XDR: External Data Representation Standard

STATUS OF THIS MEMO

   This RFC describes a standard that Sun Microsystems, Inc., and others
   are using, one we wish to propose for the Internet's consideration.
   Distribution of this memo is unlimited.

1. INTRODUCTION

   XDR is a standard for the description and encoding of data. It is
   useful for transferring data between different computer
   architectures, and has been used to communicate data between such
   diverse machines as the SUN WORKSTATION*, VAX*, IBM-PC*, and Cray*.
   XDR fits into the ISO presentation layer, and is roughly analogous in
   purpose to X.409, ISO Abstract Syntax Notation. The major difference
   between these two is that XDR uses implicit typing, while X.409 uses
   explicit typing.

   XDR uses a language to describe data formats. The language can only
   be used only to describe data; it is not a programming language.
   This language allows one to describe intricate data formats in a
   concise manner. The alternative of using graphical representations
   (itself an informal language) quickly becomes incomprehensible when
   faced with complexity. The XDR language itself is similar to the C
   language [1], just as Courier [4] is similar to Mesa. Protocols such
   as Sun RPC (Remote Procedure Call) and the NFS* (Network File System)
   use XDR to describe the format of their data.

   The XDR standard makes the following assumption: that bytes (or
   octets) are portable, where a byte is defined to be 8 bits of data.
   A given hardware device should encode the bytes onto the various
   media in such a way that other hardware devices may decode the bytes
   without loss of meaning. For example, the Ethernet* standard
   suggests that bytes be encoded in "little-endian" style [2], or least
   significant bit first.

2. BASIC BLOCK SIZE

   The representation of all items requires a multiple of four bytes (or
   32 bits) of data. The bytes are numbered 0 through n-1. The bytes
   are read or written to some byte stream such that byte m always
   precedes byte m+1. If the n bytes needed to contain the data are not
   a multiple of four, then the n bytes are followed by enough (0 to 3)

residual zero bytes, r, to make the total byte count a multiple of 4.

We include the familiar graphic box notation for illustration and
comparison. In most illustrations, each box (delimited by a plus
sign at the 4 corners and vertical bars and dashes) depicts a byte.
Ellipses (...) between boxes show zero or more additional bytes where
required.

```
     +--------+--------+...+--------+--------+...+--------+
     | byte 0 | byte 1 |...|byte n-1|    0   |...|    0   |   BLOCK
     +--------+--------+...+--------+--------+...+--------+
     |<----------n bytes---------->|<------r bytes------>|
     |<----------n+r (where (n+r) mod 4 = 0)----------->|
```
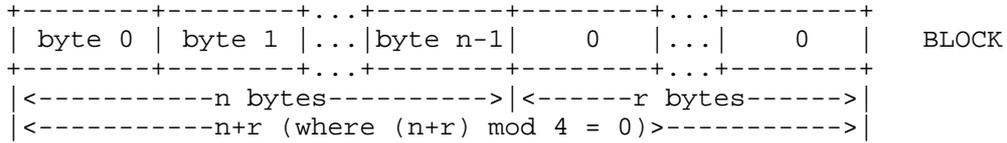
3. XDR DATA TYPES

Each of the sections that follow describes a data type defined in the
XDR standard, shows how it is declared in the language, and includes
a graphic illustration of its encoding.

For each data type in the language we show a general paradigm
declaration. Note that angle brackets (< and >) denote
variablelength sequences of data and square brackets ([and]) denote
fixed-length sequences of data. "n", "m" and "r" denote integers.
For the full language specification and more formal definitions of
terms such as "identifier" and "declaration", refer to section 5:
"The XDR Language Specification".

For some data types, more specific examples are included. A more
extensive example of a data description is in section 6: "An Example
of an XDR Data Description".

3.1 Integer

An XDR signed integer is a 32-bit datum that encodes an integer in
the range [-2147483648,2147483647]. The integer is represented in
two's complement notation. The most and least significant bytes are
0 and 3, respectively. Integers are declared as follows:

        int identifier;

```
   (MSB)                    (LSB)
   +-------+-------+-------+-------+
   |byte 0 |byte 1 |byte 2 |byte 3 |                    INTEGER
   +-------+-------+-------+-------+
   <------------32 bits------------>
```
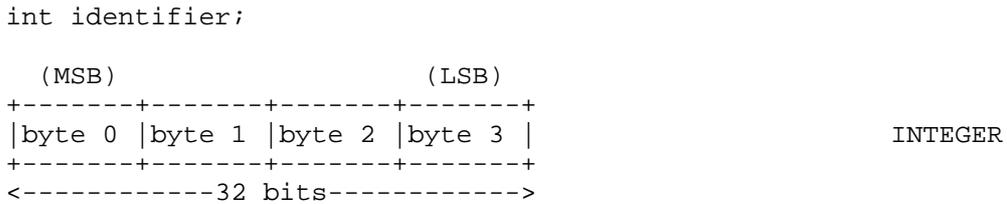
3.2.Unsigned Integer

   An XDR unsigned integer is a 32-bit datum that encodes a nonnegative
   integer in the range [0,4294967295]. It is represented by an
   unsigned binary number whose most and least significant bytes are 0
   and 3, respectively. An unsigned integer is declared as follows:

        unsigned int identifier;

```
     (MSB)                    (LSB)
     +-------+-------+-------+-------+
     |byte 0 |byte 1 |byte 2 |byte 3 |            UNSIGNED INTEGER
     +-------+-------+-------+-------+
     <-----------32 bits------------>
```

3.3 Enumeration

   Enumerations have the same representation as signed integers.
   Enumerations are handy for describing subsets of the integers.
   Enumerated data is declared as follows:

        enum { name-identifier = constant, ... } identifier;

   For example, the three colors red, yellow, and blue could be
   described by an enumerated type:

        enum { RED = 2, YELLOW = 3, BLUE = 5 } colors;

   It is an error to encode as an enum any other integer than those that
   have been given assignments in the enum declaration.

3.4 Boolean

   Booleans are important enough and occur frequently enough to warrant
   their own explicit type in the standard. Booleans are declared as
   follows:

        bool identifier;

        This is equivalent to:

        enum { FALSE = 0, TRUE = 1 } identifier;

3.5 Hyper Integer and Unsigned Hyper Integer

   The standard also defines 64-bit (8-byte) numbers called hyper
   integer and unsigned hyper integer. Their representations are the
   obvious extensions of integer and unsigned integer defined above.
   They are represented in two's complement notation. The most and
   least significant bytes are 0 and 7, respectively. Their
   declarations:

   hyper identifier; unsigned hyper identifier;

```
    (MSB)                                                    (LSB)
   +-------+-------+-------+-------+-------+-------+-------+-------+
   |byte 0 |byte 1 |byte 2 |byte 3 |byte 4 |byte 5 |byte 6 |byte 7 |
   +-------+-------+-------+-------+-------+-------+-------+-------+
   <--------------------------64 bits-------------------------->
                                          HYPER INTEGER
                                          UNSIGNED HYPER INTEGER
```
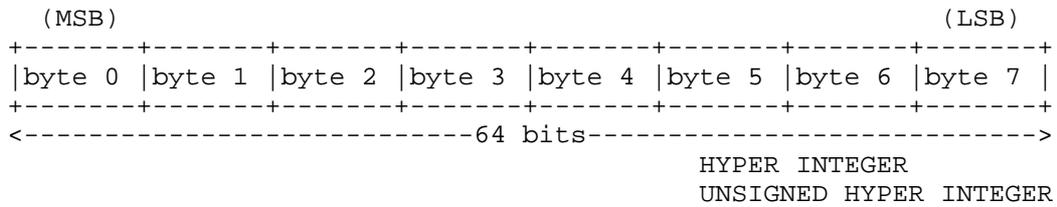
3.6 Floating-point

   The standard defines the floating-point data type "float" (32 bits or
   4 bytes). The encoding used is the IEEE standard for normalized
   single-precision floating-point numbers [3]. The following three
   fields describe the single-precision floating-point number:

      S: The sign of the number. Values 0 and 1 represent positive and
         negative, respectively. One bit.

      E: The exponent of the number, base 2. 8 bits are devoted to this
         field. The exponent is biased by 127.

      F: The fractional part of the number's mantissa, base 2. 23 bits
         are devoted to this field.

   Therefore, the floating-point number is described by:

         $(-1)^{**S} * 2^{**(E-Bias)} * 1.F$

It is declared as follows:
```
     float identifier;

     +-------+-------+-------+-------+
     |byte 0 |byte 1 |byte 2 |byte 3 |        SINGLE-PRECISION
     S|   E   |           F           |     FLOATING-POINT NUMBER
     +-------+-------+-------+-------+
     1|<- 8 ->|<-------23 bits------>|
      <-----------32 bits----------->
```
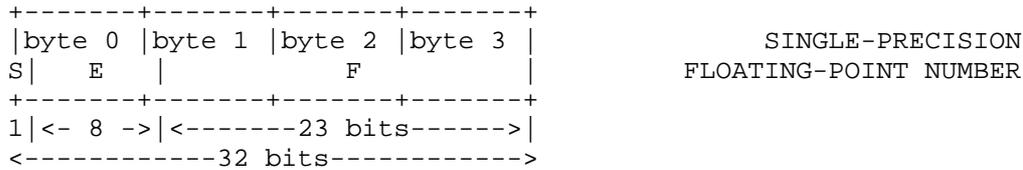
Just as the most and least significant bytes of a number are 0 and 3,
the most and least significant bits of a single-precision floating-
point number are 0 and 31. The beginning bit (and most significant
bit) offsets of S, E, and F are 0, 1, and 9, respectively. Note that
these numbers refer to the mathematical positions of the bits, and
NOT to their actual physical locations (which vary from medium to
medium).

The EEE specifications should be consulted concerning the encoding
for signed zero, signed infinity (overflow), and denormalized numbers
(underflow) [3]. According to IEEE specifications, the "NaN" (not a
number) is system dependent and should not be used externally.

3.7 Double-precision Floating-point

The standard defines the encoding for the double-precision floating-
point data type "double" (64 bits or 8 bytes). The encoding used is
the IEEE standard for normalized double-precision floating-point
numbers [3]. The standard encodes the following three fields, which
describe the double-precision floating-point number:

   S: The sign of the number. Values 0 and 1 represent positive and
      negative, respectively. One bit.

   E: The exponent of the number, base 2. 11 bits are devoted to
      this field. The exponent is biased by 1023.

   F: The fractional part of the number's mantissa, base 2. 52 bits
      are devoted to this field.

Therefore, the floating-point number is described by:

     $(-1)^{**}S * 2^{**}(E-Bias) * 1.F$

It is declared as follows:

        double identifier;
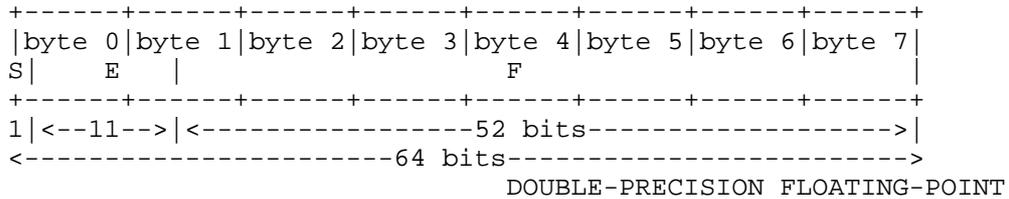
```
    +------+------+------+------+------+------+------+------+
    |byte 0|byte 1|byte 2|byte 3|byte 4|byte 5|byte 6|byte 7|
    S|   E  |                    F                          |
    +------+------+------+------+------+------+------+------+
    1|<--11-->|<----------------52 bits------------------>|
    <---------------------64 bits------------------------>
                                    DOUBLE-PRECISION FLOATING-POINT
```

Just as the most and least significant bytes of a number are 0 and 3,
the most and least significant bits of a double-precision floating-
point number are 0 and 63. The beginning bit (and most significant
bit) offsets of S, E , and F are 0, 1, and 12, respectively. Note
that these numbers refer to the mathematical positions of the bits,
and NOT to their actual physical locations (which vary from medium to
medium).

The IEEE specifications should be consulted concerning the encoding
for signed zero, signed infinity (overflow), and denormalized numbers
(underflow) [3]. According to IEEE specifications, the "NaN" (not a
number) is system dependent and should not be used externally.

3.8 Fixed-length Opaque Data

At times, fixed-length uninterpreted data needs to be passed among
machines. This data is called "opaque" and is declared as follows:

        opaque identifier[n];

where the constant n is the (static) number of bytes necessary to
contain the opaque data. If n is not a multiple of four, then the n
bytes are followed by enough (0 to 3) residual zero bytes, r, to make
the total byte count of the opaque object a multiple of four.

```
       0        1     ...
    +--------+--------+...+--------+--------+...+--------+
    | byte 0 | byte 1 |...|byte n-1|    0   |...|    0   |
    +--------+--------+...+--------+--------+...+--------+
    |<----------n bytes---------->|<------r bytes------>|
    |<----------n+r (where (n+r) mod 4 = 0)------------>|
                                        FIXED-LENGTH OPAQUE
```
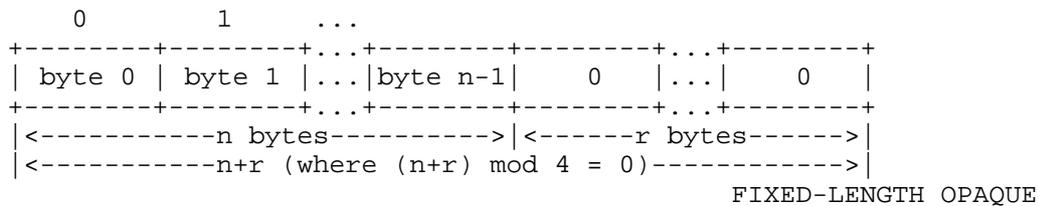
3.9 Variable-length Opaque Data

The standard also provides for variable-length (counted) opaque data,

defined as a sequence of n (numbered 0 through n-1) arbitrary bytes
to be the number n encoded as an unsigned integer (as described
below), and followed by the n bytes of the sequence.

Byte m of the sequence always precedes byte m+1 of the sequence, and
byte 0 of the sequence always follows the sequence's length (count).
If n is not a multiple of four, then the n bytes are followed by
enough (0 to 3) residual zero bytes, r, to make the total byte count
a multiple of four. Variable-length opaque data is declared in the
following way:

        opaque identifier<m>;
    or
        opaque identifier<>;

The constant m denotes an upper bound of the number of bytes that the
sequence may contain. If m is not specified, as in the second
declaration, it is assumed to be (2**32) - 1, the maximum length.
The constant m would normally be found in a protocol specification.
For example, a filing protocol may state that the maximum data
transfer size is 8192 bytes, as follows:
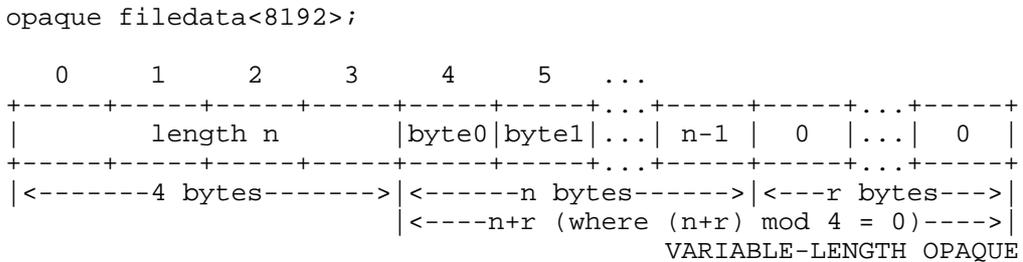
        opaque filedata<8192>;

```
      0     1     2     3     4     5   ...
   +-----+-----+-----+-----+-----+-----+...+-----+-----+...+-----+
   |        length n       |byte0|byte1|...| n-1 |  0  |...|  0  |
   +-----+-----+-----+-----+-----+-----+...+-----+-----+...+-----+
   |<-------4 bytes------->|<------n bytes------>|<---r bytes--->|
                           |<----n+r (where (n+r) mod 4 = 0)---->|
                                                 VARIABLE-LENGTH OPAQUE
```

It is an error to encode a length greater than the maximum described
in the specification.

3.10 String

The standard defines a string of n (numbered 0 through n-1) ASCII
bytes to be the number n encoded as an unsigned integer (as described
above), and followed by the n bytes of the string. Byte m of the
string always precedes byte m+1 of the string, and byte 0 of the
string always follows the string's length. If n is not a multiple of
four, then the n bytes are followed by enough (0 to 3) residual zero
bytes, r, to make the total byte count a multiple of four. Counted
byte strings are declared as follows:

```
        string object<m>;
     or
        string object<>;
```

The constant m denotes an upper bound of the number of bytes that a
string may contain. If m is not specified, as in the second
declaration, it is assumed to be (2**32) - 1, the maximum length.
The constant m would normally be found in a protocol specification.
For example, a filing protocol may state that a file name can be no
longer than 255 bytes, as follows:

```
        string filename<255>;

          0     1     2     3     4     5   ...
        +-----+-----+-----+-----+-----+-----+...+-----+-----+...+-----+
        |         length n        |byte0|byte1|...| n-1 |  0  |...|  0  |
        +-----+-----+-----+-----+-----+-----+...+-----+-----+...+-----+
        |<-------4 bytes------->|<------n bytes------>|<---r bytes--->|
                                |<----n+r (where (n+r) mod 4 = 0)---->|
                                                                STRING
```

It is an error to encode a length greater than the maximum described
in the specification.

3.11 Fixed-length Array

Declarations for fixed-length arrays of homogeneous elements are in
the following form:

```
        type-name identifier[n];
```

Fixed-length arrays of elements numbered 0 through n-1 are encoded by
individually encoding the elements of the array in their natural
order, 0 through n-1. Each element's size is a multiple of four
bytes. Though all elements are of the same type, the elements may
have different sizes. For example, in a fixed-length array of
strings, all elements are of type "string", yet each element will
vary in its length.

```
        +---+---+---+---+---+---+---+---+...+---+---+---+---+
        |   element 0   |   element 1   |...|  element n-1  |
        +---+---+---+---+---+---+---+---+...+---+---+---+---+
        |<------------------n elements------------------->|

                                    FIXED-LENGTH ARRAY
```

3.12 Variable-length Array

   Counted arrays provide the ability to encode variable-length arrays
   of homogeneous elements. The array is encoded as the element count n
   (an unsigned integer) followed by the encoding of each of the array's
   elements, starting with element 0 and progressing through element n-
   1. The declaration for variable-length arrays follows this form:

         type-name identifier<m>;
      or
         type-name identifier<>;

   The constant m specifies the maximum acceptable element count of an
   array; if m is not specified, as in the second declaration, it is
   assumed to be (2**32) - 1.

         0 1 2 3
      +--+--+--+--+--+--+--+--+--+--+--+--+...+--+--+--+--+
      | n | element 0 | element 1 |...|element n-1|
      +--+--+--+--+--+--+--+--+--+--+--+--+...+--+--+--+--+
      |<-4 bytes->|<--------------n elements------------->|
                                                    COUNTED ARRAY
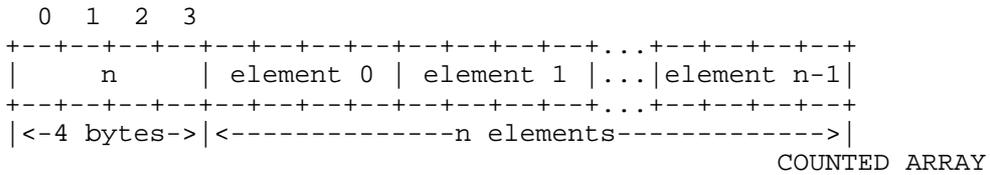
   It is an error to encode a value of n that is greater than the
   maximum described in the specification.

3.13 Structure

   Structures are declared as follows:

         struct {
            component-declaration-A;
            component-declaration-B;
            ...
         } identifier;

   The components of the structure are encoded in the order of their
   declaration in the structure. Each component's size is a multiple of
   four bytes, though the components may be different sizes.

         +-------------+-------------+...
         | component A | component B |...   STRUCTURE
         +-------------+-------------+...
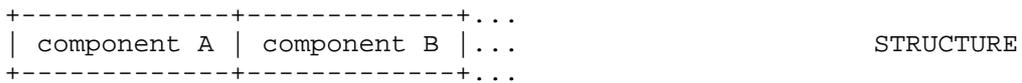
3.14 Discriminated Union

   A discriminated union is a type composed of a discriminant followed
   by a type selected from a set of prearranged types according to the

value of the discriminant. The type of discriminant is either "int",
"unsigned int", or an enumerated type, such as "bool". The component
types are called "arms" of the union, and are preceded by the value
of the discriminant which implies their encoding. Discriminated
unions are declared as follows:

```
      union switch (discriminant-declaration) {
      case discriminant-value-A:
         arm-declaration-A;
      case discriminant-value-B:
         arm-declaration-B;
      ...
      default: default-declaration;
      } identifier;
```

Each "case" keyword is followed by a legal value of the discriminant.
The default arm is optional. If it is not specified, then a valid
encoding of the union cannot take on unspecified discriminant values.
The size of the implied arm is always a multiple of four bytes.

The discriminated union is encoded as its discriminant followed by
the encoding of the implied arm.

```
        0   1   2   3
      +---+---+---+---+---+---+---+---+
      | discriminant  | implied arm   |       DISCRIMINATED UNION
      +---+---+---+---+---+---+---+---+
      |<---4 bytes--->|
```

3.15 Void

   An XDR void is a 0-byte quantity. Voids are useful for describing
   operations that take no data as input or no data as output. They are
   also useful in unions, where some arms may contain data and others do
   not. The declaration is simply as follows:
       void;

   Voids are illustrated as follows:

```
         ++
         ||                                        VOID
         ++
      --><-- 0 bytes
```

3.16 Constant

   The data declaration for a constant follows this form:

```
     const name-identifier = n;
```

"const" is used to define a symbolic name for a constant; it does not
declare any data. The symbolic constant may be used anywhere a
regular constant may be used. For example, the following defines a
symbolic constant DOZEN, equal to 12.

```
     const DOZEN = 12;
```

3.17 Typedef

"typedef" does not declare any data either, but serves to define new
identifiers for declaring data. The syntax is:

```
     typedef declaration;
```

The new type name is actually the variable name in the declaration
part of the typedef. For example, the following defines a new type
called "eggbox" using an existing type called "egg":

```
     typedef egg eggbox[DOZEN];
```

Variables declared using the new type name have the same type as the
new type name would have in the typedef, if it was considered a
variable. For example, the following two declarations are equivalent
in declaring the variable "fresheggs":

```
     eggbox  fresheggs;
     egg     fresheggs[DOZEN];
```

When a typedef involves a struct, enum, or union definition, there is
another (preferred) syntax that may be used to define the same type.
In general, a typedef of the following form:

```
     typedef <<struct, union, or enum definition>> identifier;
```

may be converted to the alternative form by removing the "typedef"
part and placing the identifier after the "struct", "union", or
"enum" keyword, instead of at the end. For example, here are the two
ways to define the type "bool":

```
        typedef enum {     /* using typedef */
           FALSE = 0,
           TRUE = 1
        } bool;

        enum bool {        /* preferred alternative */
           FALSE = 0,
           TRUE = 1
        };
```

   The reason this syntax is preferred is one does not have to wait
   until the end of a declaration to figure out the name of the new
   type.

3.18 Optional-data

   Optional-data is one kind of union that occurs so frequently that we
   give it a special syntax of its own for declaring it. It is declared
   as follows:

```
        type-name *identifier;
```

   This is equivalent to the following union:

```
        union switch (bool opted) {
        case TRUE:
           type-name element;
        case FALSE:
           void;
        } identifier;
```

   It is also equivalent to the following variable-length array
   declaration, since the boolean "opted" can be interpreted as the
   length of the array:

```
        type-name identifier<1>;
```

   Optional-data is not so interesting in itself, but it is very useful
   for describing recursive data-structures such as linked-lists and
   trees. For example, the following defines a type "stringlist" that
   encodes lists of arbitrary length strings:

```
        struct *stringlist {
           string item<>;
           stringlist next;
        };
```

It could have been equivalently declared as the following union:

```
union stringlist switch (bool opted) {
case TRUE:
   struct {
      string item<>;
      stringlist next;
   } element;
case FALSE:
   void;
};
```

   or as a variable-length array:

```
struct stringlist<1> {
   string item<>;
   stringlist next;
};
```

Both of these declarations obscure the intention of the stringlist
type, so the optional-data declaration is preferred over both of
them. The optional-data type also has a close correlation to how
recursive data structures are represented in high-level languages
such as Pascal or C by use of pointers. In fact, the syntax is the
same as that of the C language for pointers.

3.19 Areas for Future Enhancement

The XDR standard lacks representations for bit fields and bitmaps,
since the standard is based on bytes. Also missing are packed (or
binary-coded) decimals.

The intent of the XDR standard was not to describe every kind of data
that people have ever sent or will ever want to send from machine to
machine. Rather, it only describes the most commonly used data-types
of high-level languages such as Pascal or C so that applications
written in these languages will be able to communicate easily over
some medium.

One could imagine extensions to XDR that would let it describe almost
any existing protocol, such as TCP. The minimum necessary for this
are support for different block sizes and byte-orders. The XDR
discussed here could then be considered the 4-byte big-endian member
of a larger XDR family.

4. DISCUSSION

    (1) Why use a language for describing data? What's wrong with
    diagrams?

    There are many advantages in using a data-description language such
    as XDR versus using diagrams.   Languages are more formal than
    diagrams   and   lead to less ambiguous descriptions of data.
    Languages are also easier to understand and allow one to think of
    other   issues instead of the   low-level details of bit-encoding.
    Also, there is a close analogy between the types of XDR and a
    high-level language   such as C   or   Pascal.   This makes   the
    implementation of XDR encoding and decoding modules an easier task.
    Finally, the language specification itself is an ASCII string that
    can be passed from machine to machine to perform on-the-fly data
    interpretation.

    (2) Why is there only one byte-order for an XDR unit?

    Supporting two byte-orderings requires a higher level protocol for
    determining in which byte-order the data is encoded. Since XDR is
    not a protocol, this can't be done. The advantage of this, though,
    is that data in XDR format can be written to a magnetic tape, for
    example, and any machine will be able to interpret it, since no
    higher level protocol is necessary for determining the byte-order.

    (3) Why is the XDR byte-order big-endian instead of little-endian?
    Isn't this unfair to little-endian machines such as the VAX(r), which
    has to convert from one form to the other?

    Yes, it is unfair, but having only one byte-order means you have to
    be unfair to somebody. Many architectures, such as the Motorola
    68000* and IBM 370*, support the big-endian byte-order.

    (4) Why is the XDR unit four bytes wide?

    There is a tradeoff in choosing the XDR unit size. Choosing a small
    size such as two makes the encoded data small, but causes alignment
    problems for machines that aren't aligned on these boundaries. A
    large size such as eight means the data will be aligned on virtually
    every machine, but causes the encoded data to grow too big. We chose
    four as a compromise. Four is big enough to support most
    architectures efficiently, except for rare machines such as the
    eight-byte aligned Cray*. Four is also small enough to keep the
    encoded data restricted to a reasonable size.

(5) Why must variable-length data be padded with zeros?

It is desirable that the same data encode into the same thing on all
machines, so that encoded data can be meaningfully compared or
checksummed. Forcing the padded bytes to be zero ensures this.

(6) Why is there no explicit data-typing?

Data-typing has a relatively high cost for what small advantages it
may have. One cost is the expansion of data due to the inserted type
fields. Another is the added cost of interpreting these type fields
and acting accordingly. And most protocols already know what type
they expect, so data-typing supplies only redundant information.
However, one can still get the benefits of data-typing using XDR. One
way is to encode two things: first a string which is the XDR data
description of the encoded data, and then the encoded data itself.
Another way is to assign a value to all the types in XDR, and then
define a universal type which takes this value as its discriminant
and for each value, describes the corresponding data type.

5. THE XDR LANGUAGE SPECIFICATION

5.1 Notational Conventions

This specification uses an extended Back-Naur Form notation for
describing the XDR language. Here is a brief description of the
notation:

(1) The characters '|', '(', ')', '[', ']', '"', and '*' are special.
(2) Terminal symbols are strings of any characters surrounded by
double quotes.
(3) Non-terminal symbols are strings of non-special characters.
(4) Alternative items are separated by a vertical bar ("|").
(5) Optional items are enclosed in brackets.
(6) Items are grouped together by enclosing them in parentheses.
(7) A '*' following an item means 0 or more occurrences of that item.

For example, consider the following pattern:

        "a " "very" (", " "very")* [" cold " "and "] " rainy "
        ("day" | "night")

An infinite number of strings match this pattern. A few of them are:

```
        "a very rainy day"
        "a very, very rainy day"
        "a very cold and  rainy day"
        "a very, very, very cold and  rainy night"
```

5.2 Lexical Notes

   (1) Comments begin with '/*' and terminate with '*/'.
   (2) White space serves to separate items and is otherwise ignored.
   (3) An identifier is a letter followed by an optional sequence of
   letters, digits or underbar ('_'). The case of identifiers is not
   ignored.
   (4) A constant is a sequence of one or more decimal digits,
   optionally preceded by a minus-sign ('-').

5.3 Syntax Information

```
     declaration:
          type-specifier identifier
        | type-specifier identifier "[" value "]"
        | type-specifier identifier "<" [ value ] ">"
        | "opaque" identifier "[" value "]"
        | "opaque" identifier "<" [ value ] ">"
        | "string" identifier "<" [ value ] ">"
        | type-specifier "*" identifier
        | "void"

     value:
          constant
        | identifier

     type-specifier:
          [ "unsigned" ] "int"
        | [ "unsigned" ] "hyper"
        | "float"
        | "double"
        | "bool"
        | enum-type-spec
        | struct-type-spec
        | union-type-spec
        | identifier

     enum-type-spec:
        "enum" enum-body

     enum-body:
        "{"
           ( identifier "=" value )
```

```
                  ( "," identifier "=" value )*
              "}"

        struct-type-spec:
            "struct" struct-body

        struct-body:
            "{"
                ( declaration ";" )
                ( declaration ";" )*
            "}"

        union-type-spec:
            "union" union-body

        union-body:
            "switch" "(" declaration ")" "{"
                ( "case" value ":" declaration ";" )
                ( "case" value ":" declaration ";" )*
                [ "default" ":" declaration ";" ]
            "}"

        constant-def:
            "const" identifier "=" constant ";"

        type-def:
              "typedef" declaration ";"
            | "enum" identifier enum-body ";"
            | "struct" identifier struct-body ";"
            | "union" identifier union-body ";"

        definition:
              type-def
            | constant-def

        specification:
              definition *
```

5.4 Syntax Notes

   (1) The following are keywords and cannot be used as identifiers:
   "bool", "case", "const", "default", "double", "enum", "float",
   "hyper", "opaque", "string", "struct", "switch", "typedef", "union",
   "unsigned" and "void".

   (2) Only unsigned constants may be used as size specifications for
   arrays. If an identifier is used, it must have been declared
   previously as an unsigned constant in a "const" definition.

(3) Constant and type identifiers within the scope of a specification
are in the same name space and must be declared uniquely within this
scope.

(4) Similarly, variable names must be unique within the scope of
struct and union declarations. Nested struct and union declarations
create new scopes.

(5) The discriminant of a union must be of a type that evaluates to
an integer. That is, "int", "unsigned int", "bool", an enumerated
type or any typedefed type that evaluates to one of these is legal.
Also, the case values must be one of the legal values of the
discriminant. Finally, a case value may not be specified more than
once within the scope of a union declaration.

6. AN EXAMPLE OF AN XDR DATA DESCRIPTION

Here is a short XDR data description of a thing called a "file",
which might be used to transfer files from one machine to another.

```
      const MAXUSERNAME = 32;      /* max length of a user name */
      const MAXFILELEN = 65535;    /* max length of a file      */
      const MAXNAMELEN = 255;      /* max length of a file name */

      /*
       * Types of files:
       */
      enum filekind {
         TEXT = 0,       /* ascii data */
         DATA = 1,       /* raw data   */
         EXEC = 2        /* executable */
      };

      /*
       * File information, per kind of file:
       */
      union filetype switch (filekind kind) {
      case TEXT:
         void;                          /* no extra information */
      case DATA:
         string creator<MAXNAMELEN>;    /* data creator         */
      case EXEC:
         string interpretor<MAXNAMELEN>; /* program interpretor  */
      };
```

```
      /*
       * A complete file:
       */
      struct file {
         string filename<MAXNAMELEN>; /* name of file    */
         filetype type;              /* info about file */
         string owner<MAXUSERNAME>;  /* owner of file   */
         opaque data<MAXFILELEN>;    /* file data       */
      };
```

   Suppose now that there is a user named "john" who wants to store his
   lisp program "sillyprog" that contains just the data "(quit)". His
   file would be encoded as follows:

```
   OFFSET  HEX BYTES         ASCII    COMMENTS
   ------  ---------         -----    --------
     0      00 00 00 09      ....     -- length of filename = 9
     4      73 69 6c 6c      sill     -- filename characters
     8      79 70 72 6f      ypro     -- ... and more characters ...
    12      67 00 00 00      g...     -- ... and 3 zero-bytes of fill
    16      00 00 00 02      ....     -- filekind is EXEC = 2
    20      00 00 00 04      ....     -- length of interpretor = 4
    24      6c 69 73 70      lisp     -- interpretor characters
    28      00 00 00 04      ....     -- length of owner = 4
    32      6a 6f 68 6e      john     -- owner characters
    36      00 00 00 06      ....     -- length of file data = 6
    40      28 71 75 69      (qui     -- file data bytes ...
    44      74 29 00 00      t)..     -- ... and 2 zero-bytes of fill
```

7. REFERENCES

   [1] Brian W. Kernighan & Dennis M. Ritchie, "The C Programming
        Language", Bell Laboratories, Murray Hill, New Jersey, 1978.

   [2] Danny Cohen, "On Holy Wars and a Plea for Peace", IEEE Computer,
        October 1981.

   [3] "IEEE Standard for Binary Floating-Point Arithmetic", ANSI/IEEE
        Standard 754-1985, Institute of Electrical and Electronics
        Engineers, August 1985.

   [4] "Courier: The Remote Procedure Call Protocol", XEROX
        Corporation, XSIS 038112, December 1981.

8. TRADEMARKS AND OWNERS

         SUN WORKSTATION   Sun Microsystems, Inc.
         VAX   Digital Equipment Corporation
         IBM-PC   International Business Machines Corporation
         Cray   Cray Research
         NFS   Sun Microsystems, Inc.
         Ethernet   Xerox Corporation.
         Motorola 68000   Motorola, Inc.
         IBM 370   International Business Machines Corporation