Network Working Group   M. Eisler, Ed.
Request for Comments: 4506   Network Appliance, Inc.
STD: 67   May 2006
Obsoletes: 1832
Category: Standards Track

                    XDR: External Data Representation Standard

Status of This Memo

   This document specifies an Internet standards track protocol for the
   Internet community, and requests discussion and suggestions for
   improvements. Please refer to the current edition of the "Internet
   Official Protocol Standards" (STD 1) for the standardization state
   and status of this protocol. Distribution of this memo is unlimited.

Abstract

   This document describes the External Data Representation Standard
   (XDR) protocol as it is currently deployed and accepted. This
   document obsoletes RFC 1832.

```
55

56

57

58    Eisler                       Standards Track                    [Page 1]

59    

60    RFC 4506         XDR: External Data Representation Standard      May 2006

61

62

63    Table of Contents

64
```

```
103

104

105

106

107

108
```

Eisler   Standards Track   [Page 2]

RFC 4506   XDR: External Data Representation Standard   May 2006

1. Introduction

   XDR is a standard for the description and encoding of data. It is
   useful for transferring data between different computer
   architectures, and it 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 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 [KERN], just as Courier [COUR] is similar to Mesa.
   Protocols such as ONC 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 as 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 [COHE], or
   least significant bit first.

2. Changes from RFC 1832

   This document makes no technical changes to RFC 1832 and is published
   for the purposes of noting IANA considerations, augmenting security
   considerations, and distinguishing normative from informative
   references.

3. 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.

```
163
164      We include the familiar graphic box notation for illustration and
165      comparison.  In most illustrations, each box (delimited by a plus
166      sign at the 4 corners and vertical bars and dashes) depicts a byte.
167
168
169
170   Eisler                    Standards Track                   [Page 3]
171   
172   RFC 4506        XDR: External Data Representation Standard      May 2006
173
174
175      Ellipses (...) between boxes show zero or more additional bytes where
176      required.
177
178          +--------+--------+...+--------+--------+...+--------+
179          | byte 0 | byte 1 |...|byte n-1|    0   |...|    0   |   BLOCK
180          +--------+--------+...+--------+--------+...+--------+
181          |<----------n bytes---------->|<------r bytes------>|
182          |<-----------n+r (where (n+r) mod 4 = 0)----------->|
183
184   4.   XDR Data Types
185
186      Each of the sections that follow describes a data type defined in the
187      XDR standard, shows how it is declared in the language, and includes
188      a graphic illustration of its encoding.
189
190      For each data type in the language we show a general paradigm
191      declaration.  Note that angle brackets (< and >) denote variable-
192      length sequences of data and that square brackets ([ and ]) denote
193      fixed-length sequences of data.  "n", "m", and "r" denote integers.
194      For the full language specification and more formal definitions of
195      terms such as "identifier" and "declaration", refer to Section 6,
196      "The XDR Language Specification".
197
198      For some data types, more specific examples are included.  A more
199      extensive example of a data description is in Section 7, "An Example
200      of an XDR Data Description".
201
202   4.1.  Integer
203
204      An XDR signed integer is a 32-bit datum that encodes an integer in
205      the range [-2147483648,2147483647].  The integer is represented in
206      two's complement notation.  The most and least significant bytes are
207      0 and 3, respectively.  Integers are declared as follows:
208
209          int identifier;
210
211          (MSB)                        (LSB)
212          +-------+-------+-------+-------+
213          |byte 0 |byte 1 |byte 2 |byte 3 |                    INTEGER
214          +-------+-------+-------+-------+
215          <-----------32 bits------------>
216
```

```
217    4.2.  Unsigned Integer
218
219       An XDR unsigned integer is a 32-bit datum that encodes a non-negative
220       integer in the range [0,4294967295].  It is represented by an
221       unsigned binary number whose most and least significant bytes are 0
222       and 3, respectively.  An unsigned integer is declared as follows:
223
224
225
226    Eisler                       Standards Track                    [Page 4]
227    
228    RFC 4506        XDR: External Data Representation Standard       May 2006
229
230
231           unsigned int identifier;
232
233              (MSB)                       (LSB)
234               +-------+-------+-------+-------+
235               |byte 0 |byte 1 |byte 2 |byte 3 |          UNSIGNED INTEGER
236               +-------+-------+-------+-------+
237               <------------32 bits------------>
238
239    4.3.  Enumeration
240
241       Enumerations have the same representation as signed integers.
242       Enumerations are handy for describing subsets of the integers.
243       Enumerated data is declared as follows:
244
245           enum { name-identifier = constant, ... } identifier;
246
247       For example, the three colors red, yellow, and blue could be
248       described by an enumerated type:
249
250           enum { RED = 2, YELLOW = 3, BLUE = 5 } colors;
251
252       It is an error to encode as an enum any integer other than those that
253       have been given assignments in the enum declaration.
254
255    4.4.  Boolean
256
257       Booleans are important enough and occur frequently enough to warrant
258       their own explicit type in the standard.  Booleans are declared as
259       follows:
260
261           bool identifier;
262
263       This is equivalent to:
264
265           enum { FALSE = 0, TRUE = 1 } identifier;
266
267    4.5.  Hyper Integer and Unsigned Hyper Integer
268
269       The standard also defines 64-bit (8-byte) numbers called hyper
270       integers and unsigned hyper integers.  Their representations are the
```

```
271      obvious extensions of integer and unsigned integer defined above.
272      They are represented in two's complement notation.  The most and
273      least significant bytes are 0 and 7, respectively.  Their
274      declarations:
275
276      hyper identifier; unsigned hyper identifier;
277
278
279
280
281
282   Eisler                        Standards Track                   [Page 5]
283   ⬛⬛
284   RFC 4506        XDR: External Data Representation Standard       May 2006
285
286
287           (MSB)                                               (LSB)
288        +-------+-------+-------+-------+-------+-------+-------+-------+
289        |byte 0 |byte 1 |byte 2 |byte 3 |byte 4 |byte 5 |byte 6 |byte 7 |
290        +-------+-------+-------+-------+-------+-------+-------+-------+
291        <---------------------------64 bits--------------------------->
292                                                      HYPER INTEGER
293                                                      UNSIGNED HYPER INTEGER
294
295   4.6.  Floating-Point
296
297      The standard defines the floating-point data type "float" (32 bits or
298      4 bytes).  The encoding used is the IEEE standard for normalized
299      single-precision floating-point numbers [IEEE].  The following three
300      fields describe the single-precision floating-point number:
301
302         S: The sign of the number.  Values 0 and 1 represent positive and
303            negative, respectively.  One bit.
304
305         E: The exponent of the number, base 2.  8 bits are devoted to this
306            field.  The exponent is biased by 127.
307
308         F: The fractional part of the number's mantissa, base 2.  23 bits
309            are devoted to this field.
310
311      Therefore, the floating-point number is described by:
312
313            (-1)**S * 2**(E-Bias) * 1.F
314
315      It is declared as follows:
316
317            float identifier;
318
319            +-------+-------+-------+-------+
320            |byte 0 |byte 1 |byte 2 |byte 3 |              SINGLE-PRECISION
321            S|   E   |           F          |         FLOATING-POINT NUMBER
322            +-------+-------+-------+-------+
323            1|<- 8 ->|<-------23 bits------>|
324            <-----------32 bits----------->
```
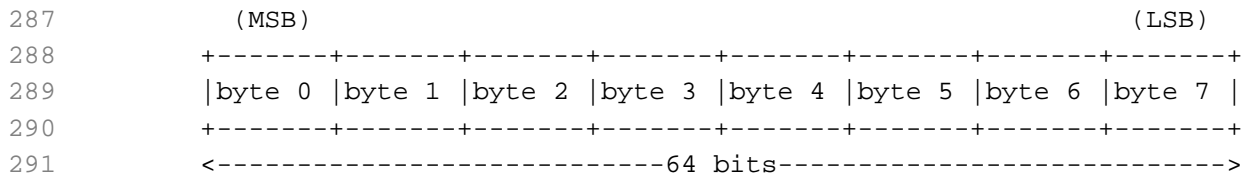
```
325
326        Just as the most and least significant bytes of a number are 0 and 3,
327        the most and least significant bits of a single-precision floating-
328        point number are 0 and 31.  The beginning bit (and most significant
329        bit) offsets of S, E, and F are 0, 1, and 9, respectively.  Note that
330        these numbers refer to the mathematical positions of the bits, and
331        NOT to their actual physical locations (which vary from medium to
332        medium).
333
334
335
336
337
338    Eisler                       Standards Track                    [Page 6]
339    ▮▮
340    RFC 4506        XDR: External Data Representation Standard      May 2006
341
342
343        The IEEE specifications should be consulted concerning the encoding
344        for signed zero, signed infinity (overflow), and denormalized numbers
345        (underflow) [IEEE].  According to IEEE specifications, the "NaN" (not
346        a number) is system dependent and should not be interpreted within
347        XDR as anything other than "NaN".
348
349    4.7.  Double-Precision Floating-Point
350
351        The standard defines the encoding for the double-precision floating-
352        point data type "double" (64 bits or 8 bytes).  The encoding used is
353        the IEEE standard for normalized double-precision floating-point
354        numbers [IEEE].  The standard encodes the following three fields,
355        which describe the double-precision floating-point number:
356
357           S: The sign of the number.  Values 0 and 1 represent positive and
358                 negative, respectively.  One bit.
359
360           E: The exponent of the number, base 2.  11 bits are devoted to
361                 this field.  The exponent is biased by 1023.
362
363           F: The fractional part of the number's mantissa, base 2.  52 bits
364                 are devoted to this field.
365
366        Therefore, the floating-point number is described by:
367
368            (-1)**S * 2**(E-Bias) * 1.F
369
370        It is declared as follows:
371
372            double identifier;
373
374            +------+------+------+------+------+------+------+------+
375            |byte 0|byte 1|byte 2|byte 3|byte 4|byte 5|byte 6|byte 7|
376            S|    E    |                    F                       |
377            +------+------+------+------+------+------+------+------+
378            1|<--11-->|<----------------52 bits------------------->|
```
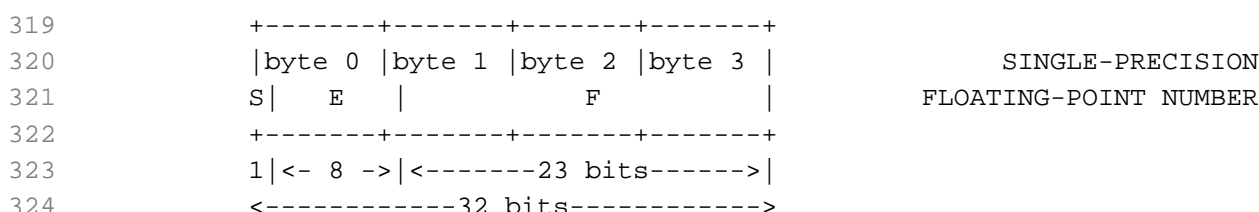
```
379              <----------------------64 bits------------------------->
380                                        DOUBLE-PRECISION FLOATING-POINT
381
382       Just as the most and least significant bytes of a number are 0 and 3,
383       the most and least significant bits of a double-precision floating-
384       point number are 0 and 63.  The beginning bit (and most significant
385       bit) offsets of S, E, and F are 0, 1, and 12, respectively.  Note
386       that these numbers refer to the mathematical positions of the bits,
387       and NOT to their actual physical locations (which vary from medium to
388       medium).
389
390
391
392
393
394    Eisler                          Standards Track                    [Page 7]
395    FF
396    RFC 4506        XDR: External Data Representation Standard        May 2006
397
398
399       The IEEE specifications should be consulted concerning the encoding
400       for signed zero, signed infinity (overflow), and denormalized numbers
401       (underflow) [IEEE].  According to IEEE specifications, the "NaN" (not
402       a number) is system dependent and should not be interpreted within
403       XDR as anything other than "NaN".
404
405    4.8.  Quadruple-Precision Floating-Point
406
407       The standard defines the encoding for the quadruple-precision
408       floating-point data type "quadruple" (128 bits or 16 bytes).  The
409       encoding used is designed to be a simple analog of the encoding used
410       for single- and double-precision floating-point numbers using one
411       form of IEEE double extended precision.  The standard encodes the
412       following three fields, which describe the quadruple-precision
413       floating-point number:
414
415         S: The sign of the number.  Values 0 and 1 represent positive and
416            negative, respectively.  One bit.
417
418         E: The exponent of the number, base 2.  15 bits are devoted to
419            this field.  The exponent is biased by 16383.
420
421         F: The fractional part of the number's mantissa, base 2.  112 bits
422            are devoted to this field.
423
424       Therefore, the floating-point number is described by:
425
426            (-1)**S * 2**(E-Bias) * 1.F
427
428       It is declared as follows:
429
430            quadruple identifier;
431
432            +------+------+------+------+------+------+-...--+------+
```
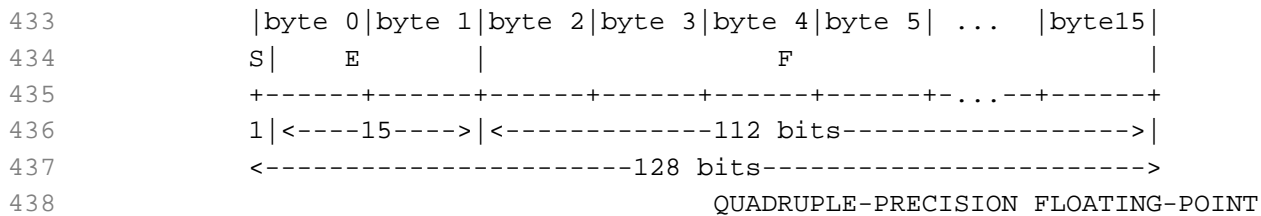
```
    |byte 0|byte 1|byte 2|byte 3|byte 4|byte 5|  ...   |byte15|
    S|    E       |                   F                        |
    +------+------+------+------+------+------+-...-+------+
    1|<----15---->|<-------------112 bits------------------>|
    <----------------------128 bits----------------------->
                                  QUADRUPLE-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 quadruple-precision
floating-point number are 0 and 127. The beginning bit (and most
significant bit) offsets of S, E , and F are 0, 1, and 16,
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).

Eisler   Standards Track   [Page 8]

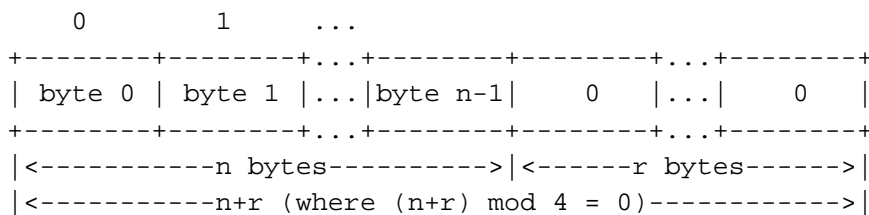RFC 4506   XDR: External Data Representation Standard   May 2006

The encoding for signed zero, signed infinity (overflow), and
denormalized numbers are analogs of the corresponding encodings for
single and double-precision floating-point numbers [SPAR], [HPRE].
The "NaN" encoding as it applies to quadruple-precision floating-
point numbers is system dependent and should not be interpreted
within XDR as anything other than "NaN".

4.9. 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
```

4.10. 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

```
487    below), and followed by the n bytes of the sequence.
488
489    Byte m of the sequence always precedes byte m+1 of the sequence, and
490    byte 0 of the sequence always follows the sequence's length (count).
491    If n is not a multiple of four, then the n bytes are followed by
492    enough (0 to 3) residual zero bytes, r, to make the total byte count
493    a multiple of four.  Variable-length opaque data is declared in the
494    following way:
495
496         opaque identifier<m>;
497      or
498         opaque identifier<>;
499
500    The constant m denotes an upper bound of the number of bytes that the
501    sequence may contain.  If m is not specified, as in the second
502    declaration, it is assumed to be (2**32) - 1, the maximum length.
503
504
505
506    Eisler                      Standards Track                   [Page 9]
507    
508    RFC 4506      XDR: External Data Representation Standard        May 2006
509
510
511    The constant m would normally be found in a protocol specification.
512    For example, a filing protocol may state that the maximum data
513    transfer size is 8192 bytes, as follows:
514
515         opaque filedata<8192>;
516
517          0     1     2     3     4     5   ...
518        +-----+-----+-----+-----+-----+-----+...+-----+-----+...+-----+
519        |       length n      |byte0|byte1|...| n-1 |  0  |...|  0  |
520        +-----+-----+-----+-----+-----+-----+...+-----+-----+...+-----+
521        |<-------4 bytes------->|<------n bytes------>|<---r bytes--->|
522                                |<----n+r (where (n+r) mod 4 = 0)---->|
523                                                        VARIABLE-LENGTH OPAQUE
524
525    It is an error to encode a length greater than the maximum described
526    in the specification.
527
528  4.11.  String
529
530    The standard defines a string of n (numbered 0 through n-1) ASCII
531    bytes to be the number n encoded as an unsigned integer (as described
532    above), and followed by the n bytes of the string.  Byte m of the
533    string always precedes byte m+1 of the string, and byte 0 of the
534    string always follows the string's length.  If n is not a multiple of
535    four, then the n bytes are followed by enough (0 to 3) residual zero
536    bytes, r, to make the total byte count a multiple of four.  Counted
537    byte strings are declared as follows:
538
539         string object<m>;
540      or
```

```
541            string object<>;

542

543        The constant m denotes an upper bound of the number of bytes that a

544        string may contain.  If m is not specified, as in the second

545        declaration, it is assumed to be (2**32) - 1, the maximum length.

546        The constant m would normally be found in a protocol specification.

547        For example, a filing protocol may state that a file name can be no

548        longer than 255 bytes, as follows:

549

550            string filename<255>;

551

552              0     1     2     3     4     5    ...

553        +-----+-----+-----+-----+-----+-----+...+-----+-----+...+-----+

554        |          length n      |byte0|byte1|...| n-1 |  0  |...|  0  |

555        +-----+-----+-----+-----+-----+-----+...+-----+-----+...+-----+

556        |<-------4 bytes------->|<------n bytes------>|<---r bytes--->|

557                                |<----n+r (where (n+r) mod 4 = 0)---->|

558                                                                STRING

559

560

561

562    Eisler                    Standards Track                 [Page 10]
```

```
565

566

567        It is an error to encode a length greater than the maximum described

568        in the specification.

569

570    4.12.  Fixed-Length Array

571

572        Declarations for fixed-length arrays of homogeneous elements are in

573        the following form:

574

575            type-name identifier[n];

576

577        Fixed-length arrays of elements numbered 0 through n-1 are encoded by

578        individually encoding the elements of the array in their natural

579        order, 0 through n-1.  Each element's size is a multiple of four

580        bytes.  Though all elements are of the same type, the elements may

581        have different sizes.  For example, in a fixed-length array of

582        strings, all elements are of type "string", yet each element will

583        vary in its length.

584

585            +---+---+---+---+---+---+---+---+...+---+---+---+---+

586            |   element 0   |   element 1   |...|  element n-1  |

587            +---+---+---+---+---+---+---+---+...+---+---+---+---+

588            |<--------------------n elements-------------------->|

589

590                                                FIXED-LENGTH ARRAY

591

592    4.13.  Variable-Length Array

593

594        Counted arrays provide the ability to encode variable-length arrays
```

```
595      of homogeneous elements.  The array is encoded as the element count n
596      (an unsigned integer) followed by the encoding of each of the array's
597      elements, starting with element 0 and progressing through element
598      n-1.  The declaration for variable-length arrays follows this form:
599
600           type-name identifier<m>;
601         or
602           type-name identifier<>;
603
604      The constant m specifies the maximum acceptable element count of an
605      array; if m is not specified, as in the second declaration, it is
606      assumed to be (2**32) - 1.
607
608             0  1  2  3
609         +--+--+--+--+--+--+--+--+--+--+--+--+...+--+--+--+--+
610         |     n     | element 0 | element 1 |...|element n-1|
611         +--+--+--+--+--+--+--+--+--+--+--+--+...+--+--+--+--+
612         |<-4 bytes->|<--------------n elements------------->|
613                                                     COUNTED ARRAY
614
615
616
617
618  Eisler                      Standards Track                    [Page 11]

620  RFC 4506        XDR: External Data Representation Standard      May 2006
621
622
623      It is an error to encode a value of n that is greater than the
624      maximum described in the specification.
625
626  4.14.   Structure
627
628      Structures are declared as follows:
629
630           struct {
631               component-declaration-A;
632               component-declaration-B;
633               ...
634           } identifier;
635
636      The components of the structure are encoded in the order of their
637      declaration in the structure.  Each component's size is a multiple of
638      four bytes, though the components may be different sizes.
639
640           +-------------+-------------+...
641           | component A | component B |...                    STRUCTURE
642           +-------------+-------------+...
643
644  4.15.   Discriminated Union
645
646      A discriminated union is a type composed of a discriminant followed
647      by a type selected from a set of prearranged types according to the
648      value of the discriminant.  The type of discriminant is either "int",
```

```
649       "unsigned int", or an enumerated type, such as "bool".  The component
650       types are called "arms" of the union and are preceded by the value of
651       the discriminant that implies their encoding.  Discriminated unions
652       are declared as follows:
653
654           union switch (discriminant-declaration) {
655           case discriminant-value-A:
656              arm-declaration-A;
657           case discriminant-value-B:
658              arm-declaration-B;
659           ...
660           default: default-declaration;
661           } identifier;
662
663       Each "case" keyword is followed by a legal value of the discriminant.
664       The default arm is optional.  If it is not specified, then a valid
665       encoding of the union cannot take on unspecified discriminant values.
666       The size of the implied arm is always a multiple of four bytes.
667
668       The discriminated union is encoded as its discriminant followed by
669       the encoding of the implied arm.
670
671
672
673
674    Eisler                      Standards Track                  [Page 12]
675    [FF]
676    RFC 4506       XDR: External Data Representation Standard     May 2006
677
678
679           0   1   2   3
680        +---+---+---+---+---+---+---+---+
681        | discriminant | implied arm |        DISCRIMINATED UNION
682        +---+---+---+---+---+---+---+---+
683        |<---4 bytes--->|
684
685    4.16.  Void
686
687       An XDR void is a 0-byte quantity.  Voids are useful for describing
688       operations that take no data as input or no data as output.  They are
689       also useful in unions, where some arms may contain data and others do
690       not.  The declaration is simply as follows:
691
692           void;
693
694       Voids are illustrated as follows:
695
696           ++
697           ||                                                  VOID
698           ++
699        --><-- 0 bytes
700
701    4.17.  Constant
702
```

```
703        The data declaration for a constant follows this form:
704
705            const name-identifier = n;
706
707        "const" is used to define a symbolic name for a constant; it does not
708        declare any data.  The symbolic constant may be used anywhere a
709        regular constant may be used.  For example, the following defines a
710        symbolic constant DOZEN, equal to 12.
711
712            const DOZEN = 12;
713
714   4.18.  Typedef
715
716        "typedef" does not declare any data either, but serves to define new
717        identifiers for declaring data.  The syntax is:
718
719            typedef declaration;
720
721        The new type name is actually the variable name in the declaration
722        part of the typedef.  For example, the following defines a new type
723        called "eggbox" using an existing type called "egg":
724
725            typedef egg eggbox[DOZEN];
726
727
728
729
730   Eisler                      Standards Track                   [Page 13]
731   FF
732   RFC 4506       XDR: External Data Representation Standard       May 2006
733
734
735        Variables declared using the new type name have the same type as the
736        new type name would have in the typedef, if it were considered a
737        variable.  For example, the following two declarations are equivalent
738        in declaring the variable "fresheggs":
739
740            eggbox  fresheggs; egg     fresheggs[DOZEN];
741
742        When a typedef involves a struct, enum, or union definition, there is
743        another (preferred) syntax that may be used to define the same type.
744        In general, a typedef of the following form:
745
746            typedef <<struct, union, or enum definition>> identifier;
747
748        may be converted to the alternative form by removing the "typedef"
749        part and placing the identifier after the "struct", "union", or
750        "enum" keyword, instead of at the end.  For example, here are the two
751        ways to define the type "bool":
752
753            typedef enum {    /* using typedef */
754                FALSE = 0,
755                TRUE = 1
756            } bool;
```

```
757
758              enum bool {        /* preferred alternative */
759                  FALSE = 0,
760                  TRUE = 1
761              };
762
763       This syntax is preferred because one does not have to wait until the
764       end of a declaration to figure out the name of the new type.
765
766    4.19.  Optional-Data
767
768       Optional-data is one kind of union that occurs so frequently that we
769       give it a special syntax of its own for declaring it.  It is declared
770       as follows:
771
772              type-name *identifier;
773
774       This is equivalent to the following union:
775
776              union switch (bool opted) {
777              case TRUE:
778                  type-name element;
779              case FALSE:
780                  void;
781              } identifier;
782
783
784
785
786    Eisler                        Standards Track                   [Page 14]
787    
788    RFC 4506        XDR: External Data Representation Standard       May 2006
789
790
791       It is also equivalent to the following variable-length array
792       declaration, since the boolean "opted" can be interpreted as the
793       length of the array:
794
795              type-name identifier<1>;
796
797       Optional-data is not so interesting in itself, but it is very useful
798       for describing recursive data-structures such as linked-lists and
799       trees.  For example, the following defines a type "stringlist" that
800       encodes lists of zero or more arbitrary length strings:
801
802              struct stringentry {
803                  string item<>;
804                  stringentry *next;
805              };
806
807              typedef stringentry *stringlist;
808
809       It could have been equivalently declared as the following union:
810
```

```
811              union stringlist switch (bool opted) {
812                 case TRUE:
813                    struct {
814                       string item<>;
815                       stringlist next;
816                    } element;
817                 case FALSE:
818                    void;
819                 };
820
821       or as a variable-length array:
822
823          struct stringentry {
824             string item<>;
825             stringentry next<1>;
826          };
827
828          typedef stringentry stringlist<1>;
829
830       Both of these declarations obscure the intention of the stringlist
831       type, so the optional-data declaration is preferred over both of
832       them.  The optional-data type also has a close correlation to how
833       recursive data structures are represented in high-level languages
834       such as Pascal or C by use of pointers.  In fact, the syntax is the
835       same as that of the C language for pointers.
836
837
838
839
840
841
842    Eisler                       Standards Track                    [Page 15]
843    ▣▣
844    RFC 4506        XDR: External Data Representation Standard       May 2006
845
846
847    4.20.  Areas for Future Enhancement
848
849       The XDR standard lacks representations for bit fields and bitmaps,
850       since the standard is based on bytes.  Also missing are packed (or
851       binary-coded) decimals.
852
853       The intent of the XDR standard was not to describe every kind of data
854       that people have ever sent or will ever want to send from machine to
855       machine.  Rather, it only describes the most commonly used data-types
856       of high-level languages such as Pascal or C so that applications
857       written in these languages will be able to communicate easily over
858       some medium.
859
860       One could imagine extensions to XDR that would let it describe almost
861       any existing protocol, such as TCP.  The minimum necessary for this
862       is support for different block sizes and byte-orders.  The XDR
863       discussed here could then be considered the 4-byte big-endian member
864       of a larger XDR family.
```

```
865
866   5.  Discussion
867
868      (1) Why use a language for describing data?  What's wrong with
869          diagrams?
870
871      There are many advantages in using a data-description language such
872      as XDR versus using diagrams.  Languages are more formal than
873      diagrams and lead to less ambiguous descriptions of data.  Languages
874      are also easier to understand and allow one to think of other issues
875      instead of the low-level details of bit encoding.  Also, there is a
876      close analogy between the types of XDR and a high-level language such
877      as C or Pascal.  This makes the implementation of XDR encoding and
878      decoding modules an easier task.  Finally, the language specification
879      itself is an ASCII string that can be passed from machine to machine
880      to perform on-the-fly data interpretation.
881
882      (2) Why is there only one byte-order for an XDR unit?
883
884      Supporting two byte-orderings requires a higher-level protocol for
885      determining in which byte-order the data is encoded.  Since XDR is
886      not a protocol, this can't be done.  The advantage of this, though,
887      is that data in XDR format can be written to a magnetic tape, for
888      example, and any machine will be able to interpret it, since no
889      higher-level protocol is necessary for determining the byte-order.
890
891      (3) Why is the XDR byte-order big-endian instead of little-endian?
892          Isn't this unfair to little-endian machines such as the VAX(r),
893          which has to convert from one form to the other?
894
895
896
897
898   Eisler                      Standards Track                    [Page 16]
899   FF
900   RFC 4506        XDR: External Data Representation Standard       May 2006
901
902
903      Yes, it is unfair, but having only one byte-order means you have to
904      be unfair to somebody.  Many architectures, such as the Motorola
905      68000* and IBM 370*, support the big-endian byte-order.
906
907      (4) Why is the XDR unit four bytes wide?
908
909      There is a tradeoff in choosing the XDR unit size.  Choosing a small
910      size, such as two, makes the encoded data small, but causes alignment
911      problems for machines that aren't aligned on these boundaries.  A
912      large size, such as eight, means the data will be aligned on
913      virtually every machine, but causes the encoded data to grow too big.
914      We chose four as a compromise.  Four is big enough to support most
915      architectures efficiently, except for rare machines such as the
916      eight-byte-aligned Cray*.  Four is also small enough to keep the
917      encoded data restricted to a reasonable size.
918
```

```
919        (5) Why must variable-length data be padded with zeros?
920
921        It is desirable that the same data encode into the same thing on all
922        machines, so that encoded data can be meaningfully compared or
923        checksummed.  Forcing the padded bytes to be zero ensures this.
924
925        (6) Why is there no explicit data-typing?
926
927        Data-typing has a relatively high cost for what small advantages it
928        may have.  One cost is the expansion of data due to the inserted type
929        fields.  Another is the added cost of interpreting these type fields
930        and acting accordingly.  And most protocols already know what type
931        they expect, so data-typing supplies only redundant information.
932        However, one can still get the benefits of data-typing using XDR.
933        One way is to encode two things: first, a string that is the XDR data
934        description of the encoded data, and then the encoded data itself.
935        Another way is to assign a value to all the types in XDR, and then
936        define a universal type that takes this value as its discriminant and
937        for each value, describes the corresponding data type.
938
939   6.  The XDR Language Specification
940
941   6.1.  Notational Conventions
942
943        This specification uses an extended Back-Naur Form notation for
944        describing the XDR language.  Here is a brief description of the
945        notation:
946
947        (1) The characters '|', '(', ')', '[', ']', '"', and '*' are special.
948        (2) Terminal symbols are strings of any characters surrounded by
949        double quotes.  (3) Non-terminal symbols are strings of non-special
950        characters.  (4) Alternative items are separated by a vertical bar
951
952
953
954   Eisler                        Standards Track                    [Page 17]
955   FF
956   RFC 4506        XDR: External Data Representation Standard       May 2006
957
958
959        ("|").  (5) Optional items are enclosed in brackets.  (6) Items are
960        grouped together by enclosing them in parentheses.  (7) A '*'
961        following an item means 0 or more occurrences of that item.
962
963        For example, consider the following pattern:
964
965            "a " "very" (", " "very")* [" cold " "and "]  " rainy "
966            ("day" | "night")
967
968        An infinite number of strings match this pattern.  A few of them are:
969
970            "a very rainy day"
971            "a very, very rainy day"
972            "a very cold and  rainy day"
```

```
 973                    "a very, very, very cold and  rainy night"
 974
 975    6.2.  Lexical Notes
 976
 977       (1) Comments begin with '/*' and terminate with '*/'.  (2) White
 978       space serves to separate items and is otherwise ignored.  (3) An
 979       identifier is a letter followed by an optional sequence of letters,
 980       digits, or underbar ('_').  The case of identifiers is not ignored.
 981       (4) A decimal constant expresses a number in base 10 and is a
 982       sequence of one or more decimal digits, where the first digit is not
 983       a zero, and is optionally preceded by a minus-sign ('-').  (5) A
 984       hexadecimal constant expresses a number in base 16, and must be
 985       preceded by '0x', followed by one or hexadecimal digits ('A', 'B',
 986       'C', 'D', E', 'F', 'a', 'b', 'c', 'd', 'e', 'f', '0', '1', '2', '3',
 987       '4', '5', '6', '7', '8', '9').  (6) An octal constant expresses a
 988       number in base 8, always leads with digit 0, and is a sequence of one
 989       or more octal digits ('0', '1', '2', '3', '4', '5', '6', '7').
 990
 991    6.3.  Syntax Information
 992
 993        declaration:
 994             type-specifier identifier
 995           | type-specifier identifier "[" value "]"
 996           | type-specifier identifier "<" [ value ] ">"
 997           | "opaque" identifier "[" value "]"
 998           | "opaque" identifier "<" [ value ] ">"
 999           | "string" identifier "<" [ value ] ">"
1000           | type-specifier "*" identifier
1001           | "void"
1002
1003        value:
1004             constant
1005           | identifier
1006
1007
1008
1009
1010    Eisler                      Standards Track                 [Page 18]
1011    
1012    RFC 4506        XDR: External Data Representation Standard      May 2006
1013
1014
1015        constant:
1016           decimal-constant | hexadecimal-constant | octal-constant
1017
1018        type-specifier:
1019             [ "unsigned" ] "int"
1020           | [ "unsigned" ] "hyper"
1021           | "float"
1022           | "double"
1023           | "quadruple"
1024           | "bool"
1025           | enum-type-spec
1026           | struct-type-spec
```

```
1027                | union-type-spec
1028                | identifier
1029
1030          enum-type-spec:
1031             "enum" enum-body
1032
1033          enum-body:
1034             "{"
1035                ( identifier "=" value )
1036                ( "," identifier "=" value )*
1037             "}"
1038
1039          struct-type-spec:
1040             "struct" struct-body
1041
1042          struct-body:
1043             "{"
1044                ( declaration ";" )
1045                ( declaration ";" )*
1046             "}"
1047
1048          union-type-spec:
1049             "union" union-body
1050
1051          union-body:
1052             "switch" "(" declaration ")" "{"
1053                case-spec
1054                case-spec *
1055                [ "default" ":" declaration ";" ]
1056             "}"
1057
1058          case-spec:
1059              ( "case" value ":")
1060              ( "case" value ":") *
1061             declaration ";"
1062
1063
1064
1065
```

```
1066    Eisler                        Standards Track                     [Page 19]
1067    ⬛⬛
1068    RFC 4506          XDR: External Data Representation Standard          May 2006
1069
1070
1071          constant-def:
1072             "const" identifier "=" constant ";"
1073
1074          type-def:
1075              "typedef" declaration ";"
1076            | "enum" identifier enum-body ";"
1077            | "struct" identifier struct-body ";"
1078            | "union" identifier union-body ";"
1079
1080          definition:
```

```
1081            type-def
1082          | constant-def
1083
1084        specification:
1085            definition *
1086
1087   6.4.  Syntax Notes
1088
1089       (1) The following are keywords and cannot be used as identifiers:
1090       "bool", "case", "const", "default", "double", "quadruple", "enum",
1091       "float", "hyper", "int", "opaque", "string", "struct", "switch",
1092       "typedef", "union", "unsigned", and "void".
1093
1094       (2) Only unsigned constants may be used as size specifications for
1095       arrays.  If an identifier is used, it must have been declared
1096       previously as an unsigned constant in a "const" definition.
1097
1098       (3) Constant and type identifiers within the scope of a specification
1099       are in the same name space and must be declared uniquely within this
1100       scope.
1101
1102       (4) Similarly, variable names must be unique within the scope of
1103       struct and union declarations.  Nested struct and union declarations
1104       create new scopes.
1105
1106       (5) The discriminant of a union must be of a type that evaluates to
1107       an integer.  That is, "int", "unsigned int", "bool", an enumerated
1108       type, or any typedefed type that evaluates to one of these is legal.
1109       Also, the case values must be one of the legal values of the
1110       discriminant.  Finally, a case value may not be specified more than
1111       once within the scope of a union declaration.
1112
1113
1114
1115
1116
1117
1118
1119
1120
1121
1122   Eisler                      Standards Track                    [Page 20]
1123
1124   RFC 4506        XDR: External Data Representation Standard      May 2006
1125
1126
1127   7.  An Example of an XDR Data Description
1128
1129       Here is a short XDR data description of a thing called a "file",
1130       which might be used to transfer files from one machine to another.
1131
1132            const MAXUSERNAME = 32;     /* max length of a user name */
1133            const MAXFILELEN = 65535;   /* max length of a file      */
1134            const MAXNAMELEN = 255;     /* max length of a file name */
```

```
1135
1136            /*
1137             * Types of files:
1138             */
1139            enum filekind {
1140               TEXT = 0,        /* ascii data */
1141               DATA = 1,        /* raw data   */
1142               EXEC = 2         /* executable */
1143            };
1144
1145            /*
1146             * File information, per kind of file:
1147             */
1148            union filetype switch (filekind kind) {
1149            case TEXT:
1150               void;                            /* no extra information */
1151            case DATA:
1152               string creator<MAXNAMELEN>;      /* data creator         */
1153            case EXEC:
1154               string interpretor<MAXNAMELEN>; /* program interpretor  */
1155            };
1156
1157            /*
1158             * A complete file:
1159             */
1160            struct file {
1161               string filename<MAXNAMELEN>; /* name of file    */
1162               filetype type;               /* info about file */
1163               string owner<MAXUSERNAME>;   /* owner of file   */
1164               opaque data<MAXFILELEN>;     /* file data       */
1165            };
```

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:

Eisler   Standards Track   [Page 21]

RFC 4506   XDR: External Data Representation Standard   May 2006

        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
```

```
1189          16        00 00 00 02      ....      -- filekind is EXEC = 2
1190          20        00 00 00 04      ....      -- length of interpretor = 4
1191          24        6c 69 73 70      lisp      -- interpretor characters
1192          28        00 00 00 04      ....      -- length of owner = 4
1193          32        6a 6f 68 6e      john      -- owner characters
1194          36        00 00 00 06      ....      -- length of file data = 6
1195          40        28 71 75 69      (qui      -- file data bytes ...
1196          44        74 29 00 00      t)..      -- ... and 2 zero-bytes of fill
```

1197

1198   8.   Security Considerations

1199

1200   XDR is a data description language, not a protocol, and hence it does
1201   not inherently give rise to any particular security considerations.
1202   Protocols that carry XDR-formatted data, such as NFSv4, are
1203   responsible for providing any necessary security services to secure
1204   the data they transport.

1205

1206   Care must be take to properly encode and decode data to avoid
1207   attacks. Known and avoidable risks include:

1208

1209   *   Buffer overflow attacks. Where feasible, protocols should be
1210   defined with explicit limits (via the "<" [value] ">" notation
1211   instead of "<" ">") on elements with variable-length data types.
1212   Regardless of the feasibility of an explicit limit on the
1213   variable length of an element of a given protocol, decoders need
1214   to ensure the incoming size does not exceed the length of any
1215   provisioned receiver buffers.

1216

1217   *   Nul octets embedded in an encoded value of type string. If the
1218   decoder's native string format uses nul-terminated strings, then
1219   the apparent size of the decoded object will be less than the
1220   amount of memory allocated for the string. Some memory
1221   deallocation interfaces take a size argument. The caller of the
1222   deallocation interface would likely determine the size of the
1223   string by counting to the location of the nul octet and adding
1224   one. This discrepancy can cause memory leakage (because less
1225   memory is actually returned to the free pool than allocated),
1226   leading to system failure and a denial of service attack.

1227

1228   *   Decoding of characters in strings that are legal ASCII
1229   characters but nonetheless are illegal for the intended
1230   application. For example, some operating systems treat the '/'

1231
1232
1233

1234   Eisler   Standards Track   [Page 22]
1235   FF
1236   RFC 4506   XDR: External Data Representation Standard   May 2006
1237
1238
1239   character as a component separator in path names. For a
1240   protocol that encodes a string in the argument to a file
1241   creation operation, the decoder needs to ensure that '/' is not
1242   inside the component name. Otherwise, a file with an illegal

```
1243            '/' in its name will be created, making it difficult to remove,
1244            and is therefore a denial of service attack.
1245
1246     *      Denial of service caused by recursive decoder or encoder
1247            subroutines.  A recursive decoder or encoder might process data
1248            that has a structured type with a member of type optional data
1249            that directly or indirectly refers to the structured type (i.e.,
1250            a linked list).  For example,
1251
1252                struct m {
1253                  int x;
1254                  struct m *next;
1255                };
1256
1257            An encoder or decoder subroutine might be written to recursively
1258            call itself each time another element of type "struct m" is
1259            found.  An attacker could construct a long linked list of
1260            "struct m" elements in the request or response, which then
1261            causes a stack overflow on the decoder or encoder.  Decoders and
1262            encoders should be written non-recursively or impose a limit on
1263            list length.
1264
1265  9.  IANA Considerations
1266
1267     It is possible, if not likely, that new data types will be added to
1268     XDR in the future.  The process for adding new types is via a
1269     standards track RFC and not registration of new types with IANA.
1270     Standards track RFCs that update or replace this document should be
1271     documented as such in the RFC Editor's database of RFCs.
1272
1273  10.  Trademarks and Owners
1274
1275     SUN WORKSTATION  Sun Microsystems, Inc.
1276     VAX             Hewlett-Packard Company
1277     IBM-PC          International Business Machines Corporation
1278     Cray            Cray Inc.
1279     NFS             Sun Microsystems, Inc.
1280     Ethernet        Xerox Corporation.
1281     Motorola 68000  Motorola, Inc.
1282     IBM 370         International Business Machines Corporation
1283
1284
1285
1286
1287
1288
1289
1290  Eisler                      Standards Track                   [Page 23]
1291  [FF]
1292  RFC 4506       XDR: External Data Representation Standard      May 2006
1293
1294
1295  11.  ANSI/IEEE Standard 754-1985
1296
```

```
1297      The definition of NaNs, signed zero and infinity, and denormalized
1298      numbers from [IEEE] is reproduced here for convenience.  The
1299      definitions for quadruple-precision floating point numbers are
1300      analogs of those for single and double-precision floating point
1301      numbers and are defined in [IEEE].
1302
1303      In the following, 'S' stands for the sign bit, 'E' for the exponent,
1304      and 'F' for the fractional part.  The symbol 'u' stands for an
1305      undefined bit (0 or 1).
1306
1307      For single-precision floating point numbers:
1308
1309       Type                S (1 bit)   E (8 bits)   F (23 bits)
1310       ----                ---------   ----------   -----------
1311       signalling NaN      u           255 (max)    .0uuuuu---u
1312                                                    (with at least
1313                                                     one 1 bit)
1314       quiet NaN           u           255 (max)    .1uuuuu---u
1315
1316       negative infinity   1           255 (max)    .000000---0
1317
1318       positive infinity   0           255 (max)    .000000---0
1319
1320       negative zero       1           0            .000000---0
1321
1322       positive zero       0           0            .000000---0
1323
1324      For double-precision floating point numbers:
1325
1326       Type                S (1 bit)   E (11 bits)  F (52 bits)
1327       ----                ---------   -----------  -----------
1328       signalling NaN      u           2047 (max)   .0uuuuu---u
1329                                                    (with at least
1330                                                     one 1 bit)
1331       quiet NaN           u           2047 (max)   .1uuuuu---u
1332
1333       negative infinity   1           2047 (max)   .000000---0
1334
1335       positive infinity   0           2047 (max)   .000000---0
1336
1337       negative zero       1           0            .000000---0
1338
1339       positive zero       0           0            .000000---0
1340
1341
1342
1343
1344
1345
1346   Eisler                      Standards Track                  [Page 24]
1347   ▉▉
1348   RFC 4506        XDR: External Data Representation Standard       May 2006
1349
1350
```

```
1351      For quadruple-precision floating point numbers:
1352
1353       Type                  S (1 bit)   E (15 bits)   F (112 bits)
1354       ----                  ---------   -----------   ------------
1355       signalling NaN        u           32767 (max)   .0uuuuu---u
1356                                                        (with at least
1357                                                         one 1 bit)
1358       quiet NaN             u           32767 (max)   .1uuuuu---u
1359
1360       negative infinity     1           32767 (max)   .000000---0
1361
1362       positive infinity     0           32767 (max)   .000000---0
1363
1364       negative zero         1           0             .000000---0
1365
1366       positive zero         0           0             .000000---0
1367
1368      Subnormal numbers are represented as follows:
1369
1370       Precision            Exponent        Value
1371       ---------            --------        -----
1372       Single               0               (-1)**S * 2**(-126) * 0.F
1373
1374       Double               0               (-1)**S * 2**(-1022) * 0.F
1375
1376       Quadruple            0               (-1)**S * 2**(-16382) * 0.F
1377
1378   12.  Normative References
1379
1380      [IEEE]  "IEEE Standard for Binary Floating-Point Arithmetic",
1381              ANSI/IEEE Standard 754-1985, Institute of Electrical and
1382              Electronics Engineers, August 1985.
1383
1384   13.  Informative References
1385
1386      [KERN]  Brian W. Kernighan & Dennis M. Ritchie, "The C Programming
1387              Language", Bell Laboratories, Murray Hill, New Jersey, 1978.
1388
1389      [COHE]  Danny Cohen, "On Holy Wars and a Plea for Peace", IEEE
1390              Computer, October 1981.
1391
1392      [COUR]  "Courier: The Remote Procedure Call Protocol", XEROX
1393              Corporation, XSIS 038112, December 1981.
1394
1395      [SPAR]  "The SPARC Architecture Manual: Version 8", Prentice Hall,
1396              ISBN 0-13-825001-4.
1397
1398      [HPRE]  "HP Precision Architecture Handbook", June 1987, 5954-9906.
1399
1400
1401
1402   Eisler                    Standards Track                   [Page 25]
1403   ░░
1404   RFC 4506        XDR: External Data Representation Standard      May 2006
```

```
1405
1406
1407    14.  Acknowledgements
1408
1409       Bob Lyon was Sun's visible force behind ONC RPC in the 1980s.  Sun
1410       Microsystems, Inc., is listed as the author of RFC 1014.  Raj
1411       Srinivasan and the rest of the old ONC RPC working group edited RFC
1412       1014 into RFC 1832, from which this document is derived.  Mike Eisler
1413       and Bill Janssen submitted the implementation reports for this
1414       standard.  Kevin Coffman, Benny Halevy, and Jon Peterson reviewed
1415       this document and gave feedback.  Peter Astrand and Bryan Olson
1416       pointed out several errors in RFC 1832 which are corrected in this
1417       document.
1418
1419    Editor's Address
1420
1421       Mike Eisler
1422       5765 Chase Point Circle
1423       Colorado Springs, CO 80919
1424       USA
1425
1426       Phone: 719-599-9026
1427       EMail: email2mre-rfc4506@yahoo.com
1428
1429       Please address comments to: nfsv4@ietf.org
1430
1431
1432
1433
1434
1435
1436
1437
1438
1439
1440
1441
1442
1443
1444
1445
1446
1447
1448
1449
1450
1451
1452
1453
1454
1455
1456
1457
1458    Eisler                      Standards Track                  [Page 26]
```

1459   FF

1460   RFC 4506   XDR: External Data Representation Standard   May 2006

1461

1462

1463   Full Copyright Statement

1464

1465   Copyright (C) The Internet Society (2006).

1466

1467   This document is subject to the rights, licenses and restrictions
1468   contained in BCP 78, and except as set forth therein, the authors
1469   retain all their rights.

1470

1471   This document and the information contained herein are provided on an
1472   "AS IS" basis and THE CONTRIBUTOR, THE ORGANIZATION HE/SHE REPRESENTS
1473   OR IS SPONSORED BY (IF ANY), THE INTERNET SOCIETY AND THE INTERNET
1474   ENGINEERING TASK FORCE DISCLAIM ALL WARRANTIES, EXPRESS OR IMPLIED,
1475   INCLUDING BUT NOT LIMITED TO ANY WARRANTY THAT THE USE OF THE
1476   INFORMATION HEREIN WILL NOT INFRINGE ANY RIGHTS OR ANY IMPLIED
1477   WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

1478

1479   Intellectual Property

1480

1481   The IETF takes no position regarding the validity or scope of any
1482   Intellectual Property Rights or other rights that might be claimed to
1483   pertain to the implementation or use of the technology described in
1484   this document or the extent to which any license under such rights
1485   might or might not be available; nor does it represent that it has
1486   made any independent effort to identify any such rights. Information
1487   on the procedures with respect to rights in RFC documents can be
1488   found in BCP 78 and BCP 79.

1489

1490   Copies of IPR disclosures made to the IETF Secretariat and any
1491   assurances of licenses to be made available, or the result of an
1492   attempt made to obtain a general license or permission for the use of
1493   such proprietary rights by implementers or users of this
1494   specification can be obtained from the IETF on-line IPR repository at
1495   http://www.ietf.org/ipr.

1496

1497   The IETF invites any interested party to bring to its attention any
1498   copyrights, patents or patent applications, or other proprietary
1499   rights that may cover technology that may be required to implement
1500   this standard. Please address the information to the IETF at
1501   ietf-ipr@ietf.org.

1502

1503   Acknowledgement

1504

1505   Funding for the RFC Editor function is provided by the IETF
1506   Administrative Support Activity (IASA).

1507

1508

1509

1510

1511

1512

1513

1514   Eisler   Standards Track   [Page 27]

1515   FF

1516