



REPORT DOCUMENTATION PAGE

Form Approved

AD-A285 574



Estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data, viewing the collection of information. Send comments regarding this burden, to Washington Headquarters, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Information and Statistics, Washington, DC 20503.

OPM No.

2. REPORT

3. REPORT TYPE AND DATES

4. TITLE AND: Alsys Corporation, Compiler: AlsyCOMP_17 Version 5.4.10
Host: VAXstation 4000 Model 60 (under VAX/VMS V 5.5-2)
Target: INMOS T9000 transputer Gamma D02 installed on an INMOS VME TestBoard (Bare), VC#: 940826N1.11375

5. FUNDING

6. Authors: The National Computing Centre, Ltd.

7. PERFORMING ORGANIZATION NAME (S) AND:

The National Computing Centre, Ltd.
Oxford Road
Manchester M1 7ED
England

8. PERFORMING ORGANIZATION

9. SPONSORING/MONITORING AGENCY NAME(S) AND:

Ada Joint Program Office, Defense Information Systems Agency
Code TXEA, 701 S. Courthouse Rd.
Arlington, VA 22204-2199

10. SPONSORING/MONITORING AGENCY

11. SUPPLEMENTARY

12a. DISTRIBUTION/AVAILABILITY: Approved for Public Release; distribution unlimited

12b. DRISTRIBUTION

13. (Maximum 200:

14. SUBJECT: Ada Programming Language, Ada Compiler Validation Summary Report, Ada Compiler Validation Capability Validation Testing, Ada Validation Office, Ada Validation Facility, ANSI/MIL-STD-1815A, AJPO

15. NUMBER OF

16. PRICE

17 SECURITY CLASSIFICATION
UNCLASSIFIED

18. SECURITY
UNCLASSIFIED

19. SECURITY CLASSIFICATION
UNCLASSIFIED

20. LIMITATION OF
UNCLASSIFIED

NSN

AVF Control Number: AVF_VSR_05401/87-940831

Ada COMPILER
VALIDATION SUMMARY REPORT:
Certificate Number: #940826N1.11375
ALSYS LIMITED
AlyCOMP_017 Version 5.4.10
VAXstation 4000 Model 60 Host and
INMOS T9000 transputer Gamma D02 on
an INMOS VME TestBoard Target

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

Prepared By:
Testing Services
The National Computing Centre Limited
Oxford Road
Manchester
M1 7ED
England

Template Version 94-05-11

DTIC QUALITY INSPECTED 2

94-32008



420770

978

2

TABLE OF CONTENTS

CHAPTER 1 INTRODUCTION

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

CHAPTER 2 IMPLEMENTATION DEPENDENCIES

2.1	WITHDRAWN TESTS	1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

CHAPTER 3 PROCESSING INFORMATION

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

APPENDIX A MACRO PARAMETERS 1

APPENDIX B COMPILATION SYSTEM OPTIONS 1

APPENDIX C APPENDIX F OF THE Ada STANDARD 1

Certificate Information

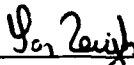
The following Ada implementation was tested and determined to pass ACVC 1.11. Testing was completed on 24 August 1994.

Compiler Name and Version: AlsyCOMP_017 Version 5.4.10
Host Computer System: VAXstation 4000 Model 60 (under VAX/VMS V 5.5-2)
Target Computer System: INMOS T9000 transputer Gamma D02 installed on an INMOS
 VME TestBoard (Bare)

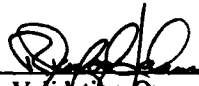
See section 3.1 for any additional information about the testing environment.

As a result of this validation effort, Validation Certificate #940826N1.11375 is awarded to Alsys Limited. This certificate expires 2 years after ANSI/MIL-STD-1815B is approved by ANSI.

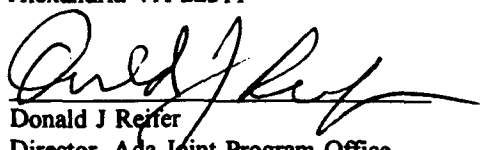
This report has been reviewed and is approved.



Jon Leigh
Manager, System Software Testing
The National Computing Centre Limited
Oxford Road
Manchester
M1 7ED
England



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



Donald J Reifer
Director, Ada Joint Program Office
Defense Information Systems Agency
Centre for Information Management
Washington DC 20301

DECLARATION OF CONFORMANCE

DECLARATION OF CONFORMANCE

The following declaration of conformance was supplied by the customer.

Declaration of Conformance

Customer: ALSYS LIMITED
Ada Validation Facility: The National Computing Centre Limited
ACVC Version: 1.11
Ada Implementation:
Ada Compiler Name and Version: AlsyCOMP_017 Version 5.4.10
Host Computer System: VAXstation 4000 Model 60 (under VAX/VMS V 5.5-2)
Target Computer System: INMOS T9000 transputer Gamma D02 installed on an INMOS VME Testboard (Bare)

Declaration:

I, the undersigned, declare that [I/we] have no knowledge of deliberate deviations from the Ada Language Standard ANSI/MIL-STD-1815A, ISO 8652-1987, FIPS 119 as tested in this validation and documented in the Validation Summary Report.

C. D. Hones
Customer Signature

24 August 94
Date

TABLE OF CONTENTS

INTRODUCTION	1
1 Implementation-Dependent Pragmas	3
1.1 INLINE	3
1.2 INLINE_GENERIC	3
1.3 INTERFACE	5
1.3.1 Calling Conventions	5
1.3.2 Parameter-Passing Conventions	6
1.3.3 Parameter Representations	7
1.3.4 Restrictions on Interfaced Subprograms	9
1.4 INTERFACE_NAME	11
1.5 NO_IMAGE	12
1.6 INDENT	12
1.7 Other Pragmas	13
2 Implementation-Dependent Attributes	15
3 Specification of the Package SYSTEM	17
4 Restrictions on Representation Clauses	21
4.1 Enumeration Types	22
4.2 Integer Types	25
4.3 Floating Point Types	28
4.4 Fixed Point Types	30
4.5 Access Types	33
4.6 Task Types	34
4.7 Array Types	36
4.8 Record Types	40

5	Conventions for Implementation-Generated Names	51
6	Address Clauses	53
6.1	Address Clauses for Objects	53
6.2	Address Clauses for Program Units	53
6.3	Address Clauses for Entries	53
7	Restrictions on Unchecked Conversions	55
8	Input-Output Packages	57
8.1	NAME Parameter	57
8.2	FORM Parameter	57
8.2.1	File Sharing	58
8.2.2	Binary Files	59
8.2.3	Buffering	60
8.2.4	Appending	61
8.3	USE_ERROR	61
9	Characteristics of Numeric Types	63
9.1	Integer Types	63
9.2	Floating Point Type Attributes	64
9.3	Attributes of Type DURATION	65
	REFERENCES	67
	INDEX	69

INTRODUCTION

Implementation-Dependent Characteristics

This appendix summarizes the implementation-dependent characteristics of the Alsys Ada Compilers for INMOS transputers. This document should be considered as the Appendix F to the Reference Manual for the Ada Programming Language ANSI/MIL-STD 1815A, January 1983, as appropriate to the Alsys Ada implementation for the transputer.

Sections 1 to 8 of this appendix correspond to the various items of information required in Appendix F [F]*; sections 9 and 10 provide other information relevant to the Alsys implementation. The contents of these sections is described below:

1. The form, allowed places, and effect of every implementation-dependent pragma.
2. The name and type of every implementation-dependent attribute.
3. The specification of the package SYSTEM [13.7].
4. The list of all restrictions on representation clauses [13.1].
5. The conventions used for any implementation-generated names denoting implementation-dependent components [13.4].
6. The interpretation of expressions that appear in address clauses.
7. Any restrictions on unchecked conversions [13.10.2].
8. Any implementation-dependent characteristics of the input-output packages [14].
9. Characteristics of numeric types.

* Throughout this manual, citations in square brackets refer to the *Reference Manual for the Ada Programming Language*, ANSI/MIL-STD-1815A, January 1983.

Throughout this appendix, the name *Ada Run-Time Executive* refers to the run-time library routines provided for all Ada programs. These routines implement the Ada heap, exceptions, tasking control, I/O, and other utility functions.

CHAPTER 1

Implementation-Dependent Pragmas

1.1 INLINE

Pragma `INLINE` is fully supported, except for the fact that it is not possible to inline a function call in a declarative part.

1.2 INLINE_GENERIC

The pragma `INLINE_GENERIC` takes the name of a generic unit or of an instance of a generic unit as argument.

When the argument is the name of a generic unit, pragma `INLINE_GENERIC` specifies that all the instances of the generic unit will be generated in line (and not in a subunit) regardless of the value of the option `GENERIC`s of the command `COMPILE`.

When the argument is the name of an instance of a generic unit, pragma `INLINE_GENERIC` specifies that this instance will be generated in line (and not in a subunit) regardless of the value of the option `GENERIC`s of the command `COMPILE`.

```
pragma INLINE_GENERIC (Ada_designator {, Ada_designator })
```

Example:

```
procedure MY_PROCEDURE is
    type MY_TYPE is (...);
    generic
        procedure MY_FIRST_GENERIC;
    generic
        type T is (<>);
    function MY_SECOND_GENERIC(L,R : T) return T;
```

```

function MY_SECOND_GENERIC(L,R : T) return T is
begin
    ...
end;

function "+" is new MY_SECOND_GENERIC(MY_TYPE);

pragma INLINE_GENERIC(MY_FIRST_GENERIC, "+");

procedure MY_FIRST_GENERIC is
begin
    ...
end;

.....

end MY_PROCEDURE ;

```

Limitations on the use of pragma INLINE_GENERIC

- This pragma is only allowed at the place of a declarative item in a declarative part or package specification, or after a library unit in a compilation but before any subsequent compilation unit. If the pragma appears at the place of a declarative item, each argument must denote a generic unit, or an instance of a generic unit, declared by an earlier declarative item of the same declarative part or package specification. If several (overloaded) subprograms satisfy this requirement, the pragma applies to all of them. If the pragma appears after a given library unit, the only name allowed is the name of this unit, which must be a generic unit.
- If an instance cannot be generated in line, in spite of a valid pragma `INLINE_GENERIC`, a warning will be emitted and the instance will be generated in a subunit. This will occur in particular when the body of the generic unit has not been compiled at the point of instantiation. This will also happen on instances appearing within the specification of a generic package (due to a restriction of the Alsys implementation).

1.3 INTERFACE

Ada programs can interface to subprograms written in occam through the use of the predefined pragma `INTERFACE` [13.9] and the implementation-defined pragma `INTERFACE_NAME`.

Pragma `INTERFACE` specifies the name of an interfaced subprogram and the name of the programming language for which calling and parameter passing conventions will be generated. Pragma `INTERFACE` takes the form specified in the *Reference Manual*:

```
pragma INTERFACE (language_name, subprogram_name);
```

where:

- *language_name* is the name of the other language whose calling and parameter passing conventions are to be used.
- *subprogram_name* is the name used within the Ada program to refer to the interfaced subprogram.

The language names currently accepted by pragma `INTERFACE` are `OCCAM` and `OCCAM_HIGH`.

The language name used in the pragma `INTERFACE` does not necessarily correspond to the language used to write the interfaced subprogram. It is used only to tell the Compiler how to generate subprogram calls, that is, which calling conventions and parameter passing techniques to use.

The language name `OCCAM` is used to refer to the standard occam calling and parameter passing conventions for the transputer (Ref. 4, Section 5.10). The programmer can use the language name `OCCAM` to interface Ada subprograms to subroutines written in any language that follows the standard occam calling conventions.

The programmer can use the language name `OCCAM_HIGH` to run the interfaced subroutine as a high priority process instead of a low priority process. The `OCCAM_HIGH` interface is otherwise identical to the `OCCAM` interface.

1.3.1 Calling Conventions

The following calling conventions are required for code to be called from Ada by use of pragma `INTERFACE`.

On entry to the subprogram, the registers A, B and C are undefined. For the T8 and T9000, the floating point registers FA, FB and FC are similarly undefined. The return address and any parameters are accessed relative to the workspace pointer, W, by the subprogram.

There are no assumptions concerning the contents of the register stacks (A, B, C and FA, FB, FC) upon return from the interfaced subprogram, other than for interfaced subprograms which are functions (see below). However, the workspace pointer, W, should contain the same address upon return from the interfaced subprogram as it contained before the call.

On the T4 and T8 the setting of the error flag is ignored on return.

1.3.2 Parameter-Passing Conventions

On entry to the subprogram, the word at offset 0 from the transputer workspace (W) pointer contains the return address of the called subprogram. Subsequent workspace locations (from W + 1 to W + n, where n is the number of parameters) contain the subprogram parameters, which are all one word in length.

There is always an implicit vector space parameter passed as the last parameter to all interfaced subprograms. This points to an area of free memory which can be used by the occam compiler to allocate arrays declared in the interfaced subprogram.

Actual parameters of mode *in* which are access values or scalars of one machine word or less in size are passed by copy. If such a parameter is less than one machine word in length it is sign extended to a full word. For all other parameters of mode *in*, and all parameters of mode *in out* or *out*, the value passed is the address of the actual parameter itself.

Since all large scalar, non-scalar and non-access parameters to interfaced subprograms are passed by address, they cannot be protected from modification by the called subprogram even though they may be formally declared to be of mode *in*. It is the programmer's responsibility to ensure that the semantics of the Ada parameter modes are honoured in these cases.

If the interfaced subprogram is a function, the result is returned as follows:

A floating point result is returned in register FPA.

Any other result is returned by value in register A if its size is at most one machine word and by address in register A otherwise.

No consistency checking is performed between the subprogram parameters declared in Ada and the corresponding parameters of the interfaced subprogram. It is the programmer's responsibility to ensure correct access to the parameters.

1.3.3 Parameter Representations

This section describes the representation of values of the types that can be passed as parameters to an interfaced subprogram. The discussion assumes no representation clauses have been used to alter the default representations of the types involved. Chapter 4 describes the effect of representation clauses on the representation of values.

Integer Types [3.5.4]

Ada integer types are represented in two's complement form and occupy a byte (`SHORT_INTEGER`) or a word (`INTEGER`).

Parameters to interfaced subprograms of type `SHORT_INTEGER` are passed by copy with the value sign extended to a full machine word. Values of type `INTEGER` are always passed by copy. The predefined type `LONG_INTEGER` is not available.

Enumeration Types [3.5.1]

Values of an Ada enumeration type are represented internally as unsigned values representing their position in the list of enumeration literals defining the type. The first literal in the list corresponds to a value of zero.

Enumeration types with 256 elements or fewer are represented in 8 bits. All other enumeration types are represented in 32 bits.

Consequently, the Ada predefined type `CHARACTER` [3.5.2] is represented in 8 bits, using the standard ASCII codes [C] and the Ada predefined type `BOOLEAN` [3.5.3] is represented in 8 bits, with `FALSE` represented by the value 0 and `TRUE` represented by the value 1.

As the representation of enumeration types is basically the same as that of integers, the same parameter passing conventions apply.

Floating Point Types [3.5.7, 3.5.8]

Ada floating-point values occupy 32 (FLOAT) or 64 (LONG_FLOAT) bits, and are held in ANSI/IEEE 754 floating point format.

Parameters to interfaced subprograms of type FLOAT are always passed by copy. Parameters of type LONG_FLOAT are passed by address.

Fixed Point Types [3.5.9, 3.5.10]

Ada fixed-point types are managed by the Compiler as the product of a signed *mantissa* and a constant *small*. The mantissa is implemented as an 8 or 32 bit integer value. *Small* is a compile-time quantity which is the power of two equal or immediately inferior to the delta specified in the declaration of the type.

The representation of an actual parameter of a fixed point type is the value of its mantissa. This is passed using the same rules as for integer types.

The attribute MANTISSA is defined as the smallest number such that:

$$2^{**} \text{MANTISSA} \geq \max(\text{abs}(\text{upper_bound}), \text{abs}(\text{lower_bound})) / \text{small}$$

The size of a fixed point type is:

<u>MANTISSA</u>	<u>Size</u>
1 .. 7	8 bits
8 .. 31	32 bits

Fixed point types requiring a MANTISSA greater than 31 are not supported.

Access Types [3.8]

Values of access types are represented internally by the address of the designated object held in single word. The value MIN_INT (the smallest integer that can be represented in a machine word) is used to represent *null*.

Array Types [3.6]

Ada arrays are passed by address; the value passed is the address of the first element of the first dimension of the array. The elements of the array are allocated by row. When an array is passed as a parameter to an interfaced subprogram, the usual consistency checking between the array bounds declared in the calling and the called subprogram is not enforced. It is the programmer's responsibility to ensure that the subprogram does not violate the bounds of the array.

When passing arrays to occam, it may be the case that some of its bounds are undefined in the source of the interfaced subprogram. If this is true, the missing bounds should be passed as extra integer value parameters to the subprogram. These parameters should be placed immediately following the array parameter itself and in the same order as the missing strides appear in the occam source.

Values of the predefined type `STRING` [3.6.3] are arrays, and are passed in the same way: the address of the first character in the string is passed. Elements of a string are represented in 8 bits, using the standard ASCII codes. The elements are packed into one or more words and occupy consecutive locations in memory.

Record Types [3.7]

Ada records are passed by address; the value passed is the address of the first component of the record. Components of a record are aligned on their natural boundaries (e.g. `INTEGER` on a word boundary) and the components may be re-ordered by the Compiler so as to minimize the total size of objects of the record type. If a record contains discriminants or components having a dynamic size, implicit components may be added to the record. Thus the default layout of the internal structure of the record may not be inferred directly from its Ada declaration. The use of a representation clause to control the layout of any record type whose values are to be passed to interfaced subprograms is recommended.

1.3.4 Restrictions on Interfaced Subprograms

Interfaced subprograms must be compiled using an error mode compatible with that used by the Ada runtime system to avoid errors when linking. For T4 and T8 mode S (`STOP`) or X (`UNIVERSAL`) should be used and for T9000 mode H (`HALT`) or X (`UNIVERSAL`) should be used.

There is no mechanism to allow runtime errors in interfaced subprograms to automatically raise an Ada exception. Interfaced subprograms should be written to explicitly deal with any errors likely to occur at runtime and return an error indication to the Ada program using an *out* parameter or a function result.

On T4 and T8, if the interfaced subprogram causes a process halt as a result of executing a `STOPP` or `STOPERR` instruction, the calling Ada task will become permanently blocked and will never terminate. On T9000 an interfaced subprogram is called with a null trap handler, so unless the subprogram installs its own handler any errors resulting in a trap will cause a process halt and will permanently block the calling Ada task.

In view of this, the use of runtime checking code added by the occam compiler is limited, and it is often convenient to suppress it using option U.

Parameters which are of a task or private type, or are access values not of mode *in*, should not be passed to interfaced subprograms.

It is not possible to interface to occam functions which have more than one return value. Unconstrained records and arrays cannot be returned from interfaced subprograms.

1.4 INTERFACE_NAME

Pragma `INTERFACE_NAME` associates the name of an interfaced subprogram, as declared in Ada, with its name in the language of origin. If pragma `INTERFACE_NAME` is not used the Ada name in uppercase is used as its name in the interfaced language.

This pragma takes the form:

```
pragma INTERFACE_NAME (subprogram_name, string_literal);
```

where:

- *subprogram_name* is the name used within the Ada program to refer to the interfaced subprogram.
- *string_literal* is the name by which the interfaced subprogram is referred to at link-time.

The use of `INTERFACE_NAME` is optional and is not needed if a subprogram has the same name in Ada as in the language of origin. It is necessary, for example, if the name of the subprogram in its original language contains characters that are not permitted in Ada identifiers or contains lowercase letters. Ada identifiers can contain only letters, digits and underscores, whereas the INMOS linker allows external names to contain other characters, for example full stops. These characters can be specified in the *string_literal* argument of the pragma `INTERFACE_NAME`.

The pragma `INTERFACE_NAME` is allowed at the same places of an Ada program as the pragma `INTERFACE` [13.9]. However, the pragma `INTERFACE_NAME` must always occur after the pragma `INTERFACE` declaration for the interfaced subprogram.

Example

```
package SAMPLE_DATA is
  function SAMPLE_DEVICE (X : INTEGER) return INTEGER;
  function PROCESS_SAMPLE (X : INTEGER) return INTEGER;
private
  pragma INTERFACE (OCCAM, SAMPLE_DEVICE);
  pragma INTERFACE (OCCAM, PROCESS_SAMPLE);
  pragma INTERFACE_NAME (PROCESS_SAMPLE, "process.sample");
end SAMPLE_DATA;
```

1.5 NO_IMAGE

The pragma `NO_IMAGE` takes the name of an enumeration type as argument. This pragma specifies to the compiler that the attributes `'IMAGE`, `'VALUE` or `'WIDTH` will never be used for this type, and that in consequence, no image table should be generated for this enumeration type. Any compilation unit containing an attribute `'IMAGE`, `'VALUE` or `'WIDTH` for a type on which a pragma `NO_IMAGE` was applied, will be rejected by the compiler.

The exception to this rule is that in the case where `'WIDTH` can be determined at compile-time (i.e. it is not applied to an enumeration subtype with dynamic bounds) its use is allowed with the pragma `NO_IMAGE`.

Example:

```
package MY_PACKAGE is
    type ENUM is (FIRST, SECOND, THIRD);
    pragma NO_IMAGE(ENUM);
    .....
end MY_PACKAGE;
```

Limitations on the use of pragma `NO_IMAGE`

- This pragma must occur in a declarative part and can be applied only to types declared in this same declarative part.

1.6 INDENT

This pragma is only used with the Alsys Reformatter (*AdaReformat*); this tool offers the functionalities of a source reformatter in an Ada environment.

The pragma is placed in the source file and interpreted by the Reformatter.

```
pragma INDENT(OFF)
```

The Reformatter does not modify the source lines after the `OFF` pragma `INDENT`.

```
pragma INDENT(ON)
```

The Reformatter resumes its action after the ON pragma INDENT. Therefore any source lines that are bracketed by the OFF and ON pragma INDENTs are not modified by the Alsys Reformatter.

1.7 Other Pragmas

Pragmas IMPROVE and PACK are discussed in detail in the section on representation clauses (Chapter 4).

Pragmas STORAGE_SIZE_RATIO and FAST_PRIMARY which are applicable only to task types are discussed in detail in section 4.6.

Pragma PRIORITY is accepted with the range of priorities running from 1 to 10 (see the definition of the predefined package SYSTEM in Chapter 3). The undefined priority (no pragma PRIORITY) is treated as though it were less than any defined priority value.

In addition to pragma SUPPRESS, it is possible to suppress checks in a given compilation by the use of the Compiler option CHECKS.

The following language defined pragmas have no effect.

CONTROLLED
MEMORY_SIZE
OPTIMIZE
STORAGE_UNIT
SYSTEM_NAME

Note that all access types are implemented by default as controlled collections as described in [4.8].

TABLE OF CONTENTS

INTRODUCTION	1
1 Implementation-Dependent Pragmas	3
1.1 INLINE	3
1.2 INLINE_GENERIC	3
1.3 INTERFACE	5
1.3.1 Calling Conventions	5
1.3.2 Parameter-Passing Conventions	6
1.3.3 Parameter Representations	7
1.3.4 Restrictions on Interfaced Subprograms	9
1.4 INTERFACE_NAME	11
1.5 NO_IMAGE	12
1.6 INDENT	12
1.7 Other Pragmas	13
2 Implementation-Dependent Attributes	15
3 Specification of the Package SYSTEM	17
4 Restrictions on Representation Clauses	21
4.1 Enumeration Types	22
4.2 Integer Types	25
4.3 Floating Point Types	28
4.4 Fixed Point Types	30
4.5 Access Types	33
4.6 Task Types	34
4.7 Array Types	36
4.8 Record Types	40

5	Conventions for Implementation-Generated Names	51
6	Address Clauses	53
6.1	Address Clauses for Objects	53
6.2	Address Clauses for Program Units	53
6.3	Address Clauses for Entries	53
7	Restrictions on Unchecked Conversions	55
8	Input-Output Packages	57
8.1	NAME Parameter	57
8.2	FORM Parameter	57
8.2.1	File Sharing	58
8.2.2	Binary Files	59
8.2.3	Buffering	60
8.2.4	Appending	61
8.3	USE_ERROR	61
9	Characteristics of Numeric Types	63
9.1	Integer Types	63
9.2	Floating Point Type Attributes	64
9.3	Attributes of Type DURATION	65
	REFERENCES	67
	INDEX	69

INTRODUCTION

Implementation-Dependent Characteristics

This appendix summarizes the implementation-dependent characteristics of the Alsys Ada Compilers for INMOS transputers. This document should be considered as the Appendix F to the Reference Manual for the Ada Programming Language ANSI/MIL-STD 1815A, January 1983, as appropriate to the Alsys Ada implementation for the transputer.

Sections 1 to 8 of this appendix correspond to the various items of information required in Appendix F [F]*; sections 9 and 10 provide other information relevant to the Alsys implementation. The contents of these sections is described below:

1. The form, allowed places, and effect of every implementation-dependent pragma.
2. The name and type of every implementation-dependent attribute.
3. The specification of the package SYSTEM [13.7].
4. The list of all restrictions on representation clauses [13.1].
5. The conventions used for any implementation-generated names denoting implementation-dependent components [13.4].
6. The interpretation of expressions that appear in address clauses.
7. Any restrictions on unchecked conversions [13.10.2].
8. Any implementation-dependent characteristics of the input-output packages [14].
9. Characteristics of numeric types.

* Throughout this manual, citations in square brackets refer to the *Reference Manual for the Ada Programming Language*, ANSI/MIL-STD-1815A, January 1983.

Throughout this appendix, the name *Ada Run-Time Executive* refers to the run-time library routines provided for all Ada programs. These routines implement the Ada heap, exceptions, tasking control, I/O, and other utility functions.

CHAPTER 1

Implementation-Dependent Pragmas

1.1 INLINE

Pragma `INLINE` is fully supported, except for the fact that it is not possible to inline a function call in a declarative part.

1.2 INLINE_GENERIC

The pragma `INLINE_GENERIC` takes the name of a generic unit or of an instance of a generic unit as argument.

When the argument is the name of a generic unit, pragma `INLINE_GENERIC` specifies that all the instances of the generic unit will be generated in line (and not in a subunit) regardless of the value of the option `GENERIC`s of the command `COMPILE`.

When the argument is the name of an instance of a generic unit, pragma `INLINE_GENERIC` specifies that this instance will be generated in line (and not in a subunit) regardless of the value of the option `GENERIC`s of the command `COMPILE`.

```
pragma INLINE_GENERIC (Ada_designator {, Ada_designator })
```

Example:

```
procedure MY_PROCEDURE is

  type MY_TYPE is (...);

  generic
    procedure MY_FIRST_GENERIC;
  generic
    type T is (<>);
  function MY_SECOND_GENERIC(L,R : T) return T;
```

```

function MY_SECOND_GENERIC(L,R : T) return T is
begin
    ...
end;

function "+" is new MY_SECOND_GENERIC(MY_TYPE);

pragma INLINE_GENERIC(MY_FIRST_GENERIC, "+");

procedure MY_FIRST_GENERIC is
begin
    ...
end;

.....

end MY_PROCEDURE ;

```

Limitations on the use of pragma INLINE_GENERIC

- This pragma is only allowed at the place of a declarative item in a declarative part or package specification, or after a library unit in a compilation but before any subsequent compilation unit. If the pragma appears at the place of a declarative item, each argument must denote a generic unit, or an instance of a generic unit, declared by an earlier declarative item of the same declarative part or package specification. If several (overloaded) subprograms satisfy this requirement, the pragma applies to all of them. If the pragma appears after a given library unit, the only name allowed is the name of this unit, which must be a generic unit.
- If an instance cannot be generated in line, in spite of a valid pragma `INLINE_GENERIC`, a warning will be emitted and the instance will be generated in a subunit. This will occur in particular when the body of the generic unit has not been compiled at the point of instantiation. This will also happen on instances appearing within the specification of a generic package (due to a restriction of the Alsys implementation).

1.3 INTERFACE

Ada programs can interface to subprograms written in occam through the use of the predefined pragma `INTERFACE` [13.9] and the implementation-defined pragma `INTERFACE_NAME`.

Pragma `INTERFACE` specifies the name of an interfaced subprogram and the name of the programming language for which calling and parameter passing conventions will be generated. Pragma `INTERFACE` takes the form specified in the *Reference Manual*:

```
pragma INTERFACE (language_name, subprogram_name);
```

where:

- *language_name* is the name of the other language whose calling and parameter passing conventions are to be used.
- *subprogram_name* is the name used within the Ada program to refer to the interfaced subprogram.

The language names currently accepted by pragma `INTERFACE` are `OCCAM` and `OCCAM_HIGH`.

The language name used in the pragma `INTERFACE` does not necessarily correspond to the language used to write the interfaced subprogram. It is used only to tell the Compiler how to generate subprogram calls, that is, which calling conventions and parameter passing techniques to use.

The language name `OCCAM` is used to refer to the standard occam calling and parameter passing conventions for the transputer (Ref. 4, Section 5.10). The programmer can use the language name `OCCAM` to interface Ada subprograms to subroutines written in any language that follows the standard occam calling conventions.

The programmer can use the language name `OCCAM_HIGH` to run the interfaced subroutine as a high priority process instead of a low priority process. The `OCCAM_HIGH` interface is otherwise identical to the `OCCAM` interface.

1.3.1 Calling Conventions

The following calling conventions are required for code to be called from Ada by use of pragma `INTERFACE`.

On entry to the subprogram, the registers A, B and C are undefined. For the T8 and T9000, the floating point registers FA, FB and FC are similarly undefined. The return address and any parameters are accessed relative to the workspace pointer, W, by the subprogram.

There are no assumptions concerning the contents of the register stacks (A, B, C and FA, FB, FC) upon return from the interfaced subprogram, other than for interfaced subprograms which are functions (see below). However, the workspace pointer, W, should contain the same address upon return from the interfaced subprogram as it contained before the call.

On the T4 and T8 the setting of the error flag is ignored on return.

1.3.2 Parameter-Passing Conventions

On entry to the subprogram, the word at offset 0 from the transputer workspace (W) pointer contains the return address of the called subprogram. Subsequent workspace locations (from W + 1 to W + n, where n is the number of parameters) contain the subprogram parameters, which are all one word in length.

There is always an implicit vector space parameter passed as the last parameter to all interfaced subprograms. This points to an area of free memory which can be used by the occam compiler to allocate arrays declared in the interfaced subprogram.

Actual parameters of mode *in* which are access values or scalars of one machine word or less in size are passed by copy. If such a parameter is less than one machine word in length it is sign extended to a full word. For all other parameters of mode *in*, and all parameters of mode *in out* or *out*, the value passed is the address of the actual parameter itself.

Since all large scalar, non-scalar and non-access parameters to interfaced subprograms are passed by address, they cannot be protected from modification by the called subprogram even though they may be formally declared to be of mode *in*. It is the programmer's responsibility to ensure that the semantics of the Ada parameter modes are honoured in these cases.

If the interfaced subprogram is a function, the result is returned as follows:

A floating point result is returned in register FPA.

Any other result is returned by value in register A if its size is at most one machine word and by address in register A otherwise.

No consistency checking is performed between the subprogram parameters declared in Ada and the corresponding parameters of the interfaced subprogram. It is the programmer's responsibility to ensure correct access to the parameters.

1.3.3 Parameter Representations

This section describes the representation of values of the types that can be passed as parameters to an interfaced subprogram. The discussion assumes no representation clauses have been used to alter the default representations of the types involved. Chapter 4 describes the effect of representation clauses on the representation of values.

Integer Types [3.5.4]

Ada integer types are represented in two's complement form and occupy a byte (SHORT_INTEGER) or a word (INTEGER).

Parameters to interfaced subprograms of type SHORT_INTEGER are passed by copy with the value sign extended to a full machine word. Values of type INTEGER are always passed by copy. The predefined type LONG_INTEGER is not available.

Enumeration Types [3.5.1]

Values of an Ada enumeration type are represented internally as unsigned values representing their position in the list of enumeration literals defining the type. The first literal in the list corresponds to a value of zero.

Enumeration types with 256 elements or fewer are represented in 8 bits. All other enumeration types are represented in 32 bits.

Consequently, the Ada predefined type CHARACTER [3.5.2] is represented in 8 bits, using the standard ASCII codes [C] and the Ada predefined type BOOLEAN [3.5.3] is represented in 8 bits, with FALSE represented by the value 0 and TRUE represented by the value 1.

As the representation of enumeration types is basically the same as that of integers, the same parameter passing conventions apply.

Floating Point Types [3.5.7, 3.5.8]

Ada floating-point values occupy 32 (FLOAT) or 64 (LONG_FLOAT) bits, and are held in ANSI/IEEE 754 floating point format.

Parameters to interfaced subprograms of type FLOAT are always passed by copy. Parameters of type LONG_FLOAT are passed by address.

Fixed Point Types [3.5.9, 3.5.10]

Ada fixed-point types are managed by the Compiler as the product of a signed *mantissa* and a constant *small*. The mantissa is implemented as an 8 or 32 bit integer value. *Small* is a compile-time quantity which is the power of two equal or immediately inferior to the delta specified in the declaration of the type.

The representation of an actual parameter of a fixed point type is the value of its mantissa. This is passed using the same rules as for integer types.

The attribute MANTISSA is defined as the smallest number such that:

$$2^{**} \text{MANTISSA} \geq \max(\text{abs}(\text{upper_bound}), \text{abs}(\text{lower_bound})) / \text{small}$$

The size of a fixed point type is:

<u>MANTISSA</u>	<u>Size</u>
1 .. 7	8 bits
8 .. 31	32 bits

Fixed point types requiring a MANTISSA greater than 31 are not supported.

Access Types [3.8]

Values of access types are represented internally by the address of the designated object held in single word. The value MIN_INT (the smallest integer that can be represented in a machine word) is used to represent *null*.

Array Types [3.6]

Ada arrays are passed by address; the value passed is the address of the first element of the first dimension of the array. The elements of the array are allocated by row. When an array is passed as a parameter to an interfaced subprogram, the usual consistency checking between the array bounds declared in the calling and the called subprogram is not enforced. It is the programmer's responsibility to ensure that the subprogram does not violate the bounds of the array.

When passing arrays to occam, it may be the case that some of its bounds are undefined in the source of the interfaced subprogram. If this is true, the missing bounds should be passed as extra integer value parameters to the subprogram. These parameters should be placed immediately following the array parameter itself and in the same order as the missing strides appear in the occam source.

Values of the predefined type `STRING` [3.6.3] are arrays, and are passed in the same way: the address of the first character in the string is passed. Elements of a string are represented in 8 bits, using the standard ASCII codes. The elements are packed into one or more words and occupy consecutive locations in memory.

Record Types [3.7]

Ada records are passed by address; the value passed is the address of the first component of the record. Components of a record are aligned on their natural boundaries (e.g. `INTEGER` on a word boundary) and the components may be re-ordered by the Compiler so as to minimize the total size of objects of the record type. If a record contains discriminants or components having a dynamic size, implicit components may be added to the record. Thus the default layout of the internal structure of the record may not be inferred directly from its Ada declaration. The use of a representation clause to control the layout of any record type whose values are to be passed to interfaced subprograms is recommended.

1.3.4 Restrictions on Interfaced Subprograms

Interfaced subprograms must be compiled using an error mode compatible with that used by the Ada runtime system to avoid errors when linking. For T4 and T8 mode S (STOP) or X (UNIVERSAL) should be used and for T9000 mode H (HALT) or X (UNIVERSAL) should be used.

There is no mechanism to allow runtime errors in interfaced subprograms to automatically raise an Ada exception. Interfaced subprograms should be written to explicitly deal with any errors likely to occur at runtime and return an error indication to the Ada program using an *out* parameter or a function result.

On T4 and T8, if the interfaced subprogram causes a process halt as a result of executing a `STOPP` or `STOPERR` instruction, the calling Ada task will become permanently blocked and will never terminate. On T9000 an interfaced subprogram is called with a null trap handler, so unless the subprogram installs its own handler any errors resulting in a trap will cause a process halt and will permanently block the calling Ada task.

In view of this, the use of runtime checking code added by the occam compiler is limited, and it is often convenient to suppress it using option U.

Parameters which are of a task or private type, or are access values not of mode *in*, should not be passed to interfaced subprograms.

It is not possible to interface to occam functions which have more than one return value. Unconstrained records and arrays cannot be returned from interfaced subprograms.

1.4 INTERFACE_NAME

Pragma `INTERFACE_NAME` associates the name of an interfaced subprogram, as declared in Ada, with its name in the language of origin. If pragma `INTERFACE_NAME` is not used the Ada name in uppercase is used as its name in the interfaced language.

This pragma takes the form:

```
pragma INTERFACE_NAME (subprogram_name, string_literal);
```

where:

- *subprogram_name* is the name used within the Ada program to refer to the interfaced subprogram.
- *string_literal* is the name by which the interfaced subprogram is referred to at link-time.

The use of `INTERFACE_NAME` is optional and is not needed if a subprogram has the same name in Ada as in the language of origin. It is necessary, for example, if the name of the subprogram in its original language contains characters that are not permitted in Ada identifiers or contains lowercase letters. Ada identifiers can contain only letters, digits and underscores, whereas the INMOS linker allows external names to contain other characters, for example full stops. These characters can be specified in the *string_literal* argument of the pragma `INTERFACE_NAME`.

The pragma `INTERFACE_NAME` is allowed at the same places of an Ada program as the pragma `INTERFACE` [13.9]. However, the pragma `INTERFACE_NAME` must always occur after the pragma `INTERFACE` declaration for the interfaced subprogram.

Example

```
package SAMPLE_DATA is
  function SAMPLE_DEVICE (X : INTEGER) return INTEGER;
  function PROCESS_SAMPLE (X : INTEGER) return INTEGER;
private
  pragma INTERFACE (OCCAM, SAMPLE_DEVICE);
  pragma INTERFACE (OCCAM, PROCESS_SAMPLE);
  pragma INTERFACE_NAME (PROCESS_SAMPLE, "process.sample");
end SAMPLE_DATA;
```

1.5 NO_IMAGE

The pragma `NO_IMAGE` takes the name of an enumeration type as argument. This pragma specifies to the compiler that the attributes `'IMAGE`, `'VALUE` or `'WIDTH` will never be used for this type, and that in consequence, no image table should be generated for this enumeration type. Any compilation unit containing an attribute `'IMAGE`, `'VALUE` or `'WIDTH` for a type on which a pragma `NO_IMAGE` was applied, will be rejected by the compiler.

The exception to this rule is that in the case where `'WIDTH` can be determined at compile-time (i.e. it is not applied to an enumeration subtype with dynamic bounds) its use is allowed with the pragma `NO_IMAGE`.

Example:

```
package MY_PACKAGE is
    type ENUM is (FIRST, SECOND, THIRD);
    pragma NO_IMAGE(ENUM);
    .....
end MY_PACKAGE;
```

Limitations on the use of pragma `NO_IMAGE`

- This pragma must occur in a declarative part and can be applied only to types declared in this same declarative part.

1.6 INDENT

This pragma is only used with the Alsys Reformatter (*AdaReformat*); this tool offers the functionalities of a source reformatter in an Ada environment.

The pragma is placed in the source file and interpreted by the Reformatter.

```
pragma INDENT(OFF)
```

The Reformatter does not modify the source lines after the `OFF` pragma `INDENT`.

```
pragma INDENT(ON)
```

The Reformatter resumes its action after the ON pragma INDENT. Therefore any source lines that are bracketed by the OFF and ON pragma INDENTs are not modified by the Alsys Reformatter.

1.7 Other Pragmas

Pragmas IMPROVE and PACK are discussed in detail in the section on representation clauses (Chapter 4).

Pragmas STORAGE_SIZE_RATIO and FAST_PRIMARY which are applicable only to task types are discussed in detail in section 4.6.

Pragma PRIORITY is accepted with the range of priorities running from 1 to 10 (see the definition of the predefined package SYSTEM in Chapter 3). The undefined priority (no pragma PRIORITY) is treated as though it were less than any defined priority value.

In addition to pragma SUPPRESS, it is possible to suppress checks in a given compilation by the use of the Compiler option CHECKS.

The following language defined pragmas have no effect.

CONTROLLED
MEMORY_SIZE
OPTIMIZE
STORAGE_UNIT
SYSTEM_NAME

Note that all access types are implemented by default as controlled collections as described in [4.8].

CHAPTER 2

Implementation-Dependent Attributes

In addition to the Representation Attributes of [13.7.2] and [13.7.3], the attributes listed in section 5 (Conventions for Implementation-Generated Names), for use in record representation clauses, and the attributes described below are provided:

<code>TDESCRIPTOR_SIZE</code>	For a prefix <code>T</code> that denotes a type or subtype, this attribute yields the size (in bits) required to hold a descriptor for an object of the type <code>T</code> , allocated on the heap or written to a file. If <code>T</code> is constrained, <code>TDESCRIPTOR_SIZE</code> will yield the value 0.
<code>TIS_ARRAY</code>	For a prefix <code>T</code> that denotes a type or subtype, this attribute yields the value <code>TRUE</code> if <code>T</code> denotes an array type or an array subtype; otherwise, it yields the value <code>FALSE</code> .

Limitations on the use of the attribute `ADDRESS`

The attribute `ADDRESS` is implemented for all prefixes that have meaningful addresses. The following entities do not have meaningful addresses. The attribute `ADDRESS` will deliver the value `SYSTEM.NULL_ADDRESS` if applied to such prefixes and a compilation warning will be issued.

- A constant or named number that is implemented as an immediate value (i.e. does not have any space allocated for it).
- A package specification that is not a library unit.
- A package body that is not a library unit or subunit.
- A function that renames an enumeration literal.

If the attribute `ADDRESS` is applied to a named number, a compilation error will be produced.

CHAPTER 3

Specification of the Package SYSTEM

package SYSTEM is

```
type NAME is (I80X86,  
              I80386,  
              MC680X0,  
              S370,  
              TRANSPUTER,  
              VAX,  
              RS_6000,  
              MIPS,  
              HP9000_PA_RISC,  
              SPARC);
```

```
SYSTEM_NAME : constant NAME := TRANSPUTER;
```

```
STORAGE_UNIT : constant := 8;  
MAX_INT       : constant := 2**31 - 1;  
MIN_INT       : constant := - (2**31);  
MAX_MANTISSA  : constant := 31;  
FINE_DELTA    : constant := 2#1.0#E-31;  
MAX_DIGITS    : constant := 15;  
MEMORY_SIZE  : constant := 2**32;  
TICK          : constant := 1.0E-6;
```

```
subtype PRIORITY is INTEGER range 1 .. 10;
```

```
type ADDRESS is private;  
NULL_ADDRESS : constant ADDRESS;
```

```
function VALUE (LEFT : in STRING) return ADDRESS;
```

```
subtype ADDRESS_STRING is STRING(1..8);
```

```
function IMAGE (LEFT : in ADDRESS) return ADDRESS_STRING;
```

```
type OFFSET is range -(2**31) .. 2**31-1;  
-- This type is used to measure a number of storage units (bytes).
```

```
function SAME_SEGMENT (LEFT, RIGHT : in ADDRESS) return BOOLEAN;
```

```
ADDRESS_ERROR : exception;
```



```

function "+" (LEFT : in ADDRESS; RIGHT : in OFFSET) return ADDRESS;
function "+" (LEFT : in OFFSET; RIGHT : in ADDRESS) return ADDRESS;
function "-" (LEFT : in ADDRESS; RIGHT : in OFFSET) return ADDRESS;

function "-" (LEFT : in ADDRESS; RIGHT : in ADDRESS) return OFFSET;

function "<=" (LEFT, RIGHT : in ADDRESS) return BOOLEAN;
function "<" (LEFT, RIGHT : in ADDRESS) return BOOLEAN;
function ">=" (LEFT, RIGHT : in ADDRESS) return BOOLEAN;
function ">" (LEFT, RIGHT : in ADDRESS) return BOOLEAN;

function "mod" (LEFT : in ADDRESS; RIGHT : in POSITIVE) return NATURAL;

type ROUND_DIRECTION is (DOWN, UP);

function ROUND (VALUE      : in ADDRESS;
                DIRECTION : in ROUND_DIRECTION;
                MODULUS    : in POSITIVE) return ADDRESS;

generic
  type TARGET is private;
function FETCH_FROM_ADDRESS (A : in ADDRESS) return TARGET;
generic
  type TARGET is private;
procedure ASSIGN_TO_ADDRESS (A : in ADDRESS; T : in TARGET);
-- These routines are provided to perform READ/WRITE operations in memory.

type OBJECT_LENGTH is range 0 .. 2**31 -1;
-- This type is used to designate the size of an object in storage units.

procedure MOVE (TO      : in ADDRESS;
                FROM    : in ADDRESS;
                LENGTH  : in OBJECT_LENGTH);

end SYSTEM;

```

The function VALUE may be used to convert a string into an address. The string is a sequence of up to eight hexadecimal characters (digits or letters in upper or lower case in the range A..F) representing the address. The exception CONSTRAINT_ERROR is raised if the string does not have the proper syntax.

The function IMAGE may be used to convert an address to a string which is a sequence of exactly eight hexadecimal digits.

CHAPTER 1 INTRODUCTION

The Ada implementation described above was tested according to the Ada Validation Procedures [Pro92] against the Ada Standard [Ada83] using the current Ada Compiler Validation Capability (ACVC). This Validation Summary Report (VSR) gives an account of the testing of this Ada implementation. For any technical terms used in this report, the reader is referred to [Pro92]. A detailed description of the ACVC may be found in the current ACVC User's Guide [UG89].

1.1 USE OF THIS VALIDATION SUMMARY REPORT

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

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

National Technical Information Service
5285 Port Royal Road
Springfield 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
Computer and Software Engineering Division
Institute for Defense Analyses
1801 North Beauregard Street
Alexandria VA 22311-1772

1.2 REFERENCES

- [Ada83] Reference Manual for the Ada Programming Language.
ANSI/MIL-STD-1815A, February 1983 and ISO 8652-1987.
- [Pro92] Ada Compiler Validation Procedures.
Version 3.1, Ada Joint Program Office, August 1992.
- [UG89] Ada Compiler Validation Capability User's Guide.
21 June 1989.

1.3 ACVC TEST CLASSES

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

The executable tests are written in a self-checking manner and produce a PASSED, FAILED, or NOT APPLICABLE message indicating the result when they are executed. Three Ada library units, the packages REPORT and 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 behaviour 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 removing some inapplicable tests (see section 2.2 and [UG89]).

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

1.4 DEFINITION OF TERMS

Ada Compiler	The software and any needed hardware that have to be added to a given host and target computer system to allow transformation of Ada programs into executable form and execution thereof.
Ada Compiler Validation Capability (ACVC)	The means for testing compliance of Ada implementations, consisting of the test suite, the support programs, the ACVC user's guide and the template for the validation summary report.
Ada Implementation	An Ada compiler with its host computer system and its target computer system.
Ada Joint Program Office (AJPO)	The part of the certification body which provides policy and guidance for the Ada certification system.
Ada Validation Facility (AVF)	The part of the certification body which carries out the procedures required to establish the compliance of an Ada implementation.
Ada Validation Organization (AVO)	The part of the certification body that provides technical guidance for operations of the Ada certification system.
Compliance of an Ada Implementation	The ability of the implementation to pass an ACVC version.
Computer System	A functional unit, consisting of one or more computers and associated software, that uses common storage for all or part of a program and also for all or part of the data necessary for the execution of the program; executes user-written or user-designated programs; performs user-designated data manipulation, including arithmetic operations and logic operations; and that can execute programs that modify themselves during execution. A computer system may be a stand-alone unit or may consist of several inter-connected units.
Conformity	Fulfilment of a product, process or service of all requirements specified.
Customer	An individual or corporate entity who enters into an agreement with an AVF which specifies the terms and conditions for AVF services (of any kind) to be performed.
Declaration of Conformance	A formal statement from a customer assuring that conformity is realized or attainable on the Ada implementation for which validation status is realized.
Host Computer System	A computer system where Ada source programs are transformed into executable form.
Inapplicable test	A test that contains one or more test objectives found to be irrelevant for the given Ada implementation.
ISO	International Organization for Standardization.

LRM	The Ada standard, or Language Reference Manual, published as ANSI/MIL-STD-1815A-1983 AND ISO 8652-1987. Citations from the LRM take the form "<section>.<subsection>.<paragraph>."
Operating System	Software that controls the execution of programs and that provides services such as resource allocation, scheduling, input/output control and data management. Usually, operating systems are predominantly software, but partial or complete hardware implementations are possible.
Target Computer System	A computer system where the executable form of Ada programs are executed.
Validated Ada Compiler	The compiler of a validated Ada implementation.
Validated Ada Implementation	An Ada implementation that has been validated successfully either by AVF testing or by registration [Pro92].
Validation	The process of checking the conformity of an Ada compiler to the Ada programming language and of issuing a certificate for this implementation.
Withdrawn test	A test found to be incorrect and not used in conformity testing. A test may be incorrect because it has an invalid test objective, fails to meet its test objective, or contains erroneous or illegal use of the Ada programming language.

CHAPTER 2 IMPLEMENTATION DEPENDENCIES
2.1 WITHDRAWN TESTS

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 22 November 1993.

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

2.2 INAPPLICABLE TESTS

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

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

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

For example:

type COLOR is (GREEN, BLACK, WHITE, RED, BLUE, YELLOW);
-- The minimum size of COLOR is 3 bits.

subtype BLACK_AND_WHITE is COLOR **range** BLACK .. WHITE;
-- The minimum size of BLACK_AND_WHITE is 2 bits.

subtype BLACK_OR_WHITE is BLACK_AND_WHITE **range** X .. X;
-- Assuming that X is not static, the minimum size of BLACK_OR_WHITE is
-- 2 bits (the same as the minimum size of the static type mark
-- BLACK_AND_WHITE).

Size: When no size specification is applied to an enumeration type or first named subtype (if any), the size of that type or first named subtype is the smallest of 32, 8, 4, 2 or 1 bit which is equal to or greater than the minimum size for the type or first named type.

When a size specification is applied to an enumeration type, this enumeration type and each of its subtypes has the size specified by the length clause. The same rule applies to a first named subtype. The size specification must of course specify a value greater than or equal to the minimum size of the type or subtype to which it applies.

For example:

type EXTENDED is
(-- The usual American ASCII characters.

NUL.	SOH.	STX.	ETX.	EOT.	ENQ.	ACK.	BEL.
BS.	HT.	LF.	VT.	FF.	CR.	SO.	SI.
DLE.	DC1.	DC2.	DC3.	DC4.	NAK.	SYN.	ETB.
CAN.	EM.	SUB.	ESC.	FS.	GS.	RS.	US.
'.'	'.'	'.'	'#'	'\$'	'%'	'&'	'.'
'('	')'	'*'	'+'	'.'	'.'	'.'	'/'
'0'	'1'	'2'	'3'	'4'	'5'	'6'	'7'
'8'	'9'	'.'	'.'	'<'	'='	'>'	'?'
'@'	'A'	'B'	'C'	'D'	'E'	'F'	'G'
'H'	'I'	'J'	'K'	'L'	'M'	'N'	'O'
'P'	'Q'	'R'	'S'	'T'	'U'	'V'	'W'
'X'	'Y'	'Z'	'['	'\'	']'	'^'	'_'
'`'	'a'	'b'	'c'	'd'	'e'	'f'	'g'
'h'	'i'	'j'	'k'	'l'	'm'	'n'	'o'
'p'	'q'	'r'	's'	't'	'u'	'v'	'w'
'x'	'y'	'z'	'{'	' '	'}'	'~'	DEL.


```
-- Extended characters
LEFT_ARROW.
RIGHT_ARROW.
UPPER_ARROW.
LOWER_ARROW.
UPPER_LEFT_CORNER.
UPPER_RIGHT_CORNER.
LOWER_RIGHT_CORNER.
LOWER_LEFT_CORNER.
...):
```

```
for EXTENDED_SIZE use 8;
-- The size of type EXTENDED will be one byte. Its objects will be represented
-- as unsigned 8 bit integers.
```

The Alsys Compiler fully implements size specifications. Nevertheless, as enumeration values are coded using integers, the specified length cannot be greater than 32 bits.

Object size: Provided its size is not constrained by a record component clause or a pragma PACK, the size of an object of an enumeration subtype is determined as follows:

When no size specification is applied to the enumeration type or its first named subtype (if any), the objects of that type or first named subtype are represented as either unsigned bytes or signed words. The compiler selects automatically the smallest such object which can hold each of the internal codes of the enumeration type (or subtype). The size of objects of the enumeration type, and of any of its subtypes, is thus 8 bits in the case of an unsigned byte, or the machine size (32 bits) in the case of signed word.

When a size specification is applied to an enumeration type or its first named subtype, objects of this enumeration type, or of any of its subtypes, have the size specified by the length clause.

Alignment: An enumeration subtype is byte aligned if the size of the subtype is less than or equal to 8 bits, word aligned otherwise.

Object address: Provided its alignment is not constrained by a record representation clause or a pragma PACK, the address of an object of an enumeration subtype is a multiple of the alignment of the corresponding subtype.

4.2 Integer Types

Predefined integer types

In the Alsys Ada implementation for the transputer the following predefined integer types are available:

```
type SHORT_INTEGER is range -2**07 .. 2**07-1;
type INTEGER       is range -2**31 .. 2**31-1;
```

Selection of the parent of an integer type

An integer type declared by a declaration of the form:

```
type T is range L .. R;
```

is implicitly derived from one of the predefined integer types. The compiler automatically selects the predefined integer type whose range is the shortest that contains the values L to R inclusive.

Encoding of integer values

Binary code is used to represent integer values, using a conventional two's complement representation.

Integer subtypes

Minimum size: The minimum size of an integer subtype is the minimum number of bits that is necessary for representing the internal codes of the subtype values in normal binary form (that is to say, in an unbiased form which includes a sign bit only if the range of the subtype includes negative values).

For a static subtype, if it has a null range its minimum size is 1. Otherwise, if m and M are the lower and upper bounds of the subtype, then its minimum size L is determined as follows. For $m \geq 0$, L is the smallest positive integer such that $M \leq 2^L - 1$. For $m < 0$, L is the smallest positive integer such that $-2^{L-1} \leq m$ and $M \leq 2^{L-1} - 1$.

For example:

subtype S is INTEGER range 0 .. 7;
-- The minimum size of S is 3 bits.

subtype D is S range X .. Y;
-- Assuming that X and Y are not static, the minimum size of
-- D is 3 bits (the same as the minimum size of the static type mark S).

Size: The sizes of the predefined integer types `SHORT_INTEGER` and `INTEGER` are respectively 8 and 32 bits.

When no size specification is applied to an integer type or to its first named subtype (if any), its size and the size of any subtypes is the smallest of 32, 8, 4, 2 or 1 bit which is equal to or greater than the minimum size for the type or first named type.

For example:

type S is range 80 .. 100;
-- S is derived from `SHORT_INTEGER`, its size is 8 bits.

type J is range 0 .. 65535;
-- J is derived from `INTEGER`, its size is 32 bits.

type N is new J range 80 .. 100;
-- N is indirectly derived from `INTEGER`, its size is 32 bits.

When a size specification is applied to an integer type, this integer type and each of its subtypes has the size specified by the length clause. The same rule applies to a first named subtype. The size specification must of course specify a value greater than or equal to the minimum size of the type or subtype to which it applies.

For example:

type S is range 80 .. 100;
for S'SIZE use 32;
-- S is derived from `SHORT_INTEGER`, but its size is 32 bits
-- because of the size specification.

type J is range 0 .. 255;
for J'SIZE use 8;
-- J is derived from `INTEGER`, but its size is 8 bits because
-- of the size specification.

```
type N is new J range 80 .. 100;
-- N is indirectly derived from INTEGER, but its size is 8 bits
-- because N inherits the size specification of J.
```

The Alsys Compiler implements size specifications. Nevertheless, as integers are implemented using machine integers, the specified length cannot be greater than 32 bits.

Object size: Provided its size is not constrained by a record component clause or a pragma PACK, the size of an object of an integer subtype is determined as follows:

When no size specification is applied to the integer type or its first named subtype (if any), the objects this type, or any of its subtypes, have the size of the predefined type from which it derives directly or indirectly.

When a size specification is applied to an integer type or its first named subtype, objects of this integer type, or of any of its subtypes, have the size specified by the length clause.

Alignment: An integer subtype is byte aligned if the size of the subtype is less than or equal to 8 bits, word aligned otherwise.

Object address: Provided its alignment is not constrained by a record representation clause or a pragma PACK, the address of an object of an integer subtype is a multiple of the alignment of the corresponding subtype.

4.3 Floating Point Types

Predefined floating point types

There are two predefined floating point types in the Alsys implementation for transputers:

Their characteristics are:

FLOAT:	
FLOAT'DIGITS	= 6
FLOAT'FIRST	= $-(2.0 - 2.0^{**}(-23)) * 2.0^{**}127$
FLOAT'LAST	= $(2.0 - 2.0^{**}(-23)) * 2.0^{**}127$
FLOAT'MACHINE_MANTISSA	= 24
LONG_FLOAT:	
LONG_FLOAT'DIGITS	= 15
LONG_FLOAT'FIRST	= $-(2.0 - 2.0^{**}(-52)) * 2.0^{**}1023$
LONG_FLOAT'LAST	= $(2.0 - 2.0^{**}(-52)) * 2.0^{**}1023$
LONG_FLOAT'MACHINE_MANTISSA	= 53

Selection of the parent of a floating point type

A floating point type declared by a declaration of the form:

type T is digits D [range L .. R];

is implicitly derived from a predefined floating point type. The Compiler automatically selects the smallest predefined floating point type whose number of digits is greater than or equal to D and which contains the values L and R.

Encoding of floating point values

In the program generated by the Compiler, floating point values are represented using the ANSI/IEEE 754 standard 32-bit and 64-bit floating point formats as appropriate.

Values of the predefined type **FLOAT** are represented using the 32-bit floating point format and values of the predefined type **LONG_FLOAT** are represented using the 64-bit floating point format as defined by the standard. The values of any other floating point type are represented in the same way as the values of the predefined type from which it derives, directly or indirectly.

Floating point subtypes

Minimum size: The minimum size of a floating point subtype is 32 bits if its base type is `FLOAT` or a type derived from `FLOAT` and 64 bits if its base type is `LONG_FLOAT` or a type derived from `LONG_FLOAT`.

Size: The sizes of the predefined floating point types `FLOAT` and `LONG_FLOAT` are respectively 32 and 64 bits.

The size of a floating point type and the size of any of its subtypes is the size of the predefined type from which it derives directly or indirectly.

The only size that can be specified for a floating point type or first named subtype using a size specification is its usual size (32, or 64 bits).

Object size: An object of a floating point subtype has the same size as its subtype.

Alignment: A floating point subtype is always word aligned.

Object address: Provided its alignment is not constrained by a record representation clause or a `pragma PACK`, the address of an object of a floating point subtype is a multiple of the alignment of the corresponding subtype.

4.4 Fixed Point Types

Small of a fixed point type

If no specification of small applies to a fixed point type, then the value of small is determined by the value of delta as defined by [3.5.9].

A specification of small can be used to impose a value of small. The value of small is required to be a power of two.

Predefined fixed point types

To implement fixed point types, the Alsys Compiler for the transputer uses a set of anonymous predefined types. These are:

```
type SHORT_FIXED is delta D range -2**7*S .. (2**7-1)*S;  
for SHORT_FIXED'SMALL use S;
```

```
type FIXED is delta D range -2**31*S .. (2**31-1)*S;  
for FIXED'SMALL use S;
```

where D is any real value and S any power of two less than or equal to D.

Selection of the parent of a fixed point type

A fixed point type declared by a declaration of the form:

```
type T is delta D range L .. R;
```

possibly with a small specification:

```
for T'SMALL use S;
```

is implicitly derived from a predefined fixed point type. The Compiler automatically selects the predefined fixed point type whose small and delta are the same as the small and delta of T and whose range is the shortest that includes the values L and R.

Encoding of fixed point values

In the program generated by the Compiler, a safe value V of a fixed point subtype F is represented as the integer:

```
V / F'BASE'SMALL
```

Fixed point subtypes

Minimum size: The minimum size of a fixed point subtype is the minimum number of binary digits that is necessary for representing the values of the range of the subtype using the small of the base type (that is to say, in an unbiased form which includes a sign bit only if the range of the subtype includes negative values).

For a static subtype, if it has a null range its minimum size is 1. Otherwise, s and S being the bounds of the subtype, if i and I are the integer representations of m and M , the smallest and the greatest model numbers of the base type such that $s < m$ and $M < S$, then the minimum size L is determined as follows. For $i \geq 0$, L is the smallest positive integer such that $I \leq 2^{L-1}$. For $i < 0$, L is the smallest positive integer such that $-2^{L-1} \leq i$ and $I \leq 2^{L-1}$.

For example:

type F is delta 2.0 range 0.0 .. 500.0;
-- The minimum size of F is 8 bits.

subtype S is F delta 16.0 range 0.0 .. 250.0;
-- The minimum size of S is 7 bits.

subtype D is S range X .. Y;
-- Assuming that X and Y are not static, the minimum size of D is 7 bits
-- (the same as the minimum size of its type mark S).

Size: The sizes of the sets of predefined fixed point types SHORT_FIXED, and FIXED are 8 and 32 bits respectively.

When no size specification is applied to an fixed point type or to its first named subtype (if any), its size and the size of any subtypes is the smallest of 32, 8, 4, 2 or 1 bit which is equal to or greater than the minimum size for the type or first named type.

For example:

type F is delta 0.01 range 0.0 .. 1.0;
-- F is derived from a 8 bit predefined fixed type, its size is 8 bits.

type L is delta 0.01 range 0.0 .. 300.0;
-- L is derived from a 32 bit predefined fixed type, its size is 32 bits.

type N is new L range 0.0 .. 2.0;
-- N is indirectly derived from a 32 bit predefined fixed type, its size is 32 bits.

When a size specification is applied to a fixed point type, this fixed point type and each of its subtypes has the size specified by the length clause. The same rule applies to a first named subtype. The size specification must of course specify a value greater than or equal to the minimum size of the type or subtype to which it applies.

For example:

```
type F is delta 0.01 range 0.0 .. 1.0;
for F'SIZE use 32;
-- F is derived from a 8 bit predefined fixed type, but its size is 32 bits
-- because of the size specification.
```

```
type L is delta 0.01 range 0.0 .. 300.0;
for F'SIZE use 16;
-- F is derived from a 32 bit predefined fixed type, but its size is 16 bits
-- because of the size specification.
-- The size specification is legal since the range contains no negative values
-- and therefore no sign bit is required.
```

```
type N is new F range 0.8 .. 1.0;
-- N is indirectly derived from a 16 bit predefined fixed type, but its size is
-- 32 bits because N inherits the size specification of F.
```

The Alsys Compiler implements size specifications. Nevertheless, as fixed point objects are represented using machine integers, the specified length cannot be greater than 32 bits.

Object size: Provided its size is not constrained by a record component clause or a pragma PACK, the size of an object of a fixed point subtype is determined as follows:

When no size specification is applied to the fixed point type or its first named subtype (if any), the objects this type, or any of its subtypes, have the size of the predefined type from which it derives directly or indirectly.

When a size specification is applied to an fixed point type or its first named subtype, objects of this integer type, or of any of its subtypes, have the size specified by the length clause.

Alignment: A fixed point subtype is byte aligned if its size is less than or equal to 8 bits, word aligned otherwise.

Object address: Provided its alignment is not constrained by a record representation clause or a pragma PACK, the address of an object of a fixed point subtype is a multiple of the alignment of the corresponding subtype.

4.5 Access Types

Collection Size

When no specification of collection size applies to an access type, no storage space is reserved for its collection, and the value of the attribute STORAGE_SIZE is then 0.

As described in [13.2], a specification of collection size can be provided in order to reserve storage space for the collection of an access type. The Alsys Compiler fully implements this kind of specification.

Encoding of access values

Access values are machine addresses represented as machine word - sized values (i.e. 32 bits).

Access subtypes

Minimum size: The minimum size of an access subtype is that of the word size of the target transputer.

Size: The size of an access subtype is the same as its minimum size.

The only size that can be specified for an access type using a size specification is its usual size.

Object size: An object of an access subtype has the same size as its subtype, thus an object of an access subtype is always one machine word long.

Alignment: An access subtype is always word aligned.

Object address: Provided its alignment is not constrained by a record representation clause or a pragma PACK, the address of an object of an access subtype is always on a word boundary, since its subtype is word aligned.

4.6 Task Types

Storage for a task activation

When no length clause is used to specify the storage space to be reserved for a task activation, the storage space indicated at bind time is used for this activation.

As described in [13.2], a length clause can be used to specify the storage space for the activation of each of the tasks of a given type. In this case the value indicated at bind time is ignored for this task type, and the length clause is obeyed.

Both the length clause and the bind time parameter specify the combined size of the task's primary and auxiliary stacks. Further bind time parameters specify the percentage of this storage size to be allocated to the primary stack and indicate whether or not to attempt to allocate the primary stack in fast internal memory. These bind time parameters indicate the default action and can be overridden using the implementation defined pragmas `STORAGE_SIZE_RATIO` and `FAST_PRIMARY`.

```
pragma STORAGE_SIZE_RATIO ( task_name , integer_literal );
```

```
pragma FAST_PRIMARY ( task_name , YES | NO );
```

These two pragmas are not allowed for derived types. They apply to the task type *task_name*. For each pragma, the pragma and the declaration of the task type to which it applies must both occur within the same declarative part or package specification, although the declaration of the task type must precede the pragma.

Pragma `STORAGE_SIZE_RATIO` specifies the percentage of the total storage size reserved for the activation of the task to be used as the task's primary stack. Any remaining storage space will be used as the task's auxiliary stack. In the absence of the pragma the default ratio specified at bind time is used for the activation.

Pragma `FAST_PRIMARY` specifies whether or not an attempt should be made to allocate the task's primary stack in fast internal memory. In the absence of the pragma the default indication specified at bind time is used for the activation.

Encoding of task values

Task values are represented as machine word sized values.

Task subtypes

Minimum size: The minimum size of a task subtype is 32 bits.

Size: The size of a task subtype is the same as its minimum size.

The only size that can be specified for a task type using a size specification is its usual size.

Object size: An object of a task subtype has the same size as its subtype. Thus an object of a task subtype is always 32 bits long.

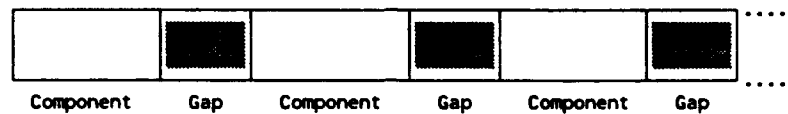
Alignment: A task subtype is always word aligned.

Object address: Provided its alignment is not constrained by a record representation clause, the address of an object of a task subtype is always on a word boundary, since its subtype is word aligned.

4.7 Array Types

Layout of an array

Each array is allocated in a contiguous area of storage units. All the components have the same size. A gap may exist between two consecutive components (and after the last one). All the gaps have the same size.



Components

If the array is not packed, the size of the components is the size of any object of the subtype of the components.

For example:

```
type A is array (1 .. 8) of BOOLEAN;
-- The size of the components of A is the size of any object of the type BOOLEAN:
-- 8 bits.
```

```
type DECIMAL_DIGIT is range 0 .. 9;
for DECIMAL_DIGIT'SIZE use 4;
type BINARY_CODED_DECIMAL is
  array (INTEGER range <>) of DECIMAL_DIGIT;
-- The size of the type DECIMAL_DIGIT is 4 bits. Thus in an array of
-- type BINARY_CODED_DECIMAL each component will be represented in
-- 4 bits as in the usual BCD representation.
```

If the array is packed and its components are neither records nor arrays, the size of the components is the size of the subtype of the components.

For example:

```
type A is array (1 .. 8) of BOOLEAN;
pragma PACK(A);
-- The size of the components of A is the size of the type BOOLEAN: 1 bit.

type DECIMAL_DIGIT is range 0 .. 9;
type BINARY_CODED_DECIMAL is
  array (INTEGER range <>) of DECIMAL_DIGIT;
pragma PACK(BINARY_CODED_DECIMAL);
-- The size of the type DECIMAL_DIGIT is 4 bits.
-- BINARY_CODED_DECIMAL is packed, each component of an array of this
-- type will be represented in 4 bits as in the usual BCD representation.
```

Packing the array has no effect on the size of the components when the components are records or arrays.

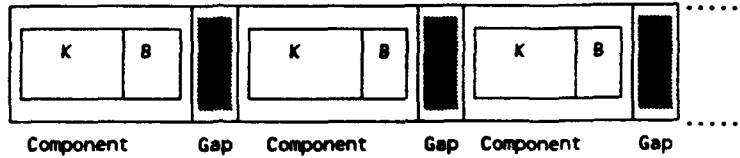
Gaps

If the components are records or arrays, no size specification applies to the subtype of the components and the array is not packed, then the Compiler may choose a representation with a gap after each component; the aim of the insertion of such gaps is to optimize access to the array components and to their subcomponents. The size of the gap is chosen so that the relative displacement of consecutive components is a multiple of the alignment of the subtype of the components. This strategy allows each component and subcomponent to have an address consistent with the alignment of its subtype

For example:

```
type R is
  record
    K : INTEGER; -- INTEGER is word aligned.
    B : BOOLEAN; -- BOOLEAN is byte aligned.
  end record;
-- Record type R is word aligned. Its size is 40 bits.

type A is array (1 .. 10) of R;
-- A gap of three bytes is inserted after each component in order to respect the
-- alignment of type R. The size of an array of type A will be 640 bits.
```



Array of type A: each subcomponent K has a word offset.

If a size specification applies to the subtype of the components or if the array is packed, no gaps are inserted.

For example:

```

type R is
  record
    K : INTEGER;
    B : BOOLEAN;
  end record;

```

```

type A is array (1 .. 10) of R;
pragma PACK(A);
-- There is no gap in an array of type A because A is packed.
-- The size of an object of type A will be 400 bits.

```

```

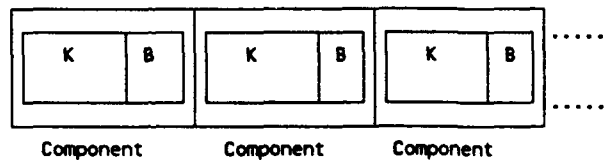
type NR is new R;
for NR'SIZE use 40;

```

```

type B is array (1 .. 10) of NR;
-- There is no gap in an array of type B because NR has a size specification.
-- The size of an object of type B will be 400 bits.

```



Array of type A or B: a subcomponent K can have any byte offset.

Array subtypes

Size: The size of an array subtype is obtained by multiplying the number of its components by the sum of the size of the components and the size of the gaps (if any). If the subtype is unconstrained, the maximum number of components is considered.

The size of an array subtype cannot be computed at compile time

- if it has non-static constraints or is an unconstrained array type with non-static index subtypes (because the number of components can then only be determined at run time).
- if the components are records or arrays and their constraints or the constraints of their subcomponents (if any) are not static (because the size of the components and the size of the gaps can then only be determined at run time).

As has been indicated above, the effect of a pragma `PACK` on an array type is to suppress the gaps and to reduce the size of the components. The consequence of packing an array type is thus to reduce its size.

If the components of an array are records or arrays and their constraints or the constraints of their subcomponents (if any) are not static, the Compiler ignores any pragma `PACK` applied to the array type but issues a warning message. Apart from this limitation, array packing is fully implemented by the Alsys Compiler.

The only size that can be specified for an array type or first named subtype using a size specification is its usual size. Nevertheless, such a length clause can be useful to verify that the layout of an array is as expected by the application.

Object size: The size of an object of an array subtype is always equal to the size of the subtype of the object.

Alignment: If no pragma `PACK` applies to an array subtype and no size specification applies to its components, the array subtype has the same alignment as the subtype of its components.

If a pragma `PACK` applies to an array subtype or if a size specification applies to its components (so that there are no gaps), the alignment of the array subtype is the lesser of the alignment of the subtype of its components and the relative displacement of the components.

Object address: Provided its alignment is not constrained by a record representation clause, the address of an object of an array subtype is a multiple of the alignment of the corresponding subtype.

4.8 Record Types

Layout of a record

Each record is allocated in a contiguous area of storage units. The size of a record component depends on its type. Gaps may exist between some components.

The positions and the sizes of the components of a record type object can be controlled using a record representation clause as described in [13.4]. In the Alsys implementation for transputer targets there is no restriction on the position that can be specified for a component of a record. If a component is not a record or an array, its size can be any size from the minimum size to the size of its subtype. If a component is a record or an array, its size must be the size of its subtype.

In a record representation clause, the first storage unit (that is, a byte) and the first bit position within a storage unit are numbered zero. Bits are ordered, and thus numbered, least significant bit first within a storage unit. Storage units are numbered such that lower numbers have the least significance in a machine word.

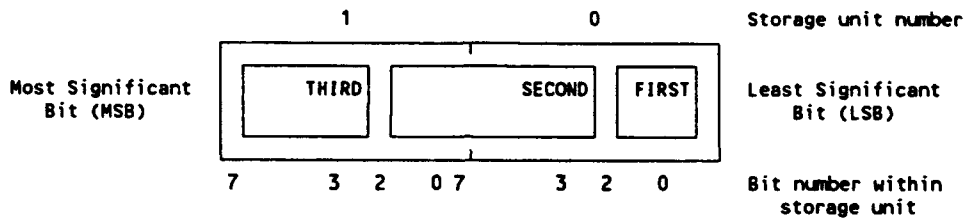
A component clause may be specified such that the component overlaps a storage unit boundary. In this case, the bits are numbered in sequence from the least significant bit of the first storage unit to the most significant bit of the last storage unit occupied by the component. For example:

```
type BIT_3 is range 0 .. 7;  
for BIT_3'SIZE use 3;  
  
type BIT_5 is range 0 .. 31;  
for BIT_5'SIZE use 5;  
  
type BIT_8 is range 0 .. 255;  
for BIT_8'SIZE use 8;
```

```

type R is
  record
    FIRST : BIT_3;
    SECOND : BIT_8;
    THIRD : BIT_5;
  end record;
for R use
  record
    FIRST at 0 range 0 .. 2;
    SECOND at 0 range 3 .. 10;
    -- Component SECOND overlaps a storage unit boundary.
    THIRD at 1 range 3 .. 7;
  end record;
for R'SIZE use 16;

```



Representation of a Record of type R

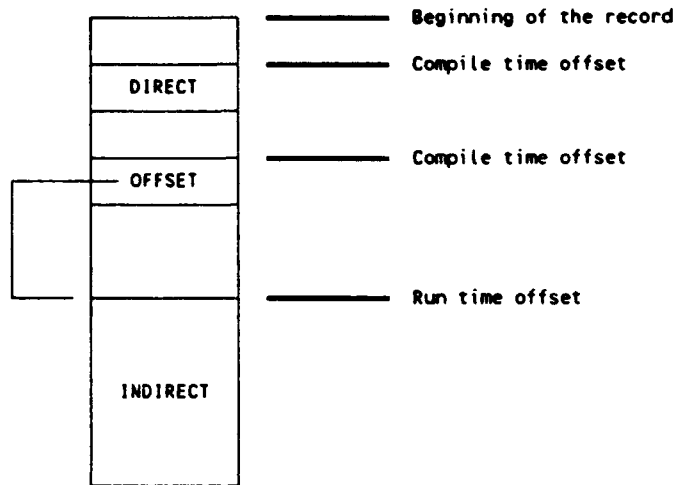
A record representation clause need not specify the position and the size for every component.

If no component clause applies to a component of a record, its size is the size of objects of its subtype. Its position is chosen by the Compiler so as to optimize access to the components of the record: the offset of the component is chosen as a multiple of the alignment of the component subtype. Moreover, the Compiler chooses the position of the component so as to reduce the number of gaps and thus the size of the record objects.

Because of these optimisations, there is no connection between the order of the components in a record type declaration and the positions chosen by the Compiler for the components in a record object.

Indirect components

If the offset of a component cannot be computed at compile time, this offset is stored in the record objects at run time and used to access the component. Such a component is said to be indirect while other components are said to be direct:



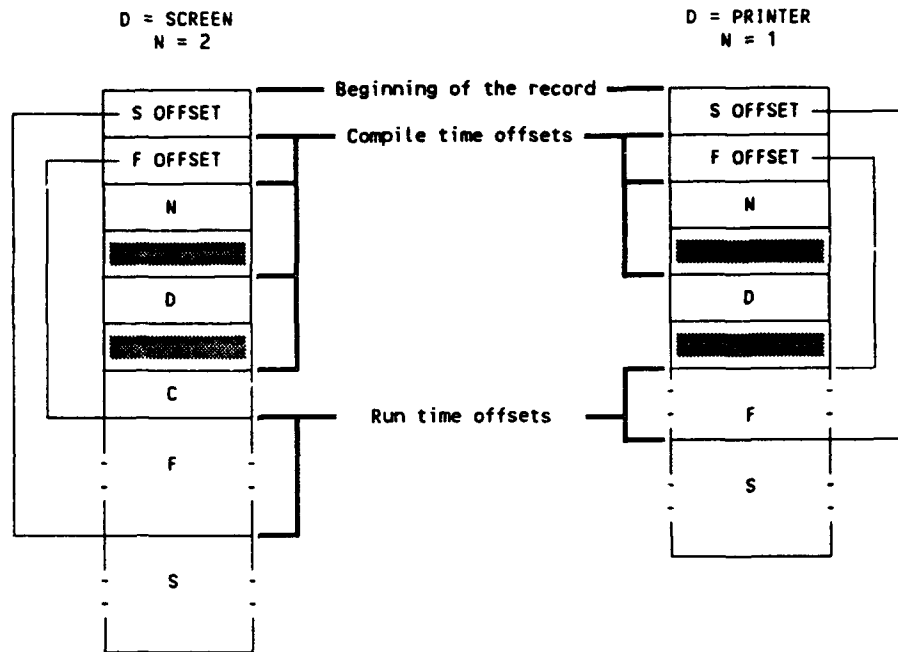
A direct and an indirect component

If a record component is a record or an array, the size of its subtype may be evaluated at run time and may even depend on the discriminants of the record. We will call these components dynamic components.

For example:

```
type DEVICE is (SCREEN, PRINTER);  
type COLOR is (GREEN, RED, BLUE);  
type SERIES is array (POSITIVE range <>) of INTEGER;  
type GRAPH (L : NATURAL) is  
  record  
    X : SERIES(1 .. L); -- The size of X depends on L  
    Y : SERIES(1 .. L); -- The size of Y depends on L  
  end record;  
  
Q : POSITIVE;  
  
type PICTURE (N : NATURAL; D : DEVICE) is  
  record  
    F : GRAPH(N); -- The size of F depends on N  
    S : GRAPH(Q); -- The size of S depends on Q  
    case D is  
      when SCREEN = >  
        C : COLOR;  
      when PRINTER = >  
        null;  
    end case;  
  end record;
```

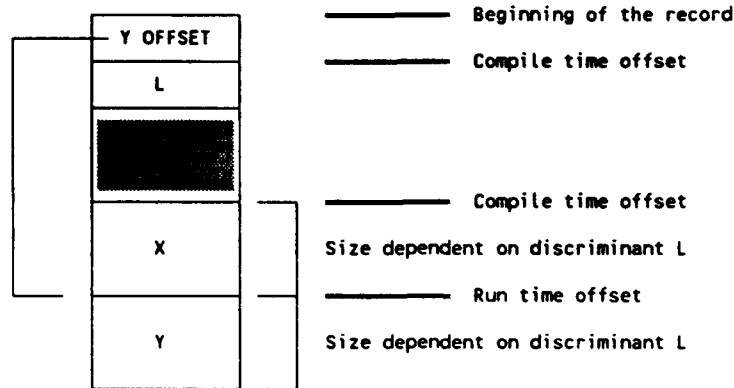
Any component placed after a dynamic component has an offset which cannot be evaluated at compile time and is thus indirect. In order to minimize the number of indirect components, the Compiler groups the dynamic components together and places them at the end of the record:



The record type PICTURE: F and S are placed at the end of the record

Thanks to this strategy, the only indirect components are dynamic components. But not all dynamic components are necessarily indirect: if there are dynamic components in a component list which is not followed by a variant part, then exactly one dynamic component of this list is a direct component because its offset can be computed at compilation time.

For example:



The record type GRAPH: the dynamic component X is a direct component.

The offset of an indirect component is always expressed in storage units.

The space reserved for the offset of an indirect component must be large enough to store the size of any value of the record type (the maximum potential offset). The Compiler evaluates an upper bound MS of this size and treats an offset as a component having an anonymous integer type whose range is 0 .. MS.

If C is the name of an indirect component, then the offset of this component can be denoted in a component clause by the implementation generated name C'OFFSET.

Implicit components

In some circumstances, access to an object of a record type or to its components involves computing information which only depends on the discriminant values. To avoid unnecessary recomputation, the Compiler stores this information in the record objects, updates it when the values of the discriminants are modified and uses it when the objects or their components are accessed. This information is stored in special components called implicit components.

An implicit component may contain information which is used when the record object or several of its components are accessed. In this case the component will be included in any record object (the implicit component is considered to be declared before any variant part in the record type declaration). There can be two components of this kind; one is called `RECORD_SIZE` and the other `VARIANT_INDEX`.

On the other hand an implicit component may be used to access a given record component. In this case the implicit component exists whenever the record component exists (the implicit component is considered to be declared at the same place as the record component). Components of this kind are called `ARRAY_DESCRIPTORs` or `RECORD_DESCRIPTORs`.

RECORD_SIZE

This implicit component is created by the Compiler when the record type has a variant part and its discriminants are defaulted. It contains the size of the storage space necessary to store the current value of the record object (note that the storage effectively allocated for the record object may be more than this).

The value of a `RECORD_SIZE` component may denote a number of bits or a number of storage units. In general it denotes a number of storage units, but if any component clause specifies that a component of the record type has an offset or a size which cannot be expressed using storage units, then the value designates a number of bits.

The implicit component `RECORD_SIZE` must be large enough to store the maximum size of any value of the record type. The Compiler evaluates an upper bound `MS` of this size and then considers the implicit component as having an anonymous integer type whose range is `0 .. MS`.

If `R` is the name of the record type, this implicit component can be denoted in a component clause by the implementation generated name `R'RECORD_SIZE`.

VARIANT_INDEX

This implicit component is created by the Compiler when the record type has a variant part. It indicates the set of components that are present in a record value. It is used when a discriminant check is to be done.

Component lists that do not contain a variant part are numbered. These numbers are the possible values of the implicit component `VARIANT_INDEX`.

For example:

```
type VEHICLE is (AIRCRAFT, ROCKET, BOAT, CAR);  
type DESCRIPTION (KIND : VEHICLE := CAR) is  
  record  
    SPEED : INTEGER;  
    case KIND is  
      when AIRCRAFT | CAR =>  
        WHEELS : INTEGER;  
        case KIND is  
          when AIRCRAFT => -- 1  
            WINGSPAN : INTEGER;  
          when others => -- 2  
            null;  
        end case;  
      when BOAT => -- 3  
        STEAM : BOOLEAN;  
      when ROCKET => -- 4  
        STAGES : INTEGER;  
    end case;  
  end record;
```

The value of the variant index indicates the set of components that are present in a record value:

Variant Index	Set
1	{KIND, SPEED, WHEELS, WINGSPAN}
2	{KIND, SPEED, WHEELS}
3	{KIND, SPEED, STEAM}
4	{KIND, SPEED, STAGES}

A comparison between the variant index of a record value and the bounds of an interval is enough to check that a given component is present in the value:

Component	Interval
KIND	--
SPEED	--
WHEELS	1 .. 2
WINGSPAN	1 .. 1
STEAM	3 .. 3
STAGES	4 .. 4

The implicit component `VARIANT_INDEX` must be large enough to store the number `V` of component lists that don't contain variant parts. The Compiler treats this implicit component as having an anonymous integer type whose range is `1 .. V`.

If `R` is the name of the record type, this implicit component can be denoted in a component clause by the implementation generated name `R^VARIANT_INDEX`.

ARRAY_DESCRIPTOR

An implicit component of this kind is associated by the Compiler with each record component whose subtype is an anonymous array subtype that depends on a discriminant of the record. It contains information about the component subtype.

The structure of an implicit component of kind `ARRAY_DESCRIPTOR` is not described in this documentation. Nevertheless, if a programmer is interested in specifying the location of a component of this kind using a component clause, he can obtain the size of the component using the `ASSEMBLY` parameter in the `COMPILE` command.

The Compiler treats an implicit component of the kind `ARRAY_DESCRIPTOR` as having an anonymous record type. If `C` is the name of the record component whose subtype is described by the array descriptor, then this implicit component can be denoted in a component clause by the implementation generated name `C^ARRAY_DESCRIPTOR`.

RECORD_DESCRIPTOR

An implicit component of this kind is associated by the Compiler with each record component whose subtype is an anonymous record subtype that depends on a discriminant of the record. It contains information about the component subtype.

The structure of an implicit component of kind `RECORD_DESCRIPTOR` is not described in this documentation. Nevertheless, if a programmer is interested in specifying the location of a component of this kind using a component clause, he can obtain the size of the component using the `ASSEMBLY` parameter in the `COMPILE` command.

The Compiler treats an implicit component of the kind `RECORD_DESCRIPTOR` as having an anonymous record type. If `C` is the name of the record component whose subtype is described by the record descriptor, then this implicit component can be denoted in a component clause by the implementation generated name `CRECORD_DESCRIPTOR`.

Suppression of implicit components

The Alsys implementation provides the capability of suppressing the implicit components `RECORD_SIZE` and/or `VARIANT_INDEX` from a record type. This can be done using an implementation defined pragma called `IMPROVE`. The syntax of this pragma is as follows:

```
pragma IMPROVE ( TIME | SPACE , [ON = >] simple_name );
```

The first argument specifies whether `TIME` or `SPACE` is the primary criterion for the choice of the representation of the record type that is denoted by the second argument.

If `TIME` is specified, the Compiler inserts implicit components as described above. If on the other hand `SPACE` is specified, the Compiler only inserts a `VARIANT_INDEX` or a `RECORD_SIZE` component if this component appears in a record representation clause that applies to the record type. A record representation clause can thus be used to keep one implicit component while suppressing the other.

A pragma `IMPROVE` that applies to a given record type can occur anywhere that a representation clause is allowed for this type.

Record subtypes

Size: Unless a component clause specifies that a component of a record type has an offset or a size which cannot be expressed using storage units, the size of a record subtype is rounded up to a whole number of storage units.

The size of a constrained record subtype is obtained by adding the sizes of its components and the sizes of its gaps (if any). This size is not computed at compile time

- when the record subtype has non-static constraints,
- when a component is an array or a record and its size is not computed at compile time.

The size of an unconstrained record subtype is obtained by adding the sizes of the components and the sizes of the gaps (if any) of its largest variant. If the size of a

component or of a gap cannot be evaluated exactly at compile time, an upper bound of this size is used by the Compiler to compute the subtype size.

The only size that can be specified for a record type or first named subtype using a size specification is its usual size. Nevertheless, such a length clause can be useful to verify that the layout of a record is as expected by the application.

Object size: An object of a constrained record subtype has the same size as its subtype.

An object of an unconstrained record subtype has the same size as its subtype if this size is less than or equal to 4 Kbyte. If the size of the subtype is greater than this, the object has the size necessary to store its current value; storage space is allocated and released as the discriminants of the record change.

Alignment: When no record representation clause applies to its base type, a record subtype has the same alignment as the component with the highest alignment requirement.

When a record representation clause that does not contain an alignment clause applies to its base type, a record subtype has the same alignment as the component with the highest alignment requirement which has not been overridden by its component clause.

When a record representation clause that contains an alignment clause applies to its base type, a record subtype has an alignment that obeys the alignment clause.

Object address: Provided its alignment is not constrained by a representation clause, the address of an object of a record subtype is a multiple of the alignment of the corresponding subtype.

CHAPTER 5

Conventions for Implementation-Generated Names

Special record components are introduced by the Compiler for certain record type definitions. Such record components are implementation-dependent; they are used by the Compiler to improve the quality of the generated code for certain operations on the record types. The existence of these components is established by the Compiler depending on implementation-dependent criteria. Attributes are defined for referring to them in record representation clauses. An error message is issued by the Compiler if the user refers to an implementation-dependent component that does not exist. If the implementation-dependent component exists, the Compiler checks that the storage location specified in the component clause is compatible with the treatment of this component and the storage locations of other components. An error message is issued if this check fails.

There are five such attributes described below (also see section 4.8):

T RECORD_SIZE	For a prefix T that denotes a record type. This attribute refers to the record component introduced by the Compiler in a record to store the size of the record object. This component exists for objects of a record type with defaulted discriminants when the sizes of the record objects depend on the values of the discriminants.
T VARIANT_INDEX	For a prefix T that denotes a record type. This attribute refers to the record component introduced by the Compiler in a record to assist in the efficient implementation of discriminant checks. This component exists for objects of a record type with variant type.
C ARRAY_DESCRIPTOR	For a prefix C that denotes a record component of an array type whose component subtype definition depends on discriminants. This attribute refers to the record component introduced by the Compiler in a record to store information on subtypes of components that depend on discriminants.

CRECORD_DESCRIPTOR

For a prefix **C** that denotes a record component of a record type whose component subtype definition depends on discriminants. This attribute refers to the record component introduced by the Compiler in a record to store information on subtypes of components that depend on discriminants.

COFFSET

For a prefix **C** that denotes an indirect component of a record type. This attribute refers to the offset component introduced by the Compiler in a record to store the offset of **C**.

CHAPTER 6

Address Clauses

6.1 Address Clauses for Objects

An address clause can be used to specify an address for an object as described in [13.5]. When such a clause applies to an object no storage is allocated for it in the program generated by the Compiler. The program accesses the object using the address specified in the clause.

An address clause is not allowed for task objects, nor for unconstrained records whose maximum possible size is greater than 4 Kbytes, or for a constant.

6.2 Address Clauses for Program Units

Address clauses for program units are not implemented.

6.3 Address Clauses for Entries

Address clauses for entries are not implemented.

CHAPTER 7

Restrictions on Unchecked Conversions

Unconstrained arrays are not allowed as target types.

Unconstrained record types without defaulted discriminants are not allowed as target types.

If the source and the target types are each scalar or access types, the sizes of the objects of the source and target types must be equal. If a composite type is used either as the source type or as the target type this restriction on the size does not apply.

If the source and the target types are each of scalar or access type or if they are both of composite type, the effect of the function is to return the operand.

In other cases the effect of unchecked conversion can be considered as a copy:

- if an unchecked conversion is achieved of a scalar or access source type to a composite target type, the result of the function is a copy of the source operand: the result has the size of the source.
- if an unchecked conversion is achieved of a composite source type to a scalar or access target type, the result of the function is a copy of the source operand: the result has the size of the target.

CHAPTER 8

Input-Output Packages

The predefined input-output packages `SEQUENTIAL_IO` [14.2.3], `DIRECT_IO` [14.2.5], `TEXT_IO` [14.3.10] and `IO_EXCEPTIONS` [14.5] are implemented as described in the Language Reference Manual.

It should be noted that, in order to generate output, calls to `PUT` procedures should be followed by a call to either `PUT_LINE` or `NEW_LINE`.

The package `LOW_LEVEL_IO` [14.6], which is concerned with low-level machine-dependent input-output, is not implemented.

All access to the services of the host system are provided through the INMOS supplied server (e.g. *iserver* Ref.3), so much of Ada input-output is host independent. Any restrictions that apply to the server will also apply to the Ada input-output.

8.1 NAME Parameter

No special treatment is applied to the `NAME` parameter supplied to the Ada procedures `CREATE` or `OPEN` [14.2.1]. This parameter is passed immediately on to the INMOS server and from there to the host operating system. The file name can thus be in any format acceptable to the host system.

8.2 FORM Parameter

The `FORM` parameter comprises a set of attributes formulated according to the lexical rules of [2], separated by commas. The `FORM` parameter may be given as a null string except when `DIRECT_IO` is instantiated with an unconstrained type; in this case the record size attribute must be provided. Attributes are comma-separated; blanks may be inserted between lexical elements as desired. In the descriptions below the meanings of *natural*, *positive*, etc., are as in Ada; attribute keywords (represented in upper case) are identifiers [2.3] and as such may be specified without regard to case.

`USE_ERROR` is raised if the `FORM` parameter does not conform to these rules.

The attributes are as follows:

8.2.1 File Sharing

This attribute allows control over the sharing of one external file between several internal files within a single program. In effect it establishes rules for subsequent OPEN and CREATE calls which specify the same external file. If such rules are violated or if a different file sharing attribute is specified in a later OPEN or CREATE call, USE_ERROR will be raised. The syntax is as follows:

```
NOT_SHARED |  
SHARED => access_mode
```

where

```
access_mode ::= READERS | SINGLE_WRITER | ANY;
```

A file sharing attribute of:

NOT_SHARED	implies only one internal file may access the external file.
SHARED => READERS	imposes no restrictions on internal files of mode IN_FILE, but prevents any internal files of mode OUT_FILE or INOUT_FILE being associated with the external file.
SHARED => SINGLE_WRITER	is as SHARED => READERS, but in addition allows a single internal file of mode OUT_FILE or INOUT_FILE.
SHARED => ANY	places no restriction on external file string.

If a file of the same name has previously been opened or created, the default is taken from that file's sharing attribute, otherwise the default depends on the mode of the file: for mode IN_FILE the default is SHARED => READERS, for modes INOUT_FILE and OUT_FILE the default is NOT_SHARED.

8.2.2 Binary Files

Two FORM attributes, RECORD_SIZE and RECORD_UNIT, control the structure of binary files.

A binary file can be viewed as a sequence (sequential access) or a set (direct access) of consecutive RECORDS.

The structure of such a record is:

[HEADER] OBJECT [UNUSED_PART]

and it is formed from up to three items:

- an OBJECT with the exact binary representation of the Ada object in the executable program, possibly including an object descriptor
- a HEADER consisting of two word sized fields:
 - the length of the object in bytes
 - the length of the descriptor in bytes
- an UNUSED_PART of variable size to permit full control of the record's size

The HEADER is implemented only if the actual parameter of the instantiation of the IO package is unconstrained.

The file structure attributes take the form:

RECORD_SIZE = > size_in_bytes
RECORD_UNIT = > size_in_bytes

Their meaning depends on the object's type (constrained or not) and the file access mode (sequential or direct access):

- a) If the object's type is constrained:
 - The RECORD_UNIT attribute is illegal
 - If the RECORD_SIZE attribute is omitted, no UNUSED_PART will be implemented: the default RECORD_SIZE is the object's size

- If present, the RECORD_SIZE attribute must specify a record size greater than or equal to the object's size, otherwise the exception USE_ERROR will be raised
- b) If the object's type is unconstrained and the file access mode is direct:
- The RECORD_UNIT attribute is illegal
 - The RECORD_SIZE attribute has no default value, and if it is not specified, a USE_ERROR will be raised
 - An attempt to input or output an object larger than the given RECORD_SIZE will raise the exception DATA_ERROR
- c) If the object's type is unconstrained and the file access mode is sequential:
- The RECORD_SIZE attribute is illegal
 - The default value of the RECORD_UNIT attribute is 1 (byte)
 - The record size will be the smallest multiple of the specified (or default) RECORD_UNIT that holds the object and its header. This is the only case where records of a file may have different sizes.

In all cases the value given must not be smaller than a minimum size. For constrained types, this minimum size is $ELEMENT_TYPE'SIZE / SYSTEM.STORAGE_UNIT$; USE_ERROR will be raised if this rule is violated. For unconstrained types, the minimum size is $ELEMENT_TYPE'DESCRIP'TOR_SIZE / SYSTEM.STORAGE_UNIT$ plus the size of the largest record which is to be read or written. If a larger record is processed, DATA_ERROR will be raised by the READ or WRITE.

8.2.3 Buffering

This attribute controls the size of the buffer used as a cache for input-output operations:

`BUFFER_SIZE => size_in_bytes`

The default value for BUFFER_SIZE is 0; which means no buffering.

8.2.4 Appending

This attribute may only be used in the FORM parameter of the OPEN command. If used in the FORM parameter of the CREATE command, USE_ERROR will be raised.

The effect of this attribute is to cause writing to commence at the end of the existing file.

The syntax of the APPEND attribute is simply:

APPEND

The default is APPEND => FALSE, but this is overridden if this attribute is specified.

In normal circumstances, when an external file is opened, an index is set which points to the beginning of the file. If the APPEND attribute is present for a sequential or for a text file, then data transfer will commence at the end of the file. For a direct access file, the value of the index is set to one more than the number of records in the external file.

This attribute is not applicable to terminal devices.

8.3 USE_ERROR

The following conditions will cause USE_ERROR to be raised:

- Specifying a FORM parameter whose syntax does not conform to the rules given above.
- Specifying the RECORD_SIZE FORM parameter attribute to have a value of zero, or failing to specify RECORD_SIZE for instantiations of DIRECT_IO for unconstrained types.
- Specifying a RECORD_SIZE FORM parameter attribute to have a value less than that required to hold the element for instantiations of DIRECT_IO and SEQUENTIAL_IO for constrained types.
- Violating the file sharing rules stated above.
- Attempting to perform an input - output operation which is not supported by the INMOS iserver due to restrictions of the host operating system.
- Errors detected whilst reading or writing (e.g. writing to a file on a read-only disk).

CHAPTER 9

Characteristics of Numeric Types

9.1 Integer Types

The ranges of values for integer types declared in package STANDARD are as follows:

SHORT_INTEGER -128 .. 127 -- $2^{**7} - 1$

INTEGER -2147483648 .. 2147483647 -- $2^{**31} - 1$

Other Integer Types

For the packages DIRECT_IO and TEXT_IO, the ranges of values for types COUNT and POSITIVE_COUNT are as follows:

COUNT 0 .. 2147483647 -- $2^{**31} - 1$

POSITIVE_COUNT 1 .. 2147483647 -- $2^{**31} - 1$

For the package TEXT_IO, the range of values for the type FIELD is as follows:

FIELD 0 .. 255 -- $2^{**8} - 1$

9.2 Floating Point Type Attributes

FLOAT

		Approximate value
DIGITS	6	
MANTISSA	21	
EMAX	84	
EPSILON	$2.0^{** -20}$	9.54E-7
SMALL	$2.0^{** -85}$	2.58E-26
LARGE	$2.0^{** 84} * (1.0 - 2.0^{** -21})$	1.93E+25
SAFE_EMAX	125	
SAFE_SMALL	$2.0^{** -126}$	1.18E-38
SAFE_LARGE	$2.0^{** 125} * (1.0 - 2.0^{** -21})$	4.25E+37
FIRST	$-2.0^{** 127} * (2.0 - 2.0^{** -23})$	-3.40E+38
LAST	$2.0^{** 127} * (2.0 - 2.0^{** -23})$	3.40E+38
MACHINE_RADIX	2	
MACHINE_MANTISSA	24	
MACHINE_EMAX	128	
MACHINE_EMIN	-125	
MACHINE_ROUNDS	TRUE	
MACHINE_OVERFLOWS	TRUE	
SIZE	32	

LONG_FLOAT

		Approximate value
DIGITS	15	
MANTISSA	51	
EMAX	204	
EPSILON	$2.0^{** -50}$	8.88E-16
SMALL	$2.0^{** -205}$	1.94E-62
LARGE	$2.0^{** 204} * (1.0 - 2.0^{** -51})$	2.57E + 61
SAFE_EMAX	1021	
SAFE_SMALL	$2.0^{** -1022}$	2.22E-308
SAFE_LARGE	$2.0^{** 1021} * (1.0 - 2.0^{** -51})$	2.25E + 307
FIRST	$-2.0^{** 1023} * (2.0 - 2.0^{** -52})$	-1.79E + 308
LAST	$2.0^{** 1023} * (2.0 - 2.0^{** -52})$	1.79E + 308
MACHINE_RADIX	2	
MACHINE_MANTISSA	53	
MACHINE_EMAX	1024	
MACHINE_EMIN	-1021	
MACHINE_ROUNDS	TRUE	
MACHINE_OVERFLOWS	TRUE	
SIZE	64	

9.3 Attributes of Type DURATION

		Approximate value
DURATION'DELTA	$2.0^{** -14}$	
DURATION'SMALL	$2.0^{** -14}$	
DURATION'LARGE	$2.0^{** 17} - 2.0^{** -14}$	131072
DURATION'FIRST	-131072.0	
DURATION'LAST	$2.0^{** 17} - 2.0^{** -14}$	131072

REFERENCES

- [1] Reference Manual for the Ada Programming Language
(ANSI/MIL-STD-1815A-1983).
- [2] Occam2 Reference Manual.
INMOS Limited
Prentice Hall, 1988.
- [3] Occam2 Toolset User Manual.
INMOS Limited.
See release notes for version number.
- [4] Transputer Instruction Set - A Compiler Writer's Guide
INMOS Limited
Prentice Hall, 1988
- [5] The T9000 Transputer Instruction Set Manual
INMOS Limited

INDEX

- Ada_designator 3
- ADDRESS attribute 15
 - restrictions 15
- Append attribute 61
- ARRAY_DESCRIPTOR attribute 51
- ASCII 7, 9
- Attributes 15
 - ARRAY_DESCRIPTOR 51
 - DESCRIPTOR_SIZE 15
 - IS_ARRAY 15
 - OBJECT 52
 - RECORD_DESCRIPTOR 52
 - RECORD_SIZE 51, 57
 - representation attributes 15
 - VARIANT_INDEX 51
- BOOLEAN 7
- C'OFFSET attribute 52
- CHARACTER 7
- COUNT 63
- DESCRIPTOR_SIZE attribute 15, 60
- DIRECT_IO 57, 63
- DURATION
 - attributes 65
- Enumeration types 7
 - BOOLEAN 7
 - CHARACTER 7
- FAST_PRIMARY 13, 34
- FIELD 63
- File sharing attribute 58
- Fixed point types 8
 - DURATION 65
- FLOAT 8, 28, 64
- Floating point types 8
 - FLOAT 8, 64
 - LONG_FLOAT 8, 65
- FORM parameter 57
- FORM parameter attributes
 - append 61
 - file sharing attribute 58
 - record_size attribute 61
- Implementation-dependent attributes 15
- Implementation-dependent pragma 5
- Implementation-generated names 51
- IMPROVE 13
- INDENT 12
- INLINE 3
- INLINE_GENERIC 3
- Input-Output packages 57
 - DIRECT_IO 57
 - IO_EXCEPTIONS 57
 - LOW_LEVEL_IO 57
 - SEQUENTIAL_IO 57
 - TEXT_IO 57
- INTEGER 7, 63
 - Integer types 7, 63
 - COUNT 63
 - FIELD 63
 - INTEGER 7, 63
 - POSITIVE_COUNT 63
 - SHORT_INTEGER 7, 63
- INTERFACE 5
- INTERFACE_NAME 5, 11
- IO_EXCEPTIONS 57
- IS_ARRAY attribute 15
- Language_name 5
- LONG_FLOAT 8, 28, 65
- LOW_LEVEL_IO 57
- NO_IMAGE 12
- NOT_SHARED 58
- Numeric types
 - characteristics 63
 - Fixed point types 65
 - integer types 63

OCCAM 5
OCCAM_HIGH 5
PACK 13
Parameter representations 7
 Access types 8
 Array types 8
 Enumeration types 7
 Fixed point types 8
 Floating point types 8
 Integer types 7
 Record types 9
Parameter-passing conventions 6
POSITIVE COUNT 63
Pragma INLINE 3
Pragma INLINE_GENERIC 3
Pragma INTERFACE 5
 language_name 5
 OCCAM 5
 subprogram_name 5
Pragma INTERFACE_NAME 5
 string_literal 11
 subprogram_name 11
Pragma NO_IMAGE 12
Pragmas
 FAST_PRIMARY 13, 34
 IMPROVE 13
 INDENT 12
 INTERFACE 5
 INTERFACE_NAME 11
 PACK 13
 PRIORITY 13
 STORAGE_SIZE_RATIO 13, 34
 SUPPRESS 13
PRIORITY 13
RECORD_DESCRIPTOR attribute 52
RECORD_SIZE attribute 51, 57, 61
Representation attributes 15
Representation clauses 21
 restrictions 21
SEQUENTIAL_IO 57

SHARED 58
SHORT_INTEGER 7, 63
STORAGE_SIZE_RATIO 13, 34
STRING 9
 String literal 11
 Subprogram_name 5, 11
SUPPRESS 13
SYSTEM package 17
TEXT_IO 57, 63
 Unchecked conversions 55
 restrictions 55
USE_ERROR 57, 61
VARIANT_INDEX attribute 51