AD-A239 784

ATION PAGE

Form Approved
OPM No. 0704-0188

Public r... response, including the time for reviewing instructions, searching existing data sources gathering and maintaining the data needed, ... estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headqu... Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Information and Regulatory Affairs, Office of Manage...

1. AGE... USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
		Final: 04Feb 1991 to 01Jun 1993

4. TITLE AND SUBTITLE
Harris Corporation, Computer Systems Division, Harris Ada 5.1, Harris NH-4400 (Host & Target), 900918W1.11028

5. FUNDING NUMBERS

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

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

8. PERFORMING ORGANIZATION REPORT NUMBER
AVF-VSR-388.291

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

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

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

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*
Harris Corporatopm. Computer Systems Division Harris Ada 5.1, Wright-Patterson AFB, OH, Harris NH-4400 (under CX/UX 5.1)(Host & Target), ACVC 1.11.

DTIC
ELECTE
AUG 26 1991
S B D

91-08769

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

15. NUMBER OF PAGES

16. PRICE CODE

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

Certificate Information

The following Ada implementation was tested and determined to pass ACVC 1.11. Testing was completed on 18 September 1991.

Compiler Name and Version: Harris Ada 5.1

Host Computer System: Harris NH-4400 (under CX/UX 5.1)

Target Computer System: Harris NH-4400 (under CX/UX 5.1)

Customer Agreement Number: 90-06-25-HAR

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

As a result of this validation effort, Validation Certificate 900918W1.11028 is awarded to Harris Corporation, Computer Systems Division. This certificate expires on 1 March 1993.

This report has been reviewed and is approved.

Steven P. Wilson

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

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

DTIC COPY INSPECTED 6

John P. Solomond

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

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

AVF Control Number: AVF-VSR-388.0291
4 February 1991
90-06-25-HAR

Ada COMPILER
VALIDATION SUMMARY REPORT:
Certificate Number: 900918W1.11028
Harris Corporation, Computer Systems Division
Harris Ada 5.1
Harris NH-4400 => Harris NH-4400

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

Certificate Information

The following Ada implementation was tested and determined to pass ACVC 1.11. Testing was completed on 18 September 1991.

Compiler Name and Version: Harris Ada 5.1

Host Computer System: Harris NH-4400 (under CX/UX 5.1)

Target Computer System: Harris NH-4400 (under CX/UX 5.1)

Customer Agreement Number: 90-06-25-HAR

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

As a result of this validation effort, Validation Certificate 900918W1.11028 is awarded to Harris Corporation, Computer Systems Division. This certificate expires on 1 March 1993.

This report has been reviewed and is approved.

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

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

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

DECLARATION OF CONFORMANCE

Customer: Harris Corporation, Computer Systems Division

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

ACVC Version: 1.11

Ada Implementation:

Compiler Name and Version: Harris Ada 5.1

Host Computer System: Harris NH-4400 (under CX/UX 5.1)

Target Computer System: Harris NH-4400 (under CX/UX 5.1)

Customer's Declaration

I, the undersigned, representing Harris Corporation, Computer Systems Division (HCSD), declare that HCSD has no knowledge of deliberate deviations from the Ada Language Standard ANSI/MIL-STD-1815A in the implementation listed in this declaration. I declare that HCSD is the licensee, owner, and distributor of the above implementation and the certificates shall be awarded in the name of Harris Corporation.

Wendell Norton Date: 8-1-90

Wendell Norton, Director of Contracts
Harris Corporation, Computer Systems Division
2101 West Cypress Creek Rd
Ft. Lauderdale, FL 33309-1892

TABLE OF CONTENTS

CHAPTER 1 INTRODUCTION

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

CHAPTER 2 IMPLEMENTATION DEPENDENCIES

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

CHAPTER 3 PROCESSING INFORMATION

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

APPENDIX A MACRO PARAMETERS

APPENDIX B COMPILATION SYSTEM OPTIONS

APPENDIX C APPENDIX F OF THE Ada STANDARD

CHAPTER 1

INTRODUCTION

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

1.1 USE OF THIS VALIDATION SUMMARY REPORT

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

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

National Technical Information Service
5285 Port Royal Road
Springfield VA 22161

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

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

1.2 REFERENCES

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

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

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

1.3 ACVC TEST CLASSES

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

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

Class B tests check that a compiler detects illegal language usage. Class B tests are not executable. Each test in this class is compiled and the resulting compilation listing is examined to verify that all violations of the Ada Standard are detected. Some of the class B tests contain legal Ada code which must not be flagged illegal by the compiler. This behavior is also verified.

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

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

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

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

1.4 DEFINITION OF TERMS

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

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.
Operating System	Software that controls the execution of programs and that provides services such as resource allocation, scheduling, input/output control, and data management. Usually, operating systems are predominantly software, but partial or complete hardware implementations are possible.
Target Computer System	A computer system where the executable form of Ada programs are executed.
Validated Ada Compiler	The compiler of a validated Ada implementation.
Validated Ada Implementation	An Ada implementation that has been validated successfully either by AVF testing or by registration [Pro90].
Validation	The process of checking the conformity of an Ada compiler to the Ada programming language and of issuing a certificate for this implementation.
Withdrawn test	A test found to be incorrect and not used in conformity testing. A test may be incorrect because it has an invalid test objective, fails to meet its test objective, or contains erroneous or illegal use of the Ada programming language.

CHAPTER 2

IMPLEMENTATION DEPENDENCIES

2.1 WITHDRAWN TESTS

The following tests have been withdrawn by the AVO. The rationale for withdrawing each test is available from either the AVO or the AVF. The publication date for this list of withdrawn tests is 2 September 1990.

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

2.2 INAPPLICABLE TESTS

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

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

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

The following 21 tests check for the predefined type LONG_INTEGER:

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

C35702A, C35713B, C45423B, B86001T, and C86006H check for the predefined type SHORT_FLOAT.

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

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

C45531I..L (4 tests) and C45532I..L (4 tests) check fixed-point operations for types that require a SYSTEM.MAX_MANTISSA of 31 or greater. For this implementation SYSTEM.MAX_MANTISSA = 30.

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

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

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

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

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

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

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

The tests listed in the following table are not applicable because the given file operations are supported for the given combination of mode and file access method.

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

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

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

CE3115A attempts resetting of an external file with OUT_FILE mode, which is not supported with multiple internal files associated with the same external file when they have different modes.

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

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

2.3 TEST MODIFICATIONS

Modifications (see section 1.3) were required for 16 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.

B24009A	B33301B	B38003A	B38003B	B38009A	B38009B
B85008G	B85008H	B91001H	BC1303F	BC3005B	BD2B03A
BD2D03A	BD4003A	BD8004C			

CE3804H requires that string "-3.525" can be read from a file using FLOAT_IO and that it equal the numeric literal "-3.525"; however, because -3.525 is not a model number this equality need not hold. This implementation reports FAILED and prints (only) the Report.Failed message from line 116, "WIDTH CHARACTERS NOT READ"; the AVO ruled that the test be graded as passed by Evaluation Modification.

CHAPTER 3

PROCESSING INFORMATION

3.1 TESTING ENVIRONMENT

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

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

Jeff Hollensen
Harris Corporation, Computer Systems Division
2101 W. Cypress Creek Rd.
Ft. Lauderdale FL 33309

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

Harris Computer Systems Division
Marketing Communication
(305) 973-5124

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

3.2 SUMMARY OF TEST RESULTS

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

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

a) Total Number of Applicable Tests	3814
b) Total Number of Withdrawn Tests	74
c) Processed Inapplicable Tests	81
d) Non-Processed I/O Tests	0
e) Non-Processed Floating-Point Precision Tests	201
f) Total Number of Inapplicable Tests	282
g) Total Number of Tests for ACVC 1.11	4170

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

3.3 TEST EXECUTION

Version 1.11 of the ACVC comprises 4170 tests. When this compiler was tested, the tests listed in section 2.1 had been withdrawn because of test errors. The AVF determined that 282 tests were inapplicable to this implementation. All inapplicable tests were processed during validation testing except for 201 executable tests that use floating-point precision exceeding that supported by the implementation In addition, the modified tests mentioned in section 2.3 were also processed.

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.

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

Option | Setting	Effect
-el	If warning or errors occur during compilation, generate a full source listing with the warning/error messages included in the listing.

-w	Suppress compilation warning messages.
-L	Generate a full source listing even if no errors or warnings occurred during compilation.
-Df	This option causes file name paths in error listings to only include the file portion of the path name, instead of the entire root directory (e.g.: bc3205d.a, instead of: /usr2/ada/acvc/1.11/prevals/bc/bc3205d.a).

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

APPENDIX A

MACRO PARAMETERS

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

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

MACRO PARAMETERS

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

Macro Parameter	Macro Value
$MAX_IN_LEN	499
$ACC_SIZE	32
$ALIGNMENT	8
$COUNT_LAST	2147483647
$DEFAULT_MEM_SIZE	3221225469
$DEFAULT_STOR_UNIT	8
$DEFAULT_SYS_NAME	HARRIS_88K
$DELTA_DOC	2.0**(-30)
$ENTRY_ADDRESS	SYSTEM.PHYSICAL_ADDRESS(16)
$ENTRY_ADDRESS1	SYSTEM.PHYSICAL_ADDRESS(17)
$ENTRY_ADDRESS2	SYSTEM.PHYSICAL_ADDRESS(18)
$FIELD_LAST	2147483647
$FILE_TERMINATOR	''
$FIXED_NAME	NO_SUCH_FIXED_TYPE
$FLOAT_NAME	NO_SUCH_FLOAT_TYPE
$FORM_STRING	""
$FORM_STRING2	"CANNOT_RESTRICT_FILE_CAPACITY"
$GREATER_THAN_DURATION	100000.0
$GREATER_THAN_DURATION_BASE_LAST	10000000.0
$GREATER_THAN_FLOAT_BASE_LAST	3.5E+38
$GREATER_THAN_FLOAT_SAFE_LARGE	1.0E38

```
$GREATER_THAN_SHORT_FLOAT_SAFE_LARGE
                        1.0E308

$HIGH_PRIORITY          9

$ILLEGAL_EXTERNAL_FILE_NAME1
                        /no/such/file/name

$ILLEGAL_EXTERNAL_FILE_NAME2
                        /this/file/does/not/exist

$INAPPROPRIATE_LINE_LENGTH
                        -1

$INAPPROPRIATE_PAGE_LENGTH
                        -1

$INCLUDE_PRAGMA1        PRAGMA INCLUDE("A28006D1.TST")

$INCLUDE_PRAGMA2        PRAGMA INCLUDE("B28006D1.TST")

$INTEGER_FIRST          -2147483648

$INTEGER_LAST           2147483647

$INTEGER_LAST_PLUS_1    2147482648

$INTERFACE_LANGUAGE     C

$LESS_THAN_DURATION     -100000.0

$LESS_THAN_DURATION_BASE_FIRST
                        -10000000.0

$LINE_TERMINATOR        ASCII.LF

$LOW_PRIORITY           0

$MACHINE_CODE_STATEMENT
                        code_3'(or_r,r0,r0,r0);

$MACHINE_CODE_TYPE      operand

$MANTISSA_DOC           30

$MAX_DIGITS             15

$MAX_INT                2147483647

$MAX_INT_PLUS_1         2147483648

$MIN_INT                -2147483648
```

MACRO PARAMETERS

```
$NAME                   TINY_INTEGER

$NAME_LIST              HARRIS_88K

$NAME_SPECIFICATION1
     /gx1d9/ada/acvc/1.11/preval_gcx_1.11/SRC/ce/X2120A

$NAME_SPECIFICATION2
     /gx1d9/ada/acvc/1.11/preval_gcx_1.11/SRC/ce/X2120B

$NAME_SPECIFICATION3
     /gx1d9/ada/acvc/1.11/preval_gcx_1.11/SRC/ce/X3119A

$NEG_BASED_INT          16#F000000E#

$NEW_MEM_SIZE           3221225469

$NEW_STOR_UNIT          8

$NEW_SYS_NAME           HARRIS_88K

$PAGE_TERMINATOR        ASCII.LF & ASCII.FF

$RECORD_DEFINITION
 type code_1(op:opcode) is record oprnd_1: operand; end record;

$RECORD_NAME            code_1

$TASK_SIZE              32

$TASK_STORAGE_SIZE      10240

$TICK                   0.01

$VARIABLE_ADDRESS       FCNDECL.SPACE(1024)

$VARIABLE_ADDRESS1      FCNDECL.SPACE(1024)

$VARIABLE_ADDRESS2      FCNDECL.SPACE(1024)

$YOUR_PRAGMA            EXTERNAL_NAME
```

APPENDIX B

COMPILATION SYSTEM OPTIONS

The compiler and 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 compiler documentation and not to this report.

Harris Ada Compiler and Link Options

Usage: ada [options] ada_source.a ... [a.ld options] [object_file.o]

Options:

[-b]	(object)	Symbolic object listing to stdout
[-d]	(dependencies)	Only check for dependencies
[-dr]	(data record)	Allow variables to be "data recorded"
[-e]	(errors)	List syntax errors to stdout
[-eh]	(errors hed)	Errors to source, call hed
[-el]	(errors list)	Errors & source to stdout
[-ev]	(errors vi)	Errors to source, call vi
[-m]	(map)	Print an object map (only with -M option)
[-o name]	(output)	Name the executable program
[-p]	(profiling)	Profiling, instrument code for a.prof(1)
[-pp "options"]	(preprocess)	Pass options to a.pp
[-u]	(update)	Force update of ada library
[-v]	(verbose)	Print info about the compile
[-w]	(warnings)	Suppress warnings
[-Df]		File name paths only include file portion of the path name,
[-E[l] [name]]	(errors)	Errors/source to stdout and file
[-G]	(call graph)	Profiling, instrument code for a.gprof(1)
[-H]	(help)	Print this description and stop
[-K]	(keep)	Keep the IL file after compile
[-L]	(list)	Generate a source listing to stdout
[-M [unit[.a]]	(main)	Call a.ld to create a program
[-N]	(not shared)	Set default of pragma SHARE_BODY to FALSE
[-O[level]]	(optimize)	Select level of code optimization (0-3)
[-R [library]]	(recompile)	Force updating of instantiations
[-S]	(suppress)	Apply pragma suppress
[-T]	(timings)	Print wall and CPU times, memory usage

COMPILATION SYSTEM OPTIONS

Usage: a.ld [options] unit_name [ld options|arguments]

Options:

[-o exec_file]	(output)	Name the generated program exec_file, instead of the default name, a.out.
[-m]	(map)	Print out an object map.
[-shmem params]	(shared memory)	The quoted params string contains a set of shared_package shared memory config. parameters.
[-v]	(verbose)	Print the link stream before execution.
[-w]	(warnings)	Suppress warning messages.
[-F]	(files)	List dependent files, suppressing execution.
[-H]	(help)	Print this description and stop.
[-Q]	(quiet)	Inhibit terminal status line messages.
[-U]	(units)	List dependent units, suppressing execution.
[-V]	(verify)	Print the link stream, suppressing execution.

APPENDIX C

APPENDIX F OF THE Ada STANDARD

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

```
package STANDARD is
   ..........
   type INTEGER is range -2_147_483_648 .. 2_147_483_647 ;
   type SHORT_INTEGER is range -32_768 .. 32_767 ;
   type TINY_INTEGER is range -128 .. 127 ;

   type FLOAT is digits 6 range -1.70141E+38 ..1.70141E+38 ;
   type LONG_FLOAT is digits 15 range -1.79769313486232E+308
                                   .. 1.79769313486232E+308 ;

   type DURATION is delta 2.0**(-13) range -131_072.0 .. 131_072.0 ;
   ..........
end STANDARD;
```

APPENDIX F OF THE Ada STANDARD

Appendix F of the Ada Language Reference Manual

Harris Ada v5.1 (ACVC 1.11, Host and Target = Harris NH-4400)

PROGRAM STRUCTURE AND COMPILATION

A "main" program must be a non-generic subprogram that is either a procedure or a function returning an Ada STANDARD.INTEGER (the predefined type). A "main" program cannot be a generic subprogram or an instantiation of a generic subprogram.

PRAGMAS

Implementation-Dependent Pragmas

Pragma CONTROLLED is recognized by the implementation but does not have an effect in this release.

Pragma ELABORATE is implemented as described in Appendix B of the Ada RM.

Pragma INLINE is implemented as described in section 6.3.2 and Appendix B of the Ada RM. See section 2.17 of this manual for more details.

Pragma INTERFACE is recognized by the implementation and support calls to C and FORTRAN language functions. The Ada specifications can be either functions or procedures. All parameters must have mode IN.

For C, the types of parameters and the result type for functions must be scalar, access, or the predefined type ADDRESS defined in the package SYSTEM. Record and array objects can be passed by reference using the ADDRESS attribute. The default link name is the symbolic representation of the simple name converted to lowercase. The link name of interface routines can be changed via the implementation-defined pragma external_name.

For FORTRAN, all parameters are passed by reference. The parameter types must have the type ADDRESS defined in the package SYSTEM. The result type for a FORTRAN function must be a scalar type. Care should be taken when using tasking and FORTRAN functions. Since FORTRAN is not reentrant, it is recommended that an Ada controller task be used to access FORTRAN functions. The default link name is the symbolic representation of the simple name converted to lowercase, with a leading and trailing underscore ("_") character. The link name of interface routines can be changed via the implementation-defined pragma external_name.

For FORTRAN, the implementation also detects usage of this pragma at link time see (a.ld) and includes a call to the system supplied FORTRAN initialization routine as part of the elaboration of the Ada program. Additionally, the default system FORTRAN libraries are included in the linking of the Ada program.

Pragma LIST is implemented as described in Appendix B of the Ada RM.

Pragma MEMORY_SIZE is recognized by the implementation but has no visible effect. The implementation restricts the argument to the predefined value in the package system.

Pragma OPTIMIZE is recognized by the implementation but has no effect in this release. See the -O option for ada for code optimization options, or the implementation defined pragma, OPT_LEVEL.

Pragma PACK causes the compiler to choose a non-aligned representation for elements of composite types. Application of the pragma will cause objects to be packed to the bit level.

Pragma PAGE is implemented as described in Appendix B of the Ada RM.

Pragma PRIORITY is implemented as described in Appendix B of the Ada RM. Priorities range from 0 to 9, with 9 being the most urgent.

Pragma SHARED is recognized by the implementation but has no effect.

Pragma STORAGE_UNIT is recognized by the implementation but has no visible effect. The implementation restricts the argument to the predefined value in the package system.

Pragma SUPPRESS in the single parameter form is supported and applies from the point of occurrence to the end of the innermost enclosing block. DIVISION_CHECK and OVERFLOW_CHECK for floating point types will reduce the amount of overhead associated with checking, but is not fully repressable. The double parameter form of the pragma, with a name of an object, type, or subtype is recognized, but has no effect.

Pragma SYSTEM_NAME is recognized by the implementation but has no visible effect. The implementation provides only one enumeration value for SYSTEM_NAME in the package SYSTEM.

Implementation-Defined Pragmas

Pragma EXTERNAL_NAME provides a method for specifying an alternative link name for variables, functions and procedures. The required parameters are the simple name of the object and a string constant representing the link name. An underscore is automatically prepended to the specified name, unless the first character of the name is an underscore. Note that this pragma is useful for referencing functions and procedures that have had pragma INTERFACE applied to them, in such cases where the functions or procedures have link names that do not conform to Ada identifiers. The pragma must occur after any such applications of pragma INTERFACE and within the same declarative part or package specification that contains the object.

Pragma INTERFACE_OBJECT provides an interface to objects defined externally from the Ada compilation model, or an object defined in a foreign language.

For example, a variable defined in the run-time system may be accessed via the pragma. This pragma has two required parameters, the first being the simple name of an Ada variable to be associated with the foreign object. The second parameter is a string constant that defines the link name of the object. The variable declaration must occur before the pragma and both must occur within the same declarative part or package specification.

Pragma INTERFACE_SHARED_OBJECT provides an interface to objects defined in foreign languages which exist in CX/UX shared memory segments. Specifically, this allows for the sharing of data between Ada objects and FORTRAN or C objects defined within the same process or in a separate process.

Pragma INTERFACE_SHARED_OBJECT associates an Ada variable with a CX/UX shared memory segment. It has two required parameters. The first parameter is the simple name of the Ada variable to be associated with the foreign object. The second parameter is a string constant that defines the external link name of the object as defined in the foreign language. The variable declaration must occur before the pragma and both must occur within the same declarative part or package specification.

Variables marked with the pragma must have a static size. It is recommended that an explicit length clause be specified for composite objects to ensure conformance with the size as defined by the foreign language. Additionally, record representation clauses may be used to define the layout of records to match the foreign language definitions.

The association of the shared memory segment to the Ada variable is effected at program startup time, by the HAPSE run-time system. However, specific control over the configuration of the shared memory is defined externally from the Ada compilation model and requires user intervention. The CX/UX shmdefine utility has been provided to aid the user in defining the configuration of shared memory segments. The utility produces a link-ready file and a loader command file which must be included in the link of any Ada program using pragma INTERFACE_SHARED_OBJECT. To include these files in the link process, the user should invoke the HAPSE prelinker, a.ld , adding the names of these files to the end of the command line. See section 11.2.2 for an example application of the pragma. Refer to the CX/UX User's Reference Manual for details on the shmdefine utility.

Pragma LINK_OPTION allows a command to the linker ld(1) to be specified in an Ada compilation unit. The pragma has one required parameter, a string within quotes containing the command to be passed to ld. The string specified will be passed directly to ld by the a.ld tool. For example, if a compilation unit references the CX/UX curses library, the pragma:

```
pragma LINK_OPTION("-lcurses");
```

can be used in the compilation unit to link the CX/UX curses library with the resulting Ada program. An example usage of this pragma may be found in the HAPSE harrislib library's MATH package. The body of this package contains a LINK_OPTION pragma that causes all programs that "with" package

MATH to be automatically linked with the CX/UX math library.

Pragma SHARED_PACKAGE provides for the sharing and communication of library level packages. All variables declared in a package marked pragma SHARED_PACKAGE (henceforth referred to as a shared package) are allocated in shared memory that is created and maintained by the implementation. The pragma can only be applied to library level package specifications. Each package specification nested in a shared package will also be shared and all objects declared in the nested packages reside in the same shared memory as the outer package.

The implementation restricts the kinds of objects that can be declared in a shared package. Unconstrained or dynamically sized objects cannot be declared in a shared package; access type objects cannot be declared in a shared package; and explicit initialization of objects cannot occur in a shared package. If any of these restrictions are violated, a warning message is issued and the package is not shared. These restrictions apply to nested packages as well. Note that if a nested package violates one of the above restrictions, it prevents the sharing of all enclosing packages as well. It is also important to note that some objects can be implicitly initialized by the compiler. Declarations of these objects will also cause a warning message to be issued, and will prevent sharing of the package. Records with gaps and arrays with gaps are examples of objects that can be implicitly initialized.

Task objects are allowed within shared packages, however, the tasks as well as the data defined within those tasks are not shared.

Pragma SHARED_PACKAGE accepts as an optional argument, "params", that, if specified, must be a string constant containing a comma separated list of CX/UX shared segment configuration parameters, as defined by the following:

> key= name, which identifies the CX/UX shared segment key to be used in subsequent shmget system calls, which are done automatically by the implementation in configuring the shared segment. name is considered to be a CX/UX filename which will be translated to a shared segment key using the CX/UX ftok(3C) service. By default, HAPSE applies "key= {absolute HAPSE library path}/.shmem/package_name to the shared package. Note that relative path names may be specified and would cause key translation to be dependent on the user's current working directory when program execution is initiated. If name is a decimal integer literal, HAPSE interprets this as the actual CX/UX key, and does not translate it using the ftok service.
>
> ipc= (IPC_CREAT, IPC_EXCL, IPC_PRIVATE), which allows the user to specify details about the initialization of the shared segment. By default, HAPSE applies ipc= (IPC_CREAT) to the shared package, thereby creating the shared segment if it did not previously exist. If any ipc parameters are given, they entirely replace the default ipc specification.
>
> SHM_RDONLY, which specifies that the segment is only available for

READ operations. HAPSE defaults shared package segments to READ/WRITE. CAUTION: Use of the 'LOCK or 'UNLOCK attributes with a SHM_RDONLY shared memory segment will raise PROGRAM_ERROR at runtime.

mode = n, where n is assumed to be a 3 digit octal number defining the access to the shared segment. By default, HAPSE applies mode=644 to the shared package, (owner read/write, group read, other read).

SHM_LOCAL, which requests that pages for the shared segment be allocated from the local memory pool (GCX only). If a program attempts to attach to a segment which has been allocated from local memory on a different CPU, then the attachment will fail. See shmget(2).

SHM_HARD, which when used in conjunction with SHM_LOCAL, specifies that pages for the shared segment MUST be allocated from the local memory pool (GCX only). If pages are not available from local memory then the signal SIGSEGV is delivered to the process. See shmget(2).

SHM_IO, which specifies that the segment will be bound to I/O memory. See shmget(2).

bind=n, where n is assumed to be an octal number. The segment will be attached to the physical memory address specified by n. Root user access is required for this operation. WARNING: If the shmbind(2) attempt fails due to EBUSY or EREGSTALE, the implementation will ignore the error and continue, assuming that another program has already bound the segment to the desired location. Shared memory segments bound to physical memory should be freed manually by the user via ipcrm(1).

no_bsem, which prohibits the use of shared package lock attributes ('LOCK and 'UNLOCK). In shared packages marked with this parameter, no binary semaphore space is initialized in the shared memory segment. Any attempt to invoke the lock attributes in a shared package marked with this parameter, will result in PROGRAM_ERROR being raised. Unlike SHM_RDONLY shared packages, "no_bsem" packages have READ/WRITE capability.

Caution: By default, every shared package that is available for READ/WRITE has a binary semaphore initialized which occupies the last 12 bytes of the segment. If a shared package is bound to a device using the bind= parameter, be aware that the contents of these 12 bytes may change during package elaboration and in the presence of 'LOCK and 'UNLOCK attribute usage. The initialization of this binary semaphore can be suppressed if SHM_RDONLY or no_bsem is applied to the shared package. In this case, references to 'LOCK or 'UNLOCK will result in PROGRAM_ERROR being raised.

A detailed explanation of the IPC and SHM flags, and access modes may be found in the CX/UX Programmer's Reference Manual. Chapter 2.

The pragma must appear within the specification of the library level package. The pragma may also be repeated in the package body to allow the user to override the shared memory configuration parameters that were associated with the pragma in the specification. Additionally, these configuration parameters, as defined above, may also be specified at link time to a.ld, via the -shmem "params" option, where "params" is defined as above with the addition that the first item in the list must be the name of a shared package. If this option is used, then it replaces all previous information that may have been provided with all pragmas for that package.

With the valid application of pragma SHARED_PACKAGE to a library level package, the following assumptions can be made about the objects declared in the package:

> The lifetime of such objects is greater than the lifetime defined by the complete execution of a single program.
>
> The lifetime of such objects is guaranteed to extend from the elaboration of the shared package by the first concurrent program until the termination of execution of the last concurrent program.

In the assumptions above, a concurrent program is defined to be any Ada program which elaborates the body of a shared package, whose span of execution, from elaboration of such a package to termination, overlaps that of another such program.

In actuality, the shared memory segments created by these programs remain even after the last concurrent program has exited. The values of objects within these segments remains valid until the segment is destroyed, or until the system is rebooted. Segments may be explicitly removed through the shared memory service shmctl, to which an interface is provided in the HAPSE package shared_memory_support. Alternatively, the user may obtain information about active shared memory segments through the CX/UX utility ipcs(3). These segments may be removed via the CX/UX utility iprm(1).

Programs that attempt to reference the contents of objects declared in shared packages that have not been implicitly or explicitly initialized are technically erroneous as defined by the RM (3.2.1(18)). This implementation, however, does not prevent such references and, in fact expects them.

The above discussion describes the intent that several Ada programs may begin, continue, and complete their execution simultaneously, with the contents of the variables in the shared packages consistent with the execution of those programs.

Since packages that contain objects that are initialized are not candidates for pragma SHARED_PACKAGE, the implementation suggests that programs be created for the sole purpose of initializing objects in the shared package.

The association of a CX/UX shared memory segment with the shared package is

effected during the elaboration of the package body. If this association should fail due to system shared memory constraints, access, or improper use of shared memory configuration parameters, one of several predefined exceptions will be raised. The exceptions are of the form:

shared_package_error.{name of package}.{service}.{code}

where .{code} is a CX/UX error code mnemonic.

For example, shared_package_error.package.shmat.EMFILE would be raised to indicate that the shared package attachment failed because it would exceed the system imposed limit on active shared segments. These exceptions are not available to the user since exceptions generated from the elaboration of library level package bodies have no enclosing scope from which to supply a handler. Refer to the CX/UX Programmer's Reference Manual for a detailed list of the error conditions for shmget(2) and shmop(2).

So that programs can define critical sections to reference and update variables within the shared packages, HAPSE has provided semaphore operations. See the description of the implementation-defined attributes P'LOCK and P'UNLOCK.

Pragma SHARE_BODY is used to indicate whether or not an instantiation is to be shared. The pragma may reference the generic unit or the instantiated unit. When it references a generic unit, it sets sharing on/off for all instantiations of the generic, unless overridden by specific SHARE_BODY pragmas for individual instantiations. When it references an instantiated unit, sharing is on/off only for that unit. The default is to share all generics that can be shared, unless the unit uses pragma INLINE.

Pragma SHARE_BODY is only allowed in the following places: immediately within a declarative part, immediately within a package specification, or after a library unit in a compilation, but before any subsequent compilation unit. The form of this pragma is

pragma SHARE_BODY (generic_name, boolean_literal)

Note that a parent instantiation is independent of any individual instantiation, therefore recompilation of a generic with different parameters has no effect on other compilations that reference it. The unit that caused compilation of a parent instantiation need not be referenced in any way by subsequent units that share the parent instantiation.

Sharing generics causes a slight execution time penalty because all type attributes must be indirectly referenced (as if an extra calling argument were added). However, it substantially reduces compilation time in most circumstances and reduces program size.

Pragma SUPPRESS_ALL give permission to the implementation to suppress all run-time checks. There are no parameters to pragma SUPPRESS_ALL. It is allowed to appear immediately within a declarative part. It's effects are equivalent to a complete list of SUPPRESS pragmas, each naming a different

check.

Pragma OPT_LEVEL controls the level of optimization performed by the compiler. This pragma takes one of the following as an argument: NONE, MINIMAL, GLOBAL, or MAXIMAL. The default is MINIMAL. NONE produces inefficient code but allows for faster compilation time. MINIMAL produces more efficient code with the compilation time slightly degraded. GLOBAL produces highly optimized code but the compilation time is significantly impacted. MAXIMAL is an extension of GLOBAL that can produce even better code but may change the meaning of the program. MAXIMAL attempts strength reduction optimizations that may raise OVERFLOW exceptions when dealing with values that approach the limits of the architecture of the machine. The pragma is allowed within any declarative part. The specified optimization level will apply to all code generated for the specifications and bodies associated with the immediately enclosing declarative part.

In general, programs should be developed and debugged using OPT_LEVEL (MINIMAL), reserving GLOBAL and MAXIMAL for a thoroughly tested product.

The following optimizations are performed at the various levels.
OPT_LEVEL NONE:
- Short circuit boolean tests
- Use of machine idioms
- Literal pooling

OPT_LEVEL MINIMAL: (in addition to those done with NONE)
- Binding of intermediate results to registers
- Determination of optimal execution order
- Simplification of algebraic expressions
- Re-association of expressions to collect constants
- Detection of unreachable instructions
- Elimination of jumps to adjacent labels
- Elimination of jumps over jumps
- Replacement of a series of simple adjacent instructions by a single faster complex instruction
- Constant folding

OPT_LEVEL GLOBAL: (in addition to those done with MINIMAL)
- Elimination of unreachable code
- Insertion of zero trip tests
- Elimination of dead code
- Constant propagation
- Variable propagation
- *Constraint propagation*
- Folding of control flow constructs with constant tests
- Elimination of local and global common sub-expressions
- Move loop invariant code out of loops
- Reordering of blocks to minimize branching
- Binding variables to registers
- Detection of uninitialized uses of variables
- Partial folding of Boolean expressions
- Direct branching to exception handlers

OPT_LEVEL MAXIMAL: (in addition to those done with GLOBAL)
- Comprehensive strength reduction
- Test replacement
- Induction variable elimination
- Elimination of dead regions
- Register reallocation and redundant move elimination
- Instruction scheduling and reordering

Pragma UNIVERSE allows the specification of the CX/UX universe for a compilation unit. The pragma has one required parameter, the literal denoting the desired universe. The pragma takes the form:

```
pragma UNIVERSE(universe_literal);
```

where the universe_literal is one of ucb or att. Details on the effects of the CX/UX universe switch can be found in the man page for universe(1).

Pragma UNIVERSE has effects at compile time, link time and execution time. At compile time, if the value specified differs from the current CX/UX universe value a warning message is printed. At link time, if the program contains units which have been compiled with conflicting values of pragma UNIVERSE a warning message is printed. Otherwise, the universe will be set as specified by the pragma for the duration of the link operation. At execution time, a call to the CX/UX setuniverse(2) service will be performed prior to the elaboration of the program's library packages if the program contains a compilation unit marked with pragma UNIVERSE and if conflicting values of pragma UNIVERSE have not been specified in any other compilation units. Pragma VOLATILE accepts a list of simple variable names, which the compiler assumes to occupy volatile storage bases. All subsequent reads and writes of these variables will result in memory references.

IMPLEMENTATION-DEPENDENT ATTRIBUTES

HAPSE has defined the following attributes for use in conjunction with the implementation-defined pragma SHARED_PACKAGE:

```
P'KEY
P'SHM_ID
P'LOCK
P'UNLOCK
```

where the prefix P denotes a package marked with pragma SHARED_PACKAGE.

The 'KEY attribute is an overloaded parameterless function which returns the key used to identify the CX/UX shared segment associated with the package. One specification of the function returns the predefined type string, and returns a value specifying the filename used in the key translation (ftok(3C)). If an integer literal key was specified in the pragma shared_package parameters, this function returns a null string. The other specification of the function returns the predefined type universal_integer, and returns a value specifying the translated integer

key. The latter form of the function will raise the predefined exception PROGRAM_ERROR if the shared package body has not yet been elaborated.

The 'SHM_ID attribute returns the shared memory identifier obtained by the implementation by the shmget(2) service call.

The 'LOCK and 'UNLOCK attributes are parameterless procedures which manipulate the "state" of a shared package. HAPSE defines all shared packages to have two states: LOCKed and UNLOCKed. Upon return from the 'LOCK procedure, the state of the package will be LOCKed. If upon invocation, 'LOCK finds the state already LOCKed, it will wait until it becomes UNLOCKed before altering the state and returning. 'UNLOCK sets the state of the package to UNLOCKed and then returns. At the point of unlocking the package, if another process waiting in the 'LOCK procedure has a more favorable CX/UX priority, the system will immediately schedule its execution.

Note that if 'LOCK is waiting, it may be interrupted by the HAPSE run-time system's time slice for tasks which may cause another task within the process to become active. Eventually, HAPSE will again transfer control to the 'LOCK procedure in the original task, and it will continue waiting or return to the task.

The state of the package is only meaningful to the 'LOCK and 'UNLOCK attribute procedures that set and query the state. A LOCK state does not prevent concurrent access to objects in the shared package. These attributes only provide indivisible operations for the setting and testing of implicit semaphores that could be used to control access to shared package objects. CAUTION: The current shared memory implementation does not allow the use of the 'LOCK and 'UNLOCK attributes with a SHM_RDONLY shared memory segment.

HAPSE provides the package, shared_memory_support. This package contains Ada type, subprogram definitions, and interfaces to aid the user in manually interfacing to the CX/UX shared memory services.

This includes:

System defines and records layouts as defined by the CX/UX C Programming Language include files, <sys/shm.h> and <sys/ipc.h>.

Interface specifications to shared memory system calls: shmbind, shmget, shmat, shmctl, shmdt.

Interface specifications to the CX/UX binary semaphore operators: binsemget, lockbinsem, unlockbinsem.

APPENDIX F OF THE Ada STANDARD

SPECIFICATION OF PACKAGE SYSTEM

```
package SYSTEM is
    type ADDRESS is private;
    type NAME is (Harris_88K);

    SYSTEM_NAME     : constant NAME := Harris_88K;

    -- System-Dependent Constraints

    STORAGE_UNIT    : constant := 8;
    MEMORY_SIZE     : constant := 3_221_225_469;

    -- System-Dependent Named Numbers

    MIN_INT         : constant := -2_147_483_648;
    MAX_INT         : constant := 2_147_483_647;
    MAX_DIGITS      : constant := 15;
    MAX_MANTISSA    : constant := 30;
    FINE_DELTA      : constant := 2.0**(-30);
    TICK            : constant := 0.01;

    -- Other System-dependent Declarations

    subtype PRIORITY is INTEGER range 0 ..  9;

    MAX_REC_SIZE    : INTEGER :=  268435455;  -- 16#0FFFFFFF#

    NO_ADDR : constant ADDRESS ;

         pragma suppress( elaboration_check );

         --
    -- The following functions provide conversions to "address"
    -- from integer values.
    --
    function LOGICAL_ADDRESS  (i : integer) return ADDRESS ;

    --
    -- The following function should ONLY be used to supply a
    -- machine address to an object's address clause statement.
    --
    -- The parameter is an integer describing the physical machine
    -- address of the object.
    --
    -- Optimization of subsequent references to the object with the
    -- address clause will result if the parameter to this function
    -- is an integer literal.
    --
```

```
    function MACHINE_ADDRESS (i : INTEGER) return ADDRESS ;

    function ADDR_GT   (A, B : ADDRESS) return BOOLEAN ;
    function ADDR_LT   (A, B : ADDRESS) return BOOLEAN ;
    function ADDR_GE   (A, B : ADDRESS) return BOOLEAN ;
    function ADDR_LE   (A, B : ADDRESS) return BOOLEAN ;
    function ADDR_DIFF (A, B : ADDRESS) return INTEGER ;
    function INCR_ADDR (A : ADDRESS ; INCR : INTEGER) return ADDRESS ;
    function DECR_ADDR (A : ADDRESS ; DECR : INTEGER) return ADDRESS ;
    function ">"  (A, B : ADDRESS) return BOOLEAN renames ADDR_GT ;
    function "<"  (A, B : ADDRESS) return BOOLEAN renames ADDR_LT ;
    function ">=" (A, B : ADDRESS) return BOOLEAN renames ADDR_GE ;
    function "<=" (A, B : ADDRESS) return BOOLEAN renames ADDR_LE ;
    function "-"  (A, B : ADDRESS) return INTEGER renames ADDR_DIFF ;
    function "+"  (A : ADDRESS ; INCR : INTEGER) return ADDRESS
       renames INCR_ADDR ;
    function "-"  (A : ADDRESS ; DECR : INTEGER) return ADDRESS
       renames DECR_ADDR ;

    pragma inline (ADDR_GT) ;
    pragma inline (ADDR_LT) ;
    pragma inline (ADDR_GE) ;
    pragma inline (ADDR_LE) ;
    pragma inline (ADDR_DIFF) ;
    pragma inline (INCR_ADDR) ;
    pragma inline (DECR_ADDR) ;
    pragma inline (LOGICAL_ADDRESS) ;
    pragma built_in (MACHINE_ADDRESS) ;

private

    type ADDRESS is new INTEGER;

    NO_ADDR : constant ADDRESS := 0 ;

end SYSTEM;
```

RESTRICTIONS ON REPRESENTATION CLAUSES

Pragma PACK

Pragma PACK is fully supported. Objects and components are packed to the nearest and smallest bit boundary when pragma PACK is applied. Length Clauses The specification T'SIZE is fully supported for all scalar and composite types, except for floating point. For floating point, access, and task types, the supplied static expression must conform to an existing supported machine representation.

Type	Size
floating point	32 or 64
access	32
task	32

T'SIZE applied to a composite type will cause compression of scalar component types and the gaps between the components. T'SIZE applied to a composite type whose components are composite types does not imply compression of the inner composite objects. To achieve such compression, the implementation requires explicit application of T'SIZE or pragma PACK to the inner composite type.

Composite types which contain components that have had T'SIZE applied to them, will adhere to the specified component size, even if it causes alignment of components on non STORAGE_UNIT boundaries.

The size of a non-component object of a type whose size has been adjusted, via T'SIZE or pragma PACK, will be exactly the specified size; however, the implementation will choose an alignment for such objects that provides optimal performance.

Record Representation Clauses

The simple expression following the keywords "at mod" in an alignment clause specifies the STORAGE_UNIT alignment restrictions for the record; values of 1, 2, 4, or 8 are allowed.

The simple expression following the keyword "at" in a component clause specifies the STORAGE_UNIT (relative to the beginning of the record) at which the following range is applicable. The static range following the keyword range specifies the bit range of the component. Components may overlap word boundaries (4 STORAGE_UNITs). Components that are themselves composite types must be aligned on a STORAGE_UNIT boundary.

A component clause applied to a component that is a composite type does not imply compression of that component. For such component types, the implementation requires that T'SIZE or pragma PACK be applied, if compression beyond the default size is desired.

Address Clauses

Address clauses are only supported for variables, constants, and task entries.

For variables and constants, both logical and machine addresses are supported. A "logical addresses" refers to a virtual memory address in the execution program's address space. A "machine addresses" refers to a physical memory address.

Logical address clauses:

The function LOGICAL_ADDRESS is defined in the package SYSTEM to provide conversion from INTEGER values to ADDRESS values for logical addresses only.

Both static and variable logical addresses are supported.

The value supplied to the address clause must be a valid logical address in the user's program.

Machine addresses clauses:

When a machine address is desired, the expression supplied on the address clause MUST be an invocation of the function MACHINE_ADDRESS, found in package SYSTEM. Any other expression supplied to the address clause will cause it to be interpreted as a logical address.

Both static and variable machine addresses are supported.

The argument to MACHINE_ADDRESS must be a valid integer physical memory address.

If the argument to MACHINE_ADDRESS is an integer literal, then static address translation can occur, thereby removing any additional overhead involved in accessing the variable at runtime.

You must be "superuser" to have the ACC_SHMBIND bit set in your access vector in order to use machine address clauses.

WARNING: It is the user's responsibility to ensure that the supplied address is a valid physical memory address.

Machine addresses clauses are implemented via CX shared memory segments, which are bound to the specified physical memory address at elaboration time. These shared memory segments are removed at the end of normal execution of a program. If the program is terminated abnormally, the user is responsible for removing the shared memory segments left in the system (see ipcs(2), ipcrm(2)).

Interrupts

Interrupt entries (UNIX signals) are supported. This feature allows Ada programs to bind a UNIX signal to an interrupt entry by using a for clause with a signal number. There is not any protection against two tasks binding the same signal. The result is undefined. Interrupt entries should not have any parameters and can be called explicitly by the program. See SIGVEC(2).

The HAPSE run-time uses SIGALRM (14) to perform time slicing and delays. The result of establishing a signal handler for SIGALRM is undefined.

The following example program uses an interrupt entry that prints a message

when the process receives SIGINT.

```
with TEXT_IO, SYSTEM;
use TEXT_IO;
procedure INTR is
-- This program waits for the user to generate SIGINT (<CONTROL>C)

SIGINT_NUMBER : constant := 2;

task SIGINT_HANDLER is
  entry SIGINT;
  for SIGINT use at SYSTEM.PHYSICAL_ADDRESS(SIGINT_NUMBER);
end SIGINT_HANDLER;
task body SIGINT_HANDLER is
  begin
    accept SIGINT;
    PUT_LINE("Control-C received");
  end SIGINT_HANDLER;

begin
  null;
end INTR;
```

OTHER REPRESENTATION IMPLEMENTATION-DEPENDENCIES

The ADDRESS attribute is not supported for the following entities: static constants, packages, tasks, and entries. Application of the attribute to these entities generated a compile time warning and a value of 0 at runtime.

CONVENTIONS FOR IMPLEMENTATION-GENERATED NAMES

Implementation-generated names do not exist.

UNCHECKED PROGRAMMING

UNCHECKED_CONVERSION

The following describes the transfer of data between the source and target operands when performing unchecked conversion. When possible, the implementation may optimize the conversion operation such that no transfer of data actually occurs.

The implementation considers all objects of simple types to be "right justified" within the storage allocated, and all objects of composite types to be "left justified". If, for alignment reasons, an object is placed in storage which is larger than the object's 'SIZE value, the significant bits of an object of a simple type will be placed in the low order bits of storage, right justified, with any padding in the high order bits. Likewise, should an object of a composite type be allocated storage which is larger than the type's 'SIZE, the significant bits will be placed in the high order bits, and any padding will be placed in the low order bits.

Simple Type to Simple Type Conversions

For all access, task, and scalar types, unchecked conversion is implemented using the most efficient transfer instruction to move a 1, 2, 4, or 8 byte object to its destination, unless the type has been explicitly given a 'SIZE which is not a power of two, in which case, a bit transfer will be used.

If the sizes of the source and target differ, then the smallest size is used.

If the target has a larger size than the source, the source is moved to the low order bits of the target. If the target type is signed, then the high bit of the source is sign extended through the high bits of the target. Otherwise, the high order bits of the target are zero filled.

If the target has a smaller size than the source, the low order bits of the source are copied to the target.

Composite Type to Composite Type Conversions

All conversions logically occur by moving bits from the source to the target, starting at the highest order bit of the source and target.

If the sizes of the source and target differ, then the smallest size is used.

If the target has a larger size than the source, the source is moved to the high order bits of the target, and the low order bits of the target are zero filled.

If the target has a smaller size than the source, the high order bits of the source are copied to the target.

Simple Type to Composite Type Conversions

Conversions from simple types to composite types are implemented by moving the low order, right justified, bits of the source to the high order, left justified bits of the target.

If the sizes of the source and target differ, then the smallest size is used.

If the target has a larger size than the source, the source is moved to the high order bits of the target, and the low order bits of the target are zero filled.

If the target has a smaller size than the source, the low order bits of the source are copied to the target.

Composite Type to Simple Type Conversions

Conversions from composite types to simple types are implemented by moving the high order, left justified, bits of the source to the low order, right justified bits of the target.

If the sizes of the source and target differ, then the smallest size is used.

If the target has a larger size than the source, the source is moved to the low order bits of the target. If the target type is signed, then the high bit of the source is sign extended through the high bits of the target. Otherwise, the high order bits of the target are zero filled.

If the target has a smaller size than the source, the high order bits of the source are copied to the target.

UNCHECKED_DEALLOCATION UNCHECKED_DEALLOCATION is supported. In the current release, it has no effect on access objects which designate tasks.

IMPLEMENTATION CHARACTERISTICS OF I/O PACKAGES

Implementation-Dependent Characteristics of DIRECT I/O

Instantiations of DIRECT_IO use the value MAX_REC_SIZE as the record size (expressed in STORAGE_UNITs) when the size of ELEMENT_TYPE exceeds that value. For example, for unconstrained arrays such as a string where ELEMENT_TYPE'SIZE is very large, MAX_REC_SIZE is used instead. MAX_REC_SIZE is defined in SYSTEM as 4_000_000 storage units.

Implementation-Dependent Characteristics of SEQUENTIAL I/O

Instantiations of SEQUENTIAL_IO use the value MAX_REC_SIZE as the record size (expressed in STORAGE_UNITs) when the size of ELEMENT_TYPE exceeds that value. For example, for unconstrained arrays such as a string where ELEMENT_TYPE'SIZE is very large, MAX_REC_SIZE is used instead. MAX_REC_SIZE is defined in SYSTEM as 4_000_000 storage units.

MACHINE CODE INSERTIONS

The general definition of the package MACHINE_CODE provides an assembly language interface for the target machine including the necessary record types needed in the code statement, an enumeration type containing all of the opcode mnemonics, a set of register definitions, and a set of addressing mode functions. Also supplied (for use only in units that WITH MACHINE CODE) is the implementation defined attribute 'REF.

Machine code statements accept operands of type OPERAND, a private type that forms the basis of all machine code address formats for the target.

The general syntax for a machine code statement is

```
CODE_n'(opcode, operand {, operand});
```

In the example shown below, CODE_3 is a record 'format' whose first argument is an enumeration value of the type OPCODE followed by three operands of type OPERAND.

```
code_3' (sub, r4, r4, +1);
```

The opcode must be an enumeration literal (ie., it cannot be an object, an attribute, or a rename). An operand can only be an entity defined in MACHINE_CODE or by the 'REF attribute.

For an object, arguments to any of the functions defined in MACHINE_CODE must be static expressions, string literals, or the functions defined in MACHINE_CODE. The 'REF attribute may not be used as an argument in any of these functions.

The 'REF attribute denotes the effective address of the first of the storage units allocated to the object. For a label, it refers to the address of the machine code associated with the corresponding body or statement. The attribute is of type OPERAND defined in the package MACHINE_CODE and is allowed only within a machine code procedure. 'REF is only supported for simple objects and labels.

Registers - The full set of 32 general purpose registers for the 88K target is supported (R0 through R31).

Addressing modes - All of the 88K addressing modes are supported by the compiler. They are accessed through the following functions provided in MACHINE_CODE.

SPECIFICATION OF THE PACKAGE MACHINE_CODE

```
package machine_code is
--
   type opcode is (
      add, add_ci, add_cio, add_co, addu,
      addu_ci, addu_cio, addu_co, and_r, and_c, and_u, bb0,
      bb0_n, bb1, bb1_eq, bb1_ne, bb1_gt, bb1_le, bb1_lt, bb1_ge,
      bb1_hi, bb1_ls, bb1_lo, bb1_hs, bb1_n, bb1_n_eq, bb1_n_ne,
      bb1_n_gt, bb1_n_le, bb1_n_lt, bb1_n_ge, bb1_n_hi, bb1_n_ls,
      bb1_n_lo, bb1_n_hs, bcnd_0, bcnd_10, bcnd_11, bcnd_15, bcnd_4,
      bcnd_5, bcnd_6, bcnd_7, bcnd_8, bcnd_9, bcnd_eq0, bcnd_ge0,
      bcnd_gt0, bcnd_le0, bcnd_lt0, bcnd_ne0, bcnd_n_0, bcnd_n_10,
      bcnd_n_11, bcnd_n_15, bcnd_n_4, bcnd_n_5, bcnd_n_6, bcnd_n_7,
      bcnd_n_8, bcnd_n_9, bcnd_n_eq0, bcnd_n_ge0, bcnd_n_gt0, bcnd_n_le0,
      bcnd_n_lt0, bcnd_n_ne0, br, br_n, bsr, bsr_n,
      clr, cmp, div, divu, ext, extu, fadd_ddd,
      fadd_dds, fadd_dsd, fadd_dss, fadd_sdd, fadd_sds, fadd_ssd, fadd_sss,
      fcmp_sdd, fcmp_sds, fcmp_ssd, fcmp_sss, fdiv_ddd, fdiv_dds, fdiv_dsd,
      fdiv_dss, fdiv_sdd, fdiv_sds, fdiv_ssd, fdiv_sss, ff0, ff1, flt_ds,
      flt_ss, fmul_ddd, fmul_dds, fmul_dsd, fmul_dss, fmul_sdd, fmul_sds,
      fmul_ssd, fmul_sss, fsub_ddd, fsub_dds, fsub_dsd, fsub_dss, fsub_sdd,
      fsub_sds, fsub_ssd, fsub_sss, int_sd, int_ss, jmp, jmp_n,
```

```
      jsr, jsr_n, ld, ld_b, ld_bu, ld_d, ld_h, ld_hu, lda,
      lda_b, lda_d, lda_h, mak, mask, mask_u, mul, nint_sd,
      nint_ss, or_r, or_c, or_u, rot, set, st, st_b,
      st_d, st_h, sub, sub_ci, sub_cio, sub_co, subu, subu_ci, subu_cio,
      subu_co, tb0, tb1, tbnd, tcnd_0, tcnd_10, tcnd_11, tcnd_15,
      tcnd_4, tcnd_5, tcnd_6, tcnd_7, tcnd_8, tcnd_9, tcnd_eq0, tcnd_ge0,
      tcnd_gt0, tcnd_le0, tcnd_lt0, tcnd_ne0, trnc_sd, trnc_ss, xmem,
      xmem_bu, xor_r, xor_c, xor_u, fldcr, fstcr, fxcr, ld_usr, ld_b_usr,
      ld_bu_usr, ld_d_usr, ld_h_usr, ld_hu_usr, ldcr, rte, st_usr, st_b_usr,
      st_d_usr, st_h_usr, stcr, xcr, xmem_usr, xmem_b_usr
   ) ;

   type scale_select is (unscaled, scaled) ;

   type operand is private ;

   type operand_seq is array (positive range <>) of operand ;
   n : positive ;

   --
   --  Instruction formats.
   --

   type code_0 (op : opcode) is
      record
         null ;
      end record ;

   type code_1 (op : opcode) is
      record
         oprnd_1 : operand ;
      end record ;

   type code_2 (op : opcode) is
      record
         oprnd_1 : operand ;
         oprnd_2 : operand ;
      end record ;

   type code_3 (op : opcode) is
      record
         oprnd_1 : operand ;
         oprnd_2 : operand ;
         oprnd_3 : operand ;
      end record ;

   type code_4 (op : opcode) is
      record
         oprnd_1 : operand ;
         oprnd_2 : operand ;
         oprnd_3 : operand ;
         oprnd_4 : operand ;
```

```
   end record ;

--
--  Registers.
--

r0  : constant operand ;
r1  : constant operand ;
r2  : constant operand ;
r3  : constant operand ;
r4  : constant operand ;
r5  : constant operand ;
r6  : constant operand ;
r7  : constant operand ;
r8  : constant operand ;
r9  : constant operand ;
r10 : constant operand ;
r11 : constant operand ;
r12 : constant operand ;
r13 : constant operand ;
r14 : constant operand ;
r15 : constant operand ;
r16 : constant operand ;
r17 : constant operand ;
r18 : constant operand ;
r19 : constant operand ;
r20 : constant operand ;
r21 : constant operand ;
r22 : constant operand ;
r23 : constant operand ;
r24 : constant operand ;
r25 : constant operand ;
r26 : constant operand ;
r27 : constant operand ;
r28 : constant operand ;
r29 : constant operand ;
r30 : constant operand ;
r31 : constant operand ;

sp  : constant operand ;

--
--  Addressing modes.
--
   type five_bit_range is new integer range 0..31 ;

--
-- Assembler Notation :
--   lo16(name)
--
   function ext_lo (name : string) return operand ;
   --
```

```
      -- Description :
      --   Provides the low 16 bits of the address of the external
      --   object denoted by name.
      --

   --
   -- Assembler Notation :
   --   hi16(name)
   --
      function ext_hi (name : string) return operand ;
      --
      -- Description :
      --   Provides the high 16 bits of the address of the external
      --   object denoted by name.
      --

   --
   -- Assembler Notation :
   --   lo16(xxx)
   --
      function absol_lo (disp : integer) return operand ;
      --
      -- Description :
      --   Provides the low 16 bits of the address specified.
      --

   --
   -- Assembler Notation :
   --   hi16(xxx)
   --
      function absol_hi (disp : integer) return operand ;
      --
      -- Description :
      --   Provides the high 16 bits of the address specified.
      --


   --
   -- Assembler Notation :
   --   Rn,0
   --
      function indr (addr_reg : operand) return operand ;
      --
      -- Description :
      --   The address of the operand is in the address register specified
      --   by the register field.
      --

   --
   -- Assembler Notation :
   --   Rn,I16
   --
```

```
   function disp (reg : operand ; disp : integer) return operand ;
   --
   -- Description :
   --   The address of the operand is the 32-bit unsigned sum of the
   --   address in the address register and the zero-extended 16-bit
   --   displacement integer.
   --

--
-- Assembler Notation :
--   Rn,lo16(name)
--
   function indr_lo (addr_reg : operand ; name : string) return operand ;
   --
   -- Description :
   --   The address of the operand is in the address register specified
   --   by the register field, indexed by the low 16 bits of the
   --   displacement of the external specified by "name".  A typical
   --   sequence to load the value of and external variable might be:
   --
   --   code_3' (or_u, r20, r0, ext_hi("name")) ;
   --   code_2' (ld, r21, indr_lo(20,"name")) ;
   --

--
-- Assembler Notation :
--   Rn,Rn
--
   function index (base_reg : operand ;
                   index_reg : operand) return operand ;
   --
   -- Description :
   --   The address of the operand is the sum of the address in the address
   --   register and the 32-bit contents of the index register.
   --   Normal unsigned 32 bit address arithmetic is used.
   --

--
-- Assembler Notation :
--   Rn[Rn]
--
   function index (base_reg  : operand ;
                   index_reg : operand ;
                   scale_factor : scale_select) return operand ;

--
-- Assembler Notation :
--   W5<05>
--
function bit_field (width  : five_bit_range ;
                    offset : five_bit_range) return operand ;
```

```
   --
   -- Assembler Notation :
   --   IMM16
   --
   --   An 16-bit unsigned immediate operand
   --
   function "+" (right : integer) return operand ;

 private

   type operand is new integer ;

   r0  : constant operand :=  0 ;
   r1  : constant operand :=  1 ;
   r2  : constant operand :=  2 ;
   r3  : constant operand :=  3 ;
   r4  : constant operand :=  4 ;
   r5  : constant operand :=  5 ;
   r6  : constant operand :=  6 ;
   r7  : constant operand :=  7 ;
   r8  : constant operand :=  8 ;
   r9  : constant operand :=  9 ;
   r10 : constant operand := 10 ;
   r11 : constant operand := 11 ;
   r12 : constant operand := 12 ;
   r13 : constant operand := 13 ;
   r14 : constant operand := 14 ;
   r15 : constant operand := 15 ;
   r16 : constant operand := 16 ;
   r17 : constant operand := 17 ;
   r18 : constant operand := 18 ;
   r19 : constant operand := 19 ;
   r20 : constant operand := 20 ;
   r21 : constant operand := 21 ;
   r22 : constant operand := 22 ;
   r23 : constant operand := 23 ;
   r24 : constant operand := 24 ;
   r25 : constant operand := 25 ;
   r26 : constant operand := 26 ;
   r27 : constant operand := 27 ;
   r28 : constant operand := 28 ;
   r29 : constant operand := 29 ;
   r30 : constant operand := 30 ;
   r31 : constant operand := 31 ;

   sp  : constant operand := 31 ;

   pragma built_in (absol_lo) ;
   pragma built_in (absol_hi) ;
   pragma built_in (disp) ;
   pragma built_in (ext_lo) ;
   pragma built_in (ext_hi) ;
```

```
   pragma built_in (index) ;
   pragma built_in (indr) ;
   pragma built_in (indr_lo) ;
   pragma built_in ("+") ;
--
end machine_code ;
```