Technical Report

CMU/SEI-93-TR-10
ESC-TR-93-187

The Use of ASN.1 and XDR for Data Representation in Real-Time Distributed Systems

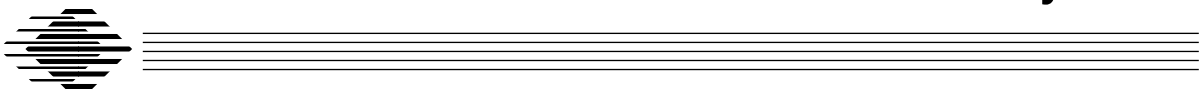
B. Craig Meyers
Gary Chastek

October 1993

Technical Report

CMU/SEI-93-TR-10
ESC-TR-93-187
October 1993

The Use of ASN.1 and XDR for Data Representation in Real-Time Distributed Systems



B. Craig Meyers
Gary Chastek

Open Systems Architectures 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.

Use of any trademarks in this report is not intended in any way to infringe on the rights of the trademark holder.

Table of Contents

List of Figures

List of Tables

The Use of ASN.1 and XDR for Data Representation in Real-Time Distributed Systems

Abstract: This report provides an overview of two standards that are used for data specification and representation in distributed systems. The standards considered are the Abstract Syntax Notation One (ASN.1) and the external data representation (XDR). Standards for data representation are appropriate for the development of real-time distributed systems, particularly loosely coupled, heterogeneous systems. The report presents an example of the use of each standard. Several performance metrics are also introduced that are suitable for comparing the space and time costs of using the different standards. Several issues are discussed that are appropriate to a system designer. An Ada implementation of ASN.1 *encode* and *decode* routines for floating point types is included in an appendix.

1. Introduction

Hard real-time systems are characterized by the presence of timing deadlines that must be met to assure system correctness. In the case of a distributed system, the deadlines are often characterized as *end-to-end* deadlines that involve multiple application programs. In the process associated with an end-to-end deadline, a sending application encodes data in some structure that a receiving application is able to decode. In the case of hard deadlines, performance of a system in regard to data manipulation is an important issue, particularly when the system can be composed of heterogeneous machines.

This report examines two well-known standards for data specification and representation, namely

- The Abstract Syntax Notation One (ASN.1). ISO/IEC 8824 [2] defines the specification of ASN.1, while ISO/IEC 8825 [3] defines the basic encoding rules (BER) that are used in the representation of data.

- The external data representation (XDR), defined in reference [4].

The orientation of this report is more toward the methods used to *represent* as opposed to *specify* data. For our purposes, data representation involves the encoding and decoding of data, usually for transfer between system elements. Data specification involves the use of some type of abstract syntax to describe data. While these are clearly related, we are concerned more with the manner in which data is encoded and decoded by elements of a distributed system.

This document is organized as follows: Chapter 2 presents an overview of the problems addressed. Chapter 3 provides an overview of the ASN.1 basic encoding rules. The use of

the external data representation is considered in Chapter 4. A comparison of these two approaches is then discussed in Chapter 5, which also contains a discussion of issues related to the two standards. A brief summary of the report appears in Chapter 7. Throughout this report, we use a representative example, that of a track update message, to illustrate the various possible encodings. Appendices A and B contain the BNF specifications for ASN.1 and XDR, respectively. Appendix C contains an example, written in Ada, of ASN.1 *encode* and *decode* routines.

The work reported in this document was performed by the Open Systems Architecture Project at the Software Engineering Institute (SEI). The SEI is a federally funded research and development center operated by Carnegie Mellon University under contract to the Department of Defense. This work was supported in part by the Navy Next Generation Computer Resources Program.

2. Overview

2.1. Problem Considered

A typical communication mechanism used in distributed systems is a remote procedure call (RPC). This is illustrated in Figure 2-1. The client makes a request (a call) to a server. Part of the request is an indication of the procedure to be called. The server performs the requested operation and then returns a response to the client, completing the procedure call. Note that RPC may be implemented either synchronously or asynchronously.

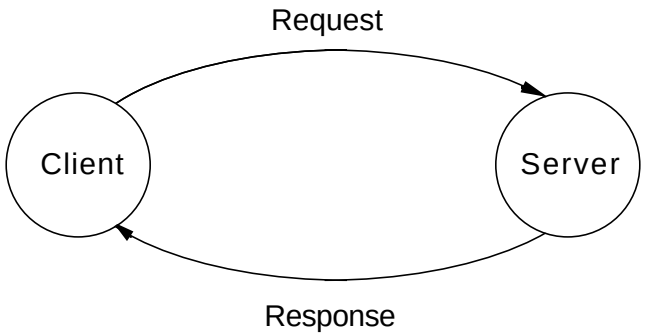


Figure 2-1: Model of a Remote Procedure Call

The data exchanged in an RPC consists of two parts, as indicated in Figure 2-2. The first is a protocol-specific component that conveys information about the protocol being used. For example, in the case of the Sun RPC [5] the protocol component contains

- An identifier that uniquely associates calls and responses.
- A *body* that can be either a call body or a reply body. In the case of a call, for example, the call body may contain the RPC version number, remote program number, remote program version number, remote procedure number, and authentication information.

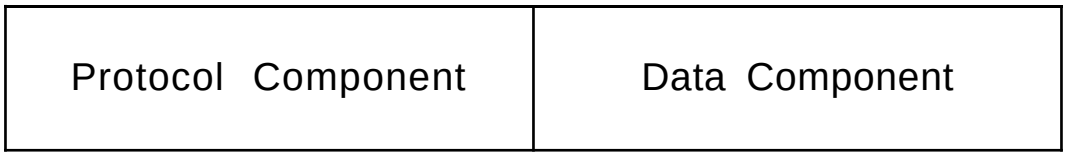


Figure 2-2: Encapsulation of Protocol Information

The second component of the information transferred is a data component that contains the information to be passed in the remote procedure call. In the case of the Sun RPC [5] and the Versatile Message Transaction Protocol (VMTP) [6], the data component is represented using the XDR standard [4].

There are several performance issues to consider in the use of a RFC model. Of concern in this report is the mechanism that is used to encode (and decode) the data component of the RPC. Application programs require flexible representations that are also efficient in their encodings. Frequently, however, greater flexibility of data representations implies a more complicated representation, which in turn requires more complicated *encode* (and *decode*) routines. These more complicated routines are larger and require more processing time to execute. As we shall see, the notions of flexibility and efficiency are not necessarily compatible.

2.2. Role of Standards

There is a current trend toward the increased use of standards in DoD real-time systems developments. It is hoped that the use of standards will increase system interoperability. When a standard contains more attributes than are believed necessary for the real-time domain, a *profile* of the standard can be developed. For example, an embedded system may not need all the functionality of a standard for a general purpose operating system.

Dkata representation standards such as ASN.1/BER and XDR, which are considered in this report, are appealing in that they lend themselves to automatic code generation. For example, one may create a data specification and then use a tool to automatically generate the routines to encode and decode the data. The existence of such a tool would eliminate the need for hand-coding the *encode* and *decode* operations on data structures.

2.3. An Example

A typical requirement of a real-time system is the need to perform track management.[1] We will consider the example of a track update message that can be exchanged by components of a distributed system. This example will be used throughout this report to illustrate the use of different data representation and encoding schemes.

2.3.1. Specification

A track update message can be transmitted when information about some track changes, or on a periodic basis. Typically, this message would contain the following information:

- Message length: the number of words in the message (we assume a 16-bit word).
- Message ID: a unique identifier for the class of the message.
- Track index: a unique identifier of the track being reported.

[1] A *track* represents the set of information about an object in the external environment, such as a radar report of an aircraft.

- Positional information: the location of the track in some particular coordinate system.

- Velocity information: the velocity components of the track. Note that these components may be reported in a different coordinate system than that used to specify the track position.

- Track category: an indication of the type of track, such as an air or surface track.

- Maneuver indicator: an indication that the track is maneuvering. This information is used in certain algorithms that estimate the state of the track.

- Track quality: a measure of the quality of the position and velocity information contained in the message.

- Sensor: an indication of the sensor that is reporting the information. Typically, many sensors may be available, and it is necessary to know which particular sensor is reporting the data.

- Clock: the time at which the track data was collected.

In reality, typical systems may report more information than that listed above. For our purposes, however, the above list captures the essence of the data and will serve to illustrate the issues associated with data representation. Although presented in the context of a message interchange, the model may easily be extended to an RPC model where the requested function is to update a remote track database.

2.3.2. Representation

The information that appears in the track update message is formatted in a predefined manner as specified in some interface requirements specification (IRS). The IRS specifies not only the format of the data, but the permitted range of values of the message components and other information about the communication protocol that is used between systems that exchange such data.

A typical representation of a track update message appears in Figure 2-3. The following are to be noted about the data representation:

- The *number of words* (NW) is a constant, having the value 12 (decimal), and is contained in one byte.

- The *message ID* (MID) is a constant, having the value 202 (octal), and is contained in one byte.

- The *track index* is represented as an unsigned integer having range 1 to 7777 (octal).

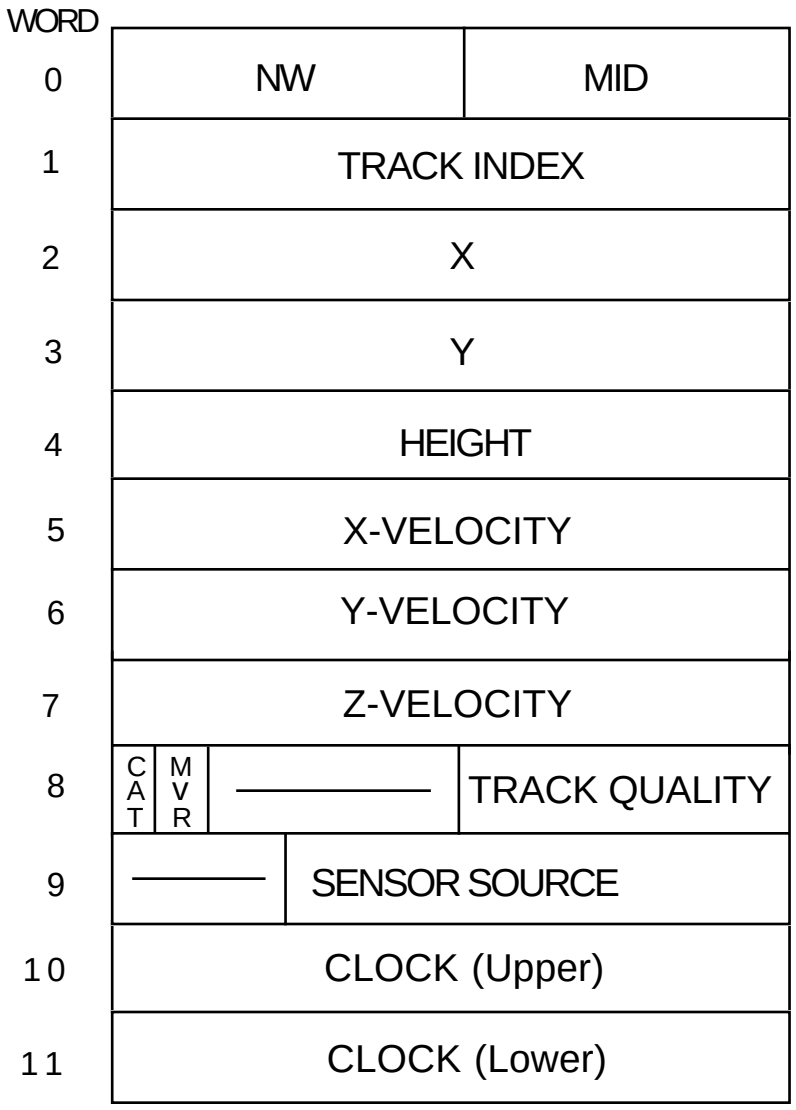


Figure 2-3: Structure of Track Update Message

- The *X-Position* (X) and *Y-Position* (Y) are fixed-point types with the assumed (binary) decimal point located between bits 6 and 7 (we assume bits are numbered from right to left starting with 0). The units for these quantities are in data miles.[2] and the range is [-255 $\frac{127}{128}$, 255 $\frac{127}{128}$].

- The *height* is a fixed-point type with the assumed (binary) decimal place between bits 7 and 8. The height is reported in units of data miles and is in the range [0, 255 $\frac{255}{256}$].

- The *velocity* data is a fixed-point type with the assumed (binary) decimal place

[2]One data mile is defined to be 6000 feet.

between bits 12 and 13. Velocity data is reported in units of data miles per second and is within the range [0, $\frac{8191}{8192}$].

- The *track category* (CAT) is a Boolean; if TRUE, it indicates a surface track; otherwise, it indicates an air track.

- The *maneuver* (MVR) is represented as a Boolean; if TRUE, it indicates that the track is maneuvering.

- The *track quality* occupies one byte and is represented by an integer in the range from 0 to 100 with larger values denoting higher track qualities.

- The *sensor source* is 10 bits wide, with each bit indicating a particular sensor. For example, if bit 6 is set, the track data is being reported by sensor 6, which we will label as SENSOR_G.

- The *clock* is an unsigned integer value of the time that the data was collected. The time is reported in milliseconds in the range [0, 37777777777] octal. Note that the high-order bit is used for data, not a sign.

There are two points about the representation of the track update message that are worth noting. First, the representation seeks to minimize the amount of storage; for example, the track category and maneuver request bits are packed in a byte and the sensor is represented as an array of type Boolean, which is also packed. Second, note the number of fixed-point types that are contained in the message.

The packing of data and the use of fixed-point types are typical of many existing systems. This is in part due to concern over buffer management (perhaps at the expense of additional processing time to decode and encode a message). The prevalence of fixed-point types is frequently driven by hardware considerations; that is, data is presented directly from a hardware device.

Finally, let us consider an instance of a track update message. Assume that track index 165 (decimal) is reported at a position given by (100.5, 67.75, 2.00) data miles with corresponding velocity components (0.50, 2.875, 0.0) in units of data miles per second. Assume that the track is an air track that is maneuvering, and that the track quality is 90 (decimal). Also assume that the track is reported by Sensor_B at a clock time of 496 (decimal) milliseconds. In this case, the actual bit stream for the track update message (in hex) would appear as

```
0C8200A532402170020010005800000405A0002000001F0
```

The total length of the track update message, illustrated above, is 24 bytes. Of these, only 12 bits are not used in the representation of the message.

3. ASN.1 Basic Encoding Rules

3.1. Overview

The Abstract Syntax Notation One (ASN.1) is an international standard for the specification and representation of data. It is well known and used in other international standards. Because it is an international standard, it is a natural candidate to consider for data specification and representation.

An ASN.1 specification is encapsulated within a *module*. A module can include an *import* and *export* list to indicate references to types and objects declared in other modules. A module can also contain *definitions* that can represent type declarations or value declarations. Finally, an ASN.1 module can contain *macro* declarations. A macro declaration can be used to change the syntax of the ASN.1 specification language when an ASN.1 specification is being compiled. Although unique in concept, use of the macro facility complicates the development of a compiler for ASN.1 specifications.[3]

It is important to note that ASN.1 can be used to produce very general specifications and has significant expressive capabilities. In this report we will be concerned with only a subset of the ASN.1 specification to illustrate its applicability to the real-time domain, which is predominated by certain data types. We will also be concerned with the ASN.1 specification mechanism only to the extent that it is used in the example of the track update message. However, the ASN.1 specification grammar appears in Appendix A in a BNF form.

3.2. Encoding Scheme

The mechanism by which data is represented in ASN.1 is defined as a set of basic encoding rules (BER), specified in ISO/IEC 8825 [3]. These rules are used by the presentation service provider in the ISO model. Each encoding consists of the following three items:

1. Identifier

2. Length

3. Value

[3]Rose discusses some issues regarding the macro facility [8]. This reference also contains a critical examination of ASN.1 in general.

The structure of an encoding is presented in Figure 3-1. The identifier is encoded in one octet and contains the following fields:

- *Class:* ASN.1 specifies 4 classes of encodings, namely, universal, application-wide, context-specific, and private, which are encoded in the first 2 bits of the identifier using the values 0, 1, 2, and 3, respectively. A universal tag is used to define an application-independent data type that is of general use. An application-wide tag is used to define an application-oriented data type that needs to be distinguished from other data types. A context-specific tag is used to distinguish among alternatives of a data type, such as members of a set. Finally, a private tag is used to define data types that are used in a limited domain.

- *Form:* An encoding may be *primitive* or *constructed*, depending upon the data type. For example, an integer is encoded as a primitive, while a sequence is encoded in a constructed form, meaning that it is an encapsulation of encodings. The encoding method, denoted by *PC* in Figure 3-1, is contained in one bit, with primitive and constructed being denoted by 0 and 1 respectively.

- *Tag:* The tag number is an indication of the data type.

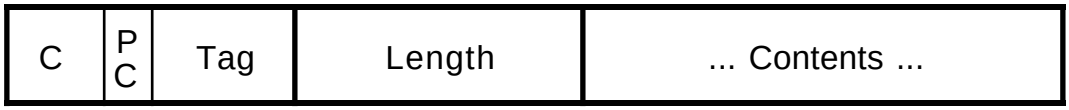


Figure 3-1: Structure of an ASN.1 Encoding

The *length* is contained in one or more octets and denotes the number of octets of data present. In this report we will be concerned with a definite form of length which is one that is encoded in a single octet. Other forms are possible, including a specification of an indefinite form where the length of the data is determined from special characters contained in the data.

The *value* of an encoding, also referred to as the *contents*, contains the actual data. For certain encodings, there may not be any values present.

3.3. Components

In discussing the encodings used in ASN.1, we will partition the ASN.1 data types into components consisting of (i) primitive, (ii) constructed, and (iii) dual encodings. These are discussed in the following subsections.

3.3.1. Primitive Components

A primitive component is one that is encoded as a single entity. The integer, Boolean, enumerated, real, and null are included within this class and are discussed below.

3.3.1.1. Integer

An integer is encoded using one or more contents bytes, represented as a two's complement binary number. The encoding is such that the contents field is encoded using the smallest number of bytes. The universal class tag for an integer is 2.

3.3.1.2. Boolean

The contents field of a Boolean is encoded using one byte. The value FALSE is encoded as zero, and TRUE is encoded as any non-zero value. The universal class tag for a Boolean is 1.

3.3.1.3. Enumerated

The encoding of an enumerated type is the same as that of the integer value that is associated with the type. The universal class tag for an enumerated type is 10.

3.3.1.4. Real

The representation of a real value may include one of the following: (i) the value zero, (ii) a binary representation of a real value, (iii) a character-based decimal representation of a real number, and (iv) certain special real values. The universal class tag for a real type is 9.

The value 0.0 is encoded by setting the length field to 0. That is, there are no contents octets.

If bit 8 of the first contents byte has the value 1, the encoding is that of a binary representation of a real number. A real value is represented in the following form:

$$S\ N\ 2^F\ B^E$$

where S is the sign, N is a number related to the mantissa, F is a scale factor in the range [0, 4), B is the base, and E is the exponent. The elements of the encoding are based on the following:

- The sign is contained in bit 7 of the first contents octet. The values positive and negative are represented by 0 and 1, respectively.

- The base is contained in bits 6 and 5 of the first contents octet and can have the values 0, 1, or 2, denoting, respectively, base 2, 8, or 16.

- The scale factor F is contained in bits 4 and 3 of the first contents octet and directly represent a value i the range [0, 4).

- The format of the exponent is indicated in bits 2 and 1 of the first contents octet and has the following interpretation:

- If bits 2 and 1 have the value 00, then the second contents octet contains the value of the exponent.

- If bits 2 and 1 have the value 01, then the second and third contents octet contains the value of the exponent.

- If bits 2 and 1 have the value 10, then the second, third and fourth contents octets contains the value of the exponent.

- If bits 2 and 1 have the value 11, then the second octet specifies the number of octets that are used for the exponent. The actual value of the exponent appears in the following number of octets.

- In all cases, the exponent is stored as a two's complement binary number.

- The value of N is encoded as a binary integer in the remaining contents octets.

If bits 8 and 7 of the first contents byte have the value "00," the encoding is that of a character-based decimal encoding. This is similar to an ASCII representation with the interpretation of the remainder of the encoding being specified by certain international standards.

If bits 8 and 7 of the first contents byte contains the value "01," a special real value is present. The two values defined by the standard are plus and minus infinity, encoded as the bitstrings 01000000 and 01000001 respectively. When a special real value is present, there will be only one contents byte. Other possible encodings are reserved for future use.

It may help to clarify matters to consider an example of an encoding of a real type. Consider, for example, the interpretation of the hex string "090680E60ADF0A8B." The interpretation of this is illustrated in Figure 3-2.

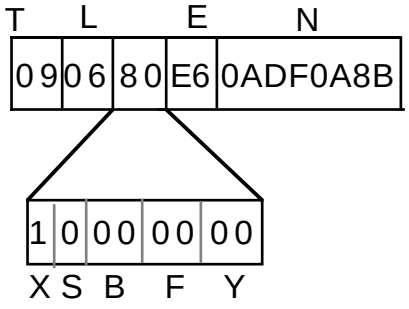


Figure 3-2: Example of a Real Type in ASN.1

The following are to be noted about the interpretation:

- The first byte contains the tag T, which in this case is 9, representing a real type.

- The second byte contains the length L.
- The third byte contains the first byte of the contents and is interpreted as follows:
    - The first bit (bit 8) indicates that the real value is encoded in binary.
    - Bit 7 indicates that the sign of the number is positive.
    - Bits 6 and 5 represent the base B. The value of zero indicates base 2.
    - Bits 4 and 3 contain the value of the scale factor F, which is 0.
    - Bits 2 and 1 indicate that the exponent occupies one byte of storage.
- The value contained in byte 4 denotes the exponent E; in this case, the value is -26.
- The last four bytes of the encoding contain the value of N.

When the value of N is converted to decimal and multiplied by the factor of 2^{-26}, the result is the value of e, the base of natural logarithms.

It is important to note that there are multiple valid BER encodings for a real number. For example, the value of e as discussed above could also be represented by the hex string "0906817FE9ADF84D." In this case, the header indicates that the exponent is represented in two octets, with the remaining three octets representing the mantissa.

The encoding specified for a real type by ASN.1 does not relate to any specific hardware, nor is the encoding based on any other standard such as the IEEE Standard for floating point numbers [1]. However, it is stated in ISO/IEC 8825 [3] that the ASN.1 representation was chosen to be easily converted to and from other formats. See Appendix C.

3.3.1.5. Null

An object of the null type is encoded such that the length field is 0 and there is no contents field present. The universal class tag for a null type is 5.

3.3.2. Constructed Components

A constructed encoding is one in which the encoding comprises one or more data values. Such encodings are used for structures and records, for example.

3.3.2.1. Sequence

ASN.1 provides for specifying a sequence or sequence-of data types. The encoding encapsulates the encodings of the component data types. In the case of a sequence, the encoding must appear in the same order as defined in the specification, but the sequence can consist of many different data types. For a sequence-of, the order must also be preserved,

but all components of a sequence-of must be of the same data type. The universal class tag for a sequence or sequence-of is 16.

3.3.2.2. Set

Similar to a sequence, ASN.1 provides for a set and set-of that represent an unordered collection of objects. The encoding for the elements of the set can be chosen at will by the encoder. Also, as with the sequence and sequence-of, a set can consist of components of many different types, while a set-of consists only of components of the same type. The universal class tag for a set or set-of will be 17.

3.3.3. Dual Encoded Types

Certain of the ASN.1 types can be encoded using either a primitive or constructed form. These are discussed below.

3.3.3.1. Bit String

A bit string is encoded as a sequence of zero, one, or more octets of data. The universal class tag for a bit string type is 3.

3.3.3.2. Octet String

An octet string represents zero, one, or mode octets of data. The universal class tag for an octet string type is 4.

3.3.4. Additional Points

The preceding has covered the basic components of ASN.1 types and associated basic encoding rules. It is to be noted that ASN.1 includes additional items that may be useful to a designer. For example, one can specify objects of generalized and universal time (although they are encoded as character strings). ASN.1 also includes subtypes and a macro facility that is quite powerful; for example, one can extend the syntax of ASN.1. For further discussion of the above items the reader is referred to ISO/IEC 8824 [2], ISO/IEC 8825 [3], and Steedman's *ASN.1: The Tutorial and Reference* [9].

3.4. Example

The development of the ASN.1 specification is fairly straightforward. Thus, in reference to Figure 2-3, we note the following:

- The track update message is encapsulated in a module called *track messages*. The use of a module specification in ASN.1 allows common data types to be grouped together. In addition, modules can export, import, and reference specifications defined in other modules; this is an attractive feature of ASN.1.

- The message itself is represented as a sequence. This implies that the encoding of the message will be the ordered encodings of the message components.

- The number of words, message type, track index, track quality, and the value of the clock are encoded as integers.

- The track category and maneuver indicator are encoded as Boolean types.

- The sensor type is specified as an enumerated type with the enumerated values appearing in the specification.

- The position and velocity data is represented as a string of octets. This choice is made because ASN.1 does not provide for the specification of a fixed-point type. This implies that a tool to decode the message would have to perform special processing for the position and velocity data.

The ASN.1 specification is presented in Figure 3-3.

```
SEQUENCE {
   NUMBER_WORDS                INTEGER (1 .. 32767),
   MESSAGE_TYPE                INTEGER (1 .. 32767),
   TRACK_INDEX                 INTEGER (1 .. 511),
   POSITION_AND_VELOCITY_DATA  OCTET STRING (SIZE (12)),
   TRACK_CATEGORY              BOOLEAN,
   MANUEVER_INDICATOR          BOOLEAN,
   TRACK_QUALITY               INTEGER (0 .. 100),
   SENSOR                      ENUMERATED
                               {SENSOR_A = 1,   SENSOR_B = 2,
                                SENSOR_C = 4,   SENSOR_D = 8,
                                SENSOR_E = 16,  SENSOR_F = 32,
                                SENSOR_G = 64,  SENSOR_H = 128,
                                SENSOR_I = 256, SENSOR_J = 512 },
   CLOCK                       INTEGER
          }
```

Figure 3-3: ASN.1 Specification of Track Update Message

In terms of the values specified in Section 2.3.2, the ASN.1 encoding of the track update message would appear as follows:

```
301F020C028202A5041032402170020100058000000010001FF
025A0A020201F0
```

The total length of the message in the ASN.1 BER encoding is 33 bytes. The representation discussed in Section 2.3 required a total of 24 bytes.

4. External Data Representation

4.1. Overview

The external data representation (XDR) is a standard developed by Sun Microsystems for the description of data [4]. This standard is used in the definition of Sun RPC [5] as well as the Versatile Message Transaction Protocol (VMTP), defined by Cheriton [6].

Some of the characteristics of XDR include the following:

- A data representation carries no information about the data type. It is argued that receivers know what data types are expected and that including such information is not of particular use.

- The data encoding is assumed to be a big endian format. This format is used on IBM and Motorola machines, but not, for example, by the VAX family, which uses little endian.

- XDR assumes that all data are encoded as a multiple of four bytes. This means, for example, that a Boolean object will be represented in terms of 32 bits.

Our interest is more in the data representation aspect of XDR. Appendix B contains a BNF specification of the language associated with XDR. In the following subsections, we present the components of the XDR representation.

4.2. Components

4.2.1. Integer

An integer is represented as a 32-bit unit in the range [-2147483648, 2147483647]. The integer is represented in two's complement notation. XDR also supports an unsigned integer with values in the range [0, 4294967295]. In the latter case, the high-order bit is interpreted as data, not a sign bit.

XDR also supports a hyper integer and an unsigned hyper integer type. These are extensions of the integer and unsigned integer types, but they occupy eight bytes as opposed to four.

4.2.2. Boolean

A Boolean is represented with the value TRUE as 1, and FALSE is represented as 0. In each case, the object assumes 4 bytes.

4.2.3. Enumerated

XDR supports enumerated types; an example of the syntax is as follows:

```
enum { RED = 2, BLUE = 12, GREEN = 15 } colors;
```

The representation of an enumerated type is identical to that of an integer type.

4.2.4. Floating Point

The representation of a floating point type is that specified by IEEE 745-1985 [1] for normalized floating point numbers. The representation includes the sign, exponent, and fraction, described as follows:

- The sign of the number is represented as one bit, with the values 0 and 1 denoting positive and negative numbers, respectively.
- The exponent is represented in base 2 with a bias of 127. A total of 8 bits are used to store the exponent.
- The fraction is represented in base 2 and is stored in 23 bits.

The result of the above is that a (single precision) floating point number is described as:

```
(-)**S * 2**(Exponent - 127) * 1.Fraction
```

The IEEE standard also describes special floating point types that may be included as part of an XDR specification. Examples of this include signed zero and signed infinity (to represent overflow). XDR also provides for a specification of a double precision floating point type that is an extension of the single precision floating point type, described above. The double precision floating point is also represented according to the IEEE standard [1].[4]

4.2.5. Opaque Data

Opaque data is defined in XDR as uninterpreted data that is passed between machines. An XDR specification may describe either fixed-length or variable-length opaque data. In each case, the data is represented as a multiple of four bytes with additional zero-byte padding as needed.

An opaque data specification can be used to describe data that will be processed by the data rather than the presentation services. For example, a file can be encoded as opaque data indicating that no operation is performed on the data by an automatically generated tool. In this sense, opaque data can be used to hide the internals of the data representation.

[4]The IEEE floating point standard [1], also specifies *extended* floating point types. These are not included in the XDR specification.

4.2.6. String

A string is represented in terms of unsigned integers and assumes the ASCII character set. The first 4 bytes of the representation define the length of the string. A length declaration may be omitted, in which case the maximum length of 2**32 -1 bytes is assumed. The data follows the length specification and must be a multiple of 4 bytes.

4.2.7. Arrays

A group of homogeneous elements can be encoded in XDR as an array. The specification provides for both fixed-length and variable-length encodings. In the latter case, the length is included as the first four bytes of the representation, encoded as an unsigned integer.

4.2.8. Structure

A structure declaration in XDR is based on that defined in the C programming language and has the syntax:

```
struct {
  component-1;
  component-2;
  ...
  component-n;
        } structure-name;
```

Each component of the structure is encoded in the order in which it is declared. The size of each component must be a multiple of four bytes, although each component may have different sizes.

4.2.9. Discriminated Union

XDR permits a specification of a discriminated type, which is called a *discriminated union*. An example of a discriminated union appears below:

```
enum { XTP = 1, OSI-CONNECTIONLESS = 2 }
  protocol;

union switch (PROTOCOL)
  {
    case XTP:
      /* XTP protocol data specification */
    case OSI-CONNECTIONLESS:
      /* Connectionless protocol data specification */
  } ;
```

The enumerated type specifies two different protocols, namely XTP and OSI Connectionless, along with the particular values of each instance. Following this are the specifications for each particular protocol instance.

The representation of a discriminated union consists of two parts. First, the discriminant is

encoded in four bytes. The second part of the encoding is the data for the specified discriminant value.

4.2.10. Void

A void is a zero-byte quantity that can be used in XDR specifications. A void specification is useful for operations that do not require any data as input or output, for example. It can also be used to specify a null branch of a discriminated union.

4.2.11. Additional Points

The XDR specification also allows for the specification of constants, optional data, and identifiers used for declaring other data. These are discussed in [4].

4.3. Example

We now consider the XDR specification for the track update message, introduced in Section 2.3. The message will be represented as a structure whose components are the elements in the track update message. The XDR specification is presented in Figure 4-1.

```
struct {
   int    NUMBER_WORDS;
   int    MESSAGE_TYPE;
   int    TRACK_INDEX;
   opaque POSITION_AND_VELOCITY_DATA [12];
   bool   TRACK_CATEGORY;
   bool   MANUEVER_INDICATOR;
   int    TRACK_QUALITY;
   enum   {SENSOR_A = 1, SENSOR_B = 2, SENSOR_C = 4,
           SENSOR_D = 8, SENSOR_E = 16, SENSOR_F = 32,
           SENSOR_G = 64, SENSOR_H = 128, SENSOR_I = 256,
           SENSOR_J = 512 } SENSOR;
   unsigned int CLOCK;
        }  TRACK_UPDATE_MESSAGE;
```

Figure 4-1: XDR Specification of Track Update Message

The development of the XDR specification is fairly straightforward, for example:

- The number of words in the message, message type, track index, and track quality are represented as integers.
- The track category and maneuver indication are represented as Boolean.
- The sensor type is represented as an enumerated type with each sensor identified.
- The value of the clock is represented as an unsigned integer.

There is, however, one issue in the use of XDR to specify the track update message. Several of the quantities in the message are defined as fixed-point types, although XDR does not provide such a specification. We could have represented these as some other type, such as an integer, but instead, chose to represent the fixed-point types in terms of *opaque* data. Thus, as shown in Figure 4-1, the position and velocity data is simply declared as an array of length 12 of opaque data. Recall, the use of opaque data hides the internal representation of the data. In the present case, it would not be possible, for example, for an automated tool to encode or decode opaque data.[5]

In terms of the parameters given in Section 2.3.2, the XDR-encoded track update message would appear as follows:

```
0000000C00000082000000A5324021700200100058000000
00000000000000001000000AA000000020000001F0
```

The total length of the message, in the XDR form, is seen to be 44 bytes. Recall that the storage required for the track update message, discussed in Section 2.3, only required a total of 24 bytes.

[5]Strictly speaking, one could modify a tool to handle the fixed-point types. However, this is tantamount to extending the definition of the XDR specification.

5. Comparison of Approaches

5.1. Qualitative

A qualitative comparison of ASN.1 and XDR can be made based on the preceding as well as the standards definitions contained in ISO/IEC 8824 [2], ISO/IEC 8825 [3], and the XDR Standard [4]. The following points are relevant:

- The specification of ASN.1 is more expressive than that of XDR. This is achieved, in part, through the use of modules, which allows for the application of modern software engineering practices. In addition, the presence of subtyping multiple class tags also provides general capabilities to an application. Also, the macro capability of ASN.1 permits one to generate new specifications; in fact, one can redefine the syntax of ASN.1 through this capability.

- The data specification and representation capabilities of ASN.1 are more general than that of the XDR standard. The encoding of data in minimal length formats, such as integer and Boolean types, conserves buffer space and may have implications for network bandwidth allocation. It was noted in the text that the real type declared by ASN.1 is not suited to any particular hardware representation, and this further illustrates the generality of the ASN.1 approach.

Of course, an application could pay a significant price for the generality contained in the ASN.1 specification and encoding standards. XDR maps all data onto 32-bit aligned boundaries, and employs current standards for floating point representation, such as the IEE floating point standard [1]. Clearly, XDR is oriented toward systems in widespread use today and seeks to achieve an efficient data representation.

Many factors are involved in a decision concerning the use of a data specification and representation standard. One concern in the real-time domain, where end-to-end deadlines are important, is that of performance. In the following section, we present some discussion of quantitative measures to compare ASN.1 and XDR.

5.2. Quantitative

It is desirable to have quantitative metrics to compare ASN.1 and XDR. Such metrics can help application designers in the selection of a data specification and representation standard. In the following two sections we present two simple metrics regarding buffer sizes and processing times.

5.2.1. Buffer Sizes

Given a set of data to be encoded according to some specification, one concern is the amount of storage required for the resulting data. The storage required is important because it contributes to the bandwidth required to transmit the data. In addition, a greater amount of storage requires more data movement among components (for example, backplanes and protocol chip sets).

One applicable metric is the ratio of buffer sizes for two representations. Let $S(encoding_i)$ denote the required buffer size to encode data according to standard i. We then consider the ratio:

$$S(i,j) = \frac{S(encoding_i)}{S(encoding_j)}$$

In relation to the example considered in the text for the track update message, let S(raw) denote the size of the data as specified in Figure 2-3. Then, we have:

S(ASN1, raw) = 33/24 = 1.4,

S(XDR, raw) = 44/24 = 1.8,

and

S(XDR, ASN1) = 44/33 = 1.3.

In the case of the track update message, the use of the XDR representation nearly doubles the size of the buffer required to store the message.

5.2.2. Processing Times

A second appropriate metric concerns the amount of time to encode and decode data. Let T(i; k, λ) denote the amount of time to perform operation λ (*encode* or *decode*) on an object of type k according to data representation standard i. We then define

$$T(i, j; k, \lambda) = \frac{T(i; k, \lambda)}{T(j; k, \lambda)}$$

to be the ratio of times to perform operation λ on an object of type k for standards i and j.

It is not our intent here to perform a detailed investigation of T(i, j; k, λ) because such results are influenced by:

- Design method (automated or hand-coded, for example).
- Implementation language.
- Target machine (underlying instruction set architecture).

To illustrate an application of the encode times, consider the case of an integer such that its

value can be represented in 32 bits. An estimate of T, expressed in terms of number of machine instructions, would give

$$T(XDR; integer; \lambda) = 2.$$

In other words, a 4-byte integer could be encoded or decoded in two instructions. This estimate assumes one instruction to fix a pointer at the address and the second instruction to move the 32 data bits. Recall that the XDR specification requires boundary alignment of 4 bytes on each data type.

A corresponding estimate for the case of ASN.1 would be roughly

$$T(ASN.1; integer; \lambda) = 7,$$

where the majority of the instructions are used to extract (and verify) tag and length information. The preceding indicates that decoding a 32-bit integer requires about 3 times as many instructions in ASN.1 than in XDR. In the case of a floating point value exchanged among machines that conform to the XDR standard (that is, that employ the IEEE floating point standard representation [1]), in XDR the data could be encoded in 2 instructions. An estimate for ASN.1/BER *encode* and *decode* routines appears in Appendix C. There, it is illustrated that the *encode* and *decode* operations, using ASN.1 BER, are considerably more expensive than when done in XDR.

The inclusion of tag information in the ASN.1 encoding must be accounted for in any performance metric to encode or decode data of a particular type. To some applications, this could be viewed as unnecessary overhead.

5.2.3. Space and Time Tradeoffs

It is possible to combine the buffer size and processing time metrics, resulting in a *composite* metric. Such a metric would convey, in a simple manner, the two metrics introduced above. To this end, define

$$S' = S(i, j),$$

and

$$T' = T(i, j; k, \lambda).$$

The composite metric is obtained by displaying data in the S'-T' plane. An example of this is presented in Figure 5-1 where i and j correspond to ASN.1 and XDR, respectively. The point labeled x in this figure represents the values of S' and T' for a 32-bit integer. In this case, ASN.1 requires about 0.75 times the buffer space of XDR. However, as noted above, ASN.1 requires about 3.5 times the amount of assembler instructions to decode the data. The shaded area in Figure 5-1 is that domain in the S'-T' plane for which the representation specified by standard i is more efficient, in space and time, than that specified by standard j.

We are unaware of any detailed metrics for comparing ASN.1 and XDR, nor are we aware of any type of benchmark suite that would seek to compare these standards. We believe that the existence of such work would help to establish quantitative results that application systems need in order to complete a detailed performance estimate of the impact of using either of the standards discussed here. A similar approach to that outlined above is easily seen to apply to other data representation standards.

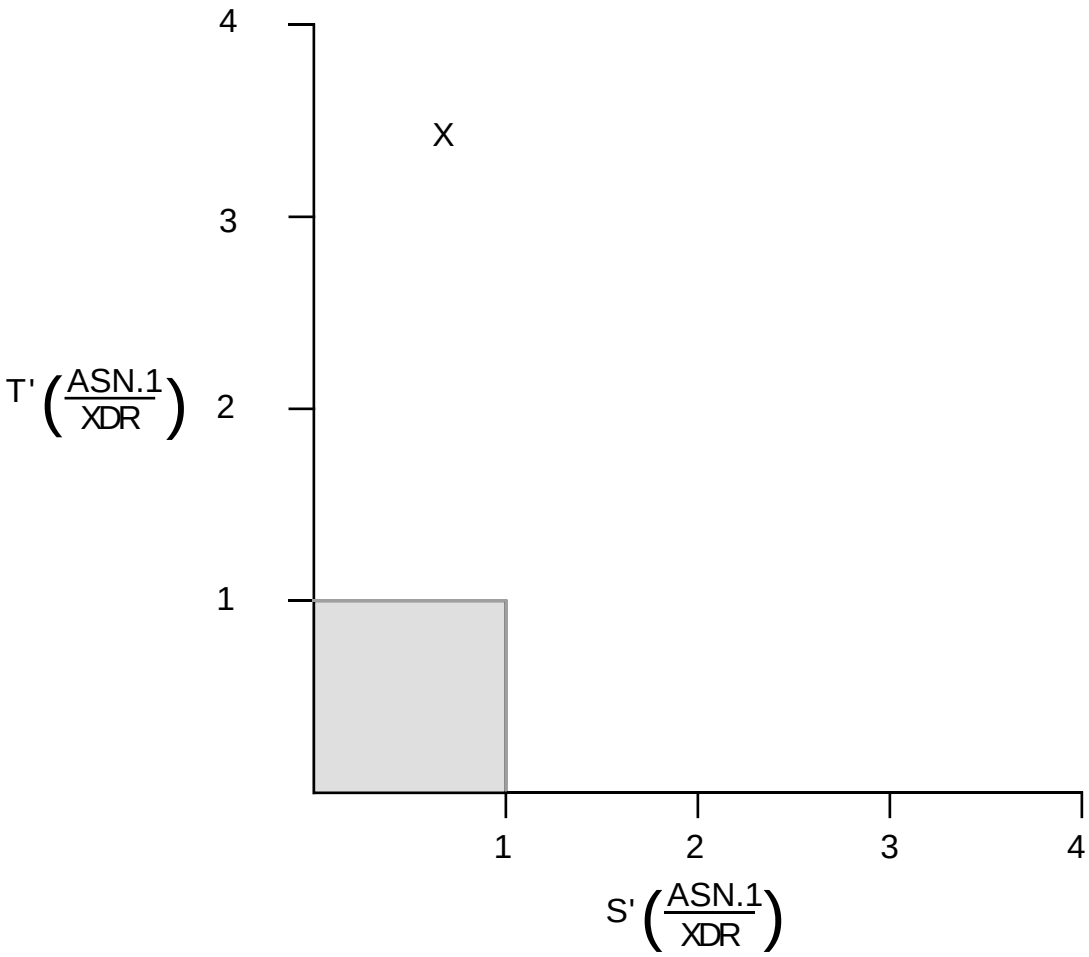


Figure 5-1: A Sample Composite Metric

5.3. Issues

5.3.1. Typing Considerations

A major difference between ASN.1 and XDR data representations is that the former carries type information while the latter does not. The utility of type information is an issue. For example, many systems are expecting data of a predefined type and to encode the type information requires additional storage as well as processing time.

Neither XDR nor ASN.1 include any provision for fixed-point types. This was noted in the

examples considered, where it was necessary to encode fixed-point data in some other form. In the case of the XDR specification of the track update message, for example, the position and velocity data were encoded as opaque data. The failure of a specification to support a particular type has implications for the ability to automate the processing of application data.

5.3.2. Byte Ordering and Alignment

The XDR specification requires a big endian byte ordering. This form is used in IBM and Motorola class machines. However, the VAX family uses a little endian byte ordering.

The criteria for byte alignment may be important in system considerations. The XDR specification is based on a four-byte wide representation, while the ASN.1 specification is of variable length.

5.3.3. Use of Other Representations

This report has considered the use of the ASN.1 Specification and basic encoding rules and the external data representation. Both of these are well known and widely used in distributed systems. There are other systems that address similar issues as these which deserve consideration. A notable case in point is the interface description language (IDL) developed by Nestor et al [7]. Although originally intended for use in compiler technology (IDL is the *de facto* intermediate representation for Diana, which is used in Ada compiler technology), IDL applies to the problems considered in this report. In fact, IDL provides capabilities not found in either of the systems considered.

5.3.4. Automated Tools

If one were to use a data specification and representation standard, such as ASN.1, XDR, or IDL, it would clearly be advantageous to have automated tooling to support *encode* and *decode* operations. Such a tool would allow a designer to specify the data, and the tool would generate the *encode* and/or *decode* routine. There are such tools available for each of the above three specifications.

5.3.5. Revisions to ASN.1

During the preparation of this report, we became aware of changes that are being proposed to the basic encoding rules to ASN.1, ISO/IEC 8825 [3]. These include the following:

- A set of *packed encoding rules* (PER) that results in a more compressed encoding than ASN.1 basic encoding rules. For example an integer in the range (100 ... 103) can be encoded in the PER using 2 bits. This is achieved by adding an offset of 100 to the encoded value, thereby requiring less storage than that for a basic encoding.

- A set of *canonical and distinguishing encoding rules* that are based on the basic encoding rules. These encodings would require, for example, that a Boolean be encoded as 1 if the value were TRUE and 0 if FALSE (recall the BER state that a Boolean having the value TRUE can be encoded as *any* non-zero value).

The advantage of a canonical and distinguished encoding is that it standardizes the encodings of certain elements.

- There is also some possibility that a set of *light weight encoding rules* (LWER) will be standardized. These are intended to be faster to encode and decode than the basic encoding, although the encodings may require more storage.

It is our understanding that the first two encodings listed above are draft international standards, while the LWER is under consideration.

6. System Design Considerations

Although this report has focused on the data representation issues, the question of data representation exists in a larger context. In particular, the use of data representation services (such as *encode* and *decode*) represent a *part* of the execution time expended in support of end-to-end deadline processing. In the following, we examine some system considerations related to this larger context.

6.1. Analysis

We will now present a simple analysis of the contribution to the end-to-end completion time in the case of heterogeneous systems. For such systems, we assume that a data representation scheme, such as ASN.1 BER, is used. In the case of a peer-to-peer communication, there are two contributions to the end-to-end completion time that must be considered, namely

- The time for the sender to encode the data before message transmission.
- The time for the receiver to decode the received data.

Consider the case in which a message is transmitted at a periodic rate R times per second. Assume that the message can contain n_i values of data of type i. For example, one could delineate the set whose elements represent integer types, Boolean types, floating point types, etc. To account for the encoding and decoding times we define

- $\lambda_{i,p}^{enc}$ to be the time required to encode a data type i on processor p.
- $\lambda_{i,p}^{dec}$ to be the time required to decode a data type i on processor p.

The inclusion of the subscript *p* in the above is to distinguish that the times to perform *encode* and *decode* operations are processor-dependent.[6]

Based on the above, the contribution to the end-to-end completion time T is given by the following expression:

$$T = R \sum_i n_i \, (\lambda_{i,p}^{enc} + \lambda_{i,p}^{dec})$$

It is possible to extend the development of the formalism in a natural manner. For example, it can be developed for a specific message type or be extended to include worst-case times for processing messages that are transmitted in a multicast manner. We will not explore further development of the formalism here. Rather, our intent is to be able to illustrate the impact of *encode* and *decode* operations in a heterogeneous context.

[6]There are also compiler and language dependencies, but they need not be included for purposes of the current discussion.

6.2. Example

We now consider an example of the formalism developed above. Consider the case of the track update message, described in Section 2.3. We will assume as part of the transition to an open system architecture that the fixed-point data in the message are replaced by floating point types, of which there are six (three for position data and three for velocity data). Based on the results in Appendix C, we use the following values for floating point types:

$$\lambda_{fp,p}^{enc} = 74.0\ \mu sec$$

$$\lambda_{fp,p}^{dec} = 365.0\ \mu sec$$

Approximately 83 percent of the total encode/decode time is spent in the *decode* operation. Applying the above formula for the time required to encode and decode a floating point type T_{fp}, we have

$$T_{fp} = 6(74.0+365.0)R\ \mu sec$$

$$= 2.634R\ \ msec$$

The above result indicates that approximately 2.5 msec are required to encode and decode the six floating point type values contained in the track update message. Of this, approximately 0.5 msec is required for the *encode* operation, and 2.0 msec for the *decode* operation. The results indicate an upper bound of about 380 messages per second not counting other network effects (physical transmission rates, protocol layer processing) and also not counting the time to encode and decode the other data types in the message. Furthermore, when one examines the number of track update messages sent per second for each track, it is apparent that the total processing time for *encode* and *decode* operations is considerable.[7]

6.3. Alternatives

It is apparent that the results presented above and in Appendix C may cause concern to some system designers. Clearly, the amount of time required to perform data transformations because of concerns about heterogeneity can be substantial. The case in which the communicating peers are components of a homogeneous system does not present the problems of the heterogeneous case. We now briefly consider some of the issues from a system development standpoint.

[7]For a network that can support a bandwidth of one megabit per second, this implies that the time spent in encoding and decoding the floating point values in several hundred track messages is equal to the time it takes to move one megabit of data. Note the throughput implications for system bandwidth.

6.3.1. Role of the Interface Requirements Specification

As background, let us note that systems typically contain documentation describing the way that data is exchanged among system components. Such documentation is often contained in an interface requirements specification (IRS), or an interface design specification (IDS). To examine issues of heterogeneity, it is interesting to examine the role of an IRS in a particular context. Toward this end, we define the following:

- An IRS is *unconstrained* if the specification of data elements is independent of a particular hardware architecture.

- An IRS is *constrained* if the specification of data elements is dependent on a particular hardware architecture.

For example, in an *unconstrained* form of the IRS, an element of a message may be specified as a floating point type. In contrast, in the case of a *constrained* IRS, the same data type would be specified in terms of a particular representation, such as the IEEE format. Since it hides implementation knowledge from the communicating systems, an unconstrained specification may be more suited to use in heterogeneous systems.[8]

6.3.2. Design Approaches

There are two basic approaches for dealing with the problems of heterogeneous communication and data representation. In the first case, one may want to apply a standard, such as ASN.1 BER. One consideration is to optimize the procedures that perform the *encode* and *decode* operations. As noted in Appendix C, Ada was used for purposes of readability. It is clear that one may be able to optimize the *encode* and *decode* operations by using assembler language, for example. However, it is not clear how much improvement would result from the use of assembler language.

The second approach is to develop *application-specific protocols*. For example, a message can be defined in the following manner:

```
Message ::=
        architecture_type;
        seq_of {message_elements};
end Message;
```

where, for example,

```
architecture_type ::= {sparc, motorola68K, intel};
```

denotes the hardware architecture that is used to represent the message components and the *{message_elements}* denotes the set of message elements, such as float, integer, and enumerated. Note that the above specification is presented in an *unconstrained* form, since the *representation* of floating point type is not defined.

---

[8]The fact that an IRS may also contain permitted ranges of values does not affect the notion of constrained and unconstrained specifications.

To illustrate the utility of application-specific protocols, consider the case in which application A_i sends messages to application A_j. We assume that the hardware architecture of A_i is different from that of A_j. From the perspective of the sending application there are two choices, namely

- Application A_i encodes the data in its native (hardware) representation. This eliminates the encoding time required if a standard, such as ASN.1 BER, were used. When application A_j receives the message, it performs the *decode* operations from architecture A_i to its hardware architecture.

- Application A_i encodes the data in the hardware architecture of application A_j. In this case, when the message is received by A_j there is essentially no time required to decode, since that conversion was made before the transmission of the message.[9]

The utility of application-specific protocols is based on knowledge of underlying hardware architectures.[10] This is in contrast with the use of ASN.1 BER and similar representation schemes, which do not assume any knowledge of hardware architectures.

The preceding has only touched on several issues that must be addressed when one is concerned with data transfer in heterogeneous systems.[11] For those systems in which performance considerations are critical, it is important to recognize the tradeoffs in the approaches to data representation.

[9]Note however, if a message is transmitted in a *multicast* manner where there are two receiving applications, each of which has a different hardware architecture, further problems must be addressed.

[10]This assumes that message elements are represented in *native* format for a particular hardware architecture and therefore, fixed-point types would be precluded, for example.

[11]We mention that a *hybrid* approach is also possible that essentially encapsulates a constrained specification in an *opaque* type.

7. Summary

The ability of a real-time system to satisfy *end-to-end* timing deadlines can be influenced by many factors. One such factor is the encoding and decoding of data, by an application task, which is then transferred to a component of a distributed system. To assure that deadlines can be met, real-time systems require timely processing of application data.

It is clear that the use of standards in the development of real-time distributed systems is an important issue. This report has examined two such standards, namely the Abstract Syntax Notation One (ASN.1) and the external data representation (XDR), in regard to data representation. Each of these standards has unique characteristics that may make it applicable to the real-time domain. Several issues are pointed out to help a designer determine the applicability of these and other standards to the issues related to data specification and representation for the real-time distributed domain. The BNF for ASN.1 and XDR, as well as an Ada implementation of ASN.1 BER *encode* and *decode* routines for floating point numbers, are included in Appendix C.

References

1. Institute of Electrical and Electronics Engineers. *IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Standard 745-1985.* New York, New York: Institute of Electrical and Electronics Engineers, 1985.

2. International Organization for Standardization. *Information Processing Systems - Open Systems Interconnection - Specification of Abstract Syntax Notation One (ASN.1), ISO/IEC 8824.* Switzerland: ISO/IEC Copyright Office, 1990.

3. International Organization for Standardization. *Information Processing Systems - Open Systems Interconnection - Specification of Basic Encoding Rules for Abstract Syntax Notation One (ASN.1), ISO/IEC 8825.* Switzerland: ISO/IEC Copyright Office, 1990.

4. SUN Microsystems. *XDR: External Data Representation Standard.* SUN Microsystems, 1987. Published as Request for Comments (RFC) 1014.

5. SUN Microsystems. *RPC: Remote Procedure Call Protocol Specification, Version 2.* SUN Microsystems, 1988. Published as Request for Comments (RFC) 1050.

6. Cheriton, D. *VMTP: Versatile Message Transaction Protocol, Protocol Specification.* Stanford, California: Stanford University, 1988. Published as Request for Comments (RFC) 1045.

7. Nestor, John R.; Newcomer, Joseph M.; Giannini, Paola; and Stone, Donald L. *IDL: The Language and Its Implementation.* Englewood Cliffs, New Jersey: Prentice Hall, 1990.

8. Rose, Marshall T. *The Open Book: A Practical Perspective.* Englewood Cliffs, New Jersey: Prentice Hall, 1990.

9. Steedman, D. *ASN.1: The Tutorial and Reference.* London: Technology Appraisals Ltd., 1990.

Appendix A: ASN.1 BNF

```
<assignment> ::= <type_assignment> <CR> | <value_assignment> <CR>

<external_type_reference> ::= <module_reference> '.'
        <type_reference>

<module_reference> ::= <identifier>

<type_reference> ::= <identifier>

<external_value_reference> ::= <module_reference> '.'
        <value_reference>

<value_reference> ::= <identifier>

<defined_type> ::= <external_type_reference> | <identifier>

<defined_value> ::= <external_value_reference> |
                    <value_reference>

<type_assignment> ::= <identifier> '::=' <type>

<value_assignment> ::= <identifier> <type> '::=' <value>

<type> ::= <builtin_type> | <defined_type> | <subtype>

<builtin_type> ::= <boolean_type> | <integer_type> |
        <bitstring_type> | <octet_string_type> | <null_type> |
        <sequence_type> | <sequenceof_type> |
        <set_type> | <setof_type> | <choice_type> |
        <selection_type> |<tagged_type> | <any_type> |
        <object_identifier_type> | <character_string_type> |
        <useful_type> | <enumerated_type> | <real_type>

<named_type> ::= <identifier> <type> | <type> | <selection_type>

<value> ::= <builtin_value> | <defined_value>

<builtin_value> ::= <boolean_value> | <integer_value> |
        <bitstring_value> |
        <octet_string_value> | <null_value> | <sequence_value> |
        <sequenceof_value> | <set_value> | <setof_value> |
        <choice_value> | <selection_value> | <tagged_value> |
        <any_value> | <object_identifier_value> |
        <character_string_value> | <enumerated_value> |
        <real_value>

<named_value> ::= <identifier> <value> | <value>
```

```
<boolean_type> ::= ' BOOLEAN '

<boolean_value> ::= ' TRUE '  | ' FALSE '

<integer_type> ::= ' INTEGER ' |
                   ' INTEGER {' <named_number_list> '}'

<named_number_list> ::= <named_number> |
        <named_number_list>',' <named_number>

<named_number> ::= <identifier> '(' <signed_number> ')' |
                   <identifier> '(' <defined_value> ')'

<enumerated_type> ::= ' ENUMERATED {' <enumeration> '}'

<enumeration> ::= <named_number> |
        <enumeration> ',' <named_number>

<enumerated_value> ::= <identifier>

<real_type> ::= ' REAL '

<real_value> ::= <numeric_real_value> | <special_real_value>

<numeric_real_value> ::= '{' <mantissa> ',' <base> ','
        <exponent> '}' | 0

<mantissa> ::= <signed_number>

<base> ::= 2 | 10

<exponent> ::= <signed_number>

<special_real_value> ::= ' PLUS-INFINITY ' | ' MINUS-INFINITY '

<bitstring_type> ::= ' BIT STRING ' |
        ' BIT STRING {' <named_bit_list> '}'

<named_bit_list> ::= <named_bit> |
                <named_bit_list> ',' <named_bit>

<named_bit> ::= <identifier> '(' <number> ')'
        <identifier> '(' <defined_value> ')'

<bitstring_value> ::= <bstring> | <hstring> |
        '{' <identifier_list> '}' | <empty_list>

<empty_list> ::= '{}'

<identifier_list> ::= <identifier> |
                      <identifier_list>',' <identifier>
```

```
<octet_string_type> ::= ' OCTET STRING '

<octet_string_value> ::= <bstring> | <hstring>

<null_type> ::= NULL

<null_value> ::= NULL

<sequence_type> ::= ' SEQUENCE {' <element_type_list> '}' |
        ' SEQUENCE {}'

<element_type_list> ::= <element_type> |
        <element_type_list> ',' <element_type>

<element_type> ::= <named_type> |
                   <named_type> ' OPTIONAL '
                   <named_type> ' DEFAULT ' <value>
                   ' COMPONENTS OF ' <type>

<sequence_value> ::= '{' <element_value_list> '}' | <empty_list>

<element_value_list> ::= <named_value> |
        <element_value_list> ',' <named_value>

<sequenceof_type> ::= ' SEQUENCE OF ' <type> | ' SEQUENCE '

<sequenceof_value> ::= '{' <value_list> '}' | <empty_list>

<value_list> ::= <value> | <value_list> ',' <value>

<set_type> ::= ' SET {' <element_type_list> '}' | ' SET {} '

<set_value> ::= '{' <element_value_list> '}' | <empty_list>

<setof_type> ::= ' SET OF ' <type> | ' SET '

<setof_value> ::= '{' <value_list> '}' | <empty_list>

<choice_type> ::= ' CHOICE {' <alternative_type_list> '}'

<alternative_type_list> ::= <named_type> |
        <alternative_type_list> ',' <named_type>

<choice_value> ::= <named_value>

<selection_type> ::= <identifier> '<' <type>

<selection_value> ::= <named_value>

<tagged_type> ::= <tag> <type> |
                  <tag> ' IMPLICIT ' <type>
```

```
                <tag> ' EXPLICIT ' <type>

<tag> ::= '[' <class> <class_number> ']'

<class> ::= ' UNIVERSAL ' | ' APPLICATION ' |
            ' PRIVATE '   |  EPSILON

<class_number> ::= <number> | <defined_value>

<tagged_value> ::= <value>

<any_type> ::= ' ANY ' |
               ' ANY DEFINED BY ' <identifier>

<any_value> ::= <type> <value>

<object_identifier_type> ::= ' OBJECT IDENTIFIER '

<object_identifier_value> ::= '{' <obj_id_component_list> '}' |
        '{' <defined_value> <obj_id_component_list> '}'

<obj_id_component_list> ::= <obj_id_component> |
        <obj_id_component> <obj_id_component_list>

<obj_id_component> ::= <name_form> | <number_form> |
        <name_and_number_form>

<name_form> ::= <identifier>

<number_form> ::= <number> | <defined_value>

<name_and_number_form> ::= <identifier> '(' <number_form> ')'

<character_string_type> ::= 'NumericString' | 'PrintableString' |
        'TeletexString' | 'VisibleString' | 'IA5String' |
        'GraphicString' | 'GeneralString'

<character_string_value> ::= <cstring>

<useful_type> ::= <identifier>

<integer_value> ::= <signed_number> | <identifier>


-- SUBTYPE DECLARATIONS

<subtype> ::= <parent_type> <subtype_spec> |
        ' SET ' <size_constraint> ' OF ' <type>  |
        ' SEQUENCE ' <size_constraint> ' OF ' <type>
```

```
<parent_type> ::= <type>

<subtype_spec> ::= '(' <subtype_clause> ')'

<subtype_clause> ::= <subtype_value_set> <subtype_value_set_list>

<subtype_value_set_list> ::= '|' <subtype_value_set>
        <subtype_value_set_list> | EPSILON

<subtype_value_set> ::= <single_value> |
        <contained_subtype> | <value_range> |
        <permitted_alphabet> |  <size_constraint> |
        <inner_type_constraints>


<single_value> ::= <value>

<contained_subtype> ::= ' INCLUDES ' <type>

<value_range> ::= <lower_endpoint> ' .. ' <upper_endpoint>

<lower_endpoint> ::= <lower_end_value> | <lower_end_value> '~ <'

<upper_endpoint> ::= <upper_end_value> | '<' <upper_end_value>

<lower_end_value> ::= <value> | ' MIN '

<upper_end_value> ::= <value> | ' MAX '

<size_constraint> ::= ' SIZE ' <subtype_spec>

<permitted_alphabet> ::= ' FROM ' <subtype_spec>

<inner_type_constraints> ::=
        ' WITH COMPONENT ' <single_type_constraint> |
        ' WITH COMPONENTS ' <multiple_type_constraint>

<single_type_constraint> ::= <subtype_spec>

<multiple_type_constraint> ::= <full_specification>     |
                               <partial_specification>

<full_specification> ::= '{' <type_constraints> '}'

<partial_specification> ::= '{ ... , ' <type_constraints> '}'

<type_constraints> ::= <named_constraint> |
                <named_constraint> ',' <type_constraints>

<named_constraint> ::= <identifier> <constraint> | <constraint>
```

```
<constraint> ::= <value_constraint>  <presence_constraint>

<value_constraint> ::= <subtype_spec> | EPSILON

<presence_constraint> ::= ' PRESENT ' | ' ABSENT ' |
                          ' OPTIONAL '|  EPSILON

<signed_number> ::= <number> | '-' <number>

<identifier> ::= <uppercase_letter> <more_characters>

<uppercase_letter> ::= <RANGE: 'A' .. 'Z' >

<more_characters> ::= <letter> | EPSILON

<letter> ::= <RANGE: 'a' .. 'z'> | <uppercase_letter> | '-' |
             <digit>

<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Appendix B: XDR BNF

```
<XDR_Spec> ::= <XDR_Defn> <CR> <more_XDR_Defn> <CR>

<XDR_Defn> ::= <type_defn> | <constant_defn>

<more_XDR_defn> ::= <xdr_defn> | EPSILON

<type_defn> ::= 'typedef ' <decl> ';'             |
        'enum '   <identifier> <enum_body>   ';' |
        'struct ' <identifier> <struct_body> ';' |
        'union '  <identifier> <union_body>  ';'

<decl> ::= <unit_decl> <identifier>     |
        <fixed_length_unit_array>       |
        <variable_length_unit_array>    |
        <opaque_spec>                   |
        <string_spec>                   |
        <void_spec>

<unit_decl> ::= <integer_decl>          |
          <float_decl>                  |
          <boolean_decl>                |
          <enumerated_decl>             |
          <structure_decl>              |
          <union_spec>                  |
          <identifier>

<integer_decl> ::= <integer> | <unsigned_integer> |
             <hyper_integer> | <unsigned_hyper_integer>

  <integer> ::= 'int '

  <unsigned_integer> ::= 'unsigned int '

  <hyper_integer> ::= 'hyper '

  <unsigned_hyper_integer> ::= 'unsigned hyper '

<float_decl> ::= <single_float_decl> |
                 <double_float_decl>

  <single_float_decl> ::= 'float '

  <double_float_decl> ::= 'double '

<boolean_decl> ::= 'bool '

<enumerated_decl> ::= 'enum ' <enum_body>

  <enum_body> ::= '{' <enum_clause>
```

```
                    <other_enum_clause> '}'

  <enum_clause> ::= <identifier> ' = ' <integer_constant>

  <other_enum_clause> ::= ', ' <enum_clause> | EPSILON

<structure_decl> ::= <CR> 'struct ' <struct_body>

  <struct_body> ::= ' {' <struct_clause>
                        <other_struct_clause> ' }'

  <struct_clause> ::= <decl> ';'

  <other_struct_clause> ::= <struct_clause> | EPSILON

<union_spec> ::= 'union ' <union_body>

  <union_body> ::= ' switch (' <identifier> ')'
                '{ ' <union_clause> <more_union_clause>
                     <default_clause> '}'

  <union_clause> ::= <CR> 'case ' <length> ' : ' <decl> ';'

  <more_union_clause> ::= <union_clause> | EPSILON

  <default_clause> ::= 'default : ' <decl> ';' | EPSILON

<fixed_length_unit_array> ::= <unit_decl> <identifier>
                <fixed_length_spec>

<fixed_length_spec> ::= ' [' <length> ']'

  <length> ::= <positive_integer> | <identifier>

<variable_length_unit_array> ::= <unit_decl> <identifier>
                <variable_length_spec>

<variable_length_spec> ::=' <' <optional_length> '>'

  <optional_length> ::= <length> | EPSILON

<opaque_spec> ::= <fixed_length_opaque_spec> |
                  <variable_length_opaque_spec>

  <fixed_length_opaque_spec> ::= 'opaque ' <identifier>
                <fixed_length_spec>

  <variable_length_opaque_spec> ::= 'opaque ' <identifier>
                <variable_length_spec>

<string_spec> ::= 'string ' <identifier>
```

```
                <variable_length_spec>

<void_spec> ::= 'void'

<constant_defn> ::= 'const ' <identifier> ' = '
        <integer_constant> ';'

<integer_constant> ::= <RANGE: -2147483647 .. 2147483647>

<positive_integer> ::= <RANGE: 1 .. 2147483647>

<identifier> ::= <letter> <other_character>

    <letter> ::= <RANGE: 'a' .. 'z'> | <RANGE: 'A' .. 'Z'>

    <other_character> ::= <letter> | <digit> | '_'

    <digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Appendix C: ASN.1 BER Encode and Decode Routines in Ada

Performance is a critical issue in real-time systems. As shown earlier in this report, ASN.1 is a very general and powerful technique for communication in a heterogeneous distributed system. The purpose of this appendix is to explore, by means of an example, the computational cost of the generality of the ASN.1 BER. We hope to illustrate exactly what can be involved in the encoding to and the decoding from an ASN.1 representation, using the BER.

We selected the ASN.1 primitive type *real* as an example, as it appears to be the most complex of the ASN.1 primitive types. We further chose to implement the *encode* and *decode* routines in Ada. While more efficient implementations might well be possible in another language, such as assembler, we are more interested in illustrating the issues inherent in the use of ASN.1. As Ada is probably more widely understood than Sparc2 assembler language, we feel that Ada is the more appropriate choice. Further, we make no claims concerning the optimality of our code. We did not explore alternative Ada implementations. It is therefore quite possible that more efficient Ada implementations of the *encode* and *decode* routines are possible. Again, our purpose is to obtain a general idea of the possible computational costs of the generality of the ASN.1 BER.

Our implementation (included in this appendix) consists of a main program which we call the driver. To decode, the driver examines an ASN.1 encoding, determines the type and overall length of a component, and then calls the appropriate *decode* routine. To encode, the driver determines the type of the object to encode, and then calls the appropriate *encode* routine. Our driver, *encode*, and *decode* routines are implemented on a SPARCstation 2 using the self-hosted Verdix Ada compiler (VADS 6.03d).

As our purpose is to identify performance concerns inherent in the use of ASN.1 BER, we have accepted certain limitations on our implementation. For example, the *encode* and *decode* routines are implemented only for the short_float type (32 bit IEEE-754 float), and do not consider the cases of NaN (not a number), or plus or minus infinity. Only the ASN.1 binary encoding for reals has been implemented (a decimal/character encoding also exists).

The major criterion for the success of our *encode* and *decode* routines is the ability to encode a float, decode that encoding, and then verify that the resulting float is equal to the original float. This is done using both Ada float_io and formatted bit displays of the encodings and floats (see Figure C-1).

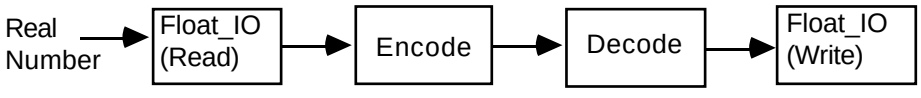


Figure C-1: Testing of Encode and Decode

There are several limitations of our testing of the *encode* and *decode* routines. We have only considered "reasonable" encodings of floats; that is, encodings that use the minimum number of octets to encode the exponent and mantissa of the represented real number. Another limitation of our testing is that we have not tested the encoding and decoding of numbers with bases other than two, although the routines are written to handle those cases. Finally, the *encode* and *decode* routines have not been tested across different machines or between different ASN.1 users.

The *encode* and *decode* procedures were compiled using three different compilers: the self hosted Verdix 6.03d compiler, the Vax Ada V2.3-3 compiler, and the XD Ada V1.2-23 compiler. The resultant code sizes for the *encode* and *decode* routines are shown in Table C-1. It is apparent that the assembler code generated is quite large. In each case, the compilation was performed with optimization of time as opposed to space.

Function	Vendor A	Vendor B	Vendor C
Encode	130	215	76
Decode	295	449	236

Table C-1: Generated Code Size for Encode/Decode Operations

There are several points to be made about the results presented in Table C-1.

- The results do not indicate of the relative merit of the compilers tested. The compilers represent different target architectures; for example the Sparc is a RISC machine, while the others are CISC machines.

- The generated code sizes do not account for the driver code that determines which routine should be called. Such generated code is expected to consist of less than 15 instructions.

- The generated code sizes do not include the elaboration code. Based on the number and types of local procedures and declarations, we expect that the elaboration code size for the *decode* routine would be at least three times greater than that of the *encode* routine.

- The *decode* procedure contains a statement that generates a call into the runtime library for each compiler tested. The call is for computing an integer base to an integer exponent.[12] An estimate of the additional assembler code for this operation is roughly 50 instructions, increasing the assembler code for the *decode* operation even more. No similar runtime calls were generated for the *encode* operation.

In view of the preceding, it is apparent that the results presented in Table C-1 underestimate the actual generated code size. What we consider interesting is the fact that the code to

[12] 2 ** (8 * Exponent_Size - 1).

perform the *decode* operation is roughly a factor of three times larger than the code to perform the *encode* operation.

As an additional experiment, we compiled our routines using the Verdix SUN-3 to MC68020, Version 6.0 cross compiler. We then tested the code on a 68020 processor that was monitored by a Tektronix DAS9200 logic analyzer. For the special case of 0.0, both the *encode* and *decode* routines ran in 26 microseconds. The results for all other tested real numbers are summarized in Table C-2.

Function	Execution Time
Encode	74 μsec
Decode	365 μsec

Table C-2: Measured Execution Times for Encode/Decode Operations

In any system, there are tradeoffs between generality and performance. Frequently, the more general a program is, the more computation it must perform. In our example, it appears that one pays a high price for the generality of ASN.1. These figures may well be of concern in a real-time system. Sensor data, for example, frequently consists of multiple real numbers. The receiver of such data would pay the decode overhead for each of the reals sent.

Note that part of the complexity of our *decode* routine stems from the fact that the ASN.1 real representation is not normalized. That is, one cannot determine, based solely on the number of octets of exponent and mantissa in the ASN.1 representation, whether a real is representable or not on a given machine. Thus the mantissa must be processed, and then the exponent computed, before it can be determined if the encoded real can be represented. For example, 2.0 can be encoded with a mantissa of 1 and exponent of 1, or a mantissa of 10 (binary) and an exponent of -1, or a mantissa of 100 (binary) and an exponent of -2, etc. There is no limit on the number of octets that can be used to represent the mantissa. Hence, assuming a base of 2 and a scaling factor of 0, any negative exponent that can be represented in an ASN.1 BER real representation can be a valid exponent for the number 2.0. Since ASN.1 BER allows up to 255 octets of exponent, this amounts to roughly 2 ** 2039 different valid representations for 2.0. Note that this is not a problem for XDR, as the XDR floating point representation is normalized.

Another problem we encountered is that ASN.1 does not provide an explicit indication of precision for reals. As demonstrated earlier, there are a number of different representations for each expressible real value. The ISO/IEC ASN.1 BER standard [3] states that the selection of a particular representation is at "... a sender's option, and can be used as a broad indication of precision." There is, however, no way for the receiver to know if the sender is using a particular representation as an indication of precision. This means that the receiver (the *decode* routine) must have some understanding of the sender (the *encode* routine) to properly interpret real data. This would require a protocol, which is beyond the scope of ASN.1. Given the generality of ASN.1, we find this odd.

As an aside on the use of Ada, one might wonder if the use of Ada9X might not improve the *encode* and *decode* routines. Since Ada9X is not currently available, it is not possible to obtain instruction counts. Ada9X will provide certain attribute functions and procedures for floats, such as *compose* and *decompose*. While *compose* and *decompose* could make the source code more readable, it is not clear that the generated object code would be more efficient than that generated by Ada83.

```
with system;
with unchecked_conversion;

package Target_Dependent_Definitions is

-- Exceptions

  ASN1_Error : exception;
  -- raised when the ASN.1 encoding cannot be decoded

-- Bit, Byte, and Word Declarations

  subtype Bit             is integer range 0..1;

  type Byte               is range 0..16#FF#;
    for Byte'size           use system.storage_unit;

  Word : constant := 4;

-- The following declarations of bit arrays, byte arrays, and
-- "special" integers are necessary as the Encode and Decode
-- routines need to examine octets, which sometimes need to be
-- treated as bits, while other times must be treated as
-- integers.

-- Declarations of arrays of Bit

  type Bit_Array is array (0..7) of Bit;
    for Bit_Array'size use system.storage_unit;
  pragma pack (Bit_Array);

  type Word_Bit_Array     is array (0..31) of Bit;
    for Word_Bit_Array'size use 4*system.storage_unit;
  pragma pack (Word_Bit_Array);

-- Declarations of arrays of Byte

  type Two_Byte_Array     is array (0..1)  of Byte;
    for Two_Byte_Array'size use 2*system.storage_unit;
  pragma pack (Two_Byte_Array);

  type Four_Byte_Array    is array (0..3)  of Byte;
    for Four_Byte_Array'size use 4*system.storage_unit;
  pragma pack (Four_Byte_Array);

-- Declarations of integers of "special" lengths for conversion
-- to and from bit and byte arrays

  type Two_Byte_Integer   is new integer range 0..65535;
    for Two_Byte_Integer'size use 2*system.storage_unit;
```

```
  type Four_Byte_Integer  is new integer range 0..system.max_int;
    for Four_Byte_Integer'size use 4*system.storage_unit;

-- Type Declarations for ASN.1 BER Real (Binary Encoding)

  type Encoding_Header is
    record
      Code           : integer range 0..1 := 1;
      Sign           : integer range 0..1;
      Base           : integer range 0..3 := 0;
      Scaling_Factor : integer range 0..3 := 0;
      Length         : integer range 0..3 := 1;
    end record;

  for Encoding_Header use
    record at mod 2;
      Code           at 0*Word range 0 .. 0;
      Sign           at 0*Word range 1 .. 1;
      Base           at 0*Word range 2 .. 3;
      Scaling_Factor at 0*Word range 4 .. 5;
      Length         at 0*Word range 6 .. 7;
    end record;
  for Encoding_Header'SIZE use system.storage_unit;

  type Contents_Array is array (natural range <>) of Byte;

  type Encoding (n : natural) is
    record
      Total_Length   : integer range 0..255;
      Header         : Encoding_Header;
      case n is
        when 0      =>
          null;
        when others =>
          Contents       : Contents_Array (1..n);
      end case;
    end record;
  pragma pack (Encoding);

  type ASN1_Encoding is access Encoding;

-- Type Declarations for IEEE floating points

  Type IEEE_Float is
    record
      Sign           : integer range 0..1;
      Exponent       : integer range 0..255;
      Mantissa       : integer range 0..8388607;
    end record;
  pragma pack (IEEE_Float);
```

```
  for IEEE_Float use
    record at mod 2;
      Sign           at 0*Word range 0 .. 0;
      Exponent       at 0*Word range 1 .. 8;
      Mantissa       at 0*Word range 9 .. 31;
    end record;

-- Unchecked Conversions necessary for Encode and Decode

  function To_Two_Byte_Array is
        new unchecked_conversion (SOURCE => Two_Byte_Integer,
                                  TARGET => Two_Byte_Array);

  function To_Four_Byte_Array is
        new unchecked_conversion (SOURCE => Four_Byte_Integer,
                                  TARGET => Four_Byte_Array);

  function To_Four_Byte_Integer is
        new unchecked_conversion (SOURCE => Four_Byte_Array,
                                  TARGET => Four_Byte_Integer);

  function To_Bit_Array is
        new unchecked_conversion (SOURCE => Byte,
                                  TARGET => Bit_Array);
  function To_Bit_Array is
        new unchecked_conversion (SOURCE => Integer,
                                  TARGET => Word_Bit_Array);

  function To_Bit_Array is
        new unchecked_conversion (SOURCE => Encoding_Header,
                                  TARGET => Bit_Array);

  function To_Short_Float is
        new unchecked_conversion (SOURCE => Word_Bit_Array,
                                  TARGET => Short_Float);

  function To_IEEE is
        new unchecked_conversion (SOURCE => Short_Float,
                                  TARGET => IEEE_Float);

-- Global Declarations necessary for Encode and Decode routines

  Zero_Float_Encoding          : ASN1_Encoding
                               := new Encoding (n => 0);

  Single_Float_Encoding        : ASN1_Encoding
                               := new Encoding (n => 5);

  Zero_Float_Encoding_Length   : constant := 1;
  Single_Float_Encoding_Length : constant := 6;
```

```
 -- binary ASN.1 BER encoding
  Single_Float_Code            : constant := 1;

 -- base 2 number
  Single_Float_Base            : constant := 0;

  Single_Float_Scaling_Factor  : constant := 0;

 -- 2 octets of exponent
  Single_Float_Length          : constant := 1;

-- Constants for the Encode routine

-- For conversion from an 8 bit two's complement to a 16 bit
-- two's complement number, with a 23 bit shift included. See
-- Notes section in Encode.

  Conversion_Constant          : constant := 32617;

-- For restoring the implied leading one missing from the IEEE
-- normalized float.

  Mantissa_Implied_1           : constant := 2 ** 23;

-- Constants for the Decode routine

-- Implementation limit on maximum length of an encoding which
-- will be accepted.

  Maximum_Encoding_Length      : constant := 10;

-- Implementation limit on maximum number of octets which will be
-- accepted.

  Maximum_Exponent_Octets      : constant := 4;

-- Number of bits in an octet.

  Bits_In_Octet                : constant := 8;

-- Bit positions needed to extract the exponent mantissa from an
-- encoding to a Short_Float.

  Bit0                         : constant := 0;
  Bit1                         : constant := 1;
  Bit8                         : constant := 8;
  Bit9                         : constant := 9;
  Bit24                        : constant := 24;
  Bit31                        : constant := 31;

end Target_Dependent_Definitions;
```

```
with Target_Dependent_Definitions;
  use Target_Dependent_Definitions;

procedure Encode (Decoded_Float :     Short_Float;
                  Encoded_Float : out ASN1_Encoding) is

 -- Encode an Single Float in an ASN.1 representation using BER

-- Notes:
--  1) The Short_Float representation is normalized.
--  2) 0.0 is a special case. The encoding contains no contents
--     octets. Only the sign bit in the header has meaning.
--  3) The Short_Float exponent is 8 bits in two's complement
--     notation. It is a "bias" representation. That is:
--       Short_Float exponent = actual exponent + bias,
--     where, for the 8 bit representation, bias = 127.
--  4) The Short_Float mantissa is 23 bits of binary fraction.
--     The mantissa of the real number represented by the
--     Short_Float is 1 greater than the Short_Float's mantissa.
--     That is, if the Short_Float stores a mantissa of :
--                    .2345
--     the mantissa of the real number represented by that
--     Short_Float is :
--                   1.2345
--  5) The ASN.1 exponent is represented in an integral number of
--     octets as a two's complement integer.
--  6) The ASN.1 mantissa is represented in an integral number of
--     octets as a binary integer (with all of its digits).
--  7) Conversion of the Short_Float to the ASN.1 encoded
--     representation conceptually involves the following:
--    a) the conversion of the Short_Float's mantissa to a binary
--       integer with its leading one restored. (i.e., add 1 and
--       multiply by 2 ** 23)
--    b) the conversion of the 8 bit two's complement
--       Short_Float's exponent to the equivalent 16 bit two's
--       complement integer (necessary since the mantissa has
--       been multiplied by 2 ** 23, the exponent must be
--       decreased by 23, which could overflow in 8 bits.
--  8) In practice, going from a two's complement exponent in 8
--     bits to a two's complement exponent in 16 bits requires a
--     change in bias from 127 to 32767 (i.e., add 32640).
--     Further the conversion of the mantissa from the IEEE
--     normalized fraction of 23 bits to the ASN.1 mantissa as a
--     binary integer requires an adjustment of -23. Hence the
--     constant of 32617 (Conversion_Constant).
--  9) The 23 bit Short_Float mantissa is converted to a 32 bit
--     (rather than a 24 bit) representation due to alignment
--     problems on the Sparc2.
-- 10) The only field in the encoding header that depends on
--     the particular Short_Float being encoded is the sign. We
--     are always using the ASN.1 binary encoding (Code = 1), a
```

```
--     base 2 number (Base = 0), a scaling factor of 0
--     (Scaling_Factor = 0), and an exponent length of 2 octets
--     (Length = 1).

  Single_Float  : IEEE_Float;
  Temp_Exp      : Two_Byte_Integer   := 0;
  Temp_Two      : Two_Byte_Array;
  Temp_Mantissa : Four_Byte_Integer;
  Temp_Four     : Four_Byte_Array;

begin

  Single_Float := To_IEEE (Decoded_Float);

  if Decoded_Float = 0.0 then
    Zero_Float_Encoding.Header.Sign  := Single_Float.Sign;
    Encoded_Float                    := Zero_Float_Encoding;
    return;
  end if;

-- Move the 8 bit exponent to a two byte integer representation

  Temp_Exp := Two_Byte_Integer(Single_Float.Exponent);

-- Adjust for the conversion from an 8 bit two's complement to a
-- 16 bit two's complement, and also for the 23 bit shift

  Temp_Exp := Temp_Exp + Conversion_Constant;

-- Set the fields in the encoding header

  Single_Float_Encoding.Header.Code
    := Single_Float_Code;

  Single_Float_Encoding.Header.Sign
    := Single_Float.Sign;

  Single_Float_Encoding.Header.Base
    := Single_Float_Base;

  Single_Float_Encoding.Header.Scaling_Factor
    := Single_Float_Scaling_Factor;

  Single_Float_Encoding.Header.Length
    := Single_Float_Length;

-- Store the converted exponent

  Temp_Two := To_Two_Byte_Array (Temp_Exp);
  Single_Float_Encoding.Contents(1) := Temp_Two(0);
  Single_Float_Encoding.Contents(2) := Temp_Two(1);
```

```
-- Convert the mantissa from the 23 bit normalized fraction to a
-- binary integer

  Temp_Mantissa := Four_Byte_Integer(Single_Float.Mantissa);

-- Add 1 to the mantissa

  if Single_Float.Exponent > 0 then
    Temp_Mantissa := Temp_Mantissa + Mantissa_Implied_1;
  end if;

-- Store the converted mantissa

  Temp_Four := To_Four_Byte_Array (Temp_Mantissa);
  for i in 1 .. 3 loop
    Single_Float_Encoding.Contents (i+2) := Temp_Four (i);
  end loop;

  Encoded_Float := Single_Float_Encoding;

end Encode;
```

```
with Target_Dependent_Definitions;
  use Target_Dependent_Definitions;

procedure Decode (Encoded_Float :     ASN1_Encoding;
                  Decoded_Float : out Short_Float) is

-- Decode an ASN.1 REAL representation (using BER) to a Float

-- Notes:
--  1) 0.0 is a special case, represented in the encoding by zero
--     octets of contents.
--  2) The ASN.1 representation for reals is not normalized. The
--     encoder of a real may use up to 255 octets to represent
--     the exponent, and an unlimited number of octets to
--     represent the mantissa. Hence it is not possible to deduce
--     whether an encoded real is representable or not on a given
--     machine without first decoding it. For example, 2.0 can be
--     encoded with a mantissa of 1 and exponent of 1, or a
--     mantissa of 10  (binary) and an exponent of -1, or a
--     mantissa of 100 (binary) and an exponent of -2, etc.
--  3) For our purposes we have limited the total encoding sizes
--     to 10 octets, and the maximum exponent size to 4.
--  4) ASN.1 allows for base 2 (Header.Base = 0), base 8
--     (Header.Base = 1), and base 16 (Header.Base = 2) numbers.
--  5) ASN.1 encodes the exponent in a series of octets that
--     follow the header. The Length field in the header
--     specifies the number of octets in that encoding. If
--     Header.Length is 0, there is 1 octet of exponent; if 1,
--     there are 2 octets of exponent; if 2, there are 3 octets
--     of exponent; if 3, the first octet contains the number of
--     following octets that contain the exponent (up to 255
--     octets). Any remaining octets contain the mantissa.
--  6) The ASN.1 exponent is stored as a two's complement binary
--     number.
--  7) The ASN.1 mantissa is stored as a binary integer.
--  8) Since there is no restriction on the number of octets in
--     the mantissa, one must search the stored mantissa from
--     high-order bit to low-order bit to find the first one bit
--     (i.e., drop the leading zeros). Since the Sparc2
--     Short_Float is normalized, that first bit will not be
--     stored in the decoded float. The 23 bits following the
--     first one bit will be moved into the mantissa field of the
--     decoded float. Since the ASN.1 representation does not
--     contain an explicit indication of precision (i.e., the
--     number of significant digits in the mantissa), the Decode
--     routine can do no more.
--  9) The position of the first one bit in the mantissa together
--     with the number of octets that contain the mantissa
--     indicate the size of the real number being represented.
--     Since the decoded float is normalized, the exponent to be
--     stored in the decoded float must be adjusted accordingly.
```

```
-- 10) Based on note 8), notice that the encoded float cannot be
--     determined to be unrepresentable on the target machine
--     (the Sparc2 in this case) until after the calculation of
--     the encoded float's exponent.


  Base           : integer           := 1;
  Exponent_Size  : integer           := 0;
  Temp_Exp       : Four_Byte_Integer := 0;
  Temp_Four      : Four_Byte_Array   := (others => 16#0# );
  Index          : natural           := 0;
  Bit_Index      : natural           := 0;
  Temp_Bits      : Bit_Array         := (others => 0);
  Temp_Float     : Word_Bit_Array    := (others => 0);
  First_One_Bit  : integer           := 0;
  Actual_Exp     : integer           := 0;
  Temp_Bit_Array : Word_Bit_Array    := (others => 0);

  function Test_First_Bit (Byte_To_Test : Byte;
                           Bit_Value    : Bit)
                          return boolean is
    Temp_Bit_Array : Bit_Array;
  begin
    Temp_Bit_Array := To_Bit_Array (Byte_To_Test);
    return (Temp_Bit_Array (Bit0) = Bit_Value);
  end Test_First_Bit;


begin

  Temp_Float (0) := Encoded_Float.Header.Sign;

  if Encoded_Float.n = 0 then
    Decoded_Float := To_Short_Float (Temp_Float);
    return;
  end if;

  if Encoded_Float.Total_Length > Maximum_Encoding_Length then
    raise ASN1_Error;
  end if;

  case Encoded_Float.Header.Base is
    when 0 =>
      Base := 1;
    when 1 =>
      Base := 3;
    when 2 =>
      Base := 4;
    when 3 =>
      raise ASN1_Error;
  end case;
```

```
-- Extract the exponent octets from the encoded float

  Temp_Four := (others => 16#0#);
  if Encoded_Float.Header.Length < 3 then
    Exponent_Size := Encoded_Float.Header.Length + 1;
    for i in 1 .. Exponent_Size loop
     Temp_Four (i+3-Exponent_Size) := Encoded_Float.Contents (i);
    end loop;
  else
    Exponent_Size := integer(Encoded_Float.Contents(1));
    if Exponent_Size > Maximum_Exponent_Octets then
      raise ASN1_Error;
    end if;
    if Encoded_Float.Contents(2) = 16#0# then
      -- test first bit of next octet. if is 0, error
      if Test_First_Bit (Encoded_Float.Contents(2), 0) then
        raise ASN1_Error;
      end if;
    elsif Encoded_Float.Contents(2) = 16#FF# then
      -- test first bit of next octet. if is 1, error
      if Test_First_Bit (Encoded_Float.Contents(2), 1) then
        raise ASN1_Error;
      end if;
    end if;
    for i in 1 .. Exponent_Size loop
      Temp_Four (i+3-Exponent_Size)
        := Encoded_Float.Contents (1+i);
    end loop;
  end if;

  Temp_Exp := To_Four_Byte_Integer (Temp_Four);

-- Find first word in mantissa which contains a non_zero bit.

  index := Exponent_Size + 1;
  while not (index > Encoded_Float.n) loop
    if Encoded_Float.Contents(index) = 16#0# then
      index := index + 1;
    else
      exit;
    end if;
  end loop;

-- Find the first one bit in the first non_zero byte of the ASN.1
--   mantissa representation.
-- Determine its position in the mantissa (i.e., compute
--   First_One_Bit).
-- Slice the 23 bits following the first one bit into the decoded
--   float's mantissa field.

  if index > Encoded_Float.n then
```

```
  -- have a mantissa of 0
    null;
  else
    Temp_Bits := To_Bit_Array (Encoded_Float.Contents(Index));
    Bit_Index := 0;
    while Temp_Bits (Bit_Index) = 0 loop
      Bit_Index := Bit_Index + 1;
    end loop;

    First_One_Bit  := Bits_In_Octet * (Encoded_Float.n - Index)
                      + 7 - Bit_Index;

    Bit_Index := Bit_Index + 1;

    for i in Bit9 .. Bit31 loop
      if Bit_Index > 7 then
        Index := Index + 1;
        Temp_Bits
          := To_Bit_Array (Encoded_Float.Contents(Index));
        Bit_Index := 0;
      end if;
      Temp_Float(i) := Temp_Bits (Bit_Index);
      Bit_Index := Bit_Index + 1;
    end loop;
  end if;

-- Compute the actual value of the exponent

  Actual_Exp := Base * integer(Temp_Exp)
                + Encoded_Float.Header.Scaling_Factor
                + First_One_Bit
                - (Base * (2 **(8 * Exponent_Size - 1) - 1));

-- If the actual value of the exponent is representable on the
-- target machine (in this case, the Sparc2), convert the value
-- to two's complement and store it in the decoded float.

  if Actual_Exp > Short_Float'Machine_Emax or
    Actual_Exp < Short_Float'Machine_Emin then
    raise ASN1_Error;
  else
    Actual_Exp := Actual_Exp + 127;
    Temp_Bit_Array := To_Bit_Array (Actual_Exp);
    Temp_Float (Bit1..Bit8) := Temp_Bit_Array (Bit24..Bit31);
  end if;

  Decoded_Float := To_Short_Float (Temp_Float);

end Decode;
```

```
with Decode;
with Encode;
with Target_Dependent_Definitions;
  use Target_Dependent_Definitions;
with text_io;
with unchecked_conversion;

procedure Driver is

-- Declarations

begin

-- Initialize the Encoding areas

-- Get the float to test

-- Encode the float to test

-- Dump Encoding

-- Decode the created encoding

-- Display the original float

-- Display the resulting float

end Driver;
```