

AD-A279 646



FORMATION PAGE

Form Approved
GPM No.Public
and m
sugge
2200-ge 1 hour per response, including the time for reviewing instructions, searching existing data sources gathering
Send comments regarding this burden estimate or any other aspect of this collection of information, including
Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA
Management and Budget, Washington, DC 20503.

1. AG

3. REPORT TYPE AND DATES

4. TITLE AND

5. FUNDING

940325S1.11346, AVF: 94ddc500_2A
DDC-I, DACS Sun SPARC/SunOS to 680x0 Bare Ada Cross Compiler
System (BASIC MO DE), Version 4.6.9

6. AUTHORS:

National Institute of Standards and Technology
Gaithersburg, Maryland

7. PERFORMING ORGANIZATION NAME(S) AND

94-15728

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

9. SPONSORING/MONITORING AGENCY NAME(S) AND

Ada Joint Program Office
The Pentagon, Rm 3E118
Washington, DC 20301-308010. SPONSORING/MONITORING
AGENCY

11. SUPPLEMENTARY

12a. DISTRIBUTION/AVAILABILITY

12b. DISTRIBUTION

Approved for Public Release; distribution unlimited

13. (Maximum 200)

Host: Sun SPARCstation IPX (under SunOS, Release 4.1.1)
Target: Lynwood j435TU (68030) (bare machine)DTIC
ELECTE
MAY 26 1994
S F D

14. SUBJECT

15. NUMBER OF

Ada programming language, Ada Compiler Validation Summary Report, Ada
Compiler Val. Capability Val. Testing, Ada Val. Office, Ada Val. Facility
ANSP/MIL-STD-1815A, AJP017. SECURITY
CLASSIFICATION
UNCLASSIFIED18. SECURITY
UNCLASSIFIED19. SECURITY
CLASSIFICATION
UNCLASSIFIED20. LIMITATION OF
UNCLASSIFIED

NNN

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

94 5 25 019 DTIC QUALITY INSPECTED 1

AVF Control Number: NIST94DDC500_2A_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 940325S1.11346
DDC-I
DACS Sun SPARC/SunOS to 680x0 Bare Ada
Cross Compiler System (BASIC_MODE), Version 4.6.9
Sun SPARCstation IPX =>
Lynwood j435TU (68030) (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	☒
DTIC TAB	☐
Unannounced	☐
Justification	
By	
Distribution /	
Availability Codes	
Dist	Avail and/or Special
A-1	

AVF Control Number: NIST94DDC500_2A_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/SunOS to 680x0 Bare Ada Cross Compiler System (BASIC_MODE), Version 4.6.9

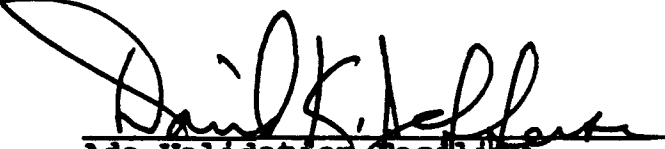
Host Computer System: Sun SPARCstation IPX running under SunOS, Release 4.1.1

Target Computer System: Lynwood j435TU (68030) (Bare Machine)

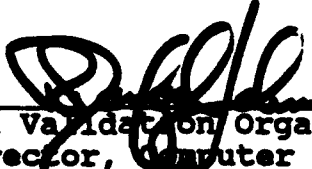
See section 3.1 for any additional information about the testing environment.

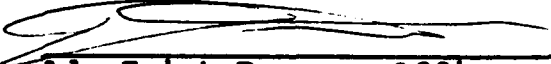
As a result of this validation effort, Validation Certificate 940325S1.11346 is awarded to DDC-I. This certificate expires 2 years after ANSI/MIL-STD-1815B is approved by ANSI.

This report has been reviewed and is approved.


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


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.

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/SunOS to 680x0 Bare Ada Cross Compiler
System (BASIC_MODE), Version 4.6.9

Host Computer System: Sun SPARCstation IPX running under SunOS, Release
4.1.1

Target Computer System: Lynwood j435TU (68030) (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.

John H. Miller
Customer Signature
Company DDC-I
Title

7-1-85
Date

James H. Miller
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.

The following 20 tests check for the predefined type `LONG_INTEGER`; for this implementation, there is no such type:

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

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 72 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.

CD2A83A was graded passed by Test Modification as directed by the AVO. This test uses a length clause to specify the collection size for an access type whose designated type is STRING; eight

designated objects are allocated, with a combined length of 30 characters. Because of this implementation's heap-management strategy and alignment requirements, the collection size at line 22 had to be increased to 812.

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	3542	
b) Total Number of Withdrawn Tests	104	
c) Processed Inapplicable Tests	524	
d) Non-Processed I/O Tests	0	
e) Non-Processed Floating-Point Precision Tests	0	
f) Total Number of Inapplicable Tests	524	(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 SPARCstation IPX and is used for downloading the executable images to the target Lynwood j435TU (68030) (Bare Machine) and to capture the results. The DDC-I Debug Monitor runs on the target Lynwood j435TU (68030) (Bare Machine) and provides communication interface between the host debugger and the executing target Lynwood j435TU (68030) (Bare Machine). The two processes communicate via ethernet to download and via RS-232 to upload.

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:

```
-nowarning -list
```

The linker options invoked explicitly for validation testing during this test were:

```
al -cpu      68030
   -fpu      68882
   -ram_base 0x1100000
   -ram      0x1000000, 0x3fffff
```

```
-main      stack_size=0x100000  
-tcb       30  
-ucc       ada_lynwood.slb
```

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	: 4
COUNT_LAST	: 2_147_483_647
DEFAULT_MEM_SIZE	: 2#1#E32
DEFAULT_STOR_UNIT	: 8
DEFAULT_SYS_NAME	: DACS_680x0
DELTA_DOC	: 2#1.0#E-31
ENTRY_ADDRESS	: FCNDECL.ENTRY.ADDRESS
ENTRY_ADDRESS1	: FCNDECL.ENTRY.ADDRESS1
ENTRY_ADDRESS2	: FCNDECL.ENTRY.ADDRESS2
FIELD_LAST	: 35
FILE_TERMINATOR	: ' '
FIXED_NAME	: NO_SUCH_TYPE
FLOAT_NAME	: NO_SUCH_TYPE
FORM_STRING	: ""
FORM_STRING2	:
	CONNOT RESTRICT_FILE_CAPACITY"
GREATER_THAN_DURATION	: 100000.0
GREATER_THAN_DURATION_BASE_LAST	: 200000.0
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	: 16#5.FFFF_F0#E+31
HIGH_PRIORITY	: 24
ILLEGAL_EXTERNAL_FILE_NAME1	: /NODIRECTORY1/FILENAME1
ILLEGAL_EXTERNAL_FILE_NAME2	: /NODIRECTORY1/FILENAME2
INAPPROPRIATE_LINE_LENGTH	: -1
INAPPROPRIATE_PAGE_LENGTH	: -1
INCLUDE_PRAGMA1	:
	PRAGMA INCLUDE ("A28006D1.ADA")
INCLUDE_PRAGMA2	:
	PRAGMA INCLUDE ("B28006E1.ADA")
INTEGER_FIRST	: -2147483648
INTEGER_LAST	: 2147483647
INTEGER_LAST_PLUS_1	: 2147483648
INTERFACE_LANGUAGE	: AS
LESS_THAN_DURATION	: -75000.0
LESS_THAN_DURATION_BASE_FIRST	: -131073.0
LINE_TERMINATOR	: ' '
LOW_PRIORITY	: 1
MACHINE_CODE_STATEMENT	:
	AA INSTR'(AA_EXIT_SUBPRGRM,0,0,0,AA INSTR_INTG'FIRST,0);
MACHINE_CODE_TYPE	: AA_INSTR
MANTISSA_DOC	: 31
MAX_DIGITS	: 15

```

MAX_INT                : 2147483647
MAX_INT_PLUS_1         : 2147483648
MIN_INT                : -2147483648
NAME                   : NO_SUCH_TYPE_AVAILABLE
NAME_LIST              : DACS_680x0
NAME_SPECIFICATION1    :
                        : /home/sun2/ada/68030/test/wrk/X2120A
NAME_SPECIFICATION2    :
                        : /home/sun2/ada/68030/test/wrk/X2120B
NAME_SPECIFICATION3    :
                        : /home/sun2/ada/68030/test/wrk/X3119A
NEG_BASED_INT          : 16#F000000E#
NEW_MEM_SIZE           : 2097152
NEW_STOR_UNIT          : 8
NEW_SYS_NAME           : DACS_680x0
PAGE_TERMINATOR        : ' '
RECORD_DEFINITION      :
                        : RECORD - INSTR_NO:INTEGER;ARG0:INTEGER;ARG1:INTEGER;
                        : ARG2:INTEGER;ARG3:INTEGER;END - RECORD;
RECORD_NAME            : AA_INSTR
TASK_SIZE              : 96
TASK_STORAGE_SIZE      : 1024
TICK                   : 2#1.0#E-14
VARIABLE_ADDRESS       : FCNDECL.VARIABLE_ADDRESS
VARIABLE_ADDRESS1      : FCNDECL.VARIABLE_ADDRESS1
VARIABLE_ADDRESS2      : FCNDECL.VARIABLE_ADDRESS2
YOUR_PRAGMA            : NOFLOAT

```

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 sublibrary. Compiler options are provided to allow the user control of optimization, run-time checks, and compiler input and output files 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.

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

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 The Invocation Command

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

```
$ ada {<option>} <source-or-unit>
```

where the options and parameters are:

DACS 680x0 Bare Ada Cross Compiler System - User's Guide

The Ada Compiler

OPTION	DESCRIPTION	REFERENCE
-{no}auto_inline	Automatic inline expansion of local subprograms.	5.1.1
-body	Compile body unit from source saved in library.	5.1.2
-check	Specifies run-time constraint checks.	5.1.3
-configuration_file	Specifies the configuration file used by the compiler	5.1.4
-{no}debug	Generate debug information.	5.1.5
-{no}fpu	Generate code for the floating point co-processors	5.1.6
-library	Specifies program library used.	5.1.7
-{no}list	Writes a source listing on the list file.	5.1.8
-mode	Protection mode.	5.1.9
-optimize	Specifies compiler optimization.	5.1.10
-{no}save_source	Inserts source text in program library.	5.1.11
-specification	Compile specification unit from source saved in library.	5.1.12
-{no}verbose	Displays compiler progress.	5.1.13
-{no}warnings	Display warning from the compiler	5.1.14
-{no}xref	Creates a cross reference listing.	5.1.15
<source-or-unit>	The name of the source file or unit to be compiled.	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. Options and option keywords may be abbreviated (characters omitted from the right) as long as no ambiguity arises. Casing is significant for options, but not for option keywords. When conflicting options are given on the command line, (e.g. `-list` and `-nolist`) the last one is used.

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 B.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.

DACS 680x0 Bare Ada Cross Compiler System - User's Guide

The Ada Compiler

A warning is issued when inline expansion is not achieved.

5.1.2 -body

-body

When using the option **-body** the Ada compiler will recompile the body of the unit specified as parameter to the Ada compiler (see section 5.1.16) into the current sublibrary. The source code saved in the program library at the previous compilation of the body is used as the source code to be compiled. If no source code is present or the body for the unit does not exist in the library, an error message is issued. This option is primarily for use by the Ada Recompiler (see chapter 7).

5.1.3 -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.

5.1.4 -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.

DACS 680x0 Bare Ada Cross Compiler System - User's Guide
The Ada Compiler

5.1.5 -[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 6.2.4.

5.1.6 -[no]fpu

-fpu (default)
-[no]fpu

If the **-fpu** option is specified the compiler will assume that a floating point co-processor is present and generate code accordingly. If the **-nofpu** option is specified the compiler will assume that a floating point co-processor is not present, and will not generate instructions for the co-processors. Floating point operations are instead implemented by calls to run time library.

5.1.7 -library

-library <file-spec>
-library \$ADA_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 (see Chapter 3). 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.

DACS 680x0 Bare Ada Cross Compiler System - User's Guide
The Ada Compiler

5.1.9 -mode

-mode ALL | BASIC | SECURE
-mode ALL (default)

The compiler generates code to execute in a non-protected BASIC mode, or in a protected SECURE mode according to the -mode option. Code can be generated to run in all protection modes by specifying ALL, this way protection mode can be decided at link time. The fastest and most compact code is generated by selecting the protection mode in which the program shall execute. Please refer to chapter 10 for details on protection modes. Mode SECURE is only usable if the program will be linked for a Motorola 68030 or 68040 processor.

5.1.10 -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:

ALL	All possible optimizations are invoked.
CHECK	Eliminates superfluous checks.
CSE	Performs common subexpression elimination including common address expressions.
FCT2PROC	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.

5.1.11 -[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 recompiler, 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-680x0 Symbolic Ada Debugger, if the source code is not saved in the library.

DACS 680x0 Bare Ada Cross Compiler System - User's Guide
The Ada Compiler

5.1.12 -specification

-specification

When using the option **-specification** the Ada compiler will recompile the specification of the unit specified as parameter to the Ada compiler (see section 5.1.16) into the current sublibrary. The source code saved in the program library at the previous compilation of the specification is used as the source code to be compiled. If no source code is present or the specification for the unit does not exist in the library, an error message is issued. This option is primarily for use by the Ada Recompiler (see chapter 7).

5.1.13 -[no]verbose

-verbose

-noverbose (default)

When **-verbose** is specified, the compiler will output information about which pass the compiler is currently running, otherwise no information will be output.

5.1.14 -[no]warnings

-warnings (default)

-nowarnings

All warnings from the Ada Compiler are displayed when option **-warnings** is specified. All compiler warnings are suppressed when **-nowarnings** is specified. See Section 5.3.2 for a description of how and when warnings are reported from the Ada Compiler.

5.1.15 -[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 5.3.1.3.

5.1.16 The Source or Unit Parameter

<source-or-unit>

This parameter specifies either the text file containing the Ada source text to be compiled or, when option **-body** or **-specification** is used, the name of the unit to be compiled. When interpreted as a file name, the file type **".ada"** is assumed by default, if the file type is omitted in the source file specification.

DACS 680x0 Bare Ada Cross Compiler System - User's Guide

The Ada Compiler

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 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 5.2.2). 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 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`.

DACS 680x0 Bare Ada Cross Compiler System - User's Guide
The Ada Compiler

- The aggregate occupies more than one line.

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

Below is a description of 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.

```
type CONFIGURATION_RECORD is
  record
    IN_FORMAT      : INFORMATTING;
    OUT_FORMAT     : OUTFORMATTING;
    ERROR_LIMIT    : INTEGER RANGE 1..32_767;
  end record;

type INPUT_FORMATS is (ASCII);

type INFORMATTING is
  record
    INPUT_FORMAT      : INPUT_FORMATS;
    INPUT_LINELENGTH : INTEGER range 72..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 5.3.2.2).

DACS 680x0 Bare Ada Cross Compiler System - User's Guide The Ada Compiler

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)
```

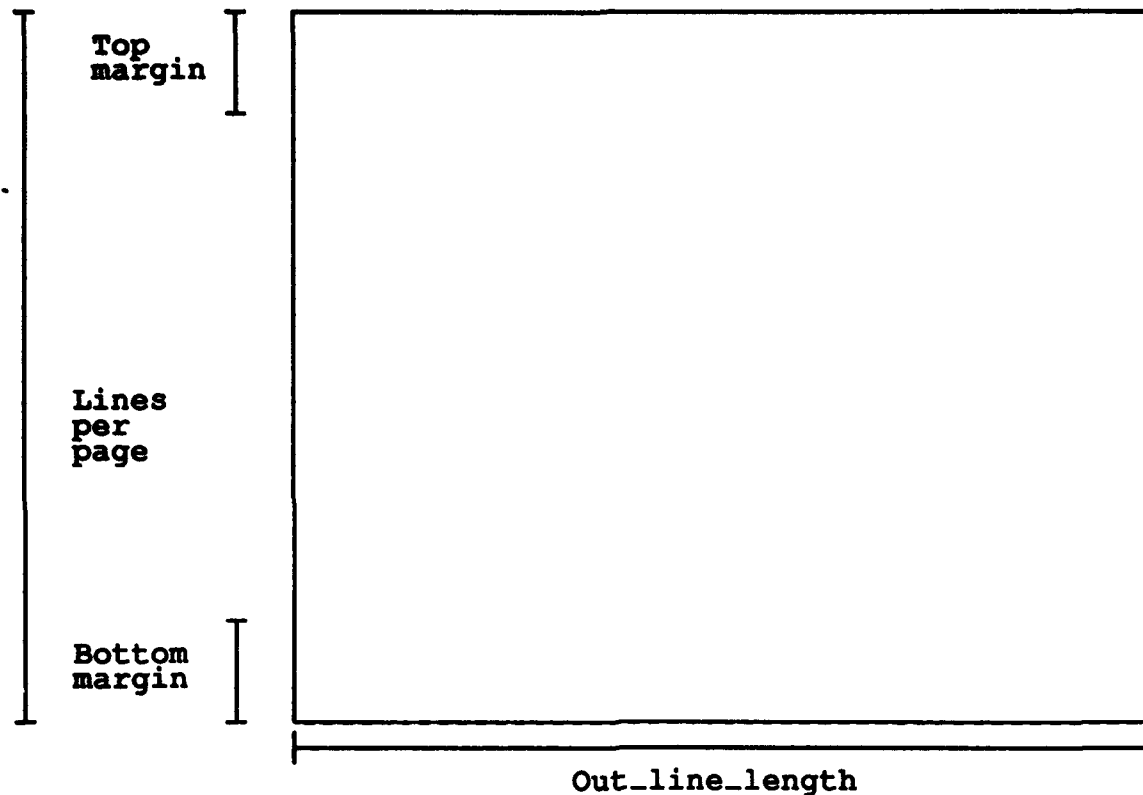


Figure 5.1: Page Layout

5.3 Compiler Output

The compiler may produce output to 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 to the list file, if the option `-list` is active.
- 2) A compilation summary is written to the list file, if `-list` is active.

DACS 680x0 Bare Ada Cross Compiler System - User's Guide

The Ada Compiler

- 3) A cross-reference listing is written to 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

If the user requests any listings by specifying the options `-list` or `-xref`, a new list file is created. 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 directory. If any such file exists prior to the compilation, the file is deleted.

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 to 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 `LIST` pragmas.
- 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.

DACS 680x0 Bare Ada Cross Compiler System - User's Guide
The Ada Compiler

- 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 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 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).

DACS 680x0 Bare Ada Cross Compiler System - User's Guide

The Ada Compiler

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 line numbers at 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.

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 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 preceded by a line showing the number of the line in the source text causing the message, and followed by 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 to the diagnostic file. Diagnostics other than warnings also appear on standard output. 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. The lines are ordered by increasing source line numbers. Line number 0 is assigned to messages not related to any particular line. On standard output 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 that they 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.

DACS 680x0 Bare Ada Cross Compiler System - User's Guide The Ada Compiler

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 also 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).

DACS 680x0 Bare Ada Cross Compiler System - User's Guide

The Ada Compiler

- 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
```

5.3.3 Return Status

The Ada Compiler's return value will have one of the following values:

- 0: The compilation was successful, warnings may have been generated.
- 1,2: Fatal internal error in the run-time system. Please contact DDC-I engineers.
- 3,4: Errors in command line options, compiler generates an error message indicating the error.
- 5: Fatal internal error in the compiler. Compiler generates an error message indicating the error. Please contact DDC-I engineers.
- 6: Severe error during compilation, e.g. a unit mentioned by a WITH clause is not present in the library. Compiler generates an error message indicating the error.
- 7: Error during compilation, e.g. most syntax errors. Compiler generates an error message indicating the error.

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 4.

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.

DACS 680x0 Bare Ada Cross Compiler System - User's Guide The Ada 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. The 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. The 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 order of compilation for generic units and describes situations in which an error will be generated during the instantiation of a generic unit.

DACS 680x0 Bare Ada Cross Compiler System - User's Guide

The Ada Compiler

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 [DoD-83] Chapter 10.3 (1).

5.5.2 Generic Formal Private Types

This 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 [DoD-83] 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 it 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.

DACS 680x0 Bare Ada Cross Compiler System - User's Guide
The Ada Compiler

5.7 Program Structure and Compilation Issues

The following limitations apply to the DACS-680x0 system:

- 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 `INTEGER`, a real literal may not exceed the range of `LONG_FLOAT`.
- The number of formal parameters permitted in a procedure is limited to 64 per parameter specification. There is no limit on the number of procedure specifications. For example the declaration:

```
procedure OVER_LIMIT (INTEGER01,  
                     INTEGER02,  
                     .....,  
                     INTEGER66: in INTEGER);
```

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

```
procedure UNDER_LIMIT (INTEGER01 : in INTEGER;  
                      INTEGER02  : in INTEGER;  
                      ...  
                      INTEGER66  : 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.

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-680x0 linker must be executed to create a program executable in the target environment. Linking is a two stage process that includes an Ada link using the information 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, the options to the DACS-680x0 linker, and the configuration of the linker.

6.1 The Link Process

The linking process can be viewed as two consecutive phases that are automatically carried out when issuing the link command `al`.

The link process is carried out in the following steps:

- Determination of Ada compilation units to include in the target program.
- Checking the validity of the included units according to the Ada rules.
- Determination of an elaboration order for the target program.
- Group units and tasks into classes (for security critical applications, see chapter 10).
- Generation of an object module to invoke the elaboration of the included Ada compilation units. This module is called the elaboration module.
- Determination of attributes of the program being linked (see section 6.7).
- Generation of an initialization module.
- Generation of option file(s) to the target linker.
- Invocation of the target linker.

The tasks of the first three steps are described in chapter 10 of the [DoD-83], the last five steps are described in detail in the following sections.

6.2 The Invocation Command

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

```
$ al {<option>} <unitname>
```

where the options and parameters are:

DACS 680x0 Bare Ada Cross Compiler System - User's Guide
The Ada Linker

OPTION	DESCRIPTION	REFERENCE
-(no)boot	Generate boot module.	6.2.1
-(no)class_file	Class file name.	6.2.2
-cpu	Select Target board CPU.	6.2.3
-(no)debug	Generate debug information.	6.2.4
-defaults	Save options as new linker defaults.	6.2.5
-(no)entry	Alternative program start label.	6.2.6
-(no)exceptions	Control of exception management.	6.2.7
-(no)executable	Name of executable file.	6.2.8
-(no)fpv	Control of which floating point processor is used.	6.2.9
-(no)heap	Control of memory management.	6.2.10
-(no)init_file	Initialization file name.	6.2.11
-interrupt_stack	Interrupt stack description.	6.2.12
-(no)itcb	Number of Interrupt Task Control Blocks allocated.	6.2.13
-(no)keep	Do not delete temporary files.	6.2.14
-library	The library used in the link.	6.2.15
-(no)log_file	Log file name.	6.2.16
-(no)logical_memory	Logical memory specification.	6.2.17
-main_task	Main task specification.	6.2.18
-(no)map	Keep linker map file.	6.2.19
-mmu_details	Setup values for MMU registers.	6.2.20
-mode	Execution mode.	6.2.21
-(no)option_file	Linker option file name.	6.2.22
-ram	Physical RAM memory specification.	6.2.23
-ram_base	Base address for RAM sections.	6.2.24
-ram_sections	Description of RAM memory sections.	6.2.25
-(no)rom	Physical ROM memory specification	6.2.26
-(no)rom_base	Base address for ROM sections.	6.2.27
-(no)rom_sections	Description of ROM memory sections.	6.2.28
-rts_stack_use	Amount of memory used by RTS.	6.2.29
-(no)scod	Supervisor code sections.	6.2.30
-(no)sdat	Supervisor data sections.	6.2.31
-(no)statistics	Print statistics.	6.2.32
-(no)target_options	Options to the target linker.	6.2.33
-task_defaults	Default values for tasks.	6.2.34
-(no)tcb	Number of Task Control Blocks allocated.	6.2.35
-ucc_library	UCC library name.	6.2.36
-(no)ucod	User code sections.	6.2.37
-(no)ucst	User constant sections.	6.2.38
-(no)udat	User data sections.	6.2.39
-(no)usr_library	A user supplied object library.	6.2.40
-(no)vector	Interrupt vector description.	6.2.41
-(no)verify	Print information about the link.	6.2.42
-(no)warnings	Print warnings.	6.2.43
<unit-name>	Name of the main unit.	6.2.44

DACS 680x0 Bare Ada Cross Compiler System - User's Guide

The Ada Linker

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

For all option values specifying a 32-bit address, 2-complement wrap-around is performed on negative numbers, e.g. `-rom_base=-1` is equivalent to `-rom_base=0xffffffff`.

6.2.1 `-[no]boot`

`-boot`
`-noboot` (default)

If `-boot` is specified an absolute file suited to gain control upon a reset is generated. The first two longwords in the `RTS_CODE` section contain the start Program Counter and the interrupt stack address. If `-noboot` is specified the absolute file does not contain the reset information. `-boot` is not valid when option `-debug` is specified, see section 6.2.4.

6.2.2 `-[no]class_file`

`-class_file <file_name>`
`-noclass_file` (default)

Specifies the name of the file containing the class specifications. The syntax of class specifications is described in chapter 10, where the concepts of classes are described as well. This option is only legal if option `-mode` is set to `SECURE` or `SAFE`.

6.2.3 `-cpu`

`-cpu 68020 | 68030 | 68040`
`-cpu <highest licensed>` (default)

Specifies the Motorola Central Processing Unit (CPU) on the target board. The `-cpu` option must match the actual CPU on the target board, as this option directs the Ada Linkers selection of RTS and supporting libraries. This option defaults to the highest CPU for which the DACS-680x0 has been licensed, with 68020 being the lowest and 68040 being the highest.

6.2.4 `-[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`

DACS 680x0 Bare Ada Cross Compiler System - User's Guide

The Ada Linker

option. See Section 5.1.5. **-debug** is not valid, when options **-boot** or **-vector INIT** are specified, see sections 6.2.1 and 6.2.41.

6.2.5 -defaults

-defaults

Saves the current setting of all options and parameters, except the **-defaults** option itself, as new defaults for the linker. The defaults are saved in the file specified by the environmental variable **ADA_LINK_DEFAULTS**. When this option is present, no actual linking will take place. For a complete description of the Ada Linker defaults system, please refer to section 6.3.

6.2.6 -[no]entry

-entry <string>
-noentry
-entry "Ada_ELAB\$Entry" (default)

The **-entry** option specifies the entry name of the program. If **-entry** is not specified the entry point is the start of the elaboration module.

6.2.7 -[no]exceptions

-exceptions (default)
-noexceptions

If **-exceptions** is specified the exception management routines are included in the target program. If **-noexceptions** is specified, the exception management routines are not included in the program, and the program will abort if the program raises any exceptions. If **-noexceptions** is specified and the target program has the exception attribute (see section 6.7) a warning is reported, and the exception management routines will not be included.

6.2.8 -[no]executable

-executable <file-name>
-noexecutable
-executable <main_unit>.x (default)

The **-executable** option specifies the file name of the absolute file created. **<file-name>** is used as name for the absolute file. If **-noexecutable** is specified the absolute file is not created.

DACS 680x0 Bare Ada Cross Compiler System - User's Guide
The Ada Linker

6.2.9 -[no]fpu

-fpu 68881 | 68882
-nofpu
-fpu 68882 (default)

Specifies the floating point co-processor available on the target system. If **-cpu 68040** has been specified, **-fpu 68881** is not allowed, as the MC68040 FPU emulates the MC68882 coprocessor and not the MC68881 coprocessor. If **-nofpu** is specified and the target program contains any floating point instructions an error message is issued. See section 6.7 concerning the float attribute. If all compilation units required for execution are compiled with **-nofpu** option no floating point instructions are generated and a link with **-nofpu** will never fail.

6.2.10 -[no]heap

-heap (default)
-noheap

If **-heap** is specified and the target program has the heap attribute (see section 6.7) then the storage management routines are included in the target program. If **-noheap** is specified the storage management routines are not included in the program. When **-noheap** is specified and the target program has the heap attribute, an error is reported and linking terminates.

6.2.11 -[no]init_file

-init_file <file-name>
-noinit_file (default)

The **-init_file** option specifies the name of a user supplied initialization file. **<file-name>** is used as name for the initialization file. If **-noinit_file** is specified, the linker generates an initialization file with the name **<prefix>_init.src**. It is assumed that the initialization file is an assembler source file.

6.2.12 -interrupt_stack

-interrupt_stack [NOSTART | START=<address>][,SIZE=<number>]
-interrupt_stack NOSTART,SIZE=10240 (default)

Specifies the creation of the interrupt stack. If **START=<address>** is specified the interrupt stack pointer is initialized to **<address>**. If **NOSTART** is specified the linker allocates the interrupt stack in the section **RTS_DATA**. **START=<address>** is not valid when **-mode** is set to **SECURE** or **SAFE**. If **SIZE=<number>** is specified the **<number>** bytes is allocated for the interrupt stack.

DACS 680x0 Bare Ada Cross Compiler System - User's Guide
The Ada Linker

6.2.13 -[no]itcb

-itcb <number>
-noitcb
-itcb 5 (default)

The **-itcb** option specifies the number of interrupt control blocks to allocate. If **-itcb <number>** is specified then **<number>** of interrupt control blocks are allocated, where **<number>** must be in the range 1..32767. If **-noitcb** is specified and the target program has the interrupt attribute (see section 6.7) then an error is reported and the absolute file is not created. If the target program does not have the interrupt attribute the **-itcb** option is ignored.

6.2.14 -[no]keep

-keep
-nokeep (default)

If **-keep** is specified temporary linker files are not deleted, otherwise they are deleted. See also section 5.5 about temporary linker files.

6.2.15 -library

-library <file-name>
-library \$ADA_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.

6.2.16 -[no]log_file

-log_file <file-name>
-nolog_file (default)

Specifies that linker information shall be written to a file named **<file-name>**. The log file will contain all verification information specified by the **-verify** option and all statistics specified with the **-statistics** option, plus warnings and errors messages, a listing of the class file (see section 6.2.2), an expanded list of the class file specifications, a detailed description of each compilation unit included in the program, and a link summary.

6.2.17 -{no}logical_memory

-logical_memory <start_addr>,<end_addr>[,<start_addr>,<end_addr>]
-nological_memory (default in BASIC mode)
-logical_memory 0x1000000,0x7fffff (default in SECURE and SAFE mode)

The **-logical_memory** specifies the logical memory areas available for task stacks and task heaps in the program. **-logical_memory** is only legal when option **-mode** is set to SECURE or SAFE. The logical memory must be disjoint from the physical memory (see section 6.2.23).

6.2.18 -main_task

-main_task [PRIORITY=<number>]
 [NOTIME_SLICE | ,TIME_SLICE=<real>]
 [NOFLOAT | ,FLOAT]
 [NOSTACK_START | ,STACK_START=<address>]
 [STACK_SIZE=<number>]
 [,HEAP_SIZE=<number>]
-main_task PRIORITY=12,NOTIME_SLICE,FLOAT,NOSTACK_START,
 STACK_SIZE=10240,HEAP_SIZE=10240 (default)

The **-main_task** option specifies priority, time slice, use of floating point co-processor, stack start, stack size and heap size for the main task. If PRIORITY=<number> is specified and the pragma PRIORITY has not been applied then the main task has the priority <number> which must be in the range 1..24, otherwise it has the priority specified in the pragma. If the TIME_SLICE=<real> is specified then the main task has the time slice <real> (<real> must be in the form <number>.<number>). If NOTIME_SLICE is specified the main program does not have a time slice. If FLOAT is specified the main program may use the floating point co-processor. The state of the co-processor will not be saved as part of the main task context. If NOFLOAT is specified the main program must NOT use the floating point co-processor. If STACK_START=<address> is specified the main stack pointer is initialized to <address>. If NOSTACK_START is specified the linker allocates the stack for the main program in the section RTS_DATA, and initializes the stack pointer. STACK_START=<address> is not valid when **-mode** is set to SECURE or SAFE. If STACK_SIZE=<number> is specified then <number> of bytes is allocated for the main program stack. If HEAP_SIZE=<number> is specified then <number> of bytes is allocated for the main program heap.

6.2.19 -{no}map

-map
-nomap (default)

-map directs the linker to keep the map file. The map file contains information about memory layout of the program. The name of the map file is <main_unit_name>.map. Please refer to [Microtec-a] for a description of the map file.

**DACS 680x0 Bare Ada Cross Compiler System - User's Guide
The Ada Linker**

6.2.20 -[no]mmu_details

```
-mmu_details [TIA=<number>][,NOTIB | ,TIB=<number>]  
              [,NOTIC | ,TIC=<number>][,NOTID | ,TID=<number>]  
              [,PAGE_SIZE=<number>][,SEGMENT_SIZE=<number>]  
-nommu_details      (default in BASIC mode)  
-mmu_details TIA=7,TIB=7,TIC=6,NOTID,PAGE_SIZE=12,SEGMENT_SIZE=25  
              (default in SECURE and SAFE mode)
```

Specifies values for the MMU Translation Control Registers. **-mmu_details** is only legal when **-mode** is set to SECURE or SAFE. **-nommu_details** is only legal when **MODE** is set to BASIC. The parameter values are all number of bits. TIA to TID specifies the number of bits to use on MMU table level A to D. PAGE_SIZE specifies the number of bits used for each page accessed by a page descriptor entry in the MMU tables. SEGMENT_SIZE specifies the number of bits used for a segment of the logical memory assigned to each task group. The segment of a task group contains the task group heap and stacks of all tasks of the task group. Please refer to [MOTOROLA-a] and [MOTOROLA-b] for a detailed description of the MMU and its registers.

A number of constraints apply to the keywords of the option:

- If **-cpu 68030** is specified, the following values are valid for the keywords of the **-mmu_details** option: TIA, TIB, TIC and TID must be in the range 2..15; NOTIB, NOTIC and NOTID can also be used. PAGE_SIZE must be in the range 8..15 for page sizes between 256 bytes and 32K bytes. SEGMENT_SIZE must be in the range 9..30 for a segment size between 512 bytes and 1 gigabyte.
- If **-cpu 68040** is specified, the following values are valid for the keywords of the **-mmu_details** option: TIA and TIB must be 7, TIC must be 5 or 6, and NOTID must be used. NOTIB and NOTIC cannot be used. PAGE_SIZE must be 12 or 13 for a page size of 4K bytes or 8K bytes. SEGMENT_SIZE must be 18 or 25 for a segment size of 256K bytes or 32M bytes.
- SEGMENT_SIZE must be equal to PAGE_SIZE + TID or PAGE_SIZE + TID + TIC or PAGE_SIZE + TID + TIC + TIB.
- PAGE_SIZE must be equal to or greater than each of TIA + 2, TIB + 2, TIC + 2, and TID + 2.
- If NOTIB is specified, both NOTIC and NOTID must be specified as well, otherwise if NOTIC is specified, NOTID must be specified as well.
- The sum of TIA, TIB, TIC, TID and PAGE_SIZE must be equal to 32.

The default value of **-mmu_details** in SECURE and SAFE mode defines a four level address translation table tree with each page having a size of 4 Kbytes and each logical segment having a size of 32 Mbytes. See section 10.5 for further description of how the values for **-mmu_details** is utilized.

DACS 680x0 Bare Ada Cross Compiler System - User's Guide

The Ada Linker

6.2.21 -mode

-mode BASIC | SECURE | SAFE
-mode BASIC (default)

Specifies how the program shall execute. BASIC means that all code executes at supervisor privilege level i.e. there is memory protection of neither code nor data. SECURE mode and SAFE mode means that code and data can be protected using the MMU and specified by use of the **-class_file** option. If **-mode** is set to SECURE or SAFE the **-class_file** option must be specified as well. In SECURE mode all objects allocated by allocators are allocated on the stack of the task executing the allocator, while in SAFE mode they are allocated on the heap of the task executing the allocator. SECURE and SAFE modes can only be selected when option **-cpu** is set to 68030 or 68040. See chapter 10 for a complete description of modes.

6.2.22 -[no]option_file

-option_file <file-name>
-nooption_file (default)

The **-option_file** option specifies the name of the target link option file. **<file-name>** is used as the name for the target link option file. If **-nooption_file** is specified the Ada linker generates an option file with the name **<main_unit_name>.opt**.

6.2.23 -ram

-ram <start_addr>,<end_addr>[,<start_addr>,<end_addr>]
-ram 0x0,0xffff (default)

The **-ram** specifies the physical RAM memory available for the executable program.

6.2.24 -ram_base

-ram_base <address>
-ram_base 0x10000 (default)

The **-ram_base** option specifies the base address for the program placed in RAM memory. The program sections specified in option **-ram_sections** are placed consecutively from the address specified with this option. In SECURE and SAFE modes, the base address will always be page aligned. The address must be within the physical RAM memory specified in option **-ram**.

DACS 680x0 Bare Ada Cross Compiler System - User's Guide
The Ada Linker

6.2.25 -ram_sections

```
-ram_sections <section_name>{,<section_name>}  
-ram_sections SUPER_CODE,SUPER_DATA,USER_CODE,\  
                USER_CONS,USER_DATA                (default)
```

The **-ram_sections** option specifies the sections to be placed in RAM memory. The sections are placed in the specified order from the address specified with option **-ram_base**. Valid section names are **SUPER_CODE**, **SUPER_DATA**, **USER_CODE**, **USER_CONS** and **USER_DATA** (see section 6.8).

6.2.26 -[no]rom

```
-rom <start_addr>,<end_addr>{,<start_addr>,<end_addr>}  
-norom                (default)
```

The **-rom** specifies the physical ROM memory available for the executable program.

6.2.27 -[no]rom_base

```
-rom_base <address>  
-norom_base      (default)
```

The **-rom_base** option specifies the base address for the program placed in ROM memory. The program sections specified in option **-rom_sections** are placed consecutively from the address specified with this option. In **SECURE** and **SAFE** modes, the base address will always be page aligned by truncating the address with the number of bits specified in option **-mmu_details** keyword **PAGE_SIZE**, i.e. the base address will be the start of the page appointed by the specified address. The address must be within the physical ROM memory specified in option **-rom**.

6.2.28 -[no]rom_sections

```
-rom_sections <section_name>{,<section_name>}  
-norom_sections      (default)
```

The **-rom_sections** option specifies the sections to be placed in ROM memory. The sections are placed in the specified order from the address specified with the **-rom_base** option. Valid section names are **SUPER_CODE**, **USER_CODE** and **USER_CONS** (see section 6.8). By default no sections are placed in ROM.

DACS 680x0 Bare Ada Cross Compiler System - User's Guide
The Ada Linker

6.2.29 -rts_stack_use

-rts_stack_use <number>
-rts_stack_use 0 (default)

Specifies the amount of extra stack space allocated in each task for the use of user supplied code in the RTS. If the Ada code interfaces to any externally supplied user code (e.g. by use of the **-usr_library** option) executing in supervisor mode, **-rts_stack_use** should be set to the amount of stack consumed by this external code. The Ada Linker determines how much RTS stack space it will need for the RTS operations, and will automatically allocate the minimum necessary RTS stack space.

6.2.30 -(no)scode

-scode <string>{,<string>}
-noscode
-scode RTS_CODE (default)

Specifies which program sections are to be placed in the supervisor code space. **-noscode** indicates that no program sections should be placed in the supervisor code space. See section 6.8 about program sections.

6.2.31 -(no)sdat

-sdat <string>{,<string>}
-nosdat
-sdat RTS_DATA (default)

Specifies which program sections are to be placed in the supervisor data space. **-nosdat** indicates that no program sections should be placed in the supervisor data space. See section 6.8 about program sections.

6.2.32 -(no)statistics

-statistics
-nostatistics (default)

-statistics specifies that statistics should be displayed about the compilation units included in the program and their dependencies, otherwise no statistics is displayed. If option **-log_file** is specified (see section 6.2.16), the statistics will be included in the log file as well.

DACS 680x0 Bare Ada Cross Compiler System - User's Guide
The Ada Linker

6.2.33 **-[no]target_options**

-target_options <string>
-notarget_options (default)

-target_options specifies additional options to the target linker. <string> will be added to the options for the target linker when the Ada Linker invokes the target linker. For instance if **-target_options** "-f c" is specified, -f c will be added to the target linker options, resulting (in this case) in the external symbol cross-reference table being included in the linker map file. If **-notarget_options** is specified no additional options, apart from the options determined by the Ada linker itself, will be added to the options for the target linker.

6.2.34 **-task_defaults**

-task_defaults [STACK_SIZE=<number>][,PRIORITY=<number>]
[,NOTIME_SLICE | ,TIME_SLICE=<real>]
-task_defaults STACK_SIZE=10240,PRIORITY=12,NOTIME_SLICE (default)

Specifies the default values to be used for task creation. The defaults specified will be used when creating tasks which do not contain pragma priority or the length clause specifying the stack size. If **STACK_SIZE=<number>** is specified then <number> of bytes is allocated for a task stack. If **PRIORITY=<number>** is specified then <number> is used as the priority of the task. The specified priority must be in the range 1 to 24. If **TIME_SLICE=<real>** is specified then <real> specifies the number of seconds to use as the time slice for the task; <real> has the form <number>.<number>. If **NOTIME_SLICE** is specified the task does not have a time slice. If the target program does not have the tasking attribute (see section 6.7) the **-task_defaults** option is ignored.

6.2.35 **-[no]tcb**

-tcb <number>
-notcb
-tcb 10 (default)

Specifies the number of task control blocks to be allocated. If **-tcb 0** or **-notcb** is specified and the target program has the tasking attribute the linker reports a error and no absolute file will be produced. This option is ignored if the target program does not have the task attribute (see section 6.7).

6.2.36 **-ucc_library**

-ucc_library <file-name>
-ucc_library \$ADA_UCC (default)

The **-ucc_library** option specifies the name of the UCC library to include in the target program. If the UCC library is not specified the environmental variable **ADA_UCC** is used as file name.

6.2.37 -[no]ucod

-ucod <string>{,<string>}
-noucod
-ucod ADA_CODE (default)

Specifies which program sections are to be placed in the user code space. **-noucod** indicates that no program sections should be placed in the user code space. See section 6.8 about program sections.

6.2.38 -[no]ucst

-ucst <string>{,<string>}
-noucst
-ucst ADA_CONS (default)

Specifies which program sections are to be placed in the user constant space. **-noucst** indicates that no program sections should be placed in the user constant space. See section 6.8 about program sections.

6.2.39 -[no]udat

-udat <string>{,<string>}
-noudat
-udat ADA_DATA (default)

Specifies which program sections are to be placed in the user data space. **-noudat** indicates that no program sections should be placed in the user data space. See section 6.8 about program sections.

6.2.40 -[no]usr_library

-usr_library <file_name>{,<file_name>}
-nouser_library (default)

When specified the object files and object libraries denoted by **file_name** is included in the link, otherwise no user library is included in the link.

6.2.41 -[no]vector

-vector [NOADDRESS | ADDRESS=<address>][,COPY | ,INIT]
-novector
-vector NOADDRESS,COPY (default)

DACS 680x0 Bare Ada Cross Compiler System - User's Guide

The Ada Linker

Specifies creation of the interrupt vector. If ADDRESS is specified the interrupt vector is placed at <address>. When NOADDRESS is specified, the interrupt vector will be placed in the section RTS_DATA. If COPY is specified the interrupt vector active when the program was invoked is copied. If INIT is specified the interrupt vector is initialized by the routine Ada_UCC_DS\$InitIV. -novector specifies that no initialization of the interrupt vector takes place. The program can hereby be invoked by an interrupt. After program invocation the interrupt vector can potentially be modified. INIT is not valid when option -debug is specified, see section 6.2.4. ADDRESS=<address> is invalid when -mode is set to SECURE or SAFE.

6.2.42 -[no]verify

-verify [ALL][,ELABORATION_ORDER][,COMMANDS][,PARAMETERS]
-noverify (default)

Determines the type and amount of information generated. If ELABORATION_ORDER is specified the elaboration order is displayed, if COMMANDS is specified the commands executing the various subprocesses are displayed, if PARAMETERS is specified the active parameters and options are displayed, and if ALL is specified all of the above mentioned information is displayed. If option -log_file is specified (see section 6.2.16) the information will be included in the log file as well.

6.2.43 -[no]warnings

-warnings
-nowarnings (default)

Specifies whether warnings should be generated or not. Warnings are generated when conflicts between target program attributes and specified options are detected, and when a package does not have a body.

6.2.44 The Main Unit Parameter

<unit-name>

The main unit must be a parameterless procedure and must be present in the library. The main unit name is a required parameter.

6.3 The Linker Defaults System

As it can be seen from the description of options above, default values exist for all options. However, it is possible to change the initial setting of default values and even have several configurations of default values for the Ada Linker. The Ada Linker default values are controlled by use of the option -defaults and the environmental variable ADA_LINK_DEFAULTS.

DACS 680x0 Bare Ada Cross Compiler System - User's Guide

The Ada Linker

The Ada Linker obtains its option and parameter values in the following way: First, options are initialized with the default values indicated in the above description of options. Second, new option and parameter defaults are loaded from the file indicated by the environmental variable `ADA_LINK_DEFAULTS`, if this points to an existing file. And third, options and parameters are given the value specified in the invocation command of the Ada Linker.

When the option `-defaults` is specified, the current value of options and parameters are saved as new defaults in the file identified by the environmental variable `ADA_LINK_DEFAULTS`. Note that `ADA_LINK_DEFAULTS` is not defined as a environmental variable when the DACS-680x0 is distributed by DDC-I, so an explicit definition is necessary.

Assume that the default value of `-itcb` should be 25 instead of 5, and that the new default settings should be saved in the file `DEFAULTS.LINK`. The following commands could be used:

```
$ setenv ADA_LINK_DEFAULTS DEFAULTS.LINK
$ al -defaults -itcb 25
```

These commands will create a new file called `DEFAULTS.LINK` in the current directory (if it does not exist already) and save the new linker default values in this file. As long as `ADA_LINK_DEFAULTS` keeps its current value of `DEFAULTS.LINK`, all linking performed in the current directory will have a `-itcb` default value of 25. Note that one should normally assign a fully expanded file name, like `/home/ada_users/user2/work/DEFAULTS.LINK`, to `ADA_LINK_DEFAULTS` to ensure that the correct default file will be found no matter in what directory the linking is performed.

Several configurations of Ada Linker defaults is possible, simply by changing `ADA_LINK_DEFAULTS` to denote different linker default files depending on the desired configuration. By the same method, different users can have different linker default values, simply by having `ADA_LINK_DEFAULTS` denote different files.

When option `-defaults` is specified no actual linking takes place; only the current value of the options and parameters are saved as new defaults. Before saving the new defaults, all options are checked to have a valid value without consideration of other options. If this is not the case, an error is reported and the new values are not saved as defaults. However, no check on interdependence between the options are performed when specifying new default values, e.g. it is possible to set `-mode SECURE` as default value without specifying a default value for `-class_file`. The interdependence between option values is checked only when an actual linking will be performed, i.e. when `-defaults` is not specified.

To reset the Ada Linker Defaults to the factory setting, simply use the command:

```
$ setenv ADA_LINK_DEFAULTS ""
```

which will ensure that no linker defaults file will be read when the linker is invoked. If the file denoted by `ADA_LINK_DEFAULTS` will not be used again, the file can be deleted. The current linker defaults setting can be viewed with the option `-verify PARAMETERS`, see section 6.2.42.

DACS 680x0 Bare Ada Cross Compiler System - User's Guide
The Ada Linker

6.4 Environmental Variables Used by the Ada Linker

When the Ada linker is executed, the following environmental variables are used:

VARIABLE	PURPOSE
ADA_LIBRARY	Identifies the default library used by all DACS tools. It is the lowest level sublibrary in the program library hierarchy. This default may be overridden by the -library option.
ADA_LINK_DEFAULTS	Identifies the file containing the Ada Linker defaults. Defaults are saved in this file when the option -defaults is used.
ADA_UCC	Identifies the library containing the User Configurable Code, e.g. an UCC library supplied by DDC-I. This default may be overridden by the -ucc_library option.

6.5 File Names Used by the Linker

During the link, the following temporary files are created in the current default directory:

```
<prefix>_init.src  
<prefix>_init.obj  
<prefix>_elab.src  
<prefix>_elab.obj  
<prefix>_end.src  
<prefix>_end.obj  
<prefix>.opt  
<prefix>_<unit_no>.obj  
<main_unit_name>.opt  
<main_unit_name>.com
```

If the **-keep** option is used **<prefix>** is the main unit name, otherwise **<prefix>** is the process identification (pid).

If linking for **SECURE** or **SAFE** mode (see section 6.2.21), the main unit and each defined class will result in the generation of a target linker option file.

6.6 Return Status

After a linking the return value of the Ada Linker will reflect if the linking was successfully completed. The following return values are possible:

DACS 680x0 Bare Ada Cross Compiler System - User's Guide

The Ada Linker

- 0: The link was successful. Warnings may have been generated during the link process.
- 1: An error occurred during the link process, e.g. the Ada Linker is unable to find the UCC file. The Ada Linker will generate an error message stating the cause of the error.
- 2: An internal error has caused the Ada Linker to abort, please contact DDC-I engineers.

6.7 Program Attributes

The linker evaluates the following attributes of the target program:

- **Tasking constructs**
The target program has the task attribute when Ada tasking constructs are used.
- **Floating point constructs**
The target program has the float attribute when the program uses the floating point co-processor. The target program will only use the co-processor instructions to implement operations on floating point types.
- **Interrupts, entries or procedures**
The target program has the interrupt attribute when the program contains address clauses for task entries or the PRAGMA INTERRUPT_HANDLER.
- **Exception handlers**
The target program has the exception attribute if the program contains any exception handlers.
- **Heap**
The target program has the heap attribute when the program contains allocations or deallocations on the heap.
- **Secure execution**
The target program has the secure attribute if the program is compiled with option -mode set to SECURE or SAFE. In this case the MC68030/MC68040 on-chip Memory Management Unit is used to protect code and data segments and for controlling storage checks.

The linker uses the attributes to generate the initialization module, to issue warnings if a combination of options is in conflict with the attributes of the target program, and to determine the proper RTS to include in the target link.

If a program contains interface calls interfacing to code which requires tasking, uses floating point instructions or storage management, the Ada compilation unit must contain a pragma to set the appropriate attribute. Please refer to Chapter 12 for details.

DACS 680x0 Bare Ada Cross Compiler System - User's Guide
The Ada Linker

6.8 Program Sections

The compiler uses the following program sections:

Section	Contents
RTS_CODE	Run-Time System code
RTS_DATA	Run-Time System data
ADA_CODE	Compiler generated code
ADA_CONS	Compiler generated constants
ADA_DATA	Compiler generated data

Table 6.1: Program sections

The program sections **RTS_DATA** and **ADA_DATA** must be in RAM memory. The program sections **RTS_CODE**, **ADA_CODE**, and **ADA_CONS** are not modified by the execution of the program and may be placed in ROM. All sections must be placed in either RAM or ROM memory.

The linker groups the program sections into 5 new sections: **SUPER_CODE**, **USER_CODE**, **USER_CONS**, **SUPER_DATA** and **USER_DATA**. CODE and CONS sections can be stored in ROM if desired. DATA sections must be placed in RAM. In **SECURE** and **SAFE** mode the **SUPER** and **USER** sections can be accessed when executing at supervisor privilege level, while only **USER** sections can be accessed when executing at user privilege level (please refer to [MOTOROLA-a] and [MOTOROLA-b] about supervisor and user privilege level). In **BASIC** mode the **SUPER** and **USER** sections can be accessed both when executing at supervisor privilege level and when executing at user privilege level.

When including user defined sections e.g. modules written in assembler, each of the compilers program sections and the user defined sections must be specified to the linker as one of the RTS program sections using the options **-udat**, **-sdat**, **-ucst**, **-ucod** and **-scod**.

6.9 The Initialization Module

The initialization module defines constants, allocates memory, and contains the code for initialization of the processor and the RTS. The initialization module for a given target program depends on the program attributes and the options given to the linker. The initialization module is generated as an assembler file with the name **<prefix>_init.src**. The assembler is invoked to produce the object file with the name **<prefix>_init.obj**. If the **-keep** option is used **<prefix>** is the main unit name, otherwise **<prefix>** is the process identification (pid).

6.9.1 The Initialization Constants

The initialization module defines the following externally visible symbols which are constants used by the run-time system.

Ada_INIT\$DisplaySize

The size of the display vector in bytes. This symbol is always defined.

DACS 680x0 Bare Ada Cross Compiler System - User's Guide
The Ada Linker

Ada_INIT\$InterruptStackSize

The size of the interrupt stack in bytes. This symbol is defined when the `-interrupt_stack` option is specified.

Ada_INIT\$MainStackSize

The byte size of the stack for the main task. This symbol is always defined.

Ada_INIT\$MainHeapSize

The byte size of the heap for the main task. This symbol is defined when option `-mode` is specified to `SECURE` or `SAFE`.

Ada_INIT\$DefaultTimeSlice

The default time slice for tasks. The symbol defines an integer, the unit is in `SYSTEM.TICKS`. The symbol is defined when the target program has the task attribute. If the `NOTIME_SLICE` is specified, the value of the symbol is `0xffffffff`.

Ada_INIT\$DefaultPriority

The default priority for tasks. The symbol is defined when the target program has the task attribute.

Ada_INIT\$MainPriority

The priority for the main task. The symbol is defined when the target program has the task attribute.

Ada_INIT\$MainTimeSlice

The time slice for the main program. Same convention as `Ada_INIT$DefaultTimeSlice`.

Ada_INIT\$DefaultStackSize

The stack size for tasks for which the `'STORAGE_SIZE` is not applied. This symbol is only defined if the target program has the task attribute.

Ada_INIT\$RTSSStackUse

The amount of memory reserved on the stack of each task to be used by the RTS.

Ada_INIT\$MainFPUse

Specifies whether or not the main task may use the floating point co-processor or the 68040 FPU.

Ada_INIT\$TCBCount

The number of task control blocks allocated minus one. This symbol is only defined when the target program has the task attribute.

**DACS 680x0 Bare Ada Cross Compiler System - User's Guide
The Ada Linker**

Ada_INIT\$ITCDBCount

The number of interrupt task control blocks allocated minus one. This symbol is only defined when the target program has the interrupt attribute.

Ada_INIT\$SuperStackSize

The size of the area to allocate supervisor stacks from, which can be allocated to tasks running on user privilege level. This symbol is only defined when option **-mode** is specified to **SECURE** or **SAFE**.

Ada_INIT\$SuperStackCount

The maximum number of supervisor stacks minus one, which can be allocated to tasks running on supervisor privilege level. This symbol is only defined when option **-mode** is specified to **SECURE** or **SAFE**.

Ada_INIT\$MainSuperStackSize

The size of the supervisor stack allocated for the main task. This symbol is only defined when option **-mode** is specified to **SECURE** or **SAFE**.

Ada_INIT\$HeapHeaderCount

The number of heap headers allocated minus one. A heap header contains a pointer to the heap and a heap semaphore (if the program contains tasking). This symbol is only defined when option **-mode** is specified to **SECURE** or **SAFE**.

Ada_INIT\$DefaultHeapSize

The size of the heap allocated for a task. This symbol is only defined when option **-mode** is specified to **SECURE** or **SAFE**.

Ada_INIT\$PageSize

The logical/physical page size measured in number of bytes. This symbol is only defined when option **-mode** is specified to **SECURE** or **SAFE**.

Ada_INIT\$PageWidth

The number of bits used as offset within a page, log 2 of page size. This symbol is only defined when option **-mode** is specified to **SECURE** or **SAFE**.

Ada_INIT\$LogSegmentWidth

The number of bits used as offset within a segment. This symbol is only defined when option **-mode** is specified to **SECURE** or **SAFE**.

Ada_INIT\$MMUTIA

The Translation Control Register's Table Index A value. This symbol is only defined when option **-mode** is specified to **SECURE** or **SAFE**.

DACS 680x0 Bare Ada Cross Compiler System - User's Guide

The Ada Linker

Ada_INIT\$MMUTIB

The Translation Control Register's Table Index B value. This symbol is only defined when option **-mode** is specified to **SECURE** or **SAFE**.

Ada_INIT\$MMUTIC

The Translation Control Register's Table Index C value. This symbol is only defined when option **-mode** is specified to **SECURE** or **SAFE**.

Ada_INIT\$MMUTID

The Translation Control Register's Table Index D value. This symbol is only defined when option **-mode** is specified to **SECURE** or **SAFE**.

Ada_INIT\$TLMMFreeCount

The size of a table describing free logical memory within all active task groups. This symbol is only defined when option **-mode** is specified to **SECURE** or **SAFE**.

6.9.2 Initialization Code

The initialization module contains code for initialization of the RTS components included in the target program. The initialization steps are executed in the same sequence as they are listed. When the initialization code is called, the interrupt priority level (IPL) mask in MC680x0 Status Register is assumed to be 7 and the active stack is assumed to be the interrupt stack.

Initialization of the Interrupt Stack

The interrupt stack pointer is initialized. This initialization is only performed when the option **-interrupt_stack** is specified.

Initialization of the Main Stack

The mode is changed to use the master stack pointer and the master stack pointer is initialized. This initialization is only performed if option **-main_task**, keyword **STACK_SIZE > 0**. If option **-mode** is set to **SECURE** or **SAFE** the supervisor main stack is initialized instead.

Initialization of the Interrupt Vector

The interrupt vector is either copied from the interrupt vector defined by the VBR register, or initialized completely, depending on the **-vector** option. If the **-novector** option is specified the interrupt vector is not initialized.

Initialization of the Virtual Memory Manager

If option **-mode** is set to **SECURE** or **SAFE** the Virtual Memory Manager is initialized. The VMM initialization generates internal data structures concerning free physical RAM memory specified by option **-ram** and free logical memory specified by option **-logical_memory**. Furthermore, MMU tables are created for supervisor code mapping all code as read only, user

DACS 680x0 Bare Ada Cross Compiler System - User's Guide

The Ada Linker

code mapping Ada code as read only, and supervisor data mapping all physical RAM memory as specified with option `-ram` as read/write.

Initialization of Classes

Initialization of classes is performed if option `-mode` is set to `SECURE` or `SAFE`. For each class a user data memory mapping table is generated. The table contains constants, stack, heap and permanent data for the class itself and data from other classes according to the rights defined in the class file in specified with option `-class_file`.

Initialization of the 680x0 Interrupt Vector Entries

The interrupt vector entries for the 680x0 exceptions that are used by the RTS are initialized by calling `Ada_UCC_E$InitMPUIV` (see the Configuration Guide [DDC-b] for more details). This initialization is only performed if `-noexceptions` has not been specified and the target program has the exception handler attribute.

Initialization of 6888x Interrupt Vector Entries

The interrupt vector entries for the 6888x or 68040 FPU exceptions that are used by the RTS are initialized by calling `Ada_UCC_F$InitFPUIV` (see the Configuration Guide [DDC-b] for more details). This initialization is only performed if `-noexceptions` has not been specified, and the program has the float and exception handler attributes.

Initialization of the Storage Manager

The parameter list defining the memory available to the storage manager is created and the storage manager is initialized. The storage manager is only initialized when the target program has the heap attribute.

Initialization of Exception Handler

Initialization for pre-handlers is performed.

Initialization of the Timer

If the target program has the task attribute, the timer is initialized by calling `Ada_UCC_A$InitTimer` (see the Configuration Guide [DDC-b] for more details). The timer may also be initialized when the package `Calendar` is included in the program, but that depends on the implementation of package `Calendar`. The implementation supplied by DDC-I will use the timer.

User Specified Initialization Code

At this point, user specified initialization code is called. Please refer to Section 6.2.11 for details on user specified initialization code.

Initialization of Frame Heap

The permanent frame heap headers on the outermost level are initialized. A frame heap header is a structure of heap elements at current block level.

Initialization of the Main Program

DACS 680x0 Bare Ada Cross Compiler System - User's Guide

The Ada Linker

The initialization of the main program allocates and initializes the display vector and initializes the frame pointer and the display pointer (A6, A5).

Initialization of the Tasking Kernel

If the target program has the task or the interrupt attribute, the tasking kernel is initialized. The initialization of the tasking kernel creates the main task. The initialization of the tasking kernel will also change the IPL to 0. If the program have neither of the mentioned attributes, the tasking kernel is not initialized, but the IPL is changed to 0.

Invocation of the Elaboration Module

The elaboration module is invoked and will execute the target program. The elaboration module is invoked by a branch and will return by a branch or a trap if linked in SECURE or SAFE mode.

Termination of the Main Task

Upon return from the elaboration module, the main program must wait until all tasks have terminated. If the target program has the task attribute, the routine in the tasking kernel terminating the main program is called. If not, the only task (the main task) is already terminated and no action is required.

Preparation for Termination of the Program

To ensure proper termination the IPL is raised to 7, the Master Stack will be the active stack afterwards.

Invocation of User Defined Termination Code

Transfers control to the user supplied termination routine Ada_UCC_B\$Exit (see [DDC-b]). Please refer to Section F.1.4 for details on user specified termination code (PRAGMA RUNDOWN).

6.9.3 Initialization

The initialize module allocates memory for the RTS data structures that depends on the target program or on options to the linker. The following data areas are defined, and made addressable by the symbols:

Ada_INIT\$InterruptVector

The address of the interrupt vector. This symbol is defined when the option -vector is specified. If ADDRESS=<address> is specified an absolute section is created at <address>, the symbol is equated to <address>, and 1024 bytes is allocated for the interrupt vector. If -novector is specified the symbol is not defined and the memory not allocated.

Ada_INIT\$InterruptStack

The start address of the interrupt stack. This symbol is defined when the option -interrupt_stack is specified. If START=<address> is specified an absolute section is created at <address>, the

**DACS 680x0 Bare Ada Cross Compiler System - User's Guide
The Ada Linker**

symbol is equated to <address> and the number of bytes specified with SIZE are reserved. Otherwise the interrupt stack is allocated in the RTS_DATA section by a DS directive.

Ada_INIT\$MainStack

The start address of the main program stack. The symbol is defined when the option -main_task is specified. If START=<address> is specified then the symbol is equated to <address> otherwise the main stack is allocated in the RTS_DATA section by a DS directive.

Ada_INIT\$MainDisplay

A pointer to the main display when the program has no tasking. The memory is only allocated when the target program has the task attribute.

Ada_INIT\$TCBAddress

The start address of the memory allocated for the task control blocks. The memory is allocated when the target program has the task attribute.

Ada_INIT\$ITCBAddress

The start address of the memory allocated for interrupt task control blocks. The memory is allocated when the target program has the interrupt attribute.

Ada_INIT\$CurrITCB<interrupt-no>

One long word is allocated for each interrupt vector entry that the target program references.

Ada_INIT\$SuperStackArea

The start address of the area from which supervisor stacks are allocated. Only supervisor stacks for tasks at user privilege level are allocated in this area. The memory is only allocated when the option -mode is set to SECURE or SAFE.

Ada_INIT\$TempHeap

The address of the memory for Frame heap header for allocation of temporary objects on the outermost lexical level.

Ada_INIT\$HeapHeaderArea

The start address of the memory allocated for heap headers. The memory is only allocated when the option -mode is set to SECURE or SAFE.

Ada_INIT\$FclTable

The start address of the memory allocated for the Function Code Lookup table used by the MMU. The address must be 16 bytes aligned. The memory is only allocated when the option -mode is set to SECURE or SAFE.

Ada_INIT\$FreePageCount

DACS 680x0 Bare Ada Cross Compiler System - User's Guide
The Ada Linker

The address of the memory allocated for the number of free physical pages. The memory is only allocated when the option **-mode** is set to **SECURE** or **SAFE**.

Ada_INIT\$FreePageList

The address of a pointer, addressing the list describing the free physical pages for each task group (see chapter 10). The memory is only allocated when the option **-mode** is set to **SECURE** or **SAFE**.

Ada_INIT\$FreePageIndex

The address of the memory pointing at the next free physical memory area in **Ada_INIT\$FreePageTable**. The memory is only allocated when the option **-mode** is set to **SECURE** or **SAFE**.

Ada_INIT\$FreePageTable

The start address of the memory allocated for the table of physical RAM memory areas used initially. The contents of the table is derived from the option **-ram**. The memory is only allocated when the option **-mode** is set to **SECURE** or **SAFE**.

Ada_INIT\$FreeSegmentIndex

The address of the memory pointing at the next entry with a free segment in **Ada_INIT\$FreeSegmentTable**. The memory is only allocated when the option **-mode** is set to **SECURE** or **SAFE**.

Ada_INIT\$FreeSegmentTable

The start address of the memory allocated for the table of free logical memory segments. The memory is only allocated when the option **-mode** is set to **SECURE** or **SAFE**.

Ada_INIT\$FreeSegmentTop

The address of the top of the free segment stack. The memory is only allocated when the option **-mode** is set to **SECURE** or **SAFE**.

Ada_INIT\$FreeSegmentStack

The start address of the memory allocated for the stack of deallocated segments. The memory is only allocated when the option **-mode** is set to **SECURE** or **SAFE**.

Ada_INIT\$TLMMFreeArea

The start address of the memory allocated for the table describing the free logical memory within all active task groups. The memory is only allocated when the option **-mode** is set to **SECURE** or **SAFE**.

DACS 680x0 Bare Ada Cross Compiler System - User's Guide The Ada Linker

6.10 The Elaboration Module

The elaboration module is generated as an assembly file, and the assembler is invoked to produce the actual object module. The assembly file is named `<prefix>_elab.arc` and the object file name is `<prefix>_elab.obj`.

The elaboration module has the entry point `Ada_ELAB$Entry`, which is invoked from the initialization module by a branch.

6.10.1 BASIC Execution Mode

The elaboration module has the following structure, when `-mode` is set to `BASIC`:

```
NAME  ADA_ELAB
CHIP  <selected cpu>
XDEF  Ada_ELAB$Entry
XREF  Ada_INIT$ElabExit
XREF  M$<main_unit_no>_1
XDEF  R$<unit_no_1>_0
XREF  M$<unit_no_1>_0
XDEF  R$<unit_no_2>_0
XREF  M$<unit_no_2>_0
...
```

SECTION ADA_CODE

```
Ada_ELAB$Entry:      bra.l    M$<unit_no_1>_0
R$<unit_no_1>_0:      bra.l    M$<unit_no_2>_0
R$<unit_no_2>_0:      ...
                      ...
                      bsr.l    M$<main_unit_no>_1
                      moveq.l  #0,d0
                      bra.l    Ada_INIT$ElabExit
```

END

The elaboration module branches to the elaboration for each of the included compilation units, and the elaboration code will branch back to the elaboration module. The elaboration code for a compilation unit will be identified by the label `M$<unit_no>_0` where `<unit_no>` is the unit number of the compilation unit. The return point is identified by the label `R$<unit_no>_0`.

6.10.2 SECURE and SAFE Execution Mode

When linking with option `-mode` set to `SECURE` or `SAFE`, the elaboration module has the following structure:

DACS 680x0 Bare Ada Cross Compiler System - User's Guide

The Ada Linker

```

NAME  ADA_ELAB
CHIP  <selected cpu>
XDEF  Ada_ELAB$Entry
XREF  Ada_INIT$ElabExit
XREF  M$<main_unit_no>_1
XREF  Ada_TK_X$Elaborate
XREF  Ada_TK_X$CallMain
XREF  Ada_INIT$MainClass
XREF  N$<class_name_1>
XREF  N$<class_name_2>
...

```

```

XDEF  R$<unit_no_1>_0
XREF  M$<unit_no_1>_0
XDEF  R$<unit_no_2>_0
XREF  M$<unit_no_2>_0
...

```

SECTION ADA_CODE

```

Ada_ELAB$Entry:  lea.l  N$<class_name>,a0
                  lea.l  M$<unit_no>-1,a1
                  bsr.l  Ada_TK_X$Elaborate
                  ...
                  lea.l  N$<class_name>,a0
                  lea.l  M$<unit_no>-1,a1
                  bsr.l  Ada_TK_X$Elaborate
                  ...
                  ...
                  lea.l  N$<class_name>,a0
                  lea.l  M$<main_unit_no>_1
                  moveq.l #0,d0
                  bsr.l  AdaTK_X$CallMain
                  lea.l  Ada_INIT$ElabExit,a0
                  moveq.l #0,d0
                  moveq.l #1,d7
                  trap   #13

```

```

R$<unit_no_1>_0:
R$<unit_no_2>_0:
...
...
R$<unit_no_<u>>_0:  moveq.l #0,d7
                   trap    #13
                   END

```

The elaboration of each compilation unit is handled by Ada_TK_X\$Elaborate which takes a class name and an elaboration code label M\$<unit_no>_0. The return point is identified by the label R\$<unit_no>_0. The elaboration of each compilation unit runs at user privilege level, but the setup for the elaboration of each compilation unit must run at supervisor privilege level. To switch from user privilege level to supervisor privilege level a trap operation is executed, and the elaboration of the next unit will proceed.

DACS 680x0 Bare Ada Cross Compiler System - User's Guide

The Ada Linker

6.10.3 Execution of the Main Program

When all the compilation units have been elaborated the main program is called. The main program is identified by the label `M$(main_unit_no)_1`. If the main program returns, register `d0` is cleared to signal successful completion, and control is passed back to the initialization module.

Please note that all transfers of control between the initialization module, the elaboration module, and the elaboration code is implemented by branch instructions. The branch instructions are used because the elaboration code may allocate objects on the stack, and consequently stack balance cannot be assumed.

6.11 Linker Examples

This section contains a number of linker examples. It is assumed that the compilation unit `example` is compiled into the default program sublibrary, that the environmental variable `ADA_LIBRARY` has been equated to the default sublibrary and that the environmental variable `ADA_UCC` has been equated to an User Configurable Code Library suitable for the target board on which the linked program will be executed. DDC-I provides UCC libraries for the Radstone CPU-3A and the Motorola MVME133, MVME143 and MVME165 boards.

Example: 1

```
$ al -noheap example
```

The program will start at address `0x10000`, and the heap is not initialized. If the target program has the heap attribute an error message is issued.

Example: 2

```
$ al -ram 0x10000,0xffff example
```

The program will start at address `0x10000`, and the heap will be placed within the address range `0x10000` to `0xffff`. This is the simplest form of a link that will support all Ada constructs.

Example: 3

```
$ al -ram_base 0x4000 -ram 0x0,0xffff example
```

The program will start at address `0x4000`, and the heap will be placed within the address range `0x4000` to `0xffff`. Physical memory below address `0x4000` is not used.

DACS 680x0 Bare Ada Cross Compiler System - User's Guide
The Ada Linker

Example: 4

```
$ al -rom_sections SUPER_CODE,USER_CODE,USER_CODE\  
    -ram_sections SUPER_DATA,USER_DATA\  
    -rom 0x200000,0x2ffff -rom_base 0x200000\  
    -ram 0x0,0xffff -ram_base 0x0 example
```

The sections SUPER_CODE, USER_CODE and USER_CONS are placed in ROM from address 0x200000. The sections SUPER_DATA and USER_DATA are placed in RAM from address 0x0. The part of the address range 0x0 to 0xffff not used by the SUPER_DATA section is used as heap space.

Example: 5

```
$ al -rom_sections SUPER_CODE -rom_base 0x200000\  
    -ram_sections SUPER_DATA -ram_base 0x0\  
    -scod RTS_CODE,ADA_CODE,ADA_CONS\  
    -noucod -noucst -vector INIT -boot\  
    -ram 0x0,0xffff -rom 0x200000,0x2ffff example
```

As example 4 but a module containing reset information is produced. The reset address is 0x200000. The interrupt vector is completely initialized. The first two long words of section RTS_CODE contains the initial PC and the initial interrupt stack pointer, consequently RTS_CODE must be the first section to load in order to control the reset address.

Example: 6

```
$ al -ram 0x10000,0xffff\  
    -vector ADDRESS=0x70000,COPY\  
    -interrupt_stack START=0x70000,SIZE=0x8000\  
    -main_task STACK_START=0x68000,STACK_SIZE=0x8000,\  
              PRIORITY=3,TIME_SLICE=0.2,FLOAT\  
    -task_defaults STACK_SIZE=0x8000,PRIORITY=4,\  
              TIME_SLICE=0.1 example
```

The start address of the program is 0x10000. The interrupt vector has the address 0x70000 and is a copy of the interrupt vector defined when the Ada program gains control. The interrupt stack has start address at 0x70000 and the size 0x8000. Please note that the first byte used in the stack is 0x6ffff and the stack grows by decrementing the stack pointer (stack grows "down" in memory). The main program stack's start address is 0x68000, and the size is 0x8000, the main program has a priority of 3 if pragma priority does not apply, the time slice is 0.2 seconds and the main program uses the floating point co-processor. The defaults used for task stack size is 0x8000, a priority of 4 and a time slice of 0.1 second.

DACS 680x0 Bare Ada Cross Compiler System - User's Guide
The Ada Linker

Example: 7

```
$ al -init_file my_file.src -option_file my_file.opt example
```

The linker was not able to generate an adequate initialization and option file, and the user decided to use his own. The `my_file.src` file contains the assembly source for the initialization module, and the `my_file.opt` contains the option file for the link. With this form of link the user has full control over the option file and the initialization module.

Example: 8

```
$ al -mode SECURE -class_file example.cls\  
-ram 0x10000,0xffff\  
-mmu_details SEGMENT_SIZE=16,PAGE_SIZE=10,\  
TIA=8,TIB=8,TIC=6 example
```

Execution of the program will be in SECURE mode. The class specification is in the file `example.cls`. Heaps and dynamic allocated stacks will be placed in address range 0x10000 to 0xffff. The MMU Translation Control Register is setup with a page size of 1K bytes, and TIA = 8, TIB = 8, TIC = 6, TID = 0, each class gets 64K bytes of logical memory for heaps and stacks.

APPENDIX C

APPENDIX F OF THE Ada STANDARD

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

package STANDARD is

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

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

type FLOAT is digits 6
range -3.4028234666385E+38 .. 3.4028234666385E+38;

type LONG_FLOAT is digits 15
range -1.7976931348623157E+308 .. 1.7976931348623157E+308;

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-680x0 required in Appendix F of the Ada Reference Manual (ANSI/MIL-STD-1815A).

F.1 Implementation-Dependent Pragmas

This section describes all implementation defined pragmas.

F1.1 PRAGMA INTERFACE_SPELLING

Format: `pragma INTERFACE_SPELLING(<subprogram-name>, <string>)`

Placement: The pragma may be placed as a declarative item.

Restrictions: Pragma `INTERFACE_SPELLING` must be applied to the subprogram denoted by `<subprogram-name>`. The `<string>` must be a string literal.

This pragma allows an Ada program to call routines with a name that is not a legal Ada name, the `<string>` provides the exact spelling of the name of the procedure.

F1.2 PRAGMA INTERFACE_TRAP

Format: `pragma INTERFACE_TRAP(<subprogram-name>, <string>, <integer>)`

Placement: The pragma may be placed as a declarative item.

Restrictions: The `<subprogram-name>` must denote a procedure or a function for which pragma interface to AS has been applied. The `<string>` must be a string literal. The `<integer>` must be greater than 3.

The pragma allows the programmer to implement assembler routines that need access to the run-time system code or data in a link mode independent manner. The string literal is used as the name for a global linker symbol, when the linker implements the call to the user supplied subroutine. The string literal must be unique when linking a program containing calls to subprograms for which `INTERFACE_TRAP` is applied. The integer is used as an index to the table of entry points in the kernel and must likewise be unique. When the integer is chosen, please consult the package `RTS_TRANSFER_INDICES` (see appendix C.8) to avoid conflicts with the indices used by the run-time system and support packages.

When control is passed to the user supplied routine register A4 contains the value of the stackpointer prior to the call; A4 is the only way to access parameters for the routine. The routine must maintain stack balance and must return by a RTS.

DACS 680x0 Bare Ada Cross Compiler System - User's Guide

Implementation Dependent Characteristics

For a program linked with the SECURE or SAFE mode it is checked that the task executing the routine has the "change mode to supervisor" privilege. The check is performed before control is passed to the user supplied routine.

F1.3 PRAGMA INITIALIZE

Format: `pragma INITIALIZE(<string_literal>)`

Placement: The pragma may be placed as a declarative item.

Restrictions: None.

When the pragma is applied the linker will, as part of the initialization code generate a call to the subprogram with the name `<string_literal>`. The call will be performed before the elaboration of the Ada program is initiated, with the interrupt mask in the Status Register at 7. If several pragmas INITIALIZE are applied to the same program the routines are called in the elaboration order, if several pragmas INITIALIZE are applied to one compilation unit the routines are called in the order of appearance. If several compilation units apply pragma INITIALIZE to the same routine the routine is only called once.

F1.4 PRAGMA RUNDOWN

Format: `pragma RUNDOWN(<string_literal>)`

Placement: The pragma may be placed as a declarative item.

Restrictions: None.

Similar to pragma initialize, but the subprogram is called after the main program have terminated and in the reverse order of the elaboration order.

F1.5 PRAGMA TASKS

Format: `pragma TASKS;`

Placement: The pragma may be placed as a declarative item.

Restrictions: None.

Marks the compilation unit with the task attribute. If the code that is interfaced by a pragma INTERFACE uses any tasking constructs, the compilation unit must be marked such that the linker includes the tasking kernel in target program at reference the compilation unit.

DACS 680x0 Bare Ada Cross Compiler System - User's Guide
Implementation Dependent Characteristics

F1.6 PRAGMA FLOATS

Format: **pragma FLOATS;**

Placement: The pragma may be placed as a declarative item.

Restrictions: None.

Marks the compilation unit with the float attribute. If the code that is interfaced by a pragma INTERFACE uses any floating point co-processor instructions, the compilation unit must be marked such that the linker includes initialization of the floating point co-processor in target programs that reference the compilation unit.

F1.7 PRAGMA INTERRUPTS

Format: **pragma INTERRUPTS;**

Placement: The pragma may be placed as a declarative item.

Restrictions: None.

Marks the compilation unit with the interrupt attribute. If the code that is interfaced by a pragma INTERFACE uses any interrupts, the compilation unit must be marked such that the linker include the interrupt handling in target programs that reference the compilation unit.

F1.8 PRAGMA STORAGE_MANAGER

Format: **pragma STORAGE_MANAGER;**

Placement: The pragma may be placed as a declarative item.

Restrictions: None.

Marks the compilation unit with the heap attribute. If the code that is interfaced by a pragma INTERFACE uses the storage manager, the compilation unit must be marked such that the linker include initialization of the storage manager in target programs that reference the compilation unit.

F1.9 PRAGMA INTERRUPT_HANDLER

The pragma INTERRUPT_HANDLER is defined with two formats.

F1.9.1 PRAGMA INTERRUPT_HANDLER for Task Entries

Format: `pragma INTERRUPT_HANDLER;`

Placement: The pragma must be placed as the first declarative item in the task specification that it applies to.

Restrictions: The task for which the pragma `INTERRUPT_HANDLER` is applied must fulfill the following requirements:

- 1) The pragma must appear first in the specification of the task and an address clause must be given to all entries defined in the task, see below.

```
task fih is
  pragma interrupt_handler;
  entry handler1;
  for handler1 use at 254;
  entry handler2;
  for handler2 use at 255;
end fih;
```

- 2) All entries of the task must be single entries with no parameters.
- 3) The entries must not be called from any tasks.
- 4) No other tasks may be specified in the body of the task.
- 5) The body of the task must consist of a single sequence of accept statements for each of the defined interrupts, see below:

```
task body fih is
  -- local simple data declaration, no tasks.
begin
  accept handler1 do
    <statementlist>;
  end handler1;
  accept handler2 do
    <statementlist>;
  end handler2;
end fih;
```

- 6) The only tasking construct that may be used from the body of an accept statement is unconditional entry calls. Several unconditional entry calls may appear in the body of an accept statement but only one entry call must be made during the handling of the interrupts.
- 7) Any procedures called from the accept body may not use any tasking constructs at all.
- 8) A given entry must only be accepted once within the body of an FIH.
- 9) No exceptions may be propagated out of the task body.

If the restrictions described above are not fulfilled, the program is erroneous and the result of the execution unpredictable. The compiler cannot and is not checking all the restrictions, but attempts to perform as many checks of the requirements as possible.

DACS 680x0 Bare Ada Cross Compiler System - User's Guide
Implementation Dependent Characteristics

The pragma **INTERRUPT_HANDLER** with no parameters allows the user to implement immediate response to exceptions.

F1.9.2 PRAGMA INTERRUPT_HANDLER for Procedures

Format: **pragma INTERRUPT_HANDLER(procedure-name, integer-literal);**

Placement: The pragma must be placed as a declarative item, in the declarative part, immediately after the procedure specification.

Restrictions: The procedure for which pragma **INTERRUPT_HANDLER** applies must fulfill the following restrictions:

- 1) The pragma must appear before the body of the procedure.
- 2) The procedure must not be called anywhere in the application.
- 3) No tasks may be declared in the body of the procedure.
- 4) The only tasking construct that may be used from the body of the procedure is unconditional entry calls. Several unconditional entry calls may appear in the body of the procedure, but only one entry call may be made during the handling of the interrupt.
- 5) Any subprograms called from the procedure must not use any tasking constructs at all.
- 6) The procedure must have no parameters.
- 7) No exceptions may be propagated out of the procedure.

If the restrictions described above is not fulfilled the program is erroneous and the result of the execution unpredictable. The compiler cannot and is not checking all the restrictions, but attempts to perform as many checks of the requirements as possible.

The pragma **INTERRUPT_HANDLER** for procedures defines the named subprogram to be an interrupt handler for the interrupt vector entry defined by the integer-literal.

F1.10 PRAGMA NO_FLOATING_POINTS

Format: **pragma NO_FLOATING_POINTS(task-id)**

Placement: The pragma must be placed as a declarative item, in the declarative part, defining the task type or object denoted by the task-id.

Restrictions: The task(s) denoted by the task-id must not execute floating-point co-processor instructions.

This pragma informs the compiler and run-time system that the task will not execute floating point co-processor instructions. Consequently the context switch needs not save and restore the state of the floating point co-processor yielding improved performance.

DACS 680x0 Bare Ada Cross Compiler System - User's Guide
Implementation Dependent Characteristics

F1.11 PRAGMA SUPERVISOR_TASK

Format: `pragma SUPERVISOR_TASK`

Placement: The pragma must be placed immediately after the task declaration of the task declaring it as a `SUPERVISOR_TASK`.

Restrictions: The pragma has no meaning if linking with BASIC mode.

This pragma informs the compiler and run-time system that the task shall execute at the supervisor privilege level, all other tasks will execute at user privilege level when linking with SECURE or SAFE mode. In BASIC mode all tasks execute at the supervisor privilege level.

F1.12 PRAGMA ACCESS_TYPE_RETAIN_HEAP

Format: `pragma ACCESS_TYPE_RETAIN_HEAP`

Placement: The pragma must be placed as a declarative item in the declarative part, immediately after the procedure specification.

Restrictions: The pragma can only be used when linking in BASIC mode.

This pragma suppresses garbage collection of access types, when leaving the scope of the access type declaration.

F.2 Implementation-Dependent Attributes

No implementation-dependent attributes are defined.

DACS 680x0 Bare Ada Cross Compiler System - User's Guide

Implementation Dependent Characteristics

F.3 Package SYSTEM

package SYSTEM is

```
type ADDRESS          is new INTEGER;
subtype PRIORITY      is INTEGER range 1 .. 24;
type NAME             is ( DACS_680X0 );
SYSTEM_NAME:         constant NAME := DACS_680X0;
STORAGE_UNIT:        constant      := 8;
MEMORY_SIZE:         constant      := 2#1#E32;
MIN_INT:             constant      := -2_147_483_648;
MAX_INT:             constant      := 2_147_483_647;
MAX_DIGITS:          constant      := 15;
MAX_MANTISSA:        constant      := 31;

FINE_DELTA:          constant      := 2#1.0#E-31;
TICK:                constant      := 2#1.0#E-14;
```

type interface_language is (AS,C);

end SYSTEM;

The basic clock period **SYSTEM.TICK** is not utilized by DACS-680x0. The real time between each successive timer tick will be a multiplum of **SYSTEM.TICK**, but the actual time between each timer tick depends on a given target board and is specified in the User Configurable Code (UCC).

F.4 Representation Clauses

The DACS-680x0 fully supports the 'SIZE representation for derived types. The representation clauses that are accepted for non-derived types are described in the following subsections.

F4.1 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 32 bits.
- **SIZE** is only obeyed for discrete types when the type is a part of a composite object, e.g. arrays or records.
- 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, the process stack area will be of the specified size.

DACS 680x0 Bare Ada Cross Compiler System - User's Guide
Implementation Dependent Characteristics

F4.2 Enumeration Representation Clauses

Enumeration representation clauses may specify representations in the range of `INTEGER'FIRST + 1..INTEGER'LAST - 1`.

F4.3 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 at a storage unit boundary (8 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 2 giga bits.
- A component must be specified with its proper size (in bits), regardless of whether the component is an array or not.
- If a non-array component has a size which equals or exceeds one storage unit 32-bits the component must start on a storage unit boundary.
- The elements in an array component should always be wholly contained in 32-bits.

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

`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.

F4.3.1 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, otherwise only longword 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 longword 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 longword; an error message is issued if this is not the case.

DACS 680x0 Bare Ada Cross Compiler System - User's Guide

Implementation Dependent Characteristics

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.

F6.1 Objects

Address clauses are supported for scalar and composite objects whose size can be determined at compile time if the address is specified.

F6.2 Task Entries

Address clauses are supported for task entries. The following restrictions applies:

- The affected entries must be defined in a task object only, not a task type.
- The entries must be single and parameterless.
- The address specified must not denote an interrupt index which the processor may trap.
- If the interrupt entry executes floating point co-processor instructions the state of the co-processor must be saved prior to execution of any floating point instructions, and restored before the return.

The address specified in the address clause denotes the interrupt vector index.

F.7 Unchecked Programming

Both **UNCHECKED_DEALLOCATION** and **UNCHECKED_CONVERSION** are supported as indicated below.

F7.1 Unchecked Deallocation

Unchecked deallocation is fully supported through the procedure **UNCHECKED_DEALLOCATION** as defined in [DoD-83] 13.10.1.

DACS 680x0 Bare Ada Cross Compiler System - User's Guide
Implementation Dependent Characteristics

F7.2 Unchecked Conversion

Unchecked conversion is fully supported through the procedure **UNCHECKED_CONVERSION** as defined in [DoD-83] 13.10.2. Unchecked conversion is only allowed between objects of the same "size". However, if a scalar type have 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.

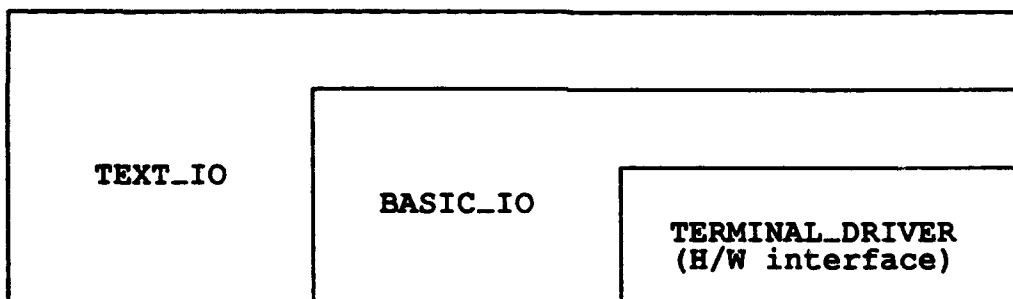


Figure F.1:

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

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

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

Table F.1: Mapping between **TEXT_IO** and ASCII

DACS 680x0 Bare Ada Cross Compiler System - User's Guide Implementation Dependent Characteristics

The services provided by the terminal driver are:

- 1) Reading a character from the communications port.
- 2) Writing a character to the communications port.

F8.1 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 2#....#
    subtype FIELD is INTEGER range 0 .. 35;

    subtype NUMBER_BASE is INTEGER range 2 .. 16;

    type TYPE_SET is (LOWER_CASE, UPPER_CASE);
```

DACS 680x0 Bare Ada Cross Compiler System - User's Guide
Implementation Dependent Characteristics

```
pragma PAGE;
-- File Management
procedure CREATE (FILE : in out FILE_TYPE;
                  MODE : in FILE_MODE := OUT_FILE;
                  NAME : in STRING := "";
                  FORM : in STRING := "")
);

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

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

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

function IS_OPEN(FILE : in FILE_TYPE) return BOOLEAN;

pragma PAGE;
-- 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;
```

DACS 680x0 Bare Ada Cross Compiler System - User's Guide
Implementation Dependent Characteristics

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

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

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

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

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

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

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

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

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

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

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

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

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

DACS 680x0 Bare Ada Cross Compiler System - User's Guide
Implementation Dependent Characteristics

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);
```

DACS 680x0 Bare Ada Cross Compiler System - User's Guide
Implementation Dependent Characteristics

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

generic
  type NUM is range <>;
package INTEGER_IO is

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

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

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

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

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

end INTEGER_IO;
```

DACS 680x0 Bare Ada Cross Compiler System - User's Guide
Implementation Dependent Characteristics

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;

DACS 680x0 Bare Ada Cross Compiler System - User's Guide
Implementation Dependent Characteristics

```
pragma PAGE;

generic
  type NUM is delta <>;
package FIXED_IO is

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

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

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

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

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

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

end FIXED_IO;
```

DACS 680x0 Bare Ada Cross Compiler System - User's Guide

Implementation Dependent Characteristics

```

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;

```

DACS 680x0 Bare Ada Cross Compiler System - User's Guide
Implementation Dependent Characteristics

F8.2 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;

F8.3 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  
-- litteral.  
--  
-- 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.  
--
```

DACS 680x0 Bare Ada Cross Compiler System - User's Guide
Implementation Dependent Characteristics

function get_real return string;

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

function get_enumeration return string;

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

function get_item(length : in integer) return string;

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

procedure put_item(item : in string);

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

-- Line and page lengths - ARM 14.3.3.

--

procedure set_line_length(to : in count);

procedure set_page_length(to : in count);

function line_length return count;

function page_length return count;

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

--

procedure new_line;

procedure skip_line;

function end_of_line return boolean;

procedure new_page;

procedure skip_page;

DACS 680x0 Bare Ada Cross Compiler System - User's Guide
Implementation Dependent Characteristics

```
function end_of_page return boolean;

function end_of_file return boolean;

procedure set_col(to   : in   positive_count);
procedure set_line(to  : in   positive_count);
function col return positive_count;
function line return positive_count;
function page return positive_count;

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

procedure get_character(item :    out character);

procedure get_string(item :    out string);

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

procedure put_character(item : in   character);
procedure put_string(item : in   string);
procedure put_line(item : in   string);

-- exceptions:

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

end BASIC_IO;
```

DACS 680x0 Bare Ada Cross Compiler System - User's Guide
Implementation Dependent Characteristics

F8.4 Package TERMINAL_DRIVER

The specification of package TERMINAL_DRIVER:

```
package terminal_driver is

  procedure put_character(ch : character);
  procedure flush;

  function get_character return character;

  procedure purge;

private
  pragma interface (AS, put_character);
  pragma interface_spelling(put_character, "Ada_UCC_G$PutByte");

  pragma interface (AS, get_character);
  pragma interface_spelling(get_character, "Ada_UCC_G$GetByte");

  pragma interface (AS, flush);
  pragma interface_spelling(flush, "Ada_UCC_G$FlushOutput");

  pragma interface (AS, purge);
  pragma interface_spelling(purge, "Ada_UCC_G$PurgeInput");

  pragma initialize("Ada_UCC_G$InitIO");
  pragma rundown ("Ada_UCC_G$CloseIO");

end terminal_driver;
```

F8.5 Package SEQUENTIAL_IO

As files are not supported, the subprograms in this package will raise USE_ERROR or STATUS_ERROR. The specification of package SEQUENTIAL_IO:

```
-- Source code for SEQUENTIAL_IO

pragma PAGE;

with IO_EXCEPTIONS;

generic

  type ELEMENT_TYPE is private;
```

DACS 680x0 Bare Ada Cross Compiler System - User's Guide
Implementation Dependent Characteristics

package SEQUENTIAL_IO is

type FILE_TYPE is limited private;

type FILE_MODE is (IN_FILE, OUT_FILE);

pragma PAGE;

-- File management

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

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

procedure CLOSE (FILE : in out FILE_TYPE);

procedure DELETE(FILE : in out FILE_TYPE);

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

procedure RESET (FILE : in out FILE_TYPE);

function MODE (FILE : in FILE_TYPE) return FILE_MODE;

function NAME (FILE : in FILE_TYPE) return STRING;

function FORM (FILE : in FILE_TYPE) return STRING;

function IS_OPEN(FILE : in FILE_TYPE) return BOOLEAN;

pragma PAGE;

-- input and output operations

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

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

function END_OF_FILE(FILE : in FILE_TYPE) return BOOLEAN;

DACS 680x0 Bare Ada Cross Compiler System - User's Guide
Implementation Dependent Characteristics

```
pragma PAGE;  
-- exceptions
```

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

```
pragma PAGE;  
private
```

```
    type FILE_TYPE is new INTEGER;  
  
end SEQUENTIAL_IO;
```

F3.6 Package DIRECT_IO

As files are not supported, the subprograms in this package will raise `USE_ERROR` or `STATUS_ERROR`. The specification of package `DIRECT_IO`:

```
pragma PAGE;  
with IO_EXCEPTIONS;
```

```
generic
```

```
    type ELEMENT_TYPE is private;
```

```
package DIRECT_IO is
```

```
    type FILE_TYPE is limited private;
```

```
    type FILE_MODE is (IN_FILE, INOUT_FILE, OUT_FILE);
```

```
    type COUNT is range 0..2_147_483_647;
```

```
    subtype POSITIVE_COUNT is COUNT range 1..COUNT'LAST;
```

DACS 680x0 Bare Ada Cross Compiler System - User's Guide
Implementation Dependent Characteristics

pragma PAGE;
-- File management

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

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

```
procedure CLOSE (FILE : in out FILE_TYPE);
```

```
procedure DELETE(FILE : in out FILE_TYPE);
```

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

```
procedure RESET (FILE : in out FILE_TYPE);
```

```
function MODE  (FILE : in FILE_TYPE) return FILE_MODE;
```

```
function NAME  (FILE : in FILE_TYPE) return STRING;
```

```
function FORM  (FILE : in FILE_TYPE) return STRING;
```

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

DACS 680x0 Bare Ada Cross Compiler System - User's Guide

Implementation Dependent Characteristics

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

```
procedure READ (FILE : in FILE_TYPE;  
               ITEM : out ELEMENT_TYPE;  
               FROM : in POSITIVE_COUNT);  
procedure READ (FILE : in FILE_TYPE;  
               ITEM : out ELEMENT_TYPE);  
  
procedure WRITE (FILE : in FILE_TYPE;  
                ITEM : in ELEMENT_TYPE;  
                TO  : in POSITIVE_COUNT);  
procedure WRITE (FILE : in FILE_TYPE;  
                ITEM : in ELEMENT_TYPE);  
  
procedure SET_INDEX(FILE : in FILE_TYPE;  
                   TO   : in POSITIVE_COUNT);  
  
function INDEX(FILE : in FILE_TYPE) return POSITIVE_COUNT;  
  
function SIZE (FILE : in FILE_TYPE) return COUNT;  
  
function END_OF_FILE(FILE : in FILE_TYPE) return BOOLEAN;
```

```
pragma PAGE;  
-- exceptions
```

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

```
pragma PAGE;  
private
```

```
type FILE_TYPE is new INTEGER;
```

```
end DIRECT_IO;
```

F.9 Package CALENDAR

Package CALENDAR is as defined in [DoD-83] section 9.6, except for a new procedure SET_TIME, which has been added to the package. SET_TIME allows setting of TIME for the duration of the executing program. SET_TIME parameters follow the same conventions as the parameters for SPLIT. The specification of package CALENDAR:

DACS 680x0 Bare Ada Cross Compiler System - User's Guide

Implementation Dependent Characteristics

```
PRAGMA PAGE;
PACKAGE calendar IS
```

```
    TYPE Time IS PRIVATE;
```

```
    SUBTYPE Year_number IS Integer RANGE 1901..2099;
    SUBTYPE Month_number IS Integer RANGE 1..12;
    SUBTYPE Day_number IS Integer RANGE 1..31;
    SUBTYPE Day_duration IS Duration RANGE 0.0..86_400.0;
```

```
    FUNCTION clock RETURN Time;
```

```
    FUNCTION year(date: Time) RETURN Year_number;
    FUNCTION month(date: Time) RETURN Month_number;
    FUNCTION day(date: Time) RETURN Day_number;
    FUNCTION seconds(date: Time) RETURN Day_duration;
```

```
    PROCEDURE split(date: IN Time;
                    year: OUT Year_number;
                    month: OUT Month_number;
                    day: OUT Day_number;
                    seconds: OUT Day_duration);
```

```
    FUNCTION time_of(year: Year_number;
                    month: Month_number;
                    day: Day_number;
                    seconds: Day_duration := 0.0) RETURN Time;
```

```
    FUNCTION "+" (left: Time;
                  right: Duration) RETURN Time;
```

```
    FUNCTION "+" (left: Duration;
                  right: Time) RETURN Time;
```

```
    FUNCTION "-" (left: Time;
                  right: Duration) RETURN Time;
```

```
    FUNCTION "-" (left: Time;
                  right: Time) RETURN Duration;
```

```
    FUNCTION "<" (left, right: Time) RETURN Boolean;
    FUNCTION "<=" (left, right: Time) RETURN Boolean;
    FUNCTION ">" (left, right: Time) RETURN Boolean;
    FUNCTION ">=" (left, right: Time) RETURN Boolean;
```

```
    PROCEDURE set_time(year    : IN Year_number;
                      month   : IN Month_number;
                      day     : IN Day_number;
                      seconds  : IN Day_duration);
```

```
    TIME_ERROR: Exception;    -- ...can be raised by
                                -- time_of , "+" and "-" .
```

```
PRIVATE
```

```
    ...
END calendar;
```

DACS 680x0 Bare Ada Cross Compiler System - User's Guide
Implementation Dependent Characteristics

F.10 Machine Code Insertions

Machine code insertions are allowed using the instructions defined in package `MACHINE_CODE`. All arguments given in the code statement aggregate must be static.

The machine language defined in package `MACHINE_CODE` is not 680x0 assembler, but rather Abstract A-code which is an intermediate language used by the compiler.