Technical Report

CMU/SEI-87-TR-17
ESD-TR-87-118

The Use of Representation Clauses and Implementation-Dependent Features in Ada:

IIIA. Qualitative Results for VAX Ada
Version 1.3

B. Craig Meyers
Andrea L. Cappellini

July 1987

Technical Report

CMU/SEI-87-TR-17
ESD/TR-87-118
July 1987

The Use of Representation Clauses and Implementation-Dependent Features in Ada: IIIA. Qualitative Results for VAX Ada

Version 1.3

B. Craig Meyers
Andrea L. Cappellini

Ada Embedded Systems Testbed Project

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

This report was prepared for the SEI Joint Program Office HQ ESC/AXS

5 Eglin Street

Hanscom AFB, MA 01731-2116

The ideas and findings in this report should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

FOR THE COMMANDER

(signature on file)

Thomas R. Miller, Lt Col, USAF, SEI Joint Program Office

This work is sponsored by the U.S. Department of Defense.

This work was created in the performance of Federal Government Contract Number F19628-95-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. The Government of the United States has a royalty-free government-purpose license to use, duplicate, or disclose the work, in whole or in part and in any manner, and to have or permit others to do so, for government purposes pursuant to the copyright license under the clause at 52.227-7013.

This document is available through Research Access, Inc. / 800 Vinial Street / Pittsburgh, PA 15212. Phone: 1-800-685-6510. FAX: (412) 321-2994. RAI also maintains a World Wide Web home page at http://www.rai.com

Copies of this document are available through the National Technical Information Service (NTIS). For information on ordering, please contact NTIS directly: National Technical Information Service / U.S. Department of Commerce / Springfield, VA 22161. Phone: (703) 487-4600.

This document is also available through the Defense Technical Information Center (DTIC). DTIC provides acess to and transfer of scientific and technical information for DoD personnel, DoD contractors and potential con tractors, and other U.S. Government agency personnel and their contractors. To obtain a copy, please contact DTIC directly: Defense Technical Information Center / 8725 John J. Kingman Road / Suite 0944 / Ft. Belvoir, VA 22060-6218. Phone: 1-800-225-3842 or 703-767-8222.

The Use of Representation Clauses and Implementation-Dependent Features in Ada: IIIA. Qualitative Results for VAX Ada Version 1.3

Abstract: This report, one in a series, provides a qualitative assessment of the support of representation clauses and implementation-dependent features in Ada provided by the VAX Ada compiler, Version 1.3. The evaluation questions that were presented in a previous report of this series form the basis of the qualitative assessment. A subjective evaluation of the support provided for these features is also presented.

1. Introduction

The Ada language was developed as a general-purpose language applicable to the development and maintenance of mission-critical systems for the Department of Defense (DoD). In the development of the language, a need to allow the language to interact with the underlying machine architecture was recognized. This coupling is discussed in Chapter 13 of *Reference Manual for the Ada Programming Language* [1].

A frequent characteristic of mission-critical systems is that they employ "packed" data structures. Furthermore, within such packed structures there may be nonstandard data representations. For example, an integer type may have a length of 12 bits, or some fixed-point type may have arbitrarily scaled precision.

Such packed data structures are defined in Ada through the use of representation clauses. The use of these clauses is highly machine dependent. That is, some compilers may provide only limited support, while others may provide a full set of capabilities. Since representation clauses are implementation dependent, their use may affect the portability of any code which uses these features.

This report is one of a series of reports related to the use of representation clauses and implementation-dependent features in Ada. The first report in the series, reference [2], provides an overview of the use of representation clauses and implementation-dependent features and contains a series of case study examples. A second report, reference [3], formulates a list of questions applicable to the evaluation of a particular compiler from the perspective of representation clauses and implementation-dependent features. This is followed by a discussion of experimental procedures and methodologies, in reference [4].

The purpose of this report is to provide a qualitative assessment of the support of representation clauses and implementation-dependent features in Ada provided by the VAX Ada compiler. Thus, the questions raised in reference [3] are answered here; the emphasis is principally qualitative. A quantitative evaluation of the VAX Ada compiler can be performed from the perspective of reference [4].

The decision to assess the VAX Ada compiler was motivated by the following reasons. First, the VAX Ada compiler was readily available and required no start-up time to become familiar with it. Second, VAX Ada is a popular product and one that is in widespread use. Thus, the results would have a greater impact by addressing a large audience. Finally, the project under which this work has been performed will be using the VAX Ada compiler to implement an application representative of typical real-time embedded systems, and an assessment of the support provided for representation clauses and machine-dependent features would benefit that task.

This report is organized in the following manner: In Chapter 2, we present a discussion of the qualitative results applicable to the use of representation clauses for the VAX Ada compiler and provide a subjective interpretation of the detailed results. In Chapter 3, the relation between the results obtained and the examples presented in reference [2] is discussed. This discussion illustrates the use of qualitative results, such as reported here, for application-specific problems. A brief summary appears in Chapter 4, followed by a list of the applicable references. The actual results, which are answers to specific questions, are provided in Appendix I.

This report has been prepared by the Ada Embedded Systems Testbed Project at the Software Engineering Institute (SEI). The SEI is a federally funded research and development center (FFRDC) sponsored by the Department of Defense and established and operated by Carnegie Mellon University. This report was prepared while the authors were on sabbatical leave at the SEI.

2. Discussion

A basic goal of reference [3] was to define a set of questions relevant to the assessment of a particular compiler from the perspective of representation clauses and implementation-dependent features. The questions, which appear in reference [3], are of a qualitative, as well as quantitative, nature. This chapter provides responses to those questions for the VAX Ada compiler. The emphasis here is on a qualitative level of assessment, and the reported results emphasize this aspect of evaluation. In other words, the focus is on whether a particular aspect of representation clauses and implementation-dependent features is supported and on the amount of support. Questions which are principally quantitative in nature may be evaluated according to the methodology described in reference [4]. This delineation of VAX Ada compiler support is quite different from the issue of VAX Ada compiler performance. While a complete assessment of support and performance of representation clauses for the VAX Ada compiler is not yet available, we believe these results will be of interest to the application development community.

We based the responses to the questions on the applicable documentation and selected generated code of the VAX Ada compiler, Version 1.3 with VMS Version 4.5. The questions and answers are presented in Appendix I. Relevant documents are cited in the following manner:

1. VLRM = *VAX Ada, Language Reference Manual* [5]
2. VRT = *VAX Ada, Programmer's Run-Time Reference Manual* [6]
3. VALM = *VAX/VMS, Volume 7, Macro and Instruction Set Reference Manual* [7]
4. DAP = *VAX Ada, Developing Ada Programs on VAX/VMS* [8]
5. RN = *VAX Ada, Version 1.1, 1.2, 1.3, Release Notes* [9]
6. DEC = [10]
7. VLINK = *VAX/VMS Linker Reference Manual* [11]

The questions have been divided into functional categories. For many questions, a full response can only be obtained in terms of detailed assessment, and such questions have been so indicated. The detailed assessment of such questions, whose response may involve performance assessments, requires quantitative procedures. Here we use the term quantitative in a broad sense. For example, a quantitative response may also include detailed examination of generated assembler code to extract information about the internals of the assessed compiler. The detailed responses for the quantitative questions can be obtained using reference [4] as a guide.

Although the responses presented in Appendix I are qualitative in nature, they contain a considerable amount of information. Naturally, the responses to the questions may stand on their own merit. However, many users will desire an overview of the support provided for representation clauses and machine-dependent features. To this end, we present a synopsis based on a subjective evaluation and the following set of subjective criteria:

1. **Full Support**: The compiler provides support for the language feature subject to natural limitations imposed on the hardware implementation.

2. **Support with Minor Limitations**: The compiler provides support for the language feature subject only to minor limitations. This implies, then, that the support provided should be satisfactory for many applications.

3. **Support with Major Limitations**: The compiler provides support for the language feature but there are major limitations. This means that use of the indicated language feature in many applications would be difficult.

4. **Unsupported**: The compiler provides no support for the language feature.

We stress that the criteria listed above are subjective in nature. There may be cases where a particular category is supported with only minor limitations. It may be, however, that the minor limitations could cause a serious problem for some applications.

The results presented below basically correspond to the categories appearing in Appendix I. One category not present in Appendix I yet deemed sufficiently important to include in this subjective evaluation is that of support facilities, in which we refer specifically to documentation and debugging facilities. Below, each category title is given, followed by its subjective rating. The rating is then followed by explanatory information where it is deemed relevant.

1. **Pragma OPTIMIZE**: Full Support.

2. **Data Types Supported**: Full Support. VAX Ada provides additional representations of integer and floating-point types beyond the scope of the language. Note, however, that fixed-point types are restricted to a length of 32 bits.

3. **Pragma PACK**: Full Support. Note that fixed- and floating-point types may not be packed; however, these objects are stored as fixed-length quantities.

4. **Length Clauses**: Supported with Minor Limitations. Fixed-point types must be represented according to a constant size. This could present problems for certain applications.

5. **Enumeration Representation Clauses**: Full Support.

6. **Record Representation Clauses**: Supported with Minor Limitations. Where a record contains a component of a fixed-point type, restrictions on the size of the component may be a problem due to the limitations imposed by the compiler.

7. **Address Clauses**: Supported with Major Limitations. Note that an address clause is only supported for objects.

8. **Data Conversion and Assignment**: Full Support. The result of UNCHECKED_CONVERSION when source and target type differ in length may cause problems for some applications.

9. **Representation Attributes**: Full Support. VAX Ada also provides two additional representation attributes which may be of benefit.

10. **Pragma INTERFACE**: Full Support. While the ability to interface with other languages is supported, it may involve considerable labor to achieve the desired result. Note that VAX Ada allows users to specify the details of parameter passing.

11. **Support Facilities**: Full Support. In general we found the documentation and debugger to be excellent. Note, however, that the release notes are difficult to use.

The support for representation clauses and implementation-dependent features by VAX Ada exhibits some variation. In some cases, such as data types, there is more support than required by the

language. In other cases, notably address clauses, there is only minimal support, or the support provided has restrictions as to details of implementation. The results illustrate the compiler-dependent aspects of support provided for representation clauses and implementation-dependent features, and point out that the selection of a particular compiler must be made with extreme care.

The results presented in this report are of a qualitative nature. They describe the support for a particular aspect of representation clauses and implementation-dependent features. Support for a language feature is clearly different from the performance of the compiler or the assessment of the effect of use of the feature. Issues of the latter type are principally quantitative in nature.

3. Relation to Examples in Volume I

In the preceding chapter, we presented a synopsis of the support provided by the VAX Ada compiler for representation clauses and implementation-dependent features. The synopsis abstracted some general results from the detailed answers.

It may be well, however, to consider the manner in which a report such as this would be used by application developers. This is clearly an important issue and is now demonstrated. Thus, in the first volume of this series, reference [2], a number of case study examples were presented. Those examples were drawn from the mission-critical systems community and contain certain characteristics representative of problems typically encountered.

A brief statement of the problem for each example in reference [2] is presented below. The implementation of the particular example is then considered, based on the VAX Ada compiler which is hosted and targeted for the MicroVAX. The ability to affect a solution to the stated problem is presented. The discussion is based on the results presented in Appendix I. That is, we provide specific references to the questions (and answers) deemed relevant for the example under consideration. It is believed that such a procedure illustrates the manner in which a report such as this may be used by application developers.

In the following paragraphs, references are made to the questions and answers appearing in Appendix I. Questions and answers are referenced by category letter followed by the number of the question. For example, "F.3" refers to question 3 under category F, which is Record Representation Clauses.

In Section 5.2 of the first report in this series, reference [2], an introductory example was given illustrating the use of representation clauses. The example was that of a message header that appears in every message used for communications between a shipboard Inertial Navigation System (INS) and some external computer (EC). This example illustrates the length, enumeration, and record clauses, with data of type integer and enumeration. The requirements for this example are within the following restrictions:

1. The size specified in a length clause for a discrete type must not exceed 32 bits (as stated in D.2).
2. Integer codes given for enumeration literals in an enumeration representation clause must be representable as a 32-bit integer (as stated in E.1).
3. Record components of a discrete type must be specified with 32 bits or less in a component clause (see F.3).
4. Record components are allowed to overlap storage unit boundaries (see F.4).

Also, the assumed value of SYSTEM.STORAGE_UNIT for the examples is equal to eight which is the default value for VAX Ada. Thus, this implementation would be valid for VAX Ada. A note should be made that as stated in A.2, VAX Ada ordering of bits and storage units is right to left, and these examples use left to right ordering. Thus, when examining the actual contents of memory, the results will be different from those implied in the diagrams. If right to left ordering is required, say by some external system, then the code presented in these examples will need to be changed.

The second example, given in Section 5.3 of reference [2], is that of a Test Message which is used to test the communications interface between the INS and EC. This example contains a message header (implemented in the previous example) and integer test data. As in the last example, the requirements here are within the restrictions listed in F.3 that the specified size for a discrete type must not exceed 32 bits, and in F.4 that storage unit boundary overlaps are allowed. Therefore, this implementation would also be valid for VAX Ada.

The next example, in Section 5.4 of reference [2], is slightly more complicated than the previous examples in that it contains both fixed- and floating-point data. A navigation message is the example, and it is used to send data such as ownship latitude, longitude, and speed to an EC. This example contains a requirement that fixed-point data be represented in less than 32 bits. VAX Ada represents fixed-point types in 32 bits as discussed in B.2. Based on this and the restriction listed in F.3 that components of all types except discrete types must be specified in a record representation clause with the actual size of the component, this implementation is illegal for VAX Ada. To implement this example on VAX Ada, a conversion routine must be considered that converts the actual data field in the message to the default VAX Ada 32-bit fixed-point representation. This will be the representation from which calculations are performed.

The fourth example, Section 5.5 of reference [2], illustrates an implementation of analog conversion. In this example, ship heading, roll, and speed data is DMA mapped to particular addresses and must be converted into actual values. The requirement that data are DMA mapped was met by the use of address clauses. This example would not be valid on VAX Ada for the following reasons:

1. SYSTEM.STORAGE_UNIT is 24 bits for this example, and VAX Ada SYSTEM.STORAGE_UNIT is 8 and this cannot be changed with pragma STORAGE_UNIT (as indicated in J.4).
2. Pragma SHARED is not supported (see J.6).
3. VAX Ada requires integer literals used in an address clause be converted to a value of type SYSTEM.ADDRESS by invoking a VAX Ada specific function (as given in G.1).

Note that the latter two constraints are easily rectified with the use of VAX Ada defined pragma VOLATILE instead of pragma SHARED and with the inclusion of the appropriate function calls for the integer to SYSTEM.ADDRESS conversion.

In Section 5.6 of reference [2], an example of a message checksum was given. This function computes the checksum of the navigation message that was implemented in Section 5.4 of reference [2] and discussed above. Recall that the implementation of the navigation message was invalid for VAX Ada. Thus, this checksum implementation would be invalid since it depends on that message implementation. A general-purpose checksum routine based on the use of the generic function UNCHECKED_CONVERSION and dealing with valid message implementations would be legal for VAX Ada since the following restrictions are met:

1. The size specified for a discrete type must not exceed 32 bits (as stated in D.2).
2. VAX Ada supports UNCHECKED_CONVERSION (reported in H.3).
3. The target type for the conversion is a legal type for VAX Ada (as shown in H.4).

The final example given in reference [2], Section 6.4.2, illustrates the use of pragma INTERFACE. The problem in this example is the need to allocate and access some data structure containing data and status information for an INS gyro. As part of this problem, a conversion must be performed to obtain a 32-bit, fixed-point quantity that has 15 bits of precision from arbitrary fixed-point quantities. This conversion was performed by an assembler routine that is accessed via pragma INTERFACE with representation attributes providing the appropriate parameters to that routine. This example is compilable on VAX Ada with the following providing relevant information to verify this:

1. There are no restrictions on the use of 'ADDRESS (see I.1).
2. There are no restrictions on the use of 'FIRST_BIT (see I.4).
3. There are no restrictions on the use of 'LAST_BIT (see I.5).
4. Pragma INTERFACE is supported (see J.5).

The preceding has illustrated the use of results presented in this report to assess problems that are representative of mission-critical systems. The purpose in presenting the above discussion, therefore, is to illustrate how reports such as this may be used. It is to be noted that the emphasis in the above discussion has been on application of qualitative results. That is, the emphasis is more toward obtaining a solution to a problem, as opposed to an assessment of how "well" the solution implements the problem requirement. Results of the latter type, requiring performance information, must be considered from the perspective of a quantitative evaluation of the VAX Ada compiler.

4. Summary

This report is one of a series dealing with the use of representation clauses and implementation-dependent features in Ada. This report provides a qualitative assessment of the VAX Ada compiler, Version 1.3. Subjective criteria were established to provide an overall assessment of the support provided by this compiler for representation clauses and implementation-dependent features. In general, this compiler provides support with minor limitations for the implementation of representation clauses and implementation-dependent features. Some exceptions, however, have been noted.

This report may be used in conjunction with the results of a detailed experimental assessment of the VAX Ada compiler to determine its suitability for specific applications.

References

1. *Reference Manual for the Ada Programming Language*, Department of Defense MIL-STD-1815, 1983.

2. B. Craig Meyers and Andrea L. Cappellini, *The Use of Representation Clauses and Implementation-Dependent Features in Ada: I. Overview*, CMU/SEI-87-TR-14, ESD-TR-87-115, July 1987.

3. B. Craig Meyers and Andrea L. Cappellini, *The Use of Representation Clauses and Implementation-Dependent Features in Ada: IIA. Evaluation Questions*, CMU/SEI-87-TR-15, ESD-TR-87-116, July 1987.

4. B. Craig Meyers and Andrea L. Cappellini, *The Use of Representation Clauses and Implementation-Dependent Features in Ada: IIB. Experimental Procedures*, CMU/SEI-87-TR-18, ESD-TR-87-126, July 1987.

5. *VAX Ada, Language Reference Manual*, DEC, February 1985.

6. *VAX Ada, Run-Time Reference Manual*, DEC, February 1985.

7. *VAX/VMS, Volume 7, MACRO and Instruction Set Reference Manual*, DEC, September 1984.

8. *VAX Ada, Developing Ada Programs on VAX/VMS*, DEC, February 1985.

9. *VAX Ada, Version 1.1, 1.2, 1.3*, Release Notes, DEC, December 1986.

10. Digital Equipment Corporation, private communication, June 23, 1987.

11. *VAX/VMS, Volume 4B, Linker Reference Manual*, DEC, September 1984.

Appendix I: Questions Relevant to the Use of Representation Clauses and Implementation-Dependent Features

A. General

1. **What is the basic unit of SYSTEM.STORAGE_UNIT? (This is useful when defining record layouts.)**

One byte, or eight bits. [VLRM, page 13-11]

2. **What is the ordering of allocation for storage units? Is it left-to-right or right-to-left with respect to each other? How are bits numbered within storage units? Is it left-to-right or right-to-left? Does the numbering always begin with zero? (This is useful when defining record layouts and verifying the actual allocation of record layouts.)**

The storage unit ordering scheme used by VAX is right to left and bits within a storage unit are also numbered right to left, starting with 0. The following example illustrates this. Suppose we had the value 1 stored in address 6, the value 0 stored in address 7, and the value 15 stored in address 8. If we examined 24 bits of storage starting at address 6, we would get the following:

00001111 00000000 00000001

3. **It is also appropriate to consider the role of the underlying architecture, particularly regarding data conversions from representation clauses to other formats. Does the machine include instructions for inserting and extracting bit-length fields? What are the restrictions on the use of such instructions (for example, what is the maximum field size to which an instruction may be applied)?**

The VAX instruction set provides three particular instructions which may be used to insert and extract variable-length bit fields. One instruction, opcode INSV, inserts the high-order bits of the source field into the target field specified by position, size and base; where base is the effective address (byte aligned), position is the beginning bit (within the byte addressed) of the field; size is the number of bits in the field.

A second instruction, opcode EXTZ, replaces the high-order bits of the target field with the sign-extended source field specified by position, size and base. A third instruction, opcode EXTZV, is similar to the EXTZ instruction. In this case, the target field is replaced with a zero-extended source field. In all three cases above, the size of the field to insert or extract must be smaller than 32 bits.

Note that the instruction set does not provide for any general-purpose bit insert and extract routines. Such an instruction would, for example, allow the source and target fields to both be specified by a position, size and base, and not necessarily restricted to accessing high-order bits in some storage unit.

[VMIS, pages 9-40, 9-42]

4. **Is pragma OPTIMIZE supported? If so, are there any restrictions on its use?**

Yes. This pragma is only allowed immediately within a declarative part of a body declaration.

[VLRM, Appendix B, page B-8]

5. **The use of representation clauses may present unusual problems throughout design and coding. What facilities exist for verifying results when representation clauses are used? We are speaking here of the debugger; thus, are there restrictions on the use of the debugger when representation clauses are used?**

The debugger provides commands for accessing addresses of variables and examining the contents of those addresses (in binary, decimal, hexadecimal, or octal) which can be used to verify the results of a representation clause. [DAP, page 7-55] The debugger also allows simultaneous display of source code and machine code, stepping through the machine code and displaying the contents of registers, for example.

The debugger does not provide support for the biasing mechanism discussed in question F.3; that is, the debugger displays the biased value stored in a record component instead of the actual value when queried. [RN]

Also, according to a DEC representative, there is no debugger support for variables specified by address clauses.

6. **Does the compiler provide a load map that contains sufficient details to identify the location of quantities specified using representation clauses?**

The Ada Compilation System (ACS) linker provides an option to create an image map. There are several options for tailoring the information provided in the map. Information such as object module synopsis, module relocatable synopsis, and a symbol cross-reference section for global symbols can be obtained.

The information provided is not sufficient since it does not, for example, give address location of variables/objects within a module.

[VLINK]

7. **Are there any restrictions on representation clauses?**

In addition to the specific restrictions discussed in this report, the following are not allowed:

- a representation clause for a generic formal type or a type that depends on a generic formal type

- a representation clause for a composite type that has a component or subcomponent of a generic formal type or a type derived from a generic formal type

[VLRM, Appendix F, page F-7]

8. **Compiler implementors currently have the option as to what degree, if any, the features in Chapter 13 of the** *Reference Manual for the Ada Programming Language* **will be supported. It is conceivable that upgraded versions of an implementation will enhance the support originally available for such features as representation clauses. How is the documentation upgraded? Is it by release notes or page changes? The manner in which this is accomplished can affect the ease with which documentation can be used.**

For our documentation, we received release notes in the form of a VMS HELP facility. The format of the release notes made it very difficult to find relevant information. Page changes would be more convenient for the user.

B. Data Types Supported

1. **What are the basic implementations of integer types?**

VAX Ada provides for three implementations of integer types. These are: INTEGER, represented in 32 bits with a range of -2^{31} .. $2^{31}-1$; SHORT_INTEGER, represented as 16 bits with a range of -2^{15} .. $2^{15}-1$; and SHORT_SHORT_INTEGER, represented as 8 bits with a range -128 .. 127. [VLRM, page 3-20]

2. **What are the basic implementations of fixed-point types?**

Each fixed-point type in VAX Ada occupies 32 bits [VLRM, page 3-34]. Values of fixed-point types are represented as signed, twos complement (binary) numbers with an implicit binary scale factor. [VRT, page 3-12]

Note the reference to an implicit binary scale factor in the above. As an illustration, a fixed-point type which has a value of *small* equal to 2^{-6} and has the value 1.5 is represented internally in the following binary format:

00000000 00000000 00000000 01100000

The low-order six bits in the above are for storage of the fraction value, which, in this case, is 0.5. The precise manner in which a fixed-point type scale factor is implemented is not documented.

3. **What are the basic implementations of floating-point types?**

Four different implementations of floating-point types are supported by VAX Ada. They are: FLOAT, represented as 32 bits which provides 6 (decimal) digits of precision; LONG_FLOAT, which may be represented as either 64 bits with 9 digits of precision or 64 bits with 15 digits of precision; and LONG_LONG_FLOAT, represented as 128 bits with 33 digits of precision. The choice of representation for the LONG_FLOAT type is determined by the use of a VAX-specific pragma. [VLRM, page 3-27]

4. **Does the compiler provide predefined, unsigned data types? If not, is it permissible for a user to define these types? For example, is the following legal:**

```
type Unsigned_Small_Int is range 0 .. 7;
for Unsigned_Small_Int'SIZE use 3;
```

VAX Ada supports several unsigned data types and operations for those types. For example, package SYSTEM contains the following:

```
type UNSIGNED_BYTE is range 0 .. 255;
for UNSIGNED_BYTE'SIZE use 8;
```

And, functions for 'and', 'not', 'or' and 'xor' are provided for this type. [VLRM, page 13-23]

VAX Ada also allows the user to define types for unsigned integers. [VRT, page 3-21]

C. Pragma PACK

1. **Does the compiler support the use of pragma PACK?**

Yes. [VLRM, page 13-3]

2. **What restrictions are placed on the use of pragma PACK? For example, are there certain types that may or may not be packed?**

Pragma PACK has an effect in VAX Ada for record or array declarations only if the record or array components are packable. A component is packable if its type allows it to be aligned on an arbitrary bit boundary. The following types are packable:

- integer
- enumeration
- record: where the record type has a compile-time constant size less than or equal to 32 bits and all its components are packable. The size of a record type is computed as the position of the last component that physically appears in the record layout plus the size of the last component [VRT, page 3-15]
- array: if the array type is itself a packed array of packable arrays or if it is an array of 1-bit components

The following types are not packable:

- fixed point
- floating point
- address
- access
- task

[VRT, page 3-19]

D. Length Clauses

1. **Does the compiler support the use of length clauses? What are the restrictions on their use?**

Yes. There are no documented restrictions. [VLRM, page 13-4]

2. **Are there restrictions on the use of the SIZE attribute designator in a length clause?**

In VAX Ada, for a discrete type the size specified must not exceed 32 bits (object sizes may be increased by the compiler for optimization purposes [RN]). The given size affects the internal representation of integer and enumeration types as follows:

- *integer*: high-order bits are sign-extended
- *enumeration*: high-order bits are zero-extended when the range of the integer codes for the enumeration literals is positive and sign-extended when the range includes negative values [VRT, page 3-21]

For all other types, the given size must equal the default size for that type, e.g., for fixed-point types, the size specified must equal 32 (bits). [RN]

For discrete types, the given size becomes the default allocation for all objects and components (in arrays and records) of that type. [VLRM, page 13-5]

3. **Are there restrictions on the use of the STORAGE_SIZE attribute designator in a length clause?**

The use of this specification in a length clause determines the amount of memory from which objects of the specified access type are allocated, and VAX Ada refers to that allocatable memory as the collection size. The collection size specified is interpreted as follows:

- *If greater than zero*: the specified size is rounded up to the next integral number of pages (where a page is 512 bytes), and that value is used as the initial size of the collection. The size of the collection is not extended when the amount is exhausted and the exception STORAGE_ERROR is raised. [VRT, page 3-17]
- *If equal to zero*: no initial storage is allocated; storage is allocated on an as needed basis until all virtual memory is depleted. (This is the default behavior.)
- *If less than zero*: a CONSTRAINT_ERROR is raised.

[VLRM, page 13-5]

When there are several collection sizes to specify, it is recommended that values chosen for the group of collection sizes be integrally related to improve efficiency of a program. For example, the group of values 512*4, 512*8, and 512*12 is better than the group of values 512*2, 512*7, and 512*13. [RN]

4. **Are there restrictions on the use of the SMALL attribute designator in a length clause?**

Consider a fixed-point type declared in the form:

<type_name> ***is delta*** <delta_value> ***range*** <range_clause>

The default representation for a fixed-point type is that the value of *small* will be the largest integral power of two which is not greater than the given <delta_value> and which satisfies the <range_clause>.

When a value of *small* is specified in a length clause, the specified value may not exceed the default. [VLRM, page 13-6]

5. **When using a SIZE attribute designator in a length clause, the** *Reference Manual for the Ada Programming Language* **states that the value of the expression specifies an upper bound for the number of bits to be allocated. The presence of a range constraint or the use of a predefined type implicitly defines the maximum number of bits required to allocate objects. If extra bits are specified in the length clause, are these extra bits allocated by the compiler?**

Not only will VAX Ada allocate the extra bits if specified, but, as explained in question I.2, VAX Ada rounds the size of a variable up to a multiple of 8 bits, though record components are subject to packing and representation requirements. (Also see question D.2.)

6. **Suppose a type, with associated length clause, has been specified storage where the number of bits is not sufficient to store the specified range of values. For example, suppose an integer type with range 10 .. 13 is defined, and three bits of storage are allocated for that type. Is an error generated for this case? If no error is generated by the compiler, how is a case such as this treated?**

How such cases are handled by VAX Ada is not documented. [VLRM, page 13-4] A test was performed with the following:

```
type S is range 10 .. 13;
for S'Size use 3;
```

and an error was generated at compilation. Detailed experimentation is necessary to investigate this further.

7. **What impact does the length clause have on the packing algorithm of composite types?**

No impact. [DEC]

8. **What is the effect of pragma OPTIMIZE (TIME) on storage allocation when length clauses are used?**

No effect. [DEC]

9. **What is the effect of pragma OPTIMIZE (SPACE) on storage allocation when length clauses are used?**

No effect. [DEC]

10. **What is the effect of pragma PACK on storage allocation when length clauses are used?**

No effect. [DEC]

E. Enumeration Representation Clauses

1. **Does the compiler support the use of enumeration representation clauses? What are the restrictions on their use?**

Yes. Each expression for the integer code must be representable as a 32-bit integer. Integer codes can be negative. [VLRM, page 13-9]

2. **Consider an enumeration type and associated enumeration representation clause where the enumerated values specified are not contiguous integers, such as:**

```
type Name is (Name_1, Name_2, Name_3, Name_4);
for Name use
  ( Name_1 => 1, Name_2 => 5, Name_3 => 12, Name_4 => 163 );
```

The enumeration type may not be efficiently implemented because of the noncontiguous nature of the integers specified in the enumeration representation clause, illustrated above. Hence, how are enumeration types represented internally, particularly in the case where enumeration clauses are specified with noncontiguous values?

Detailed experimentation is required for resolution of this.

3. **What is the effect of pragma PACK on storage allocation when enumeration representation clauses are used?**

No effect. [DEC]

4. **What is the effect of pragma OPTIMIZE (TIME) on storage allocation when enumeration representation clauses are used?**

No effect. [DEC]

5. **What is the effect of pragma OPTIMIZE (SPACE) on storage allocation when enumeration representation clauses are used?**

No effect. [DEC]

F. Record Representation Clauses

1. **Does the compiler support the use of record representation clauses? What are the restrictions on their use?**

Yes. There are no documented restrictions. [VLRM, page 13-9]

2. **What are the restrictions on the use of the alignment clause in a record representation clause?**

The alignment clause must be an integer in the range of 1 .. 512 and also an integral power of 2. In other words, the alignment clause must have a value of 2^n, where n is in the range zero through nine. Depending on how the record objects are allocated, the following restrictions exist:

- *For stack-allocated record objects*: alignment is restricted to values 2^n where n is in the range zero to two.
- *For dynamically allocated record objects*: since all dynamically allocated objects are at least longword (i.e., 4 bytes) aligned, alignment is restricted to values 2^n where n is in the range two through nine.
- *For statically allocated record objects*: there are no other restrictions.

[VLRM, page 13-10]

3. **What are the restrictions on the use of component clauses in a record representation clause?**

Legal specifications must specify enough bits in the range to represent the component type; otherwise the specification is illegal. Discrete type components must be specified with 32 bits or less. The size specified for components of all other types must be equal to the actual size of the component.

Components that are not packable must be allocated on a byte boundary, whereas packable components can be allocated without restrictions. (See also question C.2.)

[VLRM, page 13-11]

When record components are specified to be stored in an inadequate amount of space, the values are biased — that is, predictably altered. This biasing occurs only when a component clause specifies a very small amount of storage for the particular component. The method of biasing is to subtract the COMPONENT_SUBTYPE'FIRST from the component value and store the remaining value in that location. Restrictions on when biasing occurs, e.g., what constitutes a "small" amount of storage, are not documented, and detailed experimentation must be employed to gather such information. [RN]

4. **Are there restrictions on the overlap of record components with respect to the basic machine storage unit? For example, if a machine has a SYSTEM.STORAGE_UNIT equal to 16 bits, is it permitted to have components of a record that are larger than this value?**

If the specification is legal (as discussed in question F.3), boundary overlaps are permitted. That is, the size of record components can be greater than eight bits (SYSTEM.STORAGE_UNIT). Also, for example, a component can begin at storage unit five, bit seven and have a length of five bits, thus ending at storage unit six, bit three. [VLRM, page 13-11]

Variants can also be overlapped. [RN]

5. **Consider the case when a record is specified with a record representation clause. Where is a record component placed that has no associated component clause?**

Components named in a clause are allocated first, then the components not in the clause are allocated in the order in which they are written in the type declaration. Specifically, where a component is placed that has no clause depends upon the packed/unpacked allocation algorithm in effect and requires detailed experimentation to resolve. [VLRM, page 13-11]

Preliminary results show that unnamed components are placed immediately after the last specified component and in the order in which they appear in the record type. For example, a test was performed with the following:

```
type Rec is
  record
    C1 : Integer;
    C2 : Boolean;
    C3 : Boolean;
    C4 : Integer;
  end record;

for Rec use
  record
    C2 at 0 range 0 .. 7;
    C4 at 100 range 0 .. 31;
  end record;
```

With component C2 allocated at address 9588, component C4 was allocated at address 9688, component C1 was allocated at address 9692, and component C3 was allocated at address 9696.

6. **What is the effect of pragma OPTIMIZE (TIME) on storage allocation when record representation clauses are used?**

No effect. [DEC]

7. **What is the effect of pragma OPTIMIZE (SPACE) on storage allocation when record representation clauses are used?**

No effect. [DEC]

8. **What is the effect of pragma PACK on storage allocation when record representation clauses are used?**

No effect. [DEC]

G. Address Clauses

1. **Does the compiler support the use of address clauses? What are the restrictions on their use?**

VAX Ada supports address clauses with the restriction that the simple name specified must be the name of a variable. VAX Ada does not support address clauses that name constants, subprograms, packages, tasks, or single entries. VAX Ada pragmas for importing or exporting objects are not allowed in combination with an address clause, the result of which would be to ignore the pragma. [RN]

The simple expression given in the address clause must be of type SYSTEM.ADDRESS, and VAX Ada provides a conversion function, TO_ADDRESS, in package SYSTEM which converts a universal integer to type ADDRESS. This function must be invoked when specifying addresses with integer literals. [RN]

The release notes do not specify what range of universal integers will be converted to valid addresses (e.g., addresses that are user accessible). The range of universal integers converted to addresses is 0 .. MAX_INT. [DEC]

2. **What is the type SYSTEM.ADDRESS?**

The declaration of type SYSTEM.ADDRESS is not documented [VLRM, page 13-17; Appendix F, page F-3]. VAX Ada interprets values of type SYSTEM.ADDRESS as integers in the range 0 .. MAX_INT, and they refer to addresses in the user half of the VAX address space. If an address falls outside this range, the code would be erroneous, though no explicit check would be performed; thus, there is no guarantee that an exception would be raised. [VLRM, page 13-17]

The storage size of objects of type SYSTEM.ADDRESS is 32 bits. [VRT, page 3-18]

3. **What is the effect of pragma OPTIMIZE (TIME) on storage allocation when address clauses are used?**

No effect. [DEC]

4. **What is the effect of pragma OPTIMIZE (SPACE) on storage allocation when address clauses are used?**

No effect. [DEC]

5. **Does the compiler enforce strong typing in the presence of address clauses? For example, is the following recognized as erroneous by the compiler:**

```
type T_1 is range 0 .. 100;
O_1 : T_1;
for O_1use at 16#1000#;
```

```
type T_2 is digits 2 range 0.0 .. 100.0;
O_2 : T_2;
for O_2 use at 16#1000#;
```

This example compiles without errors on VAX Ada with the appropriate function calls added as discussed in G.1. Execution of this example will depend on whether the specified addresses are accessible.

6. **Does the compiler or linker recognize potential conflicts when address clauses are used? For example, suppose an address clause is present that references some address, say X. Assume that the address X is such that it lies within the address space of generated code. How is this case treated by the compiler and/or linker?**

No diagnostics are given by the compiler or linker for this case. [DEC]

H. Data Conversion and Assignment

1. **How is conversion accomplished between values of a type specified by the default representation and a type specified with a representation clause? (This refers to the use of a *new* (derived) type that is defined in terms of a representation clause.)**

Detailed experimentation is required for resolution of this.

2. **For conversions between objects of different types, does the compiler produce in-line code or generate a call to a library routine to accomplish the conversion?**

Preliminary results where an integer object is explicitly converted to a floating-point type (and vice versa) and an integer object is converted to a fixed-point type (and vice versa) show the VAX Ada compiler produces in-line code to accomplish the conversion. Detailed experimentation is required to fully investigate this issue.

3. **Is support of the generic function UNCHECKED_CONVERSION provided?**

Yes. [VLRM, page 13-55]

4. **Are there any restrictions on the use of UNCHECKED_CONVERSION? For example, are there any restrictions on the source and target types for UNCHECKED_CONVERSION? Do they have to be the same size?**

There are no restrictions on the actual subtype corresponding to the formal type SOURCE. The following are restrictions on the actual subtype corresponding to the formal type TARGET:

- must not be an unconstrained array type
- must not be an unconstrained type with discriminants

When the target type is a type with discriminants, the resulting value from a call to an instantiation of UNCHECKED_CONVERSION is checked to assure that the discriminants satisfy the constraints of the actual subtype.

For size inconsistencies between source and target, the following actions will be taken:

- *If the source size is greater than the target size*: then the high-order bits are truncated.
- *If the source size is less than the target size*: then the value is extended with zeros.

[VLRM, page 13-56]

I. Representation Attributes

1. **What are the restrictions on the use of the 'ADDRESS representation attribute? How does the compiler interpret the use of this attribute?**

There are no documented restrictions on the use of this attribute. Programming considerations for working with address values are given in [VRT, page 8-3]. VAX Ada interpretations of this attribute are the following:

- *Variables*: the value is the actual address of the variable (which can be either statically or dynamically allocated). Use of the attribute forces the variable to be stored in memory rather than a register. If the variable is not stored on a byte boundary, the value is the address of the lowest byte that holds the variable
- *Constants*: the value is the memory address of the constant, though if queried twice, the value returned may not be the same address
- *Named Numbers*: the value is ADDRESS_ZERO
- *Access Objects*: the value is the address of the designated object and the attribute is subject to an ACCESS_CHECK
- *Record Objects*: the attribute is the address of the first component and is subject to DISCRIMINANT_CHECK for an object in a variant part
- *Array Component or Slice*: the attribute is the address of the component or slice and is subject to INDEX_CHECK for the component or slice
- *Package Units*: the value is ADDRESS_ZERO
- *Subprograms*: when named in a pragma EXPORT, the value is the address that would be exported [see VLRM, page 13-45, for a discussion on exporting subprograms]. For subprograms not exported, the value is ADDRESS_ZERO

[VLRM, page 13-28]

2. **What are the restrictions on the use of the 'SIZE representation attribute? How does the compiler interpret the use of this attribute?**

There are no documented restrictions on the use of this attribute.
VAX Ada interpretations of this attribute are the following:

- *Type or Subtype*: the value must be in the range 0 .. MAX_INT, where MAX_INT equals 2^{31}-1; otherwise NUMERIC_ERROR is raised
- *Variables or Constants*: the value is its size in bits
- *Named Numbers*: the value is 0

- *Record Component*: the attribute is the size of the component and is subject to a DISCRIMINANT_CHECK for an object in a variant part
- *Array Component or Slice*: the attribute is the size of the component or slice and is subject to an INDEX_CHECK for the component or slice

[VLRM, page 13-29]
In VAX Ada, the results of T'SIZE and O'SIZE, where T is a type or subtype and O is an object of that type T, will sometimes be different. T'SIZE always returns the minimum number of bits needed to allocate any possible object of type T, whereas O'SIZE returns the number of bits actually allocated for an object. In VAX Ada, the size of a variable is always rounded up to a multiple of 8 bits, where bits are padded to enforce at least byte alignment. For discrete types, the size is always 8, 16, or 32. For example, suppose we had the following:

```
type Bool17 is new Boolean;
for Bool17'SIZE use 17;

Bool17_Object : Bool17;
```

The value of Bool17'SIZE is 1, whereas the value of Bool17_Object'SIZE is 32 (17 bits plus 15 padding bits).

For C'SIZE, where C is a component of a record, if a record representation clause is present, padding does not occur. When pragma PACK is specified the amount of padding, if any, is determined by the storage allocation algorithm used by the compiler.

[VRT, page 3-27]

3. **What are the restrictions on the use of the 'POSITION representation attribute for a record component?**

There are no documented restrictions. [VLRM, page 13-30]

4. **What are the restrictions on the use of the 'FIRST_BIT representation attribute for a record component?**

There are no documented restrictions. [VLRM, page 13-30]

5. **What are the restrictions on the use of the 'LAST_BIT representation attribute for a record component?**

There are no documented restrictions. [VLRM, page 13-30]

6. **What is the effect of pragma OPTIMIZE (TIME) on the values of the representation** attributes?

Detailed experimentation is required for a resolution of this.

7. **What is the effect of pragma OPTIMIZE (SPACE) on the values of the representation** attributes?

Detailed experimentation is required for a resolution of this.

J. Miscellaneous

1. **Suppose an object has been allocated storage where the number of bits is not sufficient to store the specified range of values. For example, suppose an object has been allocated three bits of storage, but is specified to be in the range 10 through 13. Is an error generated for this case? If no error is generated by the compiler, how is a case such as this treated?**

For objects that are components of a record where the component clause given in the record representation clause does not specify adequate storage, VAX Ada provides a biasing mechanism discussed in question F.3. It should be noted, though, that an example of this kind given in VLRM, page 13-13, states that this example is illegal.

How VAX Ada handles cases other than when an object is a component of a record is not documented. See D.6. [VLRM, page 13-4]

2. **Does the compiler support the use of pragma SUPPRESS?**

No. [VLRM, page 11-15]

3. **What restrictions are placed on the use of pragma SUPPRESS? For example, can every check be suppressed?**

Not applicable to VAX Ada. See also J.6.

4. **Is pragma STORAGE_UNIT supported? If so, are there any restrictions on the argument?**

Pragma STORAGE_UNIT is supported with the restriction that the value given *must* be 8 (bits). Since SYSTEM.STORAGE_UNIT is by default 8, one can say that VAX Ada *does not really* support this pragma. [VLRM, page 13-18]

5. **Is pragma INTERFACE supported? If so, are there any restrictions on the allowable forms and places of parameters and calls?**

Pragma INTERFACE is supported but is coupled to the VAX Ada defined pragmas IMPORT_FUNCTION, IMPORT_PROCEDURE, and IMPORT_VALUED_PROCEDURE. Pragma INTERFACE is used in combination with one of the previous pragmas. [These pragmas are discussed in VLRM, Section 13.9a1.1, and RN.]

If pragma INTERFACE is used without one of the above pragmas, the following interpretation is taken by default:

- the language name specified is ignored
- if the subprogram name applies to a single subprogram, a default import pragma is assumed; thus, for a function the default is pragma IMPORT_FUNCTION and for a procedure the default is pragma IMPORT_PROCEDURE

- if the subprogram name applies to two or more subprograms (overloaded), the pragma applies to them all, and a warning is given if the appropriate import pragmas are not given for all subprograms

The subprogram name must be an identifier or string literal.

The following rules exist for the use of pragma INTERFACE:

- if a subprogram body is given later for a subprogram named with this pragma, the body is illegal

- this pragma is illegal if it names a subprogram body

- if two pragma INTERFACEs are given, the latter is illegal

If pragma INTERFACE and pragma INLINE are used together, pragma INLINE will always be ignored.

[RN]

[Refer to VLRM, Section 13.9a, page 13-37, for a detailed discussion of import/export pragmas (for subprograms, objects, and exceptions) and parameter passing, and to the release notes for added facilities.]

6. **Is pragma SHARED supported? If so, are there any restrictions on its use?**

No, it is not supported. [VLRM, page 9-26]

7. **Are there other implementation-dependent features supported such as pragmas or** attributes?

In addition to the VAX Ada supplied pragmas discussed in question J.5, the following relevant pragmas are provided by VAX Ada:

- Pragma AST_ENTRY specifies that the given entry may be used to handle a VAX/VMS asynchronous system trap. This is one alternative to using address clauses for handling interrupts.

- Pragma LONG_FLOAT specifies which representation, either D_FLOAT or G_FLOAT, to be used for the predefined type LONG_FLOAT.

- Pragma VOLATILE specifies that the given variable may be modified asynchronously and instructs the compiler to obtain the value of the variable from memory each time it is used. This pragma is an alternative to pragma SHARED.

- Pragma SUPPRESS_ALL specifies that all run-time checks in the enclosing compilation unit be suppressed.

[VLRM, Annex B]

The following representation attributes, which are relevant to the area of representation clauses and implementation-dependent features, are provided by VAX Ada:

- X'MACHINE_SIZE yields the number of machine bits to be allocated for variables of the type or subtype X and includes any padding bits for allocating a variable.

- X'BIT yields the bit offset within the storage unit that contains the first bit of storage allocated for the object X.

[VLRM, page 13-29]

Also, VAX Ada provides subprograms in package SYSTEM to read and write to device registers. [RN]

Table of Contents