listings by specifying the options -list or -xref, a new list file is created.

The list file may include one or more of the following parts: a source listing, a cross-reference listing, and a compilation summary.

The parts of the list file are separated by page ejects. The contents of each part are described in the following sections.

The format of the output on the list file is controlled by the configuration file (see Section 5.2.2) and may therefore be controlled by the user.

5.3.1.1 Source Listing

A source listing is an unmodified copy of the source text. The listing is divided into pages and each line is supplied with a line number.

The number of lines output in the source listing is governed by the occurrence of LIST pragmas and the number of objectionable lines.

- Parts of the listing can be suppressed by the use of the LIST pragma.
- A line containing a construct that caused a diagnostic message to be produced is printed even if it occurs at a point where listing has been suppressed by a LIST pragma.

5.3.1.2 Compilation Summary

At the end of a compilation, the compiler produces a summary that is output on the list file if the option -list is active.

The summary contains information about:

- 1) The type and name of the compilation unit, and whether it has been compiled successfully or not.
- 2) The number of diagnostic messages produced for each class of severity (see Section 5.3.2.1).
- 3) Which options were active.
- 4) The full name of the source file.
- 5) The full name of the current sublibrary.
- 6) The number of source text lines.

- 7) The size of the code produced (specified in bytes).
- 8) Elapsed real time and elapsed CPU time.
- 9) A "Compilation terminated" message if the compilation unit was the last in the compilation or "Compilation of next unit initiated" otherwise.

5.3.1.3 Cross-Reference Listing

A cross-reference listing is an alphabetically sorted list of the identifiers, operators, and character literals of a compilation unit. The list has an entry for each entity declared and/or used in the unit, with a few exceptions stated below. Overloading is evidenced by the occurrence of multiple entries for the same identifier.

For instantiations of generic units, the visible declarations of the generic unit are included in the cross-reference listing as declared immediately after the instantiation. The visible declarations are the subprogram parameters for a generic subprogram and the declarations of the visible part of the package declaration for a generic package.

For type declarations, all implicitly declared operations are included in the cross-reference listing.

Cross-reference information will be produced for every constituent character literal for string literals.

The following are not included in the cross reference listing:

- Pragma identifiers and pragma argument identifiers.
- Numeric literals.
- Record component identifiers and discriminant identifiers. For a selected name whose selector denotes a record component or a discriminant, only the prefix generates cross-reference information.
- A parent unit name (following the keyword SEPARATE).

Each entry in the cross-reference listing contains:

- The identifier with, at most, 15 characters. If the identifier exceeds 15 characters, a bar ("|") is written in the 16th position and the rest of the characters are not printed.
- The place of the definition, i.e., a line number if the entity is declared in the current compilation unit, otherwise the name of the compilation unit in which the entity is declared and the line number of the declaration.
- The numbers of the lines in which the entity is used. An asterisk ("*") after a line number indicates an assignment to a variable, initialization of a constant, assignments to functions, or user-defined operators by means of RETURN statements. Please refer to Appendix B.3 for examples.

5.3.2 The Diagnostic File

The name of the diagnostic file is identical to the name of the source file except that it has the file type ".err". It is located in the current (default) directory. If any such file exists prior to the compilation, the newest version of the file is deleted. If any diagnostic messages are produced during the compilation a new diagnostic file is created.

The diagnostic file is a text file containing a list of diagnostic messages, each followed by a line showing the number of the line in the source text causing the message, and a blank line. There is no separation into pages and no headings. The file may be used by an interactive editor to show the diagnostic messages together with the erroneous source text.

5.3.2.1 Diagnostic Messages

The Ada compiler issues diagnostic messages on the diagnostic file. Diagnostics other than warnings also appear on the current output file. If a source text listing is required, the diagnostics are also found embedded in the list file (see Section 5.3.1).

In a source listing, a diagnostic message is placed immediately after the source line causing the message. Messages not related to any particular line are placed at the top of the listing. Every diagnostic message in the diagnostic file is followed by a line stating the line number of the objectional line. The lines are ordered by increasing source line numbers. Line number 0 is assigned to messages not related to any particular line. On the current output file the messages appear in the order in which they are generated by the compiler.

The diagnostic messages are classified according to their severity and the compiler action taken:

Warning: Reports a questionable construct or an error that does not influence the meaning of the program. Warnings do not hinder the generation of object code.

Example: A warning will be issued for constructs for which the compiler detects will raise `CONSTRAINT_ERROR` at run time.

Error: Reports an illegal construct in the source program. Compilation continues, but no object code will be generated.

Examples: most syntax errors; most static semantic errors.

Severe error: Reports an error which causes the compilation to be terminated immediately. No object code is generated.

Example: A severe error message will be issued if a library unit mentioned by a `WITH` clause is not present in the current program library.

Fatal error: Reports an error in the compiler system itself. Compilation is terminated immediately and no object code is produced. The user may be able to circumvent a fatal error by correcting the program or by replacing program constructs with alternatives. Please inform DDC-I about the occurrence of fatal errors.

The detection of more errors than allowed by the number specified by the `ERROR_LIMIT` parameter of the configuration file (see section 5.2.2) is considered a severe error.

5.3.2.2 Format and Content of Diagnostic Messages

For certain syntactically incorrect constructs, the diagnostic message consists of a pointer line and a text line. In other cases a diagnostic message consists of a text line only.

The pointer line contains a pointer (a carat symbol `^`) to the offending symbol or to an illegal character.

The text line contains the following information:

- the diagnostic message identification "****"
- the message code XY-Z where

X is the message number

Y is the severity code, a letter showing the severity of the error:

W: warning
E: error
S: severe error
F: fatal error

Z is an integer which, together with the message number **X**, uniquely identifies the compiler location that generated the diagnostic message; **Z** is of importance mainly to the compiler maintenance team – it does not contain information of interest to the compiler user.

The message code (with the exception of the severity code) will be suppressed if the parameter `SUPPRESS_ERROR_NO` in the configuration file has the value `TRUE` (see section 5.2.2).

- the message text; the text may include one context dependent field that contains the name of the offending symbol; if the name of the offending symbol is longer than 16 characters only the first 16 characters are shown.

Examples of diagnostic messages:

```
*** 18W-3: Warning: Exception CONSTRAINT_ERROR will be raised here
*** 320E-2: Name OBJ does not denote a type
*** 535E-0: Expression in return statement missing
```

*** 1508S-0: Specification for this package body not present in the library

5.4 The Program Library

This section briefly describes how the Ada compiler changes the program library. For a more general description of the program library, the user is referred to Chapter 3.

The compiler is allowed to read from all sublibraries constituting the current program library, but only the current sublibrary may be changed.

5.4.1 Correct Compilations

In the following examples it is assumed that the compilation units are correctly compiled, i.e., that no errors are detected by the compiler.

Compilation of a library unit which is a declaration

If a declaration unit of the same name exists in the current sublibrary, it is deleted together with its body unit and possible subunits. A new declaration unit is inserted in the sublibrary, together with an empty body unit.

Compilation of a library unit which is a subprogram body

A subprogram body in a compilation unit is treated as a secondary unit if the current sublibrary contains a subprogram declaration or a generic subprogram declaration of the same name and this declaration unit is not invalid. In all other cases it will be treated as a library unit, i.e.:

- when there is no library unit of that name
- when there is an invalid declaration unit of that name
- when there is a package declaration, generic package declaration, an instantiated package, or subprogram of that name

Compilation of a library unit which is an instantiation

A possible existing declaration unit of that name in the current sublibrary is deleted together with its body unit and possible subunits. A new declaration unit is inserted.

Compilation of a secondary unit which is a library unit body

The existing body is deleted from the sublibrary together with its possible subunits. A new body unit is inserted.

Compilation of a secondary unit which is a subunit

If the subunit exists in the sublibrary it is deleted together with its possible subunits. A new subunit is inserted.

5.4.2 Incorrect Compilations

If the compiler detects an error in a compilation unit, the program library will remain unchanged.

Note that if a file consists of several compilation units and an error is detected in any of these compilation units, the program library will not be updated for any of the compilation units.

5.5 Instantiation of Generic Units

This section describes the rules after which generic instantiation is performed.

5.5.1 Order of Compilation

When instantiating a generic unit, it is required that the entire unit, including body and possible subunits, be compiled before the first instantiation. This is in accordance with the ARM Chapter 10.3 (1).

5.5.2 Generic Formal Private Types

The present section describes the treatment of a generic unit with a generic formal private type, where there is some construct in the generic unit that requires that the corresponding actual type must be constrained if it is an array type or a type with discriminants, and there exists instantiations with such an unconstrained type (see ARM, Section 12.3.2(4)). This is considered an illegal combination. In some cases the error is detected when the instantiation is compiled, in other cases when a constraint-requiring construct of the generic unit is compiled:

- 1) If the instantiation appears in a later compilation unit than the first constraint-requiring construct of the generic unit, the error is associated with the instantiation which is rejected by the compiler.
- 2) If the instantiation appears in the same compilation unit as the first constraint-requiring construction of the generic unit, there are two possibilities:
 - a) If there is a constraint-requiring construction of the generic unit after the instantiation, an error message appears with the instantiation.
 - b) If the instantiation appears after all constraint requiring constructs of the generic unit in that compilation unit, an error message appears with the constraint-requiring construct, but will refer to the illegal instantiation.

- 3) The instantiation appears in an earlier compilation unit than the first constraint-requiring construction of the generic unit, which in that case will appear in the generic body or a subunit. If the instantiation has been accepted, the instantiation will correspond to the generic declaration only, and not include the body. Nevertheless, if the generic unit and the instantiation are located in the same sublibrary, then the compiler will consider it an error. An error message will be issued with the constraint-requiring construct and will refer to the illegal instantiation. The unit containing the instantiation is not changed, however, and will not be marked as invalid.

5.6 Uninitialized Variables

Use of uninitialized variables is not flagged by the compiler. The effect of a program that refers to the value of an uninitialized variable is undefined. A cross-reference listing may help to find uninitialized variables.

5.7 Program Structure and Compilation Issues

The following limitations apply to the DACS-80x86 Ada Compiler Systems for the Real Address Mode and 286 protected mode only:

- The Ada compiler supports a "modified large memory model for data references. The "modified large" memory model associates one data segment for each hierarchical sublibrary in the Ada program library. All package data declared within a sublibrary is efficiently referenced from Ada code compiled into the same sublibrary. A slight increase in code size results from referencing package data compiled into a different hierarchical level. Intel's medium memory model can thus be obtained by utilizing only one level of Ada program library, the root sublibrary.
- The Ada compiler supports a large memory model for executable code. Although the size of a single compilation unit is restricted to 32K words, the total size of the code portion of a program is not restricted.
- The space available for the static data of a compilation unit is 64K - 20 bytes.
- The space available for the code generated for a compilation unit is limited to 32K words.
- Any single object cannot exceed 64K - 20 bytes.

The following limitations apply to all DACS-80x86 products:

- Each source file can contain, at most, 32,767 lines of code.
- The name of compilation units and identifiers may not exceed the number of characters given in the INPUT_LINELENGTH parameter of the configuration file.
- An integer literal may not exceed the range of LONG_INTEGER, a real literal may not exceed the range of LONG_FLOAT.

DACS-80x86 User's Guide
Ada Compiler

- The number of formal parameters permitted in a procedure is limited to 127 per parameter specification. There is no limit on the number of procedure specifications. For example, the declaration:

```
procedure OVER_LIMIT (INTEGER01,  
                     INTEGER02,  
                     .....  
                     INTEGER166: in INTEGER);
```

exceeds the limit, but the procedure can be accomplished with the following:

```
procedure UNDER_LIMIT (INTEGER01 : in INTEGER;  
                      INTEGER02 : in INTEGER;  
                      .....  
                      INTEGER166 : in INTEGER);
```

The above limitations are diagnosed by the compiler. In practice these limitations are seldom restrictive and may easily be circumvented by using subunits, separate compilation, or creating new sublibraries.

5.8 Compiler Code Optimizations

DDC-I's Ada compiler for the iAPX 80x86 microprocessor family generates compact, efficient code. This efficiency is achieved, in part, by the compiler's global optimizer. Optimizations performed include:

- Common sub-expression elimination
- Elimination of redundant constraint checks
- Elimination of redundant elaboration checks
- Constant folding
- Dead code elimination
- Optimal register allocation
- Selection of optimal jumps
- Optional run-time check suppression

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.

6 THE ADA LINKER

The DACS linker must be executed to create an executable program in the target environment. Linking is a two stage process that includes an Ada link using the compilation units in the Ada program library, and a target link to integrate the application code, run-time code, and any additional configuration code developed by the user. The linker performs these two stages with a single command, providing options for controlling both the Ada and target link processes.

This chapter describes the link process, except for those options that configure the Run-Time System, which is described in detail in Chapter 7.

6.1 Invoking the Linker

Enter the following command at the shell to invoke the linker:

```
S ada_link {<option>} <unit-name>
```

where the options and parameters are:

Ada Linker Options

OPTION	DESCRIPTION	REFERENCE
-[no]debug	Links an application for use with the DACS-80x86 Symbolic Cross Debugger.	6.5.11
-enable_task_trace	Enables trace when a task terminates in unhandled exception.	6.5.28
-exception_space	Defines area for exception handling in task stack.	6.5.29
-[no]extract	Extracts Ada Object modules	6.5.14
-interrupt_entry_table	Range of interrupt entries.	6.5.27
-library	The library used in the link.	6.5.7
-[no]log	Specifies creation of a log file.	6.5.9
-lt_segment_size	Library task default segment size.	6.5.23
-lt_stack_size	Library task default stack size.	6.5.22
-mp_segment_size	Main program segment size.	6.5.25
-mp_stack_size	Main program stack size.	6.5.24
-[no]npx	Use of the 80x87 numeric coprocessor.	6.5.16
-options	Specifies target link options.	6.5.6
-priority	Default task priority.	6.5.18
-reserve_stack	Size of reserve stack.	6.5.21
-rms	Select Rate Monotonic Scheduling Run-Time Kernel (optional).	6.5.13
-[no]root_extract	Using non-DDC-I units in the root library.	6.5.10

DACS-80x86 User's Guide
The Ada Linker

-{no}rts	Includes or excludes the run-time system.	6.5.12
-searchlib	Target libraries or object modules to include in target link.	6.5.4
-selective_link	Removes uncalled code from final program.	6.5.8
-sign_on	Produce sign on and sign off messages.	6.5.30
-stop_before_link	Performs Ada link only.	6.5.5
-tasks	Maximum number of tasks or non-tasking application.	6.5.17
-task_storage_size	Tasks default storage size.	6.5.26
-template	Specifies template file.	6.5.15
-timer	Timer resolution.	6.5.20
-time_slice	Task time slicing.	6.5.19

All options may be abbreviated (characters omitted from the right) as long as no ambiguity arises. Casing is significant for options but not for options keywords.

Note: Several simultaneous links of the same program should not be performed in the same directory.

6.1.1 Diagnostic Messages

Diagnostic messages from the Ada Linker are output on the current output file and on the optional log file. The messages are output in the order they are generated by the linker.

The linker may issue two kinds of diagnostic messages: warnings and severe errors.

A warning reports something which does not prevent a successful linking, but which might be an error. A warning is issued if there is something wrong with the body unit of a program unit which formally does not need a body unit, e.g. if the body unit is invalid or if there is no object code container for the body unit. Warnings are only output on the log file, not on the current output file. The linking summary on the log file will contain the total number of warnings issued, even if the issued warnings have not been output.

A severe error message reports an error which prevents a successful linking. Any inconsistency detected by the linker will, for instance, cause a severe error message, e.g. if some required unit does not exist in the library or if some time stamps do not agree. If the linker is used for consequence examination, all inconsistencies introduced by the hypothetical recompilations are reported as errors.

A unit not marked as invalid in the program library may be reported as being invalid by the linker if there is something wrong with the unit itself or with some of the units it depends on.

6.2 The Linking Process

The linking process can be viewed as two consecutive processes. Both are automatically carried out when issuing the link command `ada_link`.

**DACS-80x86 User's Guide
The Ada Linker**

The first process constitutes the Ada link process and the second constitutes the target link process.

The Ada link process

- retrieves the required Ada object modules from the program library.
- determines an elaboration order for all Ada units.
- creates a module containing the User Configurable Data (UCD) from the specified configuration options to the linker and
- creates a shell script that carries out the target link process (i.e., `dlnkbldx86`). The locate/build phase is an integral part of the target link.

If the option `-stop_before_link` is NOT specified (default), the above script is executed automatically. Otherwise the linking process is halted at this point.

When `-stop_before_link` is specified, all temporary files are retrieved for inspection or modification. The target linker is invoked by executing the shell script.

6.2.1 Temporary Files

The following temporary files are in use during the link phase:

<code><main_program>_link.com</code>	The shell script which invokes the target linker.
<code><main_program>_elabcode.o</code>	The object code for the calling sequence of the elaboration code.
<code><main_program>_ucd.o</code>	The object code generated from the RTS configuration options (see Section 7.2).
<code><main_program>_uxxxx.o</code>	The Ada object modules which have been extracted from the program library. <code>xxxxx</code> is the unit number of the Ada unit.

DACS-80x86 User's Guide
The Ada Linker

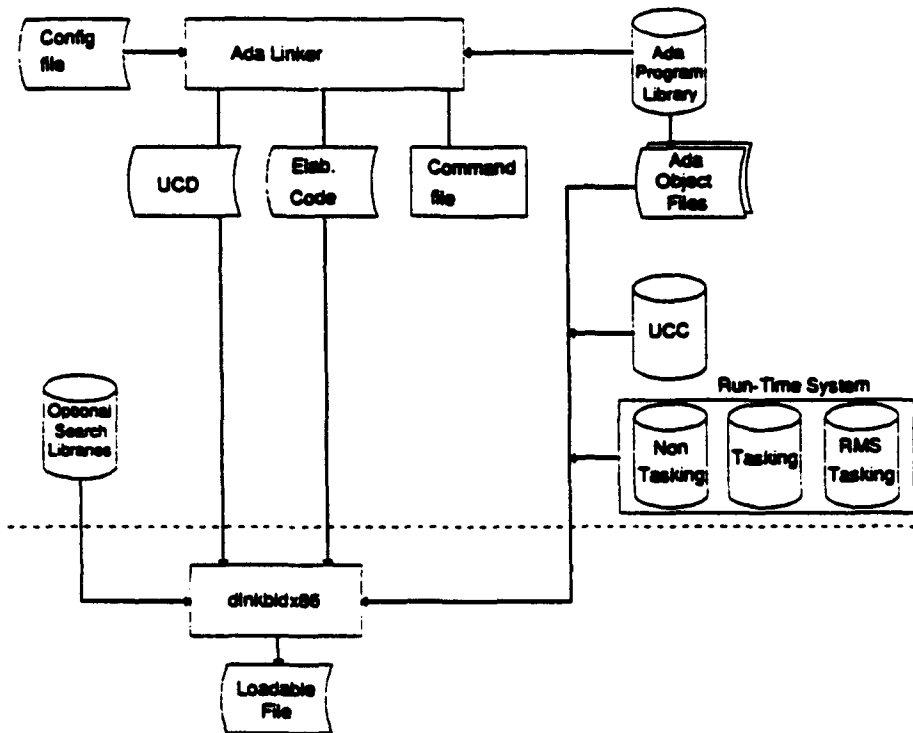


Figure 6-3. The Linking Process

The following components make up the run-time system:

- 1) User configurable portion of the RTS
 - a) User configurable data (UCD) and
 - b) User configurable code (UCC)
- 2) Permanent part of the RTS
 - a) Non-tasking RTS (r11.lib) or
 - b) Tasking RTS (r12.lib)
 - c) RMS Tasking RTS (r13.lib)

The User Configurable Code defined by the environmental variable `ada_ucc_lib` is included in the link. If no tasking has been specified, then the RTS non-tasking library (r11.lib) will be included. If tasking has been specified, then support for tasking will be included (r12.lib or, when `-rms`, r13.lib).

DACS-80x86 User's Guide
The Ada Linker

The output of the linker step is an absolute executable object file with the extension ".dat" and a map file with the extension ".mp5".

6.2.2 Environmental Variables

When a link is executed, a number of files are referred to and most are accessed through environmental variables. The locate/build phase uses the control file \$ada_ucc_dir/config.bld_ddci, the remaining variables are:

VARIABLE	PURPOSE
ada_system_library	Identifies the root library where the system compilation units reside.
ada_library	Identifies the default library used by all DACS-80x86 tools. It is the lowest level sublibrary in the program library hierarchy.
ada_root_lib	Identifies the OMF library where the system library units have been extracted from the system library. By having a separate Library for the root compilation units, the link process is much faster than otherwise having to extract each unit from the system library for each link.
ada_r11_lib	Identifies the OMF library for the Permanent Part of the non-tasking version of the Run-Time System.
ada_r12_lib	Identifies the OMF library for the Permanent Part of the tasking version of the Run-Time System.
ada_r13_lib	Identifies the OMF library for the Permanent Part of the optional Rate Monotonic scheduling Run-Time System.
ada_ucc_lib	Identifies the OMF library for the User Configurable Code portion of the Run-Time System.
ada_template	Identifies the template file for the Linker.
ada_ucc_dir	Identifies the directory of the current UCC.

With each of these environmental variables, the name will differ depending on how the system was installed (ada86, ada186 etc). Throughout this document ada is assumed. For example, the environmental variables for the root library for the 80186 version of the compiler would be `ada186_root_lib`, and the RTS UCC library environmental variables for the 8086 version would be `ada86_ucc_lib`.

6.3 Run-Time System Overview

The Run-Time System for DACS-80x86 is defined as all code and data, other than the code and data produced by the code generator, required to make an embedded system application operate properly on a specific hardware system.

In general, there are two major components that make up the Run-Time System.

- 1) Code and data assumed to exist by the code generator. This is hardware independent and known as the RTS Permanent Part.
- 2) Code and data tailoring the application with respect to the characteristics of the hardware and other requirements of the embedded systems developer. This code is called the RTS User Configurable Part.

Both of the above components consist of modular OMF libraries. The modules are only included in the user program if they are needed, i.e., if a call or reference is made to the module. This ensures a compact RTS (typical applications are 4 KB to 10 KB).

The RTS Permanent Part does not make any assumptions about the hardware other than an 80x86 and some amount of memory available.

There are several versions of the RTS User Configurable Part available for different development targets. Also, the source code is provided to allow the modification of the User Configurable Code (UCC) to operate on other targets. Refer to the RTS Configuration Guide for complete information on modifying the UCC.

DDC-I has carefully analyzed and selected the parts of the Run-Time System that must be configurable for hardware independence, freeing the user from major rewrites whenever the Run-Time System is retargeted while, still allowing for almost unlimited adaptability.

Four important features of the run-time system are:

- It is small
- It is completely ROMable
- It is configurable
- It is efficient

Conceptually, an Ada run-time system can be viewed as consisting of the following components:

- Executive, i.e., the start-up mechanism
- Storage Management
- Tasking Management
- Input/Output
- Exception Handling

- Run-Time Library Routines
- Package CALENDAR support routines

The run-time system (RTS) can be configured by the user through Ada Linker command options. The Ada Linker will generate appropriate data structures to represent the configured characteristics (UCD).

Two versions of the RTS are supplied, one including tasking and one excluding tasking. The linker selects the RTS version including tasking only if the option `-tasks` is present or `-tasks n` is present and $n > 0$. Otherwise, the linker selects the RTS version excluding tasking.

6.4 Linker Elaboration Order

The elaboration order is primarily given by the unit dependencies, but this leaves some freedom here and there to arbitrarily choose between two or more alternatives. This arbitrary is in the DACS-80x86 linker controlled by the *spelling* of the involved library units, in order for "free" units to become alphabetically sorted.

Recompiling from scratch, an entire system may thus affect the allocation of unit numbers, but the elaboration order remains the same.

It is also attempted to elaborate "body after body", so that a body having a with to a specification, will be attempted elaborated *after* the body of this specification.

Also elaboration of units from different library levels is attempted to complete elaboration of a father-level prior to the son-level.

This strategy should in many cases reduce the need for resetting pragma ELABORATE.

6.5 Ada Linker Options

This section describes in detail the Ada linker option and parameters.

6.5.1 The Parameter <unit-name>

<unit-name>

The <unit_name> must be a library unit in the current program library, but not necessarily of the current sublibrary.

Note that a main program must be a procedure without parameters, and that <unit-name> is the identifier of the procedure, not a file specification. The main procedure is not checked for parameters, but the execution of a program with a main procedure with parameters is undefined.

6.5.2 The Parameter <recompilation-spec>

The syntax of <recompilation-spec> is:

<unit_spec>[-body|-specification][,...]

This parameter tells the linker to perform a consistency check of the entire program using the hypothetical recompilation of all units designated in the <recompilation-spec>. The link process in this instance is not actually performed.

The <unit_spec> is a list of unit-names (wildcards are allowed), separated by comma (,) or plus (+). Each unit-name should include an option to indicate if the body or specification is to be hypothetically compiled (-spec is the default).

6.5.3 Required Recompilations

If the consistency check found that recompilations are required, a list of required recompilations is written to the current output file or to a text file if the -log option is specified (the name of the text file is indicated in the log file, line 8). The list will include any inconsistencies detected in the library and recompilations required by the hypothetical recompilations specified with the options -declaration and -body.

The entries in the list contain:

- 1) The unit name.
- 2) Indication of what type of unit (declaration unit, body unit, or subunit).
- 3) If the unit is specified as recompiled with the -declaration or -body option, it is marked with "-R-".
- 4) The environmental variable of the sublibrary containing the unit.

In the recompilation list the units are listed in a recommended recompilation order, consistent with the dependencies among the units.

6.5.4 -searchlib

-searchlib <file_name> {,<file_name>}

The -searchlib option directs the Ada Linker to search the specified 80x86 target libraries for object modules in order to resolve symbol references. The 80x86 target libraries for object files will be searched before the DACS Run-Time System (RTS) library normally searches for run-time routines; in this way one can replace the standard DACS RTS routines with custom routines.

The -searchlib option is also intended to specify libraries of modules referenced from Ada via pragma INTERFACE.

Examples:

```
S ada_link -searchlib interface_lib p
```

Links the subprogram p, resolving referenced symbols first with the target library interface_lib and then with the standard RTS target library.

6.5.5 -stop_before_link

-stop_before_link

The **-stop_before_link** option allows the user to introduce assemblers and linkers from third parties or to otherwise configure the link to suit the application. The link is halted with the following conditions:

The user configurable data file, <main>_ucd.o, is produced with the default or user specified linker option values included.

- The elaboration code is contained in the <main>_elabcode.o file.
- The shell script file that contains the link command is present and has not been executed. The file's name is <main>_link.com.
- The temporary Ada object file(s) used by the target linker are produced. These objects are linked and deleted when <main>_link.com is executed.
- With **-selective_link** the object files comprise all Ada units including those from the root library. At this point it is possible to disassemble the "cut" object files using **-object** with the disassembler.

To complete the link, the <main>_link.com script must be executed. To use third party tools, this file may have to be modified.

6.5.6 -options

-options <parameter>

-options allow the user to pass options onto the target linker.

6.5.7 -library

-library <file-name>
-library Sada_library (default)

The **-library** option specifies the current sublibrary, from which the linking of the main unit will take place. If this option is not specified, the sublibrary specified by the environmental variable **ada_library** is used.

6.5.8 -selective_link

-selective_link

This extracts all required object modules from the Ada library (including the root library) and cuts out exactly those parts that are actually called, in order to make the resulting target program considerably smaller. If a program uses e.g. **PUT_LINE** as the only routine from **TEXT_IO**, the contribution from the **TEXT_IO** object module will only contain **PUT_LINE** (and whatever that needs). Note that disassemblies of units used in a selective link normally will not match what is linked, because of the cutting. Such disassemblies may though be obtained by disassembling directly those units that made up the selective link, by stopping the linking before the target link phase (**-stop_before_link**), making disassemblies using **-object** and then resuming the link.

Note also that unused constants and permanent variables are not removed.

Only "level 1" subprograms may be removed. Nested subprograms (that are not called) are to be removed during compilation using the **-optimize** option. Nested subprograms are only removed, if the routine in which the nesting occurs is removed.

6.5.9 -[no]log

-log [<file-spec>]
-nolog (default)

The option specifies if a log file will be produced from the front end linker. As default, no log file is produced. If **<file-spec>** is not entered with **-log** the default file name for the log file will be **link.log** in the current directory.

The log file contains extensive information on the results of the link. The file includes:

- An elaboration order list with an entry for each unit included, showing the order in which the units will be elaborated. For each unit, the unit type, the time stamp, and the dependencies are shown. Furthermore, any elaboration inconsistencies will be reported.
- A linking summary with the following information:
 - Parameters and active options.
 - The full name of the program library (the current sublibrary and its ancestor sublibraries).

- The number of each type of diagnostic message.
- A termination message, stating if the linking was terminated successfully or unsuccessfully or if a consequence examination was terminated.
- Diagnostic messages and warnings are written on the log file.

If recompilations are required (as a result of the consistency check) a text file is produced containing excerpts of the log file. The name of this text file is written in the log file, line 8.

The log file consists of:

- Header consisting of the linker name, the linker version number, and the link time.
- The elaboration order of the compilation units. The units are displayed in the order elaborated with the unit number, compilation time, unit type, dependencies, and any linking errors.
- If recompilations are required, the units that must be recompiled are listed along with its unit type and sublibrary level.
- The linking summary that includes the main unit name, the program library, any recompilations that are required, and if any errors or warnings occurred.

6.5.10 `-[no]root_extract`

`-root_extract`
`-noroot_extract` (default)

The units contained in the Ada system library supplied by DDC-I have been extracted and inserted into the `Sada_root_lib` OMF Library, thus eliminating extractions from the system library at link time and improving link performance.

The user should normally not modify or compile into the Ada system library supplied by DDC-I. If however, a unit is compiled into the Ada system library, the `Sada_root_lib` will no longer match the Ada system library and `-root_extract` must be specified in order to link from the Ada system library.

6.5.11 `-[no]debug`

`-debug`
`-nodebug` (default)

The `-debug` option specifies that debug information is generated. The debug information is required to enable symbolic debugging. If `-nodebug` is specified, the Ada linker will skip the generation of debug information, thus saving link time, and will not insert the debug information

into the chosen sublibrary, thus saving disk space. Note that any unit which should be symbolically debugged with the DDC-I Ada Symbolic Cross Debugger must also be compiled with the `-debug` option.

6.5.12 `-(no)rts`

`-rts` (default)
`-norts`

The `-rts` option directs the Ada Linker to include the appropriate Run-Time System (RTS) in the link. `-norts` directs the Ada Linker to exclude the RTS in the link.

The ability to exclude the Run-Time System from the link allows the user to do an additional link with a private copy of a custom RTS. The Ada Linker may report unresolved references to RTS routines, but will still produce a relocatable object file.

6.5.13 `-rms`

`-rms`

This option selects the Rate Monotonic Scheduling Tasking Kernel (if tasking is selected). The default is to use the Standard Tasking Kernel. This feature is supplied as an option.

6.5.14 `-(no)extract`

`-extract` (default)
`-noextract`

This option to the linker allows the user to specify that program unit objects should not be extracted from the Ada program library. This option would be used if the user knows that many objects have not changed since the last link and does not want the linker to waste time extracting them.

To use this feature, the user should modify the template to not delete unit object files after a target link is performed. This way the object files remain in the current directory (or wherever the user decides to put them). On subsequent links the user can extract object modules of modified units from the Ada library using the standalone DACS extract tool. A new target link can then be performed using a combination of newly extracted objects and the object files from previous links that have gone unchanged. This could significantly improve linker speed when linking programs that share common and rarely modified libraries and when relinking programs that have had only a few units modified.

6.5.15 -template

-template <file-name>
-template Sada_template (default)

The template file is known to the linker via the environmental variable `ada_template`. DDC-I supplies a default template file as part of the standard release system. Please refer to appendix H for detailed information.

6.5.16 -npx

-npx (default)
-nonpx

The `-npx` option specifies that the 80x87 (8087, 80287, or 80387) numeric coprocessor is used by the Ada program. When `-npx` is specified, the 80x87 is initialized by the task initialization routine, the floating point stack is reset during exception conditions, and the 80x87 context is saved during a task switch.

Configurable Data

A 16 bit boolean constant is generated by the Ada Linker:

```

    _CD_NPX_USED boolean
= 0 - 80x87 is not used
= 1 - 80x87 is used
```

6.5.17 -tasks

-tasks [n]
(default is no tasking)

This option specifies the maximum number of tasks allowed by the RTS. If specified, `n` must be greater than zero. If `-tasks` is specified without a value for `n`, `n` defaults to 10. If `-tasks` is not specified, the RTS used will not include support for tasking. If `-tasks` is specified, the RTS used will include support for tasking.

Ada Interrupt tasks identified with `pragma INTERRUPT_HANDLER` need not be included in the count of maximum number of tasks. The main program must be counted in the maximum number of tasks. Note that the main program, which may implicitly be considered a task, will not run under control of the tasking kernel when `-notasks` is specified. See also `-rms` option.

Configurable Data

For `-tasks`, the linker generates the following configurable data:

`_CD_MAX_TASKS`

INTEGER = N

`_CD_TCBS`

N Task
Control
Blocks
(TCBS)

If `-npx` is
active, N
numeric co-
processor
save areas

Example:

```
$ ada_link -tasks 3 p
```

- Link the program P, which has at most 3 tasks, including the main program.

6.5.18 -priority

```
-priority n  
-priority 15      (default)
```

The `-priority` option specifies the default priority for task execution. The main program will run at this priority, as well as tasks which have had no priority level defined via `pragma PRIORITY`. The range of priorities is from 0 to 31.

Priorities can be set on a per task basis dynamically at run time. See section E.1 (Package `RTS_EntryPoints`) for more details.

Configurable Data

The Ada Linker generates the following constant data:

`_CD_PRIORITY`

Constant = N

Example:

```
$ ada_link -tasks -priority 8 p
```

- Link the subprogram P which has the main program and tasks running at default priority 8.

6.5.19 -time_slice

-time_slice [r] (default no time slicing is active)

The **-time_slice** options specifies whether or not time slicing will be used for tasks. If specified, R is a decimal number of seconds representing the default time slice to be used. If R is not specified, the default time slice will be 1/32 of a second. R must be in the range $\text{Duration}'\text{Small} \leq R \leq 2.0$ and must be greater than or equal to the **-timer** linker option value. Time slicing only applies to tasks running at equal priority. Because the RTS is a preemptive priority scheduler, the highest priority task will always run before any lower priority task. Only when two or more tasks are running at the same priority is time slicing applied to each task.

Time slicing can be specified on a per task basis dynamically at run-time. See Section E.1 (Package `RTS_EntryPoints`) for more details.

Time slicing is not applicable unless tasking is being used. This means that the **-tasks** option must be used for **-time_slice** to be effective.

Configurable Data

The Ada Linker generates the following data:

`_CD_TIME_SLICE_USED`
= 0 - No time slicing
= 1 - Time slicing

BOOLEAN

`_CD_TIME_SLICE`

absolute integer

- representing the number Y that satisfies $Y * \text{Duration}'\text{Small} = R$

Example:

```
$ ada_link -time_slice 0.125 -tasks p
```

- Specifies tasks of equal priority to be time sliced each eighth of a second.

6.5.20 -timer

-timer R
-timer 0.001 (default)

The **-timer** option specifies the resolution of calls to the Run-Time System routine `TIMER` (see the Run-Time System Configuration Guide for DACS-80x86 for more information). The number, R, specifies a decimal number of seconds which have elapsed for every call to `TIMER`. The default `TIMER` resolution is one millisecond. R must be in the range $\text{Duration}'\text{Small} < R < 2$.

Configurable Data

The Ada Linker generates the following 16 bit constant:

`_CD_TIMER`

Absolute Integer

- representing the number Y that satisfies $Y * \text{DURATION}'\text{SMALL} = R$

6.5.21 -reserve_stack

`-reserve_stack [n]`

The `-reserve_stack` option designates how many words are reserved on each task stack. This space is reserved for use by the RTS, which does no checking for stack overflow. This reserved space also allows the RTS to function in situations such as handling a storage error exception arising from stack overflow.

The `-reserve_stack` option also reserves part of the main program stack size, specified by the linker option `-mp_stack_size`.

Configurable Data

The Ada Linker generates the following integer constant:

`_CD_RESERVE_STACK`

INTEGER

Examples:

```
S ada_link -reserve_stack 200 -tasks p
```

- Reserve 200 words from each stack for use by the RTS.

6.5.22 -lt_stack_size

`-lt_stack_size n`

`-lt_stack_size 500(default)`

The `-lt_stack_size` option designates the library task default size in words. A library task is formed when a task object is declared at the outermost level of a package. Library tasks are created and activated during the initial main program elaboration. (See the Ada Reference Manual for more details).

For each library task, the representation spec:

FOR Task_object'STORAGE_SIZE USE N;

can be used to specify the library task stack size. However, if the representation spec is not used, the default library task size specified by `-lt_stack_size` will be used.

For efficiency reasons, all tasks created within library tasks will have stacks allocated within the same segment as the library task stack. Normally, the segment which contains the library task stack is allocated just large enough to hold the default library task stack. Therefore, one must use the option `-lt_stack_option` or the pragma `LT_SEGMENT_SIZE` to reserve more space within the segment that may be used for nested tasks' stacks. (See the implementation dependent pragma `LT_SEGMENT_SIZE` in Section F.1 for more information).

The range of this parameter is limited by physical memory size, task stack size allocated during the build phase of the link, and the maximum segment size (64K for all except the 386/486 protected mode, which is 4 GB).

Configurable Data

The Ada Linker generates the following integer constant:

`_CD_LT_STACK_SIZE`

INTEGER

Example:

```
$ ada_link -lt_stack_size 2048 -tasks p
```

- Link the subprogram P using a 2K words default library stack size.

6.5.23 `-lt_stack_size`

`-lt_segment_size n`

`-lt_segment_size (lt_stack_size + 20 + exception_stack_space)` (default)

This parameter defines in words the size of a library task segment. The library task segment contains the task stack and the stacks of all its nested tasks.

The default value is only large enough to hold one default task stack. If `-lt_stack_size` is used and specifies a value other than the default value, `-lt_segment_size` should also be specified to be the size of `<task_stack_size> + <total_of_nested_tasks_sizes> + <20_words_overhead> + exception_stack_space`.

Note that the task stack size specified by the 'STORAGE_size can be representation spec or by the option `-lt_stack_size`.

Dynamically allocated tasks receive their own segment equal in size to the `mp_segment_size`.

DACS-80x86 User's Guide
The Ada Linker

The range of this parameter is limited by physical memory size, task stack size allocated during the build phase, and the maximum segment size (64K for all except the 386/486 protected mode, which is 4 GB).

Configurable Data

The Ada Linker generates the following data structure:

`_CD_LT_SEGMENT_SIZE` INTEGER

Example:

```
S ada_link -lt_segment_size 2048 -tasks p
```

- Link the program P using a library task segment size of 2K words.

6.5.24 -mp_stack_size

`-mp_stack_size n`
`-mp_stack_size 8000` (default)

The `-mp_stack_size` option specifies the main program stack size in words.

The range of this parameter is limited by physical memory size, task stack size allocated during the build phase (in tasking programs only), the maximum segment size (64K for all except the 386/486 protected mode, which is 4 GB), and the size of `mp_segment_size`.

Configurable Data

The Ada Linker generates the following data structures for nontasking programs:

`_CD_MP_STACK_SIZE` INTEGER

`_CD_MP_STACK` MP_STACK_SIZE
words of
storage

`_CD_MP_STACK_START` Highest addr.
of MP stack

For tasking programs, the Ada Linker generates the same structures but limits the size to 1024 words. This stack is only used for the execution of the system startup code and elaboration. At main program activation, a segment for the main program equal to the size specified by `-mp_segment_size` will be allocated from the dynamic memory pool and a stack for the main program equal to the size specified by `-mp_stack_size` will be allocated from the memory pool.

Example:

```
$ ada_link -mp_stack_size 1000 p
```

- Link the subprogram P with a stack of 1000 words.

6.5.25 -mp_segment_size

```
-mp_segment_size n  
-mp_segment_size 8100 (Default)
```

The `-mp_segment_size` option specifies the size, in words, of the segment in which the main program stack is allocated. The default setting can be calculated from the formula:

$$\text{mp_segment_size} = \text{mp_stack_size} + \text{overhead} + (\text{tasks} - 1) * (\text{overhead} + \text{task_storage_size})$$

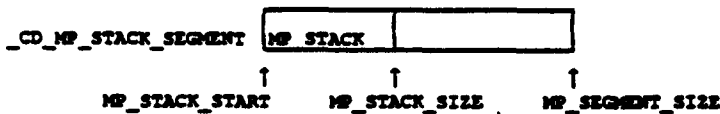
Normally, the main program segment size can be set to the size of the main program stack. However, when the main program contains nested tasks, the stacks for the nested tasks will be allocated from the data segment which contains the main program stack. Therefore, when the main program contains nested tasks, the main program stack segment must be extended via the `-mp_segment_size` option.

The range of this parameter is limited by physical memory size, task stack size allocated during the build phase (in tasking programs only), and the maximum segment size (64K for all except the 386/486 protected mode, which is 4 GB).

Note: Dynamically allocated tasks receive their own segment equal in size to `mp_segment_size`.

Configurable Data

The Ada Linker allocates the `_CD_MP_STACK` (see the `-mp_stack_size` option) within a data segment called `_CD_MP_STACK_SEGMENT`:



Example:

```
$ ada_link -tasks -mp_segment_size 32000 program_a
```

Links the subprogram PROGRAM_A, which contains tasks nested in the main program allocating 32,000 words for the main program stack segment.

6.5.26 -task_storage_size

-task_storage_size n
-task_storage_size 1024 (default)

This option sets the default storage size in words for stacks of tasks that are not library tasks. This value can be overridden with a representation clause.

The range is limited by the size of the `lt_segment_size` (if it is a subtask to a library task), or by `mp_segment_size` (if it is a subtask to the main program).

Configurable Data

The Ada Linker generates the following data structure:

`_CD_TASK_STORAGE_SIZE` INTEGER

6.5.27 -interrupt_entry_table

-interrupt_entry_table L,H

The `-interrupt_entry_table` option specifies the range of interrupt vector numbers used by the Ada program in interrupt tasks.

The number, L, specifies the lowest numbered interrupt handler. The number, H, specifies the highest numbered interrupt handler. The range for low and high interrupts is 0 to 255.

Configurable Data

If `-interrupt_entry_table` is specified, the Ada Linker will generate the following data structure:

<code>_CD_LOW_INTERRUPT</code>	CONSTANT	(L)
<code>_CD_HIGH_INTERRUPT</code>	CONSTANT	(H)
<code>_CD_INTERRUPT_VECTOR</code>	(H-L+1)*5 words reserved for Interrupt Vector	

If the user ever detects unresolved references to the symbols:

`_CD_LOW_INTERRUPT`
`_CD_HIGH_INTERRUPT`
`_CD_INTERRUPT_VECTOR`

the Ada program contains standard interrupt tasks for which the RTS requires the above data structure. You must relink the Ada program specifying the `-interrupt_entry_table` option.

Example:

```
$ ada_link -tasks -interrupt_entry_table 5,20 p
```

- Links the subprogram P, which has standard Ada interrupt entries numbered 5 through 20.

6.5.28 `-(no)enable_task_trace`

`-enable_task_trace`
`-noenable_task_trace` (default)

This option instructs the exception handler to produce a stack trace when a task terminates because of an unhandled exception.

Configurable Data

`_CD_TRACE_ENABLED`

BOOLEAN

- = 0 - task trace disabled
- = 1 - task trace enabled

6.5.29 `-exception_space`

`-exception_space n`
`-exception_space 0a0h` (default)

Each stack will have set its top area aside for exception space. When an exception occurs, the exception handler may switch stack to this area to avoid accidental overwrite below the stack bottom (which may lead to protection exceptions) if the size of the remaining part of the stack is smaller than the N value. Specifying a value =0 will never cause stack switching. Otherwise an N value below the default value is not recommended.

Configurable Data

`_CD_EXCEPTION_STACK_SPACE_SIZE`

INTEGER

Note that this value is added to all requests for task stack space, thus requiring an increase in the requirements of the appropriate segment's size

6.5.30 -sign_on

-sign_on [<string>]

When this option is specified the linker will generate code to output a sign on message, before the Ada elaboration is initiated and a sign off message when the target program has terminated successfully. If the program terminates with an uncaught exception, the sign off message is not printed.

The sign on message consists of:

START [<string>] <program name>

and the sign off message

STOP [<string>] <program name>

The <string> may contain spaces, e.g.

-sign_on "Test 3" (remember the quotes).

This facility is very useful to separate output from several target programs run after each other, and to verify that a program that produces little or no output has actually been loaded and run successfully.

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 -128 .. 127;

type INTEGER is range -32_768 .. 32_767;

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

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;

APPENDIX F - IMPLEMENTATION-DEPENDENT CHARACTERISTICS

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 Pragma

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 are invalid in Ada subprogram identifiers. This pragma must be used in conjunction with pragma INTERFACE, i.e., pragma INTERFACE must be specified for the 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");
```

The string literal may be appended 'NEAR (or 'FAR) to specify a particular method of call. The default is 'FAR. This suffix should only be used, when the called routines require a near call (writing 'FAR is however harmless). If 'NEAR is added, the routine must be in the same segment as the caller.

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 (see section 7.1).

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 from 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.1.5 Pragma MONITOR_TASK

F.1.5.1 Function

The pragma MONITOR_TASK is used to specify that a task with a certain structure can be handled in a special way by the Run-Time System, enabling a very efficient context switch operation.

F.1.5.2 Format

The format of the pragma is

```
pragma MONITOR_TASK;
```

The pragma must be given in a task specification before any entry declarations.

F.1.5.3 Restrictions

The following restrictions apply on tasks containing a pragma MONITOR_TASK :

- Only single anonymous tasks can be "monitor tasks".
- Entries in "monitor tasks" must be single entries (i.e. not family entries).
- The task and entry attributes are not allowed for "monitor tasks" and "monitor task" entries.
- The <declarative part> should only contain declaration of objects; no types or nested structures must be used.
- The structure of the task body must be one of the following:

```
1.
task body MON_TASK is
  <declarative part>
begin
  <statement list>
  loop
    select
      accept ENTRY_1<parameter_list> (do
        end);
      or
      accept ENTRY_2<parameter_list> (do
        <statement_list>
        end);
      or
      terminate
    end select;
  end loop;
end;
```

where each entry declared in the specification must be accepted unconditionally exactly once.

DACS-80x86 User's Guide Implementation-Dependent Characteristics

```
2.
task body MON_TASK is
  <declarative part>
begin
  <statement list>
  loop
    accept MON_ENTRY<parameter_list>{do
      <statement_list>
    end};
  end loop;
end;
```

where the task only has one entry.

In both cases the declarative parts, the statement lists and the parameter lists may be empty. The statement list can be arbitrarily complex, but no nested select or accept statements are allowed.

No exception handler in the monitor task body can be given.

The user must guarantee that no exceptions are propagated out of the accepts.

F.1.5.4 Example

The following tasks can be defined

```
task LIST_HANDLER is
  pragma MONITOR TASK;
  entry INSERT(ELEM:ELEM_TYPE);
  entry REMOVE(ELEM:out ELEM_TYPE);
  entry IS_PRESENT(ELEM:ELEM_TYPE,
                  RESULT:out BOOLEAN);
end LIST_HANDLER;

task body LIST_HANDLER is
  "define list"
begin
  "initialize list"
  select
    accept INSERT(ELEM:ELEM_TYPE) do
      "insert in list"
      end INSERT;
    or
    accept REMOVE(ELEM:out ELEM_TYPE) do
      "find in list and remove from list"
      end REMOVE;
    or
    accept IS_PRESENT(ELEM:ELEM_TYPE,
                     RES: out BOOLEAN) do
      "scan list"
      end IS_PRESENT;
    or
    terminate;
  end select;
end MON_TASK;
```

The task can be used

```
task type LIST_USER is
  ...
end LIST_USER;

task body LIST_USER is
  ...
```

DACS-80x86 User's Guide
Implementation-Dependent Characteristics

```
begin
  select
    LIST_HANDLER.INSERT(FIRST_ELEM);
  else
    raise INSERT_ERROR;
  end select;
  loop
    LIST_HANDLER.INSERT(NEXT_ELEM);
  end loop;
end LIST_USER;
```

F.1.6 Pragma TASK_STORAGE_SIZE (T, N)

This pragma may be used as an alternative to the attribute 'TASK_STORAGE_SIZE to designate the storage size (N) of a particular task object (T) (see section 7.1).

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:

<u>Compiler System</u>	<u>System Name</u>
DACS-8086	iAPX86
DACS-80186	iAPX186
DACS-80286 Real Mode	iAPX286
DACS-80286 Protected Mode	iAPX286_PM

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;
```

DACS-80x86 User's Guide
Implementation-Dependent Characteristics

```

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 := 291.09E-31;
TICK : constant := 0.000_000_125;

type Interface_language is
    (ASMS6, FLMS6, CS6, CS6_REVERSE,
     ASM_ACF, FLM_ACF, C_ACF, C_REVERSE_ACF,
     ASM_WOACT, FLM_WOACT, C_WOACT, C_REVERSE_WOACT);

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 MDS.
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;

```

DACS-80x86 User's Guide
Implementation-Dependent Characteristics

```

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_ACT,   PLM_ACT,   C_ACT,   C_REVERSE_ACT,
     ASM_NOACT, PLM_NOACT, C_NOACT, C_REVERSE_NOACT);

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 BDS.
end record;

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

```

F.4 Representation Clauses

The DACS-80x86™ fully supports the 'SIZE representation for derived types. The representation clauses that are accepted for non-derived types are described in the following subsections.

F.4.1 Length Clause

Some remarks on implementation dependent behavior of length clauses are necessary:

- When using the SIZE attribute for discrete types, the maximum value that can be specified is 16 bits. For DACS-80386PM/80486PM the maximum is 32 bits.
- SIZE is only obeyed for discrete types when the type is a part of a composite object, e.g. arrays or records, for example:

```

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

sixteen_bits_allocated : byte;           -- one word allocated

```

DACS-80x86 User's Guide
Implementation-Dependent Characteristics

```
eight_bit_per_element : array(0..7) of byte; -- four words allocated
type rec is
  record
    c1,c2 : byte; -- eight bits per component
  end record;
```

- Using the STORAGE_SIZE attribute for a collection will set an upper limit on the total size of objects allocated in this collection. If further allocation is attempted, the exception STORAGE_ERROR is raised.
- When STORAGE_SIZE is specified in a length clause for a task type, the process stack area will be of the specified size. The process stack area will be allocated inside the "standard" stack segment. Note that STORAGE_SIZE may not be specified for a task object.

F.4.2 Enumeration Representation Clauses

Enumeration representation clauses may specify representations in the range of -32767..+32766 (or -16#7FFF..16#7FFE).

F.4.3 Record Representation Clauses

When representation clauses are applied to records the following restrictions are imposed:

- if the component is a record or an unpacked array, it must start on a storage unit boundary (16 bits)
- a record occupies an integral number of storage units (words) (even though a record may have fields that only define an odd number of bytes)
- a record may take up a maximum of 32K bits
- a component must be specified with its proper size (in bits), regardless of whether the component is an array or not (Please note that record and unpacked array components take up a number of bits divisible by 16 (=word size))
- if a non-array component has a size which equals or exceeds one storage unit (16 bits) the component must start on a storage unit boundary, 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 elements in an array component should always be wholly contained in one storage unit
- if a component has a size which is less than one storage unit, it must be wholly contained within a single storage unit:

component at N range X .. Y;

where N is as in previous paragraph, and $0 \leq X \leq Y \leq 15$. Note that for this restriction a component is not required to start in an integral number of storage units from the beginning of the record.

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.

Pragma pack on a record type will attempt to pack the components not already covered by a representation clause (perhaps none). This packing will begin with the small scalar components and larger components will follow in the order specified in the record. The packing begins at the first storage unit after the components with representation clauses.

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 level higher 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 clause may denote a dynamic value.

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. This form requires the use of pragma `INTERRUPT_HANDLER`.
- 2) Mapping of an interrupt 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 an interrupt 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:

- Provide the fastest possible response time to an interrupt.
- Allow entry calls to other tasks during interrupt servicing.
- Allow procedure and function calls during interrupt servicing.
- Does not require its own stack to be allocated.
- Can be coded in packages with other declarations so that desired visibility to appropriate parts of the program can be achieved.
- 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 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:

- It must be a task object, not a task type.
- The pragma must appear first in the specification of the task object.
- All entries of the task object must be single entries (no families) with no parameters.
- The entries must not be called from any task.
- 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.
- 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 task, not another fast interrupt task.
- The task may only reference global variables; no data local to the task may be defined.
- The task must be declared in a library package, i.e., at the outermost level of some package.
- 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 queuing 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 and sets (E)BP to look like a frame where the interrupt return address works as return address.

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_EntryPoints` 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 80386/80486 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 option `-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. 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 occurs. 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` option 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.

If an entry call is made by a fast interrupt handler the interrupt number must be included in the `-interrupt_entry_table` option at link time. This option 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 link 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 no special actions need be performed. The Ada Link automatically recognizes Ada interrupt handlers and adds them to the IDT.

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 80386/80486 protected mode programs using fast interrupt handlers.

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;
```

Compilation and Linking:

```
$ ada Example_5  
$ ada_link -tasks - Example_5
```

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.

**DACS-80x86 User's Guide
Implementation-Dependent Characteristics**

- 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` option 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 interrupt tasks in the `-interrupt_entry_table` option 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 INTERRUPTVECTORTABLE 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` option. 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 interrupt task must be accounted for in the `-interrupt_entry_table` option. This option 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.6.7 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;
  end select;
  end loop;
end Normal_Task;
end P;
```

DACS-80x86 User's Guide
Implementation-Dependent Characteristics

```
        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.

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;
```

Compilation and Linking:

```
$ ada Example_3
$ ada_link -tasks -interrupt_entry_Table 20,20 Example_3
```

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 to 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.

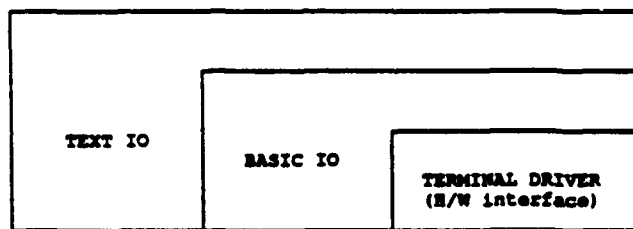
If these techniques are used for either variant of interrupt handlers, caution must be taken that other tasks do not call the task entry which reenables interrupts if this can cause adverse side effects.

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

DACS-80x86 User's Guide
Implementation-Dependent Characteristics

package that need be changed when changing communications controllers. The actual body of the `TERMINAL_DRIVER` is written in assembly language and is part of the UCC modules `DIINPUT` and `DIIGET`. The 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:

<code>TEXT_IO</code>	ASCII Character
<code>LINE_TERMINATOR</code>	<code>ASCII.CR</code>
<code>PAGE_TERMINATOR</code>	<code>ASCII.FF</code>
<code>FILE_TERMINATOR</code>	<code>ASCII.SUB (CTRL/Z)</code>
<code>NEW_LINE</code>	<code>ASCII.LF</code>

The services provided by the terminal driver are:

- 1) Reading a character from the communications port, `Get_Character`.
- 2) Writing a character to the communications port, `Put_Character`.

F.8.1 Package `TEXT_IO`

The specification of package `TEXT_IO`:

```
pragma page;
with BASIC_IO;

with IO_EXCEPTIONS;
package TEXT_IO is

  type FILE_TYPE is limited private;

  type FILE_MODE is (IN_FILE, OUT_FILE);

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

  -- max. size of an integer output field 20....0
  subtype FIELD is INTEGER range 0 .. 35;

  subtype NUMBER_BASE is INTEGER range 2 .. 16;

  type TYPE_SET is (LOWER_CASE, UPPER_CASE);

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;
```

DACS-80x86 User's Guide
Implementation-Dependent Characteristics

```
        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 COUNT;  
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;
```

DACS-80x86 User's Guide
Implementation-Dependent Characteristics

```

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);

```


DACS-80x86 User's Guide
Implementation-Dependent Characteristics

```
        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);
```

DACS-80x86 User's Guide
Implementation-Dependent Characteristics

```

        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.
```

DACS-80x86 User's Guide
Implementation-Dependent Characteristics

```
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);
```

DACS-80x86 User's Guide Implementation-Dependent Characteristics

```
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 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, "DIPUT?put_character");
  pragma interface (ASM86, get_character);
  pragma interface_spelling(get_character, "DIGET?get_character");
end TERMINAL_DRIVER;
```

F.8.5 Packages SEQUENTIAL_IO and DIRECT_IO

The specifications of SEQUENTIAL_IO and DIRECT_IO are specified in the ARM:

Since files are not supported the subprograms in these units raise USE_ERROR or STATUS_ERROR.

F.8.6 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);
```

DACS-80x86 User's Guide Implementation-Dependent Characteristics

```
-- 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.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 80386/80486 addresses. In case of other targets, the scale factor should be set to "scale_1".

```

type opcode_type is (
  -- 8086 instructions:
  _AAA, _AAD, _AAM, _AAS, _ADC, _ADD, _AND, _CALL,
  _CALLN, _CBW, _CLC, _CLD, _CLI, _CMC, _CMP, _CMPS, _CWD, _DPA,
  _DAS, _DEC, _DIV, _HLT, _IDIV, _IMUL, _IN, _INC, _INP,
  _INTO, _IRET, _JA, _JAE, _JB, _JBE, _JC, _JCKZ, _JE,
  _JG, _JGE, _JL, _JLE, _JNA, _JNAE, _JNB, _JNBE, _JNC,
  _JNE, _JNG, _JNGE, _JNL, _JNLE, _JNO, _JNP, _JNS, _JNZ,
  _JO, _JP, _JPE, _JPO, _JS, _JZ, _JMP, _LAMP, _LIP,
  _LES, _LEA, _LOCK, _LODS, _LOOP, _LOOPE,
  _LOOPNE, _LOOPNZ, _MUL, _NEG, _NOP, _NOT, _OR, _OUT,
  _POP, _POPF, _PUSH, _PUSHF, _RCL, _RCR, _ROL, _ROR,
  _REP, _REPE, _REPNE, _RET, _RETF, _RETN, _RETNP, _SHL,
  _SAL, _SAR, _SEL, _SER, _SBB, _SCAS, _STC, _STD, _STI,
  _STOS, _SUB, _TEST, _WAIT, _XCHG, _XLAT, _XOR,

  -- 8087/80187/80287 Floating Point Processor instructions:
  _FABS, _FADD, _FADDD, _FADDP, _FBLD, _FBSTP, _FCES,
  _FNCLX, _FCOM, _FCOMD, _FCOMP, _FCOMPD, _FCOMPP, _FDECSTP,
  _FDIV, _FDIVD, _FDIVP, _FDIVR, _FDIVRD, _FDIVRP, _FREE,
  _FIADD, _FIADDQ, _FICOM, _FICOMD, _FICOMP, _FICOMPD, _FIDIV,
  _FIDIVD, _FIDIVR, _FIDIVRD, _FIELD, _FIELDQ, _FIELDL, _FIMUL,
  _FIMULD, _FIMULP, _FIMULR, _FIST, _FISTD, _FISTP, _FISTPD,
  _FISTPL, _FISUB, _FISUBD, _FISUBR, _FISUBRD, _FLD, _FLDQ,
  _FLDCW, _FLDENV, _FLDLG2, _FLDLW2, _FLDL2E, _FLDL2T, _FLDPI,
  _FLDZ, _FLD1, _FMUL, _FMULD, _FMULP, _FMULR, _FRATAN,
  _FPREM, _FPATAN, _FRNDINT, _FRSTOR, _FSAVE, _FSCALE, _FSETPM,
  _FSORT, _FST, _FSTD, _FSTCW, _FSTEMV, _FSTP, _FSTPD,
  _FSTSW, _FSTSWAX, _FSUB, _FSUBD, _FSUBR, _FSUBRD,
  _FSUBRP, _FIST, _FWAIT, _FXAM, _FXCH, _FXTRACT, _FYL2X,
  _FYL2XP1, _F2XM1,

  -- 80186/80286/80386 instructions:
  -- Notice that some immediate versions of the 8086
  -- instructions only exist on these targets
  -- (shifts, rotates, push, imul, ...)
  _BOUND, _CLTS, _ENTER, _INS, _LAR, _LEAVE, _LGDT,
  _LIDT, _LSL, _OUTS, _POPA, _PUSHA, _SGDT, _SIDT,
    
```


DACS-80x86 User's Guide
Implementation-Dependent Characteristics

```

m_ARPL,    m_LLDI,    m_LMSW,    m_LTR,
-- 16 bit always...

m_SLDT, m_SMSW,    m_STR,    m_VERR, m_VERRW,
-- the 80386 specific instructions:

m_SETA,    m_SETAE, m_SETB,    m_SETBE, m_SETC,    m_SETE,
m_SETG,    m_SETGE, m_SETI,    m_SETLE, m_SETNA,    m_SETMAE,
m_SETMB,    m_SETMBE, m_SETMC,    m_SETME, m_SETMG,
m_SETNGE, m_SETNL, m_SETMLE,    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_JTS, m_LFS, m_LGS,    m_LSS, m_MOVBX, m_MOVSX,
m_MOVCB, m_MOVDB, m_MOVTR, m_SELD, m_SHRD,
-- the 80387 specific instructions:

m_FUCOM,    m_FUCOMP,    m_FUCOMPP, m_FPREMI, m_FSIN, m_FCOS,
m_FSINCOS,
-- byte/w ord/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_BTB, m_BTD, m_BTCW,
m_BTCD, m_BTRW, m_BTRD, m_BTSB, m_BTSD, m_CBWB,
m_CWOB, m_CWOW, m_CWOD, m_CQPB, m_CQPW, m_CQPD,
m_CQSB, m_CQSW, m_CQSD, m_DECB, m_DECW, m_DECD,
m_DIVB, m_DIVW, m_DIVD, m_IDIB, m_IDIW, m_IDID,
m_IMULB, m_IMULW, m_IMULD, m_INCB, m_INCW, m_INCD,
m_INSB, m_INSW, m_INSD, m_LCQSB, m_LCQSW, m_LCQSD,
m_MOVB, m_MOVW, m_MOVD, m_MCVSB, m_MCVSW, m_MCVSD,
m_MOVSB, m_MOVSW, m_MOVXB, m_MOVXBW, m_MULB, m_MULW,
m_MULD, m_NEGB, m_NEGW, m_NEGD, m_NEGB, m_NOTB, m_NOTW,
m_NOTD, m_ORB, m_ORW, m_ORD, m_ORB, m_OUTSB, m_OUTSW,
m_OUTSD, m_POPB, m_POPD, m_PUSHB, m_PUSHD, m_RCLB,
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_SELB,
m_SHLB, m_SHLD, m_SHRB, m_SHRW, m_SHRD, m_SBBB,
m_SBBW, m_SBBD, m_SCASB, m_SCASW, m_SCASD, m_STOSB,
m_STOSW, m_STOSD, m_STOSB, m_STOSW, m_STOSD, m_TESTB,
m_TESTW, m_TESTD, m_XORB, m_XORB, m_XORD, m_XORB,
m_XORD, m_XORD, m_XORB,
-- 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 :

```

DACS-80x86 User's Guide
Implementation-Dependent Characteristics

```

-- 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
ST1, ST2, ST3, ST4, ST5, ST6, ST7, -- floating registers (stack)
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 =>
immediatel : 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

```

DACS-80x86 User's Guide
Implementation-Dependent Characteristics

```

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

```

DACS-80x86 User's Guide
Implementation-Dependent Characteristics

```
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 $1 \leq x \leq 999$ and is put in the offset field in the first operand of the `MACHINE_INSTRUCTION`.
- `m_reset`: used to enable use of more than 999 labels. The label number after a `m_RESET` must be in the range $1 \leq x \leq 999$. 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 with 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 successful machine procedure. When not INLINE a block is created around the call which allows addressing of parameters, and code for exiting the procedure is also automatic.

The user takes over the responsibility for correct parameter addressing. The rules of Ada procedure calls must be followed. The calling conventions are summarized below.

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 alignment 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);
```

DACS-80x86 User's Guide
Implementation-Dependent Characteristics

Notice that machine subprograms cannot be functions.

The parameters take up:

op1	: integer	:	1 word
op2	: integer	:	1 word
res	: integer	:	1 word
<hr style="width: 20%; margin-left: 0;"/>			
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_subprgm,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:

DACS-80x86 User's Guide
Implementation-Dependent Characteristics

```
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` option 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 located 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.

Symbolic fixups are possible in certain instructions. With these you may build 'symbolic' instructions byte for byte. The instructions involved all require the operand type NAME (like used with `CALL`), and the interpretation is the following:

(name, m_DATAD, "MYNAME")	a full virtual address (offset and selector) of the symbol MYNAME (no additional offset is possible).
(name, m_DATAW, "MYNAME")	the offset part of the symbol MYNAME (no additional offset is possible).
(name, m_DATAB, "MYNAME")	the selector value of symbol MYNAME

In inlined machine instructions it may be a problem to obtain the address of a parameter (rather than the value). The `LEA` instruction may be used to get the offset part, but now the following form allows a way to load a selector value as well:

```
(system_address, LES, param'address) ES is loaded with the selector of PARAM. If this  
selector was e.g. SS, it would be pushed and popped  
into ES. LES may be substituted for LFS and LGS  
for 80386.
```

DACS-80x86 User's Guide
Implementation-Dependent Characteristics

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 except for DACS-80386PM/DACS-80486PM.

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.
```

DACS-80x86 User's Guide
Implementation-Dependent Characteristics

```

for TaskState use (Initial => 160000 ,
                  Engaged => 160080 ,
                  Running => 160100 ,
                  Delayed => 160180 ,
                  EntryCallingTimed => 160200 ,
                  EntryCallingUnconditional => 160280 ,
                  SelectingTimed => 160310 ,
                  SelectingUnconditional => 160390 ,
                  SelectingTerminable => 160410 ,
                  Accepting => 1604A0 ,
                  Synchronizing => 160530 ,
                  Completed => 1605C0 ,
                  Terminated => 160640);

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 WPXSaveArea is array(1..48) of System.UnsignedWord;

type FlagsType is
record
    WPXFlag          : 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);

pragma page;
type TaskControlBlock is
record
    sem              : System.Semaphore;
    isMonitor        : Integer;

    -- Delay queue handling

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

    -- Saved registers

    SS               : System.UnsignedWord ;

```

DACS-80x86 User's Guide
Implementation-Dependent Characteristics

```
SP          : Offset ;

-- Ready queue handling
next       : System.TaskValue ;

-- Semaphore handling
semnext   : System.TaskValue ;

-- Priority fields
priority   : System.Priority;
saved_priority : System.Priority;

-- Miscellaneous 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;
isAsynch    : Integer;
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;
```

DACS-80x86 User's Guide
Implementation-Dependent Characteristics

```
-- Auxiliary fields

ttd      : AccTaskTypeDescriptor;
FirstCaller : System.TaskValue;

-- Run-Time System fields

ACF      : ACF_type;           -- cf. User's guide 9.4.2
SOFirst  : Integer;           -- Only used in RMS
SemFirst  : Integer;           -- Only used in RMS
TBlockingTask : System.TaskValue; -- Only used in RMS
PBlockingTask : System.TaskValue; -- Only used in RMS
collection : System.Address;
partition  : Integer;

TaskCheckLimit : Offset;      -- to assure inline storage check
LastException  : System.DWord; -- 2 * 16 bits
SavedAdaAddr   : Offset;      -- to improve rendezvous's

-- NFX save area
--
-- When the application is linked with -npx, a special
-- save area for the NFX is allocated at the very end
-- of every TCB.
-- ie:
--
-- case NFX_Present is
--   when TRUE => NFXsave   : NFXSaveArea;
--   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 136 .. 136;
TCB_ok   : constant TCB_ok_value := TaskControlBlock'size / 8;

end TaskTypes;
```

F.11 RMS Tasking (OPTIONAL)

The DACS-80x86 systems may run tasking applications by means of Rate Monotonic Scheduling (RMS). RMS capability is purchased optionally, and is thus not included by default. Please contact DDC-I for more information regarding RMS and your system. RMS allows the programmer to guarantee properties of a tasking system, i.e. that tasks will meet their hard deadlines. The RMS tasking is selected by specifying `-rms` to the Ada link command.

