AD-A240 761

[REPORT DOCUME]NTATION PAGE

Form Approved
OPM No. 0704-0188

[...] 1 hour per response, including the time for reviewing instructions, searching existing data sources gathering and maintaining the data [...] this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington [...] 5 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Information and Regulatory Affairs, Office of Management and Budget, Washington, DC 20503.

1. AGENCY USE ONLY *(Leave Blank)*

2. REPORT DATE

3. REPORT TYPE AND DATES COVERED
Final: 27 Mar 1991 to 01Jun 1993

4. TITLE AND SUBTITLE
KRUPP ATLAS ELEKTRONIK GMBH, KAE Ada Compiler VVME 1.82, VAX 6000-410 VMS 5.2 (Host) to KAE MPR 2300 MOS2300 2.1 (Target), 910324I1.11136

5. FUNDING NUMBERS

6. AUTHOR(S)
IABG-AVF
Ottobrunn, Federal Republic of Germany

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
IABG-AVF, Industrieanlagen-Betriebsgeselschaft
Dept. SZT/ Einsteinstrasse 20
D-8012 Ottobrunn
FEDERAL REPUBLIC OF GERMANY

8. PERFORMING ORGANIZATION REPORT NUMBER
IABG-VSR 072

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)*

KRUPP ATLAS ELETRONIK GMBH, KAE Ada Compiler VVME 1.82, Ottobrunn, Germany, VAX 6000-410 under VMS, Version 5.2 (Host) to KAE MPR 2300 under MOS 2300, Version 2.1 (Target), ACVC 1.11.

DTIC
ELECTE
SEP 19 1991
S D
D

91-11050

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

15. NUMBER OF PAGES

16. PRICE CODE

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

NSN 7540-01-280-550

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

9 [...] 7

AVF Control Number: IABG-VSR 072
27 March, 1991.

Ada COMPILER
VALIDATION SUMMARY REPORT:
Certificate Number: #910324I1.11136
KRUPP ATLAS ELEKTRONIK GMBH
KAE Ada Compiler VVME 1.82
VAX 6000-410 => KAE MPR 2300
VMS 5.2 MOS2300 2.1

== based on TEMPLATE Version 90-08-15 ==

Accesion For
NTIS CRA&I ✓
DTIC TAB
Unannounced
Justification
By
Distribution /
Availability Codes
Dist | Avail and/or Special
A-1

Prepared By:
IABG mbH, Abt. ITE
Einsteinstr. 20
W-8012 Ottobrunn
Germany

Certificate Information

The following Ada implementation was tested and determined to pass ACVC 1.11. Testing was completed on 24 March, 1991.

Compiler Name and Version: **KAE Ada Compiler VVME 1.82**

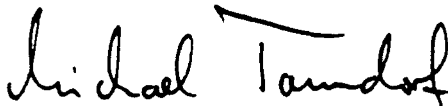
Host Computer System: **VAX 6000-410 under VMS, Version 5.2**

Target Computer System: **KAE MPR 2300 under MOS2300, Version 2.1**

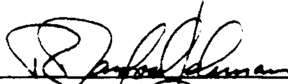
See Section 3.1 for any additional information about the testing environment.

As a result of this validation effort, Validation Certificate 910324I1.11136 is awarded to Krupp Atlas Elektronik GMBH. This certificate expires on 1 March, 1993.

This report has been reviewed and is approved.



IABG, Abt. ITE
Michael Tonndorf
Einsteinstr. 20
W-8012 Ottobrunn
Germany



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

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

DECLARATION OF CONFORMANCE

The following declaration of conformance was supplied by the customer.

Declaration of Conformance

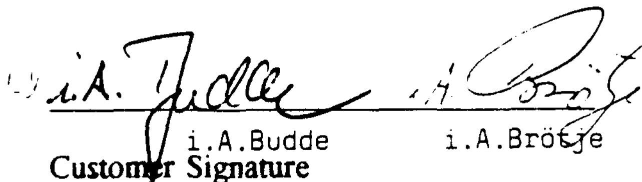
Customer:	KRUPP ATLAS ELEKTRONIK GmbH
Certificate Awardee:	KRUPP ATLAS ELEKTRONIK GmbH
Ada Validation Facility:	IABG mbH
ACVC Version:	1.11

Ada Implementation:

Ada Compiler Name and Version:	KRUPP ATLAS ELEKTRONIK Ada Compiler VVME 1.82
Host Computer System:	VAX 6000-410 under VMS, Version 5.2
Target Computer System:	KRUPP ATLAS ELEKTRONIK GmbH MPR 2300 under MOS2300, Version 2.1

Declaration:

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



i.A.Budde i.A.Brötje

Customer Signature

22.03.1991

Date

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 [Pro89] 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 [Pro89]. 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.

[Pro89] Ada Compiler Validation Procedures, Version 2.0, Ada Joint Program Office, May 1989.

[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 Validation Facility (AVF) The part of the certification body which carries out the procedures required to establish the compliance of an Ada implementation.

Ada Validation Organization (AVO) The part of the certification body that provides technical guidance for operations of the Ada certification system.

Compliance of an Ada Implementation The ability of the implementation to pass an ACVC version.

Computer System A functional unit, consisting of one or more computers and associated software, that uses common storage for all or part of a program and also for all or part of the data necessary for the execution of the program; executes user-written or user-designated programs; performs user-designated data manipulation, including arithmetic operations and logic operations; and that can execute programs that modify themselves during execution. A computer system may be a stand-alone unit or may consist of several inter-connected units.

Conformity Fulfillment by a product, process or service of all requirements specified.

Customer	An individual or corporate entity who enters into an agreement with an AVF which specifies the terms and conditions for AVF services (of any kind) to be performed.
Declaration of Conformance	A formal statement from a customer assuring that conformity is realized or attainable on the Ada implementation for which validation status is realized.
Host Computer System	A computer system where Ada source programs are transformed into executable form.
Inapplicable test	A test that contains one or more test objectives found to be irrelevant for the given Ada implementation.
Operating System	Software that controls the execution of programs and that provides services such as resource allocation, scheduling, input/output control, and data management. Usually, operating systems are predominantly software, but partial or complete hardware implementations are possible.
Target Computer System	A computer system where the executable form of Ada programs are executed.
Validated Ada Compiler	The compiler of a validated Ada implementation.
Validated Ada Implementation	An Ada implementation that has been validated successfully either by AVF testing or by registration [Pro89].
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 February 25, 1991.

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

2.2 INAPPLICABLE TESTS

A test is inapplicable if it contains test objectives which are irrelevant for a given Ada implementation. Reasons for a test's inapplicability may be supported by documents issued by ISO and the AJPO known as Approved 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 Approved Ada Commentaries are included as appropriate.

The following 159 tests have floating-point type declarations requiring more digits than SYSTEM.MAX_DIGITS:

C241130..Y (11 tests) (*)	C357050..Y (11 tests)
C357060..Y (11 tests)	C357070..Y (11 tests)
C357080..Y (11 tests)	C358020..Z (12 tests)
C452410..Y (11 tests)	C453210..Y (11 tests)
C454210..Y (11 tests)	C455210..Z (12 tests)
C455240..Z (12 tests)	C456210..Z (12 tests)
C456410..Y (11 tests)	C460120..Z (12 tests)

(*) C24113W..Y (3 tests) contain lines of length greater than 255 characters which are not supported by this implementation.

The following 20 tests check for the predefined type LONG_INTEGER:

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

C35404D, C45231D, B86001X, C86006E, and CD7101G check for a predefined integer type with a name other than INTEGER, LONG_INTEGER, or SHORT_INTEGER.

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

C41401A checks that CONSTRAINT_ERROR is raised upon the evaluation of various attribute prefixes; this implementation derives the attribute values from the subtype of the prefix at compilation time, and thus does not evaluate the prefix or raise the exception.

C45531M..P (4 tests) and C45532M..P (4 tests) check fixed-point operations for types that require a SYSTEM.MAX_MANTISSA of 48 or greater.

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

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

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

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

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

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

BD8001A, BD8003A, BD8004A..B (2 tests), and AD8011A use machine code insertions.

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

The following 19 tests check operations on sequential, direct, and text files when multiple internal files are associated with the same external file; USE_ERROR is raised when this association is attempted.

CE2107A..H	CE2107L	CD2110B	CE2110D	CE2111D
CE2111H	CE3111A..B	CE3111D..E	CE3114B	CE3115A

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.

EE2401D contains instatiations of package DIRECT_IO with unconstrained array types. This implementation raises USE_ERROR upon creation of such a file.

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.

CE3102C expects the target system to allow external files to be created using a filename with no filename extension (suffix). For this implementation all files residing on the target operating system must have filenames with a filename extension of upto three characters.

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 21 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	B24009A	B29001A	B38003A	B38009A	B38009B
B91001H	BC2001D	BC2001E	BC3204B	BC3205B	BC3205D

C34007P and C34007S were graded passed by Evaluation Modification as directed by the AVO. These tests include a check that the evaluation of the selector "all" raises CONSTRAINT_ERROR when the value of the object is null. This implementation determines the result of the equality tests at lines 207 and 223, respectively, based on the subtype of the object; thus, the selector is not evaluated and no exception is raised, as allowed by LRM 11.6(7). The tests were graded passed given that their only output from Report.Failed was the message "NO EXCEPTION FOR NULL.ALL - 2".

C41401A was graded inapplicable by Evaluation Modification as directed by the AVO. This test checks that the evaluation of attribute prefixes that denote variables of an access type raises CONSTRAINT_ERROR when the value of the variable is null and the attribute is appropriate for an array or task type. This implementation-language derives the array attribute values from the subtype; thus, the prefix is not evaluated and no exception is raised, as allowed by LRM 11.6(7), for the checks at lines 77, 87, 97, 108, 121, 131, 141, 152, 165 & 175.

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.

BC3204C..D and BC3205C..D (4 tests) were graded passed by Evaluation Modification as directed by the AVO. These tests are expected to produce compilation errors, but this implementation compiles the units without error; all errors are detected at link time. This behavior is allowed by AI-00256, as the units are illegal only with respect to units that they do not depend on.

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 sales and technical information about this Ada implementation system, see:

Dipl.-Math. Dieter Weigel
Software Engineering
Krupp Atlas Elektronik Gmbh
Sebaldsbrücker Heerstr. 235
D 2800 Bremen 44
Germany
Tel. +421 457 3058

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 [Pro89].

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	3821	
b) Total Number of Withdrawn Tests	93	
c) Processed Inapplicable Tests	97	
d) Non-Processed I/O Tests	0	
e) Non-Processed Floating-Point Precision Tests	159	
f) Total Number of Inapplicable Tests	256	(c+d+e)
g) Total Number of Tests for ACVC 1.11	4170	(a+b+f)

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 256 tests were inapplicable to this implementation. All inapplicable tests were processed during validation testing except for 159 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 Reel containing the customized test suite (see section 1.3) was taken on-site by the validation team for processing. The contents of the 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 tests were compiled and linked on the host computer system, as appropriate. The executable images were transferred to the target computer system via an Ethernet connection 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:

Compiler options :

/LIST tells the compiler to produce full listings in the specified directory, used for all class B tests, all inapplicable executable tests and all class E tests.

/LOG tells the Compiler to write additional me··ages onto the specified file.

/NOCOPY_SOURCE tells the compiler not to keep a copy of the source file in the program library.

Linker options :

/EXECUTABLE together with a parameter specifies the name of the executable program module to be produced.

/LOG tells the implicitly invoked Completer to write additional messages onto the specified file.

/NODEBUG tells the linker not to include debug information in the executable image file.

Test output, compiler and linker listings, and job logs were captured on a Magnetic Tape Reel and archived at the AVF. The listings examined on-site : 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	255 -- 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 lists all of the other macro parameters and their respective values.

Macro Parameter	Macro Value
$ACC_SIZE	32
$ALIGNMENT	4
$COUNT_LAST	2147483647
$DEFAULT_MEM_SIZE	2147483648
$DEFAULT_STOR_UNIT	8
$DEFAULT_SYS_NAME	MOTOROLA_68030_KAE
$DELTA_DOC	2#1.0#E-31
$ENTRY_ADDRESS	SYSTEM.CONVERT_ADDRESS("16#40#")
$ENTRY_ADDRESS1	SYSTEM.CONVERT_ADDRESS("16#80#")
$ENTRY_ADDRESS2	SYSTEM.CONVERT_ADDRESS("16#100#")
$FIELD_LAST	512
$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	0.0
$GREATER_THAN_DURATION_BASE_LAST	16_777_217.0
$GREATER_THAN_FLOAT_BASE_LAST	1.80E+308
$GREATER_THAN_FLOAT_SAFE_LARGE	2.251E+307
$GREATER_THAN_SHORT_FLOAT_SAFE_LARGE	4.26E+37

$HIGH_PRIORITY	56
$ILLEGAL_EXTERNAL_FILE_NAME1	/NODIRECTORY/FILENAME
$ILLEGAL_EXTERNAL_FILE_NAME2	FILENAME.*
$INAPPROPRIATE_LINE_LENGTH	-1
$INAPPROPRIATE_PAGE_LENGTH	-1
$INCLUDE_PRAGMA1	PRAGMA INCLUDE ("A28006D1.TST")
$INCLUDE_PRAGMA2	PRAGMA INCLUDE ("B28006F1.TST")
$INTEGER_FIRST	-2147483648
$INTEGER_LAST	2147483647
$INTEGER_LAST_PLUS_1	2147483648
$INTERFACE_LANGUAGE	META
$LESS_THAN_DURATION	-0.0
$LESS_THAN_DURATION_BASE_FIRST	-16_777_217.0
$LINE_TERMINATOR	' '
$LOW_PRIORITY	0
$MACHINE_CODE_STATEMENT	NULL
$MACHINE_CODE_TYPE	NO_SUCH_TYPE
$MANTISSA_DOC	31
$MAX_DIGITS	18
$MAX_INT	2147483647
$MAX_INT_PLUS_1	2147483648
$MIN_INT	-2147483648
$NAME	NO_SUCH_TYPE_AVAILABLE
$NAME_LIST	MOTOROLA_68030_KAE

$NAME_SPECIFICATION1	RAM0:/TEST/X2120A.TST-1.1
$NAME_SPECIFICATION2	RAM0:/TEST/X2120B.TST-1.1
$NAME_SPECIFICATION3	RAM0:/TEST/X3119A.TST-1.1
$NEG_BASED_INT	16#FFFFFFFE#
$NEW_MEM_SIZE	2147483647
$NEW_STOR_UNIT	8
$NEW_SYS_NAME	MOTOROLA_68030_KAE
$PAGE_TERMINATOR	' '
$RECORD_DEFINITION	NEW INTEGER
$RECORD_NAME	NO_SUCH_MACHINE_CODE_TYPE
$TASK_SIZE	32
$TASK_STORAGE_SIZE	16384
$TICK	0.01
$VARIABLE_ADDRESS	GET_VARIABLE_ADDRESS
$VARIABLE_ADDRESS1	GET_VARIABLE_ADDRESS1
$VARIABLE_ADDRESS2	GET_VARIABLE_ADDRESS2
$YOUR_PRAGMA	SQUEEZE

APPENDIX B

COMPILATION AND LINKER 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.

4. Compiling

After a program library has been created, one or more compilation units can be compiled in the context of this library. The compilation units can be placed on different source files or they can all be on the same file. One unit, a parameterless procedure, acts as the main program. If all units needed by the main program and the main program itself have been compiled successfully, they can be linked. The resulting code can then be executed on the target.

§4.1 and Chapter 5 describe in detail how to call the Compiler and the Linker. In §4.2 the Completer, which is called to generate code for instances of generic units, is described.
Chapter 6 explains the information which is given if the execution of a program is abandoned due to an unhandled exception.
The information the Compiler produces and outputs in the Compiler listing is explained in §4.4.
Finally, the log of a sample session is given in Chapter 7.

4.1 Compiling Ada Units

To start the KRUPP ATLAS ELEKTRONIK Ada Compiler, use the KAS COMPILE command.

KAS COMPILE — Command Description

Format

$ KAS COMPILE file-spec[,...]

Command Qualifiers	Defaults
/[NO]ANALYZE_DEPENDENCY	/NOANALYZE_DEPENDENCY
/LIBRARY=directory-spec	/LIBRARY=[.ADALIB]
/[NO]LOG[=file-spec]	/NOLOG
/[NO]RECOMPILE	/NORECOMPILE

Positional Qualifiers	Defaults
/[NO]CHECK	/CHECK
/[NO]COPY_SOURCE	/COPY_SOURCE
/[NO]INLINE	/INLINE
/[NO]LIST[=file-spec]	/NOLIST
/[NO]MACHINE_CODE	/NOMACHINE_CODE
/[NO]OPTIMIZE	/OPTIMIZE

Command Parameters

`file-spec`

Specifies the file(s) to be compiled. The default directory is []. The default file type is ADA. The maximum length of lines in file-spec is 255. The maximum number of source lines in file-spec is 65534. Wild cards are allowed.
Note: If you specify a wild card the order of the compilation is alphabetical, which is not always successful. Thus wild cards should be used together with `/ANALYZE_DEPENDENCY`. With this qualifier the sources can be processed in any order.

Description

The source file may contain a sequence of compilation units (cf. LRM(§10.1). All compilation units in the source file are compiled individually. When a compilation unit is compiled successfully, the program library is updated and the Compiler continues with the compilation of the next unit on the source file. If the compilation unit contained errors, they are reported (see §4.4). In this case, no update operation is performed on the program library and all subsequent compilation units in the compilation are only analyzed without generating code.

The Compiler delivers the status code WARNING on termination (cf. *VAX/VMS, DCL Dictionary*, command EXIT) if one of the compilation units contained errors. A message corresponding to this code has not been defined; hence %NONAME-W-NOMSG is printed upon notification of a batch job terminated with this status.

/ANALYZE_DEPENDENCY
/NOANALYZE_DEPENDENCY (D)

Specifies that the Compiler only performs syntactical analysis and the analysis of the dependencies on other units. The units in file-spec are entered into the library if they are syntactically corr
The actual compilation is done later with the KAS AUTOCOMPILE command.

Note: An already existing unit with the same name as the new one is replaced and all dependent units become obsolete, unless the source file of both are identical. In this case the library is *not* updated because the dependencies are already known.

By default, the normal, full compilation is done.

/LIBRARY=dir-spec

Specifies the program library the command works on. The KAS COMPILE command needs write access to the library. The default is [.ADALIB].

/LOG[=file-spec]
/NOLOG (D)

Controls whether the Compiler writes additional messages onto the specified file. The default file name is SYS$OUTPUT. The default file type is LOG.

By default, no additional messages are written.

/RECOMPILE
/NORECOMPILE (D)

Indicates that a recompilation of a previously analyzed source is to be performed. This qualifier should not be used unless the command was produced by the KRUPP ATLAS ELEKTRONIK Ada Recompiler. See the KAS RECOMPILE command.

Positional Qualifiers

/CHECK (D)
/NOCHECK

Controls whether all run-time checks are suppressed. If you specify `/NOCHECK` this is equivalent to the use of PRAGMA suppress for all kinds of checks.

By default, no run-time checks are suppressed, except in cases where PRAGMA suppress_all appears in the source.

/COPY_SOURCE (D)
/NOCOPY_SOURCE

Controls whether a copy of the source file is kept in the library. The copy in the program library is used for later access by the Debugger or tools like the Recompiler. The name of the copy is generated by the Compiler and need normally not be known by the user. The Recompiler and the Debugger know this name. You can use the KAS DIRECTORY/FULL command to see the file name of the copy. If a specified file contains several compilation units a copy containing only the source text of one compilation unit is stored in the library for each compilation unit. Thus the Recompiler can recompile a single unit.

If `/NOCOPY_SOURCE` is specified, the Compiler only stores the name of the source file in the program library. In this case the Recompiler and the Debugger are able to use the original file if still exists.

`/COPY_SOURCE` cannot be specified together with `/ANALYZE_DEPENDENCY`

/INLINE (D)
/NOINLINE

Controls whether inline expansion is performed as requested by PRAGMA inline. If you specify `/NOINLINE` these pragmas are ignored.

By default, inline expansion is performed.

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

Controls whether a listing file is created. One listing file is created for each source file compiled. If `/LIST` is placed as a command qualifier a listing file is created for all sources. If `/LIST` is placed as a parameter qualifier a listing file is created only for the corresponding source file.

The default directory for listing files is the current default directory. The default file name is the name of the source file being compiled unless /RECOMPILE is specified. In this case the name of the original source file, which is stored in the library, is taken as default. The default file type is LIS. No wildcard characters are allowed in the file specification.

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

/MACHINE_CODE
/NOMACHINE_CODE (D)

Controls whether machine code is appended at the listing file. /MACHINE_CODE has no effect if /NOLIST or /ANALYZE_DEPENDENCY is specified.

By default, no machine code is appended at the listing file.

/OPTIMIZE (D)
/NOOPTIMIZE

Controls whether full optimization is applied in generating code. There is no way to specify that only certain optimizations are to be performed.

By default, full optimization is done.

End of Command Descript.

4.2 Completing Generic Instances

Since the Compiler does not generate code for instances of generic bodies, the Completer must be used to complete such units before a program using the instances can be executed. The Completer must also be used to complete packages in the program which do not require a body. This is done implicitly when the Linker is called.
It is also possible to call the Completer explicitly with the KAS COMPLETE command.

KAS COMPLETE Command Description

Format

$ KAS COMPLETE unit[,...]

Command Qualifiers	Defaults
/[NO]CHECK	/CHECK
/[NO]INLINE	/INLINE
/LIBRARY=directory-spec	/LIBRARY=[.ADALIB]
/[NO]LIST[=file-spec]	/NOLIST
/[NO]LOG[=file-spec]	/NOLOG
/[NO]MACHINE_CODE	/NOMACHINE_CODE
/[NO]OPTIMIZE	/OPTIMIZE

Command Parameters

`unit`

specifies the unit(s) whose execution closure is to be completed.

Description

The KAS COMPLETE command invokes the KRUPP ATLAS ELEKTRONIK Ada Completer. The Completer generates code for all instantiations of generic units in the execution closure of the specified unit(s). It also generates code for packages without bodies (if necessary).

By default, the Completer is invoked implicitly by the KAS LINK command. In normal cases there is no need to invoke it explicitly.

Command Qualifiers

/CHECK (D)
/NOCHECK

Controls whether all run-time checks are suppressed. If you specify `/NOCHECK` this is equivalent

to the use of PRAGMA suppress for all kinds of checks.

By default, no run-time checks are suppressed, except in cases where PRAGMA suppress_all appears in the source.

/INLINE (D)
/NOINLINE

Controls whether inline expansion is performed as requested by PRAGMA inline. If /NOINLINE is specified these pragmas are ignored.

By default, inline expansion is performed.

/LIBRARY=dir-spec

Specifies the program library the command works on. The KAS COMPLETE command needs write access to the library.

The default library is [.ADALIB].

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

Controls whether a listing file is created.

The default directory for listing files is the current default directory. The default file nam COMPLETE. The default file type is LIS. No wildcard characters are allowed in the file specification.

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

/LOG[=file-spec]
/NOLOG (D)

Controls whether the KAS COMPLETE command writes additional messages onto the specified file. The default file name is SYS$OUTPUT.

The default file type is LOG.

By default, no additional messages are written.

/MACHINE_CODE
/NOMACHINE_CODE (D)

Controls whether a machine code listing is appended to the listing file. /MACHINE_CODE has no effect if /NOLIST is specified.

By default, no machine code listing is appended to the listing file.

/OPTIMIZE (D)

/NOOPTIMIZE

Controls whether full optimization is applied in generating code. There is no way to specify that only certain optimizations are to be performed.

By default, full optimization is done.

End of Command Description

---

5. Linking

An Ada program is a collection of units used by a main program which controls the execution. The main program must be a parameterless library procedure; any parameterless library procedure within a program library can be used as a main program.

To link a program, call the KAS LINK command.

KAS LINK Command Description

Format

$ KAS LINK unit

Command Qualifiers	Defaults
/[NO]CHECK	/CHECK
/[NO]COMPLETE	/COMPLETE
/[NO]DEBUG	/DEBUG
/EXECUTABLE=file-spec	see text
/EXTERNAL[=(file-spec,..)]	/EXTERNAL=""
/[NO]INLINE	/INLINE
/LIBRARY=directory-spec	/LIBRARY=[.ADALIB]
/[NO]LIST[=file-spec]	/NOLIST
/[NO]LOG[=file-spec]	/NOLOG
/[NO]MACHINE_CODE	/NOMACHINE_CODE
/[NO]MAP[=file-spec]	/NOMAP
/[NO]OPTIMIZE	/OPTIMIZE
/[NO]SELECTIVE	/SELECTIVE

Command Parameters

Specifies the library unit which is the main program. This must be a parameterless library procedure.

Description

The KAS LINK command invokes the KRUPP ATLAS ELEKTRONIK Ada Linker.

The Linker generates an executable image, which can be transfered to the target and executed there.

The default file name of the executable image is the file name of the source file which contained the specified library unit. The default file type is KAX (Krupp Atlas Elektronik Ada Executable).

The default directory is [].

Command Qualifiers

/CHECK (D)
/NOCHECK

This qualifier is passed to the implicitly invoked Completer. See the same qualifier with the KAS COMPLETE command.

/COMPLETE (D)
/NOCOMPLETE

Controls whether the Completer of the KRUPP ATLAS ELEKTRONIK Ada system is invoked before the linking is performed. Only specify /NOCOMPLETE if you are sure that there are no instantiations or implicit package bodies to be compiled, e.g. if you repeat the KAS LINK command with different linker options.

/DEBUG (D)
/NODEBUG

Controls whether debug information for the KRUPP ATLAS ELEKTRONIK Ada Debugger is to be generated and included in the excutable image . If the program is to run under the control of the Debugger it must be linked with the /DEBUG qualifier.

By default, debug information is included in the program image file.

/EXECUTABLE=file-spec

Specifies the name of the executable image.

The default file name of the executable image is the file name of the source file which contained the specified library unit.

The default file type is KAX.

The default directory is [].

/EXTERNAL[=(file-spec,..)]

Specifies a list of external object files in ROF format (Relocatable Object File Format); all of type OBJ.

The original files are be MOS2300 files in L-format, which are converted to ROF format files.

/INLINE (D)
/NOINLINE

This qualifier is passed to the implicitly invoked completer. See the same qualifier with the KAS COMPLETE command.

/LIBRARY=directory-spec

Specifies the program library the command works on. The KAS link command needs write access to the library unless /NOCOMPLETE is specified. If /NOCOMPLETE is specified the KAS LINK command needs only read access.

The default library is [.ADALIB].

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

This qualifier is passed to the implicitly invoked Completer. See the same qualifier with the KAS COMPLETE command.

By default the Completer does not create a listing file.

/LOG[=file-spec]

/NOLOG (D)

Controls whether the implicitly invoked Completer writes additional messages onto the specified file. The default file name is SYS$OUTPUT.

The default file type is LOG.

By default, no additional messages are written.

/MACHINE_CODE
/NOMACHINE_CODE (D)

This qualifier is passed to the implicitly invoked Completer. See the same qualifier with the KAS COMPLETE command. If /LIST and /MACHINE_CODE is specified, the Linker of the KRUPP ATLAS ELEKTRONIK Ada system generates a listing with the machine code of the program starter in the file []LINK.LIS. The program starter is a routine which contains the calls of the necessary elaboration routines and a call for the Ada subprogram which is the main program.

By default, no machine code is generated.

/MAP[=file-spec]
/NOMAP (D)

Specifies whether the map listing of the Linker and the table of symbols which are used for linking the Ada units are to be produced in the specified file. The default directory is the directory in which the program image file is located. The default file name is the name of the program image file.

The default file type is MAP.

/OPTIMIZE (D)
/NOOPTIMIZE

This qualifier is passed to the implicitly invoked Completer. See the same qualifier with the KAS COMPLETE command.

/SELECTIVE(D)
/NOSELECTIVE

Controls selective linking. Selective linking means that only the code of those subprograms which can actually be called is included in the executable image.

`/NOSELECTIVE` means that the code of all subprograms of all packages in the execution closure of the main procedure is linked into the executable image.

Note: The code of the runtime system and of the predefined units is always linked selectively.

End of Command Description

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 short_float IS DIGITS 6 RANGE
     -16#0.FFFF_F8#E32 .. 16#0.FFFF_F8#E32;

TYPE float IS DIGITS 15 RANGE
     -16#0.FFFF_FFFF_FFFF_F8#E256 .. 16#0.FFFF_FFFF_FFFF_F8#E256;

TYPE long_float IS DIGITS 18 RANGE
     -16#0.FFFF_FFFF_FFFF_FFFC#E4096 .. 16#0.FFFF_FFFF_FFFF_FFFC#E4096;

TYPE duration IS DELTA 2#1.0#E-7 RANGE
     -16_777_216.0 .. 16_777_216.0;

...

end STANDARD;
```

15. Appendix F

This chapter, together with the Chapters 16 and 17, is the Appendix F required in the LRM, in which all implementation-dependent characteristics of an Ada implementation are described.

15.1 Implementation-Dependent Pragmas

The form, allowed places, and effect of every implementation-dependent pragma is stated in this section.

15.1.1 Predefined Language Pragmas

The form and allowed places of the following pragmas are defined by the language; their effect is (at least partly) implementation-dependent and stated here.

CONTROLLED

has no effect.

ELABORATE

is fully implemented. The KRUPP ATLAS ELEKTRONIK Ada System assumes a PRAGMA elaborate, i.e. stores a unit in the library as if a PRAGMA elaborate for a unit u was given, if the compiled unit contains an instantiation of u (or a generic program unit in u) and if it is clear that u *must* have been elaborated before the compiled unit. In this case an appropriate information message is given. By this means it is avoided that an elaboration order is chosen which would lead to a PROGRAM_ERROR when elaborating the instantiation.

INLINE

Inline expansion of subprograms is supported with the following restrictions: the subprogram must not contain declarations of other subprograms, tasks, generic units or body stubs. If the subprogram is called recursively only the outer call of this subprogram will be expanded.

INTERFACE

is supported for ASSEMBLER and META.

PRAGMA interface (assembler,...) provides an interface with the internal calling conventions of the KRUPP ATLAS ELEKTRONIK Ada System. See §15.1.3 for further description.

PRAGMA interface (META, ...) provides an interface with the operating system language. See §15.1.4 for further description.

PRAGMA interface should always be used in connection with the `PRAGMA external_name` (see §15.1.2), otherwise the Compiler will generate an internal name that leads to an unsolved

reference during linking. These generated names are prefixed with an underline; therefore the user should not use names beginning with an underline.

LIST

is fully implemented. Note that a listing is only generated when the /LIST qualifier is specified with the KAS COMPILE (or KAS COMPLETE or KAS LINK) command.

MEMORY_SIZE

has no effect.

OPTIMIZE

has no effect; but see also the /OPTIMIZER qualifier with the KAS COMPILE command, §4.1

PACK

see §16.1.

PAGE

is fully implemented. Note that form feed characters in the source do not cause a new page in the listing. They are - as well the other format effectors (horizontal tabulation, vertical tabulation, carriage return, and line feed) - replaced by a ~ character in the listing.

PRIORITY

There are two implementation-defined aspects of this pragma: First, the range of the subtype `priority`, and second, the effect on scheduling (Chapter 14) of not giving this pragma for a task or main program. The range of subtype `priority` is 0 .. 56, as declared in the predefined library package `system` (see §15.3); and the effect on scheduling of leaving the priority of a task or main program undefined by not giving PRAGMA `priority` for it is the same as if the PRAGMA `priority` 0 had been given (i.e. the task has the lowest priority).

SHARED

is fully supported.

STORAGE_UNIT

has no effect.

SUPPRESS

has no effect, but see §15.1.2 for the implementation-defined PRAGMA `suppress_all`.

SYSTEM_NAME

has no effect.

15.1.2 Implementation-Defined Pragmas

BYTE_PACK

see §16.1.

EXTERNAL_NAME (<string>, <ada_name>)

<ada_name> specifies the name of a subprogram or of an object declared in a library package, <string> must be a string literal. It defines the external name of the specified item. The Compiler uses a symbol with this name in the `call` instruction for the subprogram. The subprogram declaration of <ada_name> must precede this pragma. If several subprograms with the same name satisfy this requirement the pragma refers to that subprogram which is declared last.

Upper and lower cases are distinguished within <string>, i.e. <string> must be given exactly as it is to be used by external routines. This pragma will be used in connection with the pragma `interface (...)` (see §15.1.1).)

RESIDENT (<ada_name>)

this pragma causes the value of the object to be held in memory and prevents assignments of a value to the object <ada_name> from being eliminated by the optimizer (see §4.1) of the KRUPP ATLAS ELEKTRONIK Ada Compiler. The following code sequence demonstrates the intended usage of the pragma:

```
...
x : integer;
a : SYSTEM.address;
...
BEGIN
        x := 5;
        a := x'ADDRESS;
        do_something (a);       -- let do_something be a non-local
                                -- procedure
                                -- a.ALL will be read in the body
                                -- of do_something
        x := 6;
...
```

If this code sequence is compiled by the KRUPP ATLAS ELEKTRONIK Ada Compiler with the /OPTIMIZE qualifier the statement `x := 5;` will be eliminated because from the point of view of the optimizer the value of x is not used before the next assignment to x. Therefore

```
PRAGMA resident (x);
```

should be inserted after the declaration of x.

This pragma can be applied to all those kinds of objects for which the address clause is supported (cf. §16.5).

SUPPRESS_ALL

causes all the runtime checks described in the LRM(§11.7) to be suppressed; this pragma is only allowed at the start of a compilation before the first compilation unit; it applies to the whole compilation.)

15.1.3 Pragma Interface (Assembler,...)

This section describes the internal calling conventions of the KRUPP ATLAS ELEKTRONIK Ada System, which are the same ones which are used for subprograms for which a `PRAGMA interface` (ASSEMBLER,...) is given. Thus the actual meaning of this pragma is simply that the body needs and must not be provided in Ada, but in object form using the /EXTERNAL qualifier with the KAS LINK command.

The internal calling conventions are explained in four steps:

- Parameter passing mechanism
- Ordering of parameters
- Type mapping
- Saving registers

Parameter passing:

The parameters of a call to a subprogram are placed by the caller in an area called *parameter block*. This area is aligned on a longword boundary and contains parameter values (for parameter of scalar types), descriptors (for parameter of composite types) and alignment gaps.

For a function subprogram an extra field is assigned at the beginning of the parameter block containing the function result upon return. Thus the return value of a function is treated like an anonymous parameter of mode OUT. No special treatment is required for a function result except for return values of an unconstrained array type (see below).

A subprogram is called using the *JSR* instruction. The address pointing to the beginning of the parameter block is pushed onto the stack before calling the subprogram.

In general, the ordering of the parameter values within the parameter block does not agree with the order specified in the Ada subprogram specification. When determining the position of a parameter within the parameter block the calling mechanism and the size and alignment requirements of the parameter type are considered. The size and alignment requirements and the passing mechanism are described in the following:

Scalar parameters or parameters of access types are passed by value, i.e. the values of the actual parameters of modes IN or IN OUT are copied into the parameter block before the call. Then, after

the subprogram has returned, values of the actual parameters of modes IN OUT and OUT are copied out of the parameter block into the associated actual parameters. The parameters are aligned within the parameter block according to their size: A parameter with a size of 8, 16 or 32 bits (or a multiple of 8 bits greater than 32) has an alignment of 1, 2 or 4 (which means that the object is aligned to a byte, word or longword boundary within the parameter block). If the size of the parameter is not a multiple of 8 bits (which may be achieved by attaching a size specification to the parameter's type in case of an integer, enumeration or fixed point type) it will be byte aligned. Parameters of access types are always aligned to a longword boundary.

For parameters of composite types, descriptors are placed in the parameter block instead of the complete object values. A descriptor contains the address of the actual parameter object and, possibly, further information dependent on the specific parameter type. The following composite parameter types are distinguished:

- A parameter of a constrained array type is passed by reference for all parameter modes.

- For a parameter of an unconstrained array type, the descriptor consists of the address of the actual array parameter followed by the bounds for each index range in the array (i.e. FIRST(1), LAST(1), FIRST(2), LAST(2), ...). The space allocated for the bound elements in the descriptor depends on the type of the index constraint.

- For functions whose return value is an unconstrained array type a descriptor for the array is passed in the parameter block as for parameters of mode OUT. The fields for its address and all array index bounds are filled up by the function before it returns. In contrast to the procedure for an OUT parameter, the function allocates the array in its own stack space. The function then returns without releasing its stack space. After the function has returned, the calling routine copies the array into its own memory space and then deallocates the stack memory of the function.

- A constrained record parameter is passed by reference for all parameter modes.

- For an unconstrained record parameter of mode IN, the parameter is passed by reference using the address pointing to the record.

If the parameter has mode OUT or IN OUT, the value of the `CONSTRAINED` attribute applied to the actual parameter is passed as an additional boolean IN parameter (which occupies one byte in the parameter block and is aligned to a byte boundary). The boolean IN parameter and the address are treated like two consecutive parameters in a subprogram specification, i.e.the positions of the two parameters within the parameter block are determined independently of each other. For all kinds of composite parameter types the pointer pointing to the actual parameter object is represented by a 32 bit address, which is always aligned to a longword boundary.

Ordering of parameters:

The ordering of the parameters in the parameter block is determined as follows:

The parameters are processed in the order they are defined in the Ada subprogram specification. For a function the return value is treated as an anonymous parameter of mode OUT at the start of the parameter list. Because of the size and alignment requirements of a parameter it is not always

possible to place parameters in such a way that two consecutive parameters are densely located in the parameter block. In such a situation a gap, i.e. a piece of memory space which is not associated with a parameter, exists between two adjacent parameters. Consequently, the size of the parameter block will be larger than the sum of the sizes used for all parameters. In order to minimize the size of the gaps in a parameter block an attempt is made to fill each gap with a parameter that occurs later in the parameter list. If during the allocation of space within the parameter block a parameter is encountered whose size and alignment fit the characteristics of an available gap, then this gap is allocated for the parameter instead of appending it at the end of the parameter block. As each parameter will be aligned to a byte, word or longword boundary the size of any gap may be one, two or three bytes. Every gap of size three bytes can be treated as two gaps, one of size one byte with an alignment of 1 and one of size two bytes with an alignment of 2. So, if a parameter of size two is to be allocated, a two byte gap, if available, is filled up. A parameter of size one will fill a one byte gap. If none exists but a two byte gap is available, this is used as two one byte gaps. By this first fit algorithm all parameters are processed in the order they occur in the Ada program.

A called subprogram accesses each parameter for reading or writing using the parameter block address incremented by an offset from the start of the parameter block suitable for the parameter. So the value of a parameter of a scalar type or an access type is read (or written) directly from (into) the parameter block. For a parameter of a composite type the actual parameter value is accessed via the descriptor stored in the parameter block which contains a pointer to the actual object. When standard entry code sequences are used within the assembler subprogram (see below), the parameter block address is accessible at address (8,A6).

Type mapping:

To access individual components of array or record types, knowledge about the type mapping for array and record types is required. An array is stored as a sequential concatenation of all its components. Normally, pad bits are used to fill each component to a byte, word, longword or a multiple thereof depending on the size and alignment requirements of the components' subtype. This padding may be influenced using one of the PRAGMAs `pack` or `byte_pack` (cf. §16.1). The offset of an individual array component is then obtained by multiplying the padded size of one array component by the number of components stored in the array before it. This number may be determined from the number of elements for each dimension using the fact that the array elements are stored row by row. (For unconstrained arrays the number of elements for each dimension can be found in the descriptor stored in the parameter block.)

A record object is implemented as a concatenation of its components. Initially, locations are reserved for those components that have a component clause applied to them. Then locations for all other components are reserved. Any gaps large enough to hold components without component clauses are filled, so in general the record components are rearranged. Components in record variants are overlaid. The ordering mechanism of the components within a record is in principle the same as that for ordering the parameters in the parameter block.

A record may hold implementation-dependent components (cf. §16.4). For a record component whose size depends on discriminants, a generated component holds the offset of the record component within the record object. If a record type includes variant parts there may be a generated component (cf. §16.4) holding the size of the record object. This size component is allocated as the first component within the record object if this location is not reserved by a component clause.

Since the mapping of record types is rather complex record component clauses should be introduced for each record component if an object of that type is to be passed to a non Ada subprogram to be sure to access the components correctly.

Saving registers:

The last aspect of the calling conventions discussed here is that of saving registers. The calling subprogram assumes that the values of the registers A0, A1, A5, D0-D3, FP0-FP7 will be destroyed by the called subprogram and saves them of its own accord. If the called subprogram wants to modify further registers it has to ensure that the old values are restored upon return from the subprogram.

Finally we give the appropriate code sequences for the subprogram entry and for the return, which both obey the rules stated above.

A subprogram for which PRAGMA interface (assembler,...) is specified is - in effect - called with the subprogram calling sequence

```
PEA <address of parameter block> | only for functions or
                                 | procedures with parameters
JSR <subprogram address>
```

Thus the appropriate entry code sequence is

```
LINK  A6,#-(<frame-size>+4)

CLR.L (-4,A6)                    | The field at address (-4,A6) is reserved
                                 | for use by the Ada runtime system
```

The return code sequence is then simply

```
RTS
```

for procedures without parameters and

```
RTD #4
```

for functions and procedures with parameters.

Consider the following example. A function `sin` is to be implemented by an assembler routine. Its Ada specification is as follows:

```
FUNCTION sin (x : long_float) RETURN long_float;
PRAGMA interface (assembler, sin);
PRAGMA external_name ("CPSIN", sin); )
```

It is implemented by the following assembler routine:

```
CPSIN:  LINK.W   A6,#-4         *-- allocate frame
        CLR.L    (-4,A6)        *-- clear the indicator bits
        MOVEA.L  (8,A6),A0      *-- address of parameter block
        FSIN.X   (12,A0),FP0    *-- parameter x
        FMOVE.X  FP0,(A0)       *-- store function result
        UNLK     A6             *-- remove frame
        RTD      #4             *-- return to caller
```

15.1.4 Pragma Interface (META,...)

PRAGMA interface (META, ...) cannot be compared with the PRAGMA interface (assembler, ...).

For the use of PRAGMA interface (META, ...) there are several restriction concerning the use of META and parameter passing.

A META subroutine can only be called from the context of the main program. A call from a sub-task will raise PROGRAM_ERROR.
The user semaphores 0 .. 1023 are used within the runtime system and so far not available for the programmer.

For parameter passing the programmer has to define a record in Ada with a representation clause (cf. LRM(§13.4)). This record is the one and only parameter of the subroutine.
The META program gets a pointer to this record and uses it as a pointer to an equivalent structure.

Consider the following example. A procedure `put_line` is to be implemented by a META procedure. Its Ada specification is as follows:

```
TYPE parameter IS RECORD
        length   : natural;
        addr     : system.address;
END RECORD;

FOR parameter USE RECORD AT MOD 4;
        length  AT 0 RANGE 0..31;
        addr    AT 4 RANGE 0..31;
END RECORD;

FUNCTION put_line (x : IN parameter);
PRAGMA interface (META, put_line);
PRAGMA external_name ("P_L", put_line); )
```

It is implemented by the following META procedure :

```
PROC: P_L ( PB POINTER_TO PARAMETER );

TYPE : PARAMETER STRUCT /
          LENGTH  LONG,
          ADDRESS POINTER;

...
```

15.2 Implementation-Dependent Attributes

The name, type and implementation-dependent aspects of every implementation-dependent attribute is stated in this section.

15.2.1 Language-Defined Attributes

The name and type of all the language-defined attributes are as given in the LRM. We note here only the implementation-dependent aspects.

ADDRESS

If this attribute is applied to an object for which storage is allocated, it yields the address of the first storage unit that is occupied by the object.

If it is applied to a subprogram or to a task,it yields the address of the entry point of the subprogram or task body.

If it is applied to a task entry for which an address clause is given, it yields the address given in the address clause.

For any other entity this attribute is not supported and will return the value `system.address_zero`.

IMAGE

The image of a character other than a graphic character (cf. LRM(§3.5.5(11))) is the string obtained by replacing each italic character in the indication of the character literal (given in the LRM(Annex C(13))) by the corresponding upper-case character. For example, `character'image (nul) = "NUL"`.

MACHINE_OVERFLOWS

Yields `true` for each real type or subtype.

MACHINE_ROUNDS

Yields true for each real type or subtype

STORAGE_SIZE

The value delivered by this attribute applied to an access type is as follows: If a length specification (STORAGE_SIZE, see §16.2) has been given for that type (static collection), the attribute delivers that specified value. In the case of a dynamic collection,i.e. no length specification by STORAGE_SIZE given for the access type, the attribute delivers the number of storage units currently allocated for the collection. Note that dynamic collections are extended if needed.

If the collection manager (cf. §13.3.1) is used for a dynamic collection the attribute delivers the number of storage units currently allocated for the collection. Note that in this case the number of storage units currently allocated may be decreased by release operations.

The value delivered by this attribute applied to a task type or task object is as follows: If a length specification (STORAGE_SIZE, see §16.2) has been given for the task type, the attribute delivers that specified value; otherwise, the default value is returned.

15.2.2 Implementation-Defined Attributes

There are no implementation-defined attributes.

15.3 Specification of the Package SYSTEM

The package system as required in the LRM(§13.7) is reprinted here with all implementation-dependent characteristics and extensions filled in.

```
PACKAGE system IS

        TYPE designated_by_address IS LIMITED PRIVATE;

        TYPE address IS ACCESS designated_by_address;
        FOR address'storage_size USE 0;

        address_zero : CONSTANT address := NULL;

        FUNCTION "+" (left : address; right : integer) RETURN address;

        FUNCTION "+" (left : integer; right : address) RETURN address;

        FUNCTION "-" (left : address; right : integer) RETURN address;

        FUNCTION "-" (left : address; right : address) RETURN integer;
```

```
FUNCTION symbolic_address (symbol : string) RETURN address;

SUBTYPE external_address IS STRING;

-- External addresses use hexadecimal notation with characters
-- '0'..'9', 'a'..'f' and 'A'..'F'. For instance:
-- "7FFFFFFF"
-- "80000000"
-- "8" represents the same address as "00000008"

FUNCTION convert_address (addr : external_address) RETURN address;

-- convert_address raises CONSTRAINT_ERROR if the external address
-- addr is the empty string, contains characters other than
-- '0'..'9', 'a'..'f', 'A'..'F' or if the resulting address value
-- cannot be represented with 32 bits.

FUNCTION convert_address (addr : address) RETURN external_address;

-- The resulting external address consists of exactly 8 characters
-- '0'..'9', 'A'..'F'.

TYPE name IS (motorola_68030_kae);

system_name : CONSTANT name := motorola_68030_kae;

storage_unit : CONSTANT := 8;

memory_size : CONSTANT := 2 ** 31;

min_int : CONSTANT := - 2 ** 31;

max_int : CONSTANT := 2 ** 31 - 1;

max_digits : CONSTANT := 18;

max_mantissa : CONSTANT := 31;

fine_delta : CONSTANT := 2.0 ** (-31);

tick         : CONSTANT := 0.01;

SUBTYPE priority IS integer RANGE 0 .. 56;

non_ada_error : EXCEPTION RENAMES _non_ada_error;

-- non_ada_error is raised, if some event occurs which does not
-- correspond to any situation covered by Ada, e.g.:
```

```
-- illegal instruction encountered
-- error during address translation
-- illegal address

TYPE exception_id IS NEW address;

no_exception_id        : CONSTANT exception_id := NULL;

-- Coding of the predefined exceptions:

constraint_error_id    : CONSTANT exception_id := ...
numeric_error_id       : CONSTANT exception_id := ...
program_error_id       : CONSTANT exception_id := ...
storage_error_id       : CONSTANT exception_id := ...
tasking_error_id       : CONSTANT exception_id := ...

non_ada_error_id       : CONSTANT exception_id := ...

status_error_id        : CONSTANT exception_id := ...
mode_error_id          : CONSTANT exception_id := ...
name_error_id          : CONSTANT exception_id := ...
use_error_id           : CONSTANT exception_id := ...
device_error_id        : CONSTANT exception_id := ...
end_error_id           : CONSTANT exception_id := ...
data_error_id          : CONSTANT exception_id := ...
layout_error_id        : CONSTANT exception_id := ...

time_error_id          : CONSTANT exception_id := ...

system_error_code IS NEW integer;

no_error_code          : CONSTANT system_error_code := 0;

TYPE exception_kind IS
          (      ada,        -- exception info from ada - system completely
                 non_ada,    -- exception info from os (severe errors)
                 operating_system); -- exception info with error_code from os

TYPE exception_information (excp_kind : exception_kind := ada)

-- Selection of the exception kind. The codings and
-- meanings of the kind are given above.

IS RECORD

          excp_id     : exception_id := no_exception_id;
          -- Identification of the exception. The codings of
          -- the predefined exceptions are given above.
```

```
code_addr  : address := address_zero;
-- Code address where the exception occured. Depending
-- on the kind of the exception it may be be address of
-- the instruction which caused the exception, or it
-- may be the address of the instruction which would
-- have been executed if the exception had not occured.

CASE excp_kind IS

        WHEN ada=> NULL;

        -- no additional information available

WHEN non_ada        =>

        format_vector_offset : integer;

        -- Classification and identification of the system error
        -- by the "Format Vector Offset" (FVO) of the exception
        -- frame

        access_addr : address;

        -- the address accessed in case of bus - or address - error
        -- otherwise address_zero.

WHEN operating_system =>

        error_code : system_error_code := no_error_code;

        -- Classification and identification of the system error
        -- by supplying status information of the operating system
        -- at present selected in case of i/o - exceptions
        --          name_error
        --          | use_error
        --          | end_error
        --          | data_error

        record_count : natural := 0;

        -- the record count in case of i/o - transfers

END CASE;

END RECORD;

-------------------------------------------------
-- IT IS INTENDED THAT THE FOLLOWING TWO SUBPROGRAMS ARE
-- USED ONLY WHEN INTERFACING WITH THE OPERATING SYSTEM.
```

```
PROCEDURE get_exception_information  (excp_info : OUT exception_information);

-- The subprogram get_exception_information has to be called
-- from within an exception_handler BEFORE ANY OTHER EXCEPTION
-- IS RAISED. It returns the information_record about the
-- actually handled exception.
-- Otherwise, its result is undefined.

PROCEDURE raise_exception_info  (excp_info : IN OUT exception_information);

-- The behaviour of this subprogram depends on the values
-- of excp_info:
-- If the (predefined) value "no_exception_id" for the component
-- excp_info.excp_id is NOT selected, the subprogram raises
-- the exception described by the exception_id supplied by the
-- parameter, otherwise the values of this information record are
-- handled internally and no exception is raised.
-- If the (predefined) value "address_zero" for the component
-- excp_info.code_addr is supplied, the procedure delivers
-- the return address for this procedure call in this component,
-- otherwise the given value is handled internally.

PRIVATE

...

END system;
```

15.4 Restrictions on Representation Clauses

See Chapter 16 of this manual.

15.5 Conventions for Implementation-Generated Names

There are implementation generated components but these have no names. (cf. §16.4 of this manual).

15.6 Expressions in Address Clauses

See §16.5 of this manual.

15.7 Restrictions on Unchecked Conversions

The implementation supports unchecked type conversions for all kinds of source and target types with the restriction that the target type must not be an unconstrained array type. The result value of the unchecked conversion is unpredictable, if

target_type'SIZE > source_type'SIZE)

15.8 Characteristics of the Input-Output Packages

The implementation-dependent characteristics of the input-output packages as defined in the LRM(Chapter 14) are reported in Chapter 17 of this manual.

15.9 Requirements for a Main Program

A main program must be a parameterless library procedure. This procedure may be a generic instantiation; the generic procedure need not be a library unit.

15.10 Unchecked Storage Deallocation

The generic procedure `unchecked_deallocation` is provided; the effect of calling an instance of this procedure is as described in the LRM(§13.10.1).
The implementation also provides an implementation-defined package `collection_manager`, which has advantages over unchecked deallocation in some applications (cf. §13.3.1).
Unchecked deallocation and operations of the `collection_manager` can be combined as follows:

- `collection_manager.reset` can be applied to a collection on which unchecked deallocation has also been used. The effect is that storage of all objects of the collection is reclaimed.
- After the first `unchecked_deallocation(release)` on a collection, all following calls of `release (unchecked deallocation)` until the next `reset` have no effect, i.e. storage is not reclaimed.
- after a `reset` a collection can be managed by `mark` and `release` (resp. `unchecked_deallocation`) with the normal effect even if it was managed by `unchecked_deallocation` resp. `mark` and `release` before the `reset`.

15.11 Machine Code Insertions

A package `machine_code` is not provided and machine code insertions are not supported.

15.12 Numeric Error

The predefined exception `numeric_error` is never raised implicitly by any predefined operation; instead the predefined exception `constraint_error` is raised.

16 Appendix F: Representation Clauses

In this chapter we follow the section numbering of Chapter 13 of the LRM and provide notes for the use of the features described in each section.

16.1 Pragmas

PACK

As stipulated in the LRM(§13.1), this pragma may be given for a record or array type. It causes the Compiler to select a representation for this type such that gaps between the storage areas allocated to consecutive components are minimized. For components whose type is an array or record type the PRAGMA `PACK` has no effect on the mapping of the component type. For all other component types the Compiler will choose a representation for the component type that needs minimal storage space (packing down to the bit level). Thus the components of a packed data structure will in general not start at storage unit boundaries.

BYTE_PACK

This is an implementation-defined pragma which takes the same argument as the predefined language PRAGMA `PACK` and is allowed at the same positions. For components whose type is an array or record type the PRAGMA `BYTE_PACK` has no effect on the mapping of the component type. For all other component types the Compiler will try to choose a more compact representation for the component type. But in contrast to PRAGMA `PACK` all components of a packed data structure will start at storage unit boundaries and the size of the components will be a multiple of `system.storage_unit`. Thus, the PRAGMA `BYTE_PACK` does not effect packing down to the bit level (for this see PRAGMA `PACK`).

16.2 Length Clauses

SIZE

For all integer, fixed point and enumeration types the value must be <= 32;
for `short_float` types the value must be = 32 (this is the amount of storage which is associated with these types anyway);
for `float` types the value must be = 64 (this is the amount of storage which is associated with these types anyway).
for `long_float` types the value must be = 96 (this is the amount of storage which is associated with these types anyway).
for `access` types the value must be = 32 (this is the amount of storage which is associated with these types anyway).
If any of the above restrictions are violated, the Compiler responds with a RESTRICTION error message in the Compiler listing.

STORAGE_SIZE

Collection size: If no length clause is given, the storage space needed to contain objects designated by values of the access type and by values of other types derived from it is extended dynamically at runtime as needed. If, on the other hand, a length clause is given, the number of storage units stipulated in the length clause is reserved, and no dynamic extension at runtime occurs.

Storage for tasks: The memory space reserved for a task is 16K bytes if no length clause is given (cf. Chapter 14). If the task is to be allotted either more or less space, a length clause must be given for its task type, and then all tasks of this type will be allotted the amount of space stipulated in the length clause (the activation of a small task requires about 1.4K bytes). Whether a length clause is given or not, the space allotted is not extended dynamically at runtime.

SMALL

There is no implementation-dependent restriction. Any specification for `SMALL` that is allowed by the LRM can be given. In particular those values for `SMALL` are also supported which are not a power of two.

16.3 Enumeration Representation Clauses

The integer codes specified for the enumeration type have to lie inside the range of the largest integer type which is supported; this is the type `integer` defined in package `standard`.

16.4 Record Representation Clauses

Record representation clauses are supported. The value of the expression given in an alignment clause must be 0, 1, 2 or 4. If this restriction is violated, the Compiler responds with a RESTRICTION error message in the Compiler listing. If the value is 0 the objects of the corresponding record type will not be aligned, if it is 1, 2 or 4 the starting address of an object will be a multiple of the specified alignment.
The number of bits specified by the range of a component clause must not be greater than the amount of storage occupied by this component. (Gaps between components can be forced by leaving some bits unused but not by specifying a bigger range than needed.) Violation of this restriction will produce a RESTRICTION error message.
There are implementation-dependent components of record types generated in the following cases

- If the record type includes variant parts and if it has either more than one discriminant or else the only discriminant may hold more than 256 different values, the generated component holds the size of the record object.

- If the record type includes array or record components whose sizes depend on discriminants, the generated components hold the offsets of these record components (relative to the corresponding generated component) in the record object.) But there are no implementation-generated names (cf. LRM(§13.4(8))) denoting these components. So the mapping of these

components cannot be influenced by a representation clause.

16.5 Address Clauses

Address clauses are supported for objects declared by an object declaration and for single task entries. If an address clause is given for a subprogram, package or a task unit, the Compiler responds with a RESTRICTION error message in the Compiler listing.

If an address clause is given for an object, the storage occupied by the object starts at the given address. Address clauses for single entries are described in §16.5.1.

16.6 Change of Representation

The implementation places no additional restrictions on changes of representation.

17. Appendix F: Input-Output

In this chapter we follow essentially the section numbering of Chapter 14 of the LRM and provide notes for the use of the features described in each section.

17.1 External Files and File Objects

The association of internal to external files is managed via I/O - channels provided by a standard - input - output monitor (in the following abbreviated to "STIOMO") of the MOS2300.

In general the association of multiple internal files to a single external file is not allowed. However with application of the FORM parameter "OPEN => OLD_SHARED" (see §17.1.1) the only form of file sharing which is allowed is shared reading. If two or more files are associated with the same external file at one time (regardless of whether these files are declared in the same program or task), all of these (internal) files must be opened with the mode `in_file`. An attempt to open one of these files with a mode other than `in_file` will raise the exception `USE_ERROR`.
Files associated with terminal devices (which is only legal for text files) are excepted from this restriction. Such files may be opened with an arbitrary mode at the same time and associated with the same terminal device.

The following restrictions apply to the generic actual parameter for `element_type`:

- input/output of access types is not defined.

- input/output of unconstrained array types is only possible with record mode `sequential` (see FORM parameter "RECORD_MODE", §17.1.1.)

- the size of an object to be input or output must not be greater than 32767 storage units.

Files opened or reset to mode `in_file` are locked for write - access, otherwise they are unlocked.
The number of files which are open at the same time is only restricted by the MOS2300 system resources.
Temporary files are temporary in the sense of MOS2300. They are deleted by close and delete - operations and after the job step is exited.
After completion of the main program and all library tasks all opened files are closed. The same is true if the program execution is aborted by typing ASCII.ETX (= CTRL/C) followed by "X.",";X." or ";X.X.".

17.1.1 The NAME and FORM Parameters

The **name** parameter string must be a legal MOS2300 file specification string and must not con-

tain wild cards, even if that would specify a unique file. It must not contain any MOS2300 channel mode specification. The function `name` will return a complete file specification string consisting of

<softdisc>:/<area>/<path><file>.<type>-<version>.<level>,

which is the file name of the file opened or created. It does not contain the node name specification. If necessary the node name can be supplied by a FORM parameter (see below) in an open or create operation. Only <file>.<type> is essential in the `name` parameter string, all other fields are optional.
Temporary file names are unique due to the following structure:
The first character of the filename is a "%", the second is a package identifier ("T" for `text_io`, "S" for `sequential_io` and "D" for `direct_io`). A three-digit number identifies the STIO-MO and a two-digit number the channel which is associated to the external and internal file. The numbers are separated by an underline - character. The file type is "TMP".

The syntax of the `form` parameter string is defined by:

form_parameter ::= [form_specifikation { , form_specification }]

form_specification ::= keyword [=> value]

keyword ::= identifier

value ::= identifier | string_literal | numeric_literal

For identifier, numeric_literal, string_literal see LRM(Appendix E).
Only an integer literal is allowed as numeric_literal (cf. LRM(§2.4)).
In the following, the form specifications which are allowed for all files are described. Special form specifications, which are only valid for `text_io` are described in §17.3.1.

NEW_STIOMO

Before the external file is associated to a STIOMO - channel a new STIOMO Instantiation is forced.

APPEND

This keyword provides for opening the external file at its end. It has no effect for creation of files. The `file_mode` must be `out_file`. This keyword is not applicable to `direct_io`.

OPEN => identifier

This keyword makes the use of several open - modes possible, that are known in MOS2300 - I/O. The identifiers are

ANY

If the file is opened, the file is created if not existing. If the file is created, an existing external file is overwritten. Application is not possible for terminals and devices. Possibly the exception

USE_ERROR is raised in following I/O - operations (e.g. if an existing file is write - locked). The file - date is not modified.

OLD_SHARED

An existing external file can be accessed by multiple internal files for reading. The file_mode must be in_file.

DEVICE

The internal file is associated with a device exclusively. The name parameter takes the device name. This identifier is not applicable to direct_io.

DEVICE_SHARED

Multiple internal files may be associated with a device. The name parameter takes the device name. This identifier is not applicable to direct_io.

SPOOL

A spool file is created on the SPOOL - area. The name parameter takes the name of the SPOOL - device. The file_mode must be in_file. The identifier is only allowed in a create - operation.

TERMINAL

The internal file is associated with the LOGIN - terminal for read and write operations. Multiple internal files may be associated with a terminal. The identifier is only allowed in an open - operation. The name parameter is ignored and should be set to the empty string. It is not applicable to direct_io.

RECORD_MODE => identifier

This keyword determines the MOS2300 record - mode, which means the record organization of the external file. It is not applicable to terminals and devices. It has no effect for existing files which are opened. For text_io only the record - mode SEQUENTIAL makes sense. Therefore this FORM - Parameter is not available in text_io. The identifiers are:

SEQUENTIAL

The external file is created with records of variable length. This is the default for sequential_io.

DIRECT

The external file is created with records of fixed length. The internal actual length of the records can be smaller. The actual length returned after read operations cannot be used in the predefined Ada - I/O packages. For this reason this form specification is usefull if the external file should be accessed by foreign (non Ada-) programs after creation.

FIXED

The external file is created with records of fixed length. The access to these records is fast, because the record positions can be calculated and the length information is omitted. Therefore this is the default for `direct_io`.

REMOTE => string_literal

This keyword makes the access to other nodes possible. `string_literal` is the nodename. No checking of the name syntax and the existence of the node is done.

TIMEOUT => numeric_literal

This keyword makes it possible to provide terminals and devices with timeout values. `numeric_literal` is the timeout - value in units of 10 milliseconds. No checking of `numeric_literal` takes place. The FORM parameter is only allowed in conjunction with open - parameters `DEVICE`, `DEVICE_SHARED` and `TERMINAL` and with file_mode `in_file`. It is not applicable to `direct_io`.

Multiple form specifications of the same parameter are allowed, but only the last written form specification is valid.

17.2 Sequential and Direct Files

Sequential and direct files are represented by MOS2300 - devices or files with record mode `SEQUENTIAL,FIXED` and `DIRECT`. The default is record mode `fixed` for direct files and `sequential` for sequential files. Other record modes are set up by application of the FORM - parameter `RECORD_MODE` (see FORM Parameters, §17.1.1). Each element of the file is stored in one record. The record size is determined by the number of storage units needed for the element type. It equals element_type'SIZE / system.storage_unit, if element_type is a multiple of `system.storage_unit`, otherwise it equals element_type'SIZE / system.storage_unit + 1. Unconstrained array types may only be used in files with record mode `sequential`

17.2.1 Sequential Files

The default form string for a sequential file is :

"RECORD_MODE => SEQUENTIAL"

The default form may be used for all types (except for those excluded in §17.1).

17.2.2 Direct Files

The implementation dependent type `count` defined in the package specification of `direct_io`

has an upper bound of :

COUNT'LAST = 2_147_483_647 (= INTEGER'LAST)

The default form string for a direct file is :

"RECORD_MODE => FIXED"

The default form may not be used for unconstrained array types.

17.3 Text Input-Output

Text files are represented as MOS2300 terminal devices or files with record mode `sequential`. One line is represented as a sequence of one or more records; all records except for the last one have a length of exactly MAX_RECORD_SIZE and a continuation marker (' ') at the last position. A line of length MAX_RECORD_SIZE - 1 is represented by one record of this length. A line terminator is not represented explicitly in the external file; the end of a record which is shorter than MAX_RECORD_SIZE does not have a continuation marker as its last character and is taken as a line terminator.

The value MAX_RECORD_SIZE may be specified by a form string. The values of MAX_RECORD_SIZE are restricted from 2 to 200 (this is the actual length of the internal buffer in storage_units). If the line length is equal or greater than MAX_RECORD_SIZE then records of length MAX_RECORD_SIZE are generated for file_mode out_file. If file_mode is in_file then the records expected to read must be equal to or less than the actual value of MAX_RECORD_SIZE
A page terminator is represented as a record consisting of a single ASCII.FF. A record of length zero is assumed to precede a page terminator if the record before the page terminator is another page terminator or a record of length MAX_RECORD_SIZE with a continuation marker at the last position; this implies that a page terminator is preceded by a line terminator in all cases.
A file terminator is not represented explicitly in the external file; the end of the file is taken as a file terminator. A page terminator is assumed to precede the end of the file if there is not explicitly one as the last record of the file. For input from a terminal, a file terminator is represented as ASCII.SUB (= CTRL/Z).

17.3.1 File Management

In the following, the form specifications which are only allowed for text files or have a special meaning for text files are described.

CHARACTER_IO

The predefined package `text_io` was designed for sequential text files; moreover, this implementation always uses sequential files with a record structure, even for terminal devices. It therefore offers no language-defined facilities for modifying data previously written to the terminal (e.g. changing characters in a text which is already on the terminal screen) or for outputting cha-

racters to the terminal without following them by a line terminator. It also has no language-defined provision for input of single characters from the terminal (as opposed to lines, which must end with a line terminator, so that in order to input one character the user must type in that character and then a line terminator) or for suppressing the echo on the terminal of characters typed in at the keyboard.
For these reasons, in addition to the input/output facilities with record structured external files, another form of input/output is provided for text files:
It is possible to transfer single characters from/to a terminal device. This form of input/output is specified by the keyword CHARACTER_IO in the form string. CHARACTER_IO is only allowed in conjunction with the form specification "OPEN => TERMINAL" or "OPEN => DEVICE(_SHARED)". MAX_RECORD_SIZE must be the default value (200).
For an infile, the external file (associated with a terminal) is considered to contain a single line. An ASCII.SUB (= CTRL/Z) character represents a line terminator followed by a page terminator followed by a file terminator. Arbitrary characters (including all control characters except for ASCII.SUB) may be read; a character read is not echoed to the terminal.
For an outfile, an item is written to the external file (terminal) without a linefeed. A line terminator is not represented. A page terminator is represented as ASCII.FF, a line terminator and a file terminator are not represented on the external file.

MAX_RECORD_SIZE => numeric_literal

This value specifies the maximum length of a record in the external file. Each record which is not the last record of a line has exactly this maximum record size, with a continuation marker (' ') at the last position. The value must be in the range 2 .. 200, which is the actual size of the internal buffer. The value is defaulted to 200. If a file is opened with file_mode out_file, the specifie lue (or the default of 200) is used to create records not longer than specified in this value external file. If the value is specified for an existing file it must be equal or greater than the corresponding value in the fileparameters of the external file. . The default form string for a text file is

MAX_RECORD_SIZE => 200

17.3.2 Default Input and Output Files

The association of the standard input file to external files depends on the use of MOS2300 jobfiles. This means, the "T"(transaction) - channel is linked to a file, which contains instructions to be executed by the job processor.
If the "T" - channel is linked to such a jobfile, then the channel - settings are copied to a "J_" - channel. In this case this channel is associated to the standard input file, which gets its input from the jobfile positioned to the current record. The name returned with function **name** is the fully specified name of the jobfile.
If the "T" - channel is not linked to a file (normally it is linked to a terminal), then the standard input file is associated with the MOS2300 user login terminal and expects its input from this terminal. The name returned with function **name** is the MOS2300 device name for this terminal.
It is possible to pass jobcontrol - instructions via the standard input file to the job processor. The jobcontrol - instruction has to be initiated with a ASCII.SEMICOLON as jobtrap - character in the first column. The read operation is recovered with the next input line. The passing of jobcontrols is

only possible, if the parameter "JOB_C" was passed in the commandline during start of the Ada program, because the standard input file should normally accept the ASCII.SEMICOLON in the first column as input data.
The standard output file is associated with the MOS2300 user login terminal. The name returned with function `name` is the MOS2300 device name for this terminal. If the parameter "PROT" is given in the Ada - start commandline, then additionally the standard output is written to a file with name "STANDARD.OUT" on the default area and subdirectory. If a file with this name exists, the output will be appended to the file. If a file with that name is exclusivly accessed, the file level is incremented and another try is done with this version.
The form strings for the standard input and output files are :

"OPEN => TERMINAL, MAX_RECORD_SIZE => 200"

17.3.3 Implementation-Defined Types

The implementation-dependent types `count` and `field` defined in the package specification of `text_io` have the following upper bounds :

COUNT'LAST = 2_147_483_647 (= INTEGER'LAST)

FIELD'LAST = 512

17.4 Exceptions in Input-Output

For each of `name_error`, `use_error`, `device_error` and `data_error`, the conditions under which that exception can be raised are listed. The conditions under which the other exceptions declared in the package `io_exceptions` can be raised are as described in the LRM(§14.4).

NAME_ERROR

- in an `open` operation, if the specified file does not exist;
- if the `name` parameter in a call of the `create` or `open` procedure is not a legal MOS2300 file specification string; for example, if it contains illegal characters, is too long or is syntactically incorrect; and also if it contains wild cards, even if that would specify a unique file.

USE_ERROR

- whenever an error occurred during an operation of the underlying MOS2300 system. This may happen if an internal error was detected, an operation is not possible for reasons depending on the file or device characteristics, a size restriction is violated, a capacity limit is exceeded or for similar reasons;
- if the characteristics of the external file are not appropriate for the file type; for example, if the record size of a file does not correspond to the size of the element type of a `direct_io` or

sequential_io file. In general it is only guaranteed that a file which is created by an Ada program may be reopened by another program if the file types and the form strings are the same;

- if two or more (internal) files are associated with the same external file at one time (regardless of whether these files are declared in the same program or task), and an attempt is made to open one of these files with mode other than in_file if the open mode OLD_SHARED is specified. However, files associated with terminal devices (which is only legal for text files) are excepted from this restriction. Such files may be opened with an arbitrary mode at the same time and associated with the same terminal device;
- if a given form parameter string does not have the correct syntax or if a condition on an individual form specification described in §17.1.1, §17.3.1 is not fulfilled;

DEVICE_ERROR

is never raised. Instead of this exception the exception use_error is raised whenever an error occurred during an operation of the underlying MOS2300 system.

DATA_ERROR

- the conditions under which data_error is raised by text_io are stated in the LRM; the following notes apply to the packages sequential_io and direct_io :
  - by the procedure read if the size of a record in the external file to be read exceeds the storage size of the given variable which has exactly the size element_type'SIZE.
  - In general, the exception data_error is not raised by the procedure read if the element read is not a legal value of the element type.
  - by the procedure read if an element with the specified position in a direct file does not exist; this is only possible if the file is associated with a external file with record mode fixed or direct.

17.5 Low Level Input-Output

We give here the specification of the package low_level_io :

```
PACKAGE low_level_io IS

        TYPE device_type IS (null_device);

        TYPE data_type IS   RECORD
                NULL;
        END RECORD;

PROCEDURE send_control      (device : device_type;   data : IN OUT data_type);
```

```
PROCEDURE receive_control    (device : device_type;  data : IN OUT data_type);

END low_level_io;
```

Note that the enumeration type `device_type` has only one enumeration value, `null_device`; thus the procedures `send_control` and `receive_control` can be called, but `send_control` will have no effect on any physical device and the value of the actual parameter `data` after a call of `receive_control` will have no physical significance.