XDR Protocol Specification

This appendix contains the XDR Protocol Language Specification.

XDR Protocol Introduction	*page 251*
XDR Data Type Declarations	*page 253*
The XDR Language Specification	*page 267*

XDR Protocol Introduction

External data representation (XDR) is a standard for the description and encoding of data. The XDR protocol 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 reference model's presentation layer (layer 6) and is roughly analogous in purpose to X.409, ISO Abstract Syntax Notation. The major difference between the two is that XDR uses implicit typing, while X.409 uses explicit typing.

XDR uses a language to describe data formats and only can be used to describe data; it is not a programming language. This language makes it possible to describe intricate data formats in a concise manner. The XDR language is similar to the C language. Protocols such as RPC and the NFS use XDR to describe the format of their data.

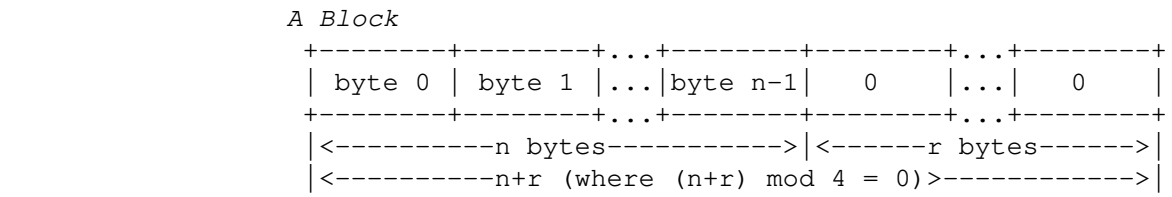
The XDR standard assumes that bytes (or octets) are portable and that a byte is defined to be 8 bits of data.

C

Graphic Box Notation

This appendix uses graphic box notation for illustration and comparison. In most illustrations, each box depicts a byte. 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`. The `n` bytes are followed by enough (0 to 3) residual zero bytes, `r`, to make the total byte count a multiple of four. Ellipses (`...`) between boxes show zero or more additional bytes where required. For example:

A Block



Basic Block Size

Choosing the XDR block size requires a trade off. Choosing a small size such as two makes the encoded data small, but causes alignment problems for machines that are not 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 large. Four was chosen as a compromise. Four is big enough to support most architectures efficiently, except for rare machines such as the eight-byte aligned Cray.

This is not to say that the computers cannot utilize standard `XDR`, just that they do so at a greater overhead per data item than 4-byte (32-bit) architectures. Four is also small enough to keep the encoded data restricted to a reasonable size.

The same data should encode into an equivalent result on all machines, so that encoded data can be compared or checksummed. So, variable length data must be padded with trailing zeros.

XDR Data Type Declarations

Each of the sections that follow:

- Describe a data type defined in the XDR standard
- Show how that data type is declared in the language
- Include a graphic illustration of the encoding

For each data type in the language we show a general paradigm declaration. Note that angle brackets (< and >) denote variable length sequences of data and square brackets ([and]) denote fixed-length sequences of data. `n`, `m` and `r` denote integers. For the full language specification, refer to "The XDR Language Specification" on page 267.

For some data types, specific examples are included. A more extensive example is given in the section, "XDR Data Description" on page 270.

Signed Integer

Description

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.

Declaration

Integers are declared:

```
int identifier;
```

Encoding

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

Unsigned Integer

Description

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

Declaration

An unsigned integer is declared as follows:

```
unsigned int identifier;
```

Encoding

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

Enumerations

Description

Enumerations have the same representation as signed integers and are handy for describing subsets of the integers.

Declaration

Enumerated data is declared as follows:

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

For example, an enumerated type could represent the three colors red, yellow, and blue as follows:

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

It is an error to assign to an `enum` an integer that has not been assigned in the `enum` declaration.

Encoding

See "Signed Integer,"shown previously.

Booleans

Description

Booleans are important enough and occur frequently enough to warrant their own explicit type in the standard. Booleans are integers of value `0` or `1`.

Declaration

Booleans are declared as follows:

`bool` *`identifier`*;

This is equivalent to:

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

Encoding

See "Signed Integer," shown in a previous section.

Hyper Integer and Unsigned Hyper Integer

Description

The standard defines 64-bit (8-byte) numbers called `hyper int` and `unsigned hyper int` whose 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.

Declaration

Hyper integers are declared as follows:

```
hyper int identifier;
unsigned hyper int identifier;
```

Encoding

```
Hyper Integer
(MSB)                                                   (LSB)
+------+------+------+------+------+------+------+------+
|byte 0|byte 1|byte2 |byte 3|byte 4|byte 5|byte 6|byte 7|
+------+------+------+------+------+------+------+------+
<------------------------64 bits----------------------->
```

Floating Point

Description

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 [1]. 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. There are eight bits in this field. The exponent is biased by 127.
- F: The fractional part of the number's mantissa, base 2. There are 23 bits are in this field.

Therefore, the floating-point number is described by:

```
(-1)**S * 2**(E-Bias) * 1.F
```

Declaration

Single-precision floating-point data is declared as follows:

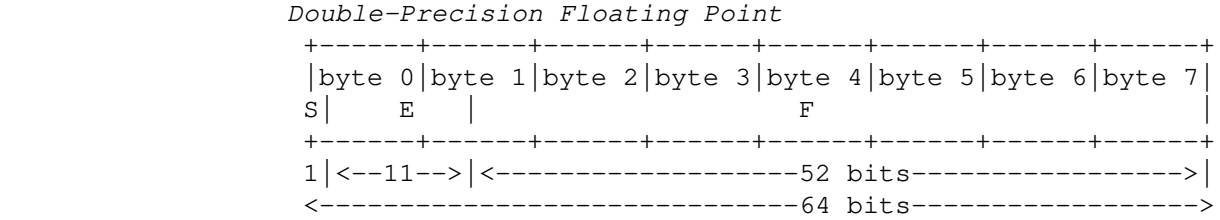
```
float identifier;
```

Declaration

`double` *identifier*;

Encoding

Double-Precision Floating Point



Just as the most and least significant bytes of an integer 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.

These offsets refer to the logical positions of the bits, *not* to their physical locations (which vary from medium to medium).

The IEEE specifications should be consulted about the encoding for signed zero, signed infinity (overflow), and de-normalized numbers (underflow) [1]. According to IEEE specifications, the `NaN` (not a number) is system dependent and should not be used externally.

Quadruple-Precision Floating Point

Description

The standard defines the encoding for the quadruple-precision floating-point data type `quadruple` (128 bits or 16 bytes). The encoding used is the IEEE standard for normalized quadruple-precision floating-point numbers [1]. The standard encodes the following three fields, which describe the quadruple-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. There are 15 bits in this field. The exponent is biased by 16383.

F: The fractional part of the number's mantissa, base 2. There are 111 bits in this field.

Therefore, the floating-point number is described by:

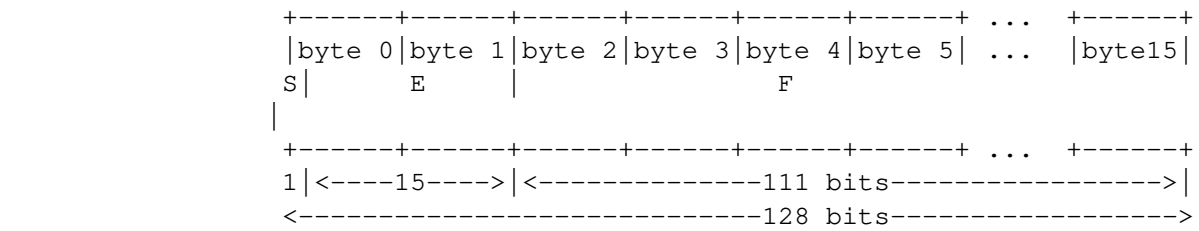
```
(-1)**S * 2**(E-Bias) * 1.F
```

Declaration

`quadruple` *identifier*`;`

Encoding

Quadruple-Precision Floating Point



Just as the most and least significant bytes of an integer 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. These offsets refer to the logical positions of the bits, *not* to their physical locations (which vary from medium to medium).

The IEEE specifications should be consulted about the encoding for signed zero, signed infinity (overflow), and de-normalized numbers (underflow) [1]. According to IEEE specifications, the `NaN` (not a number) is system dependent and should not be used externally.

Fixed-Length Opaque Data

Description

At times, fixed-length uninterpreted data needs to be passed among machines. This data is called `opaque`.

Declaration

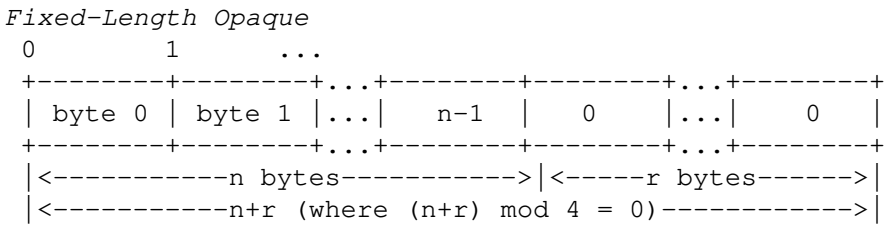
Opaque data is declared as follows:

```
opaque identifier[n];
```

where the constant *n* is the (static) number of bytes necessary to contain the opaque data.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.

Encoding

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.



Variable-Length Opaque Data

Description

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 subsequently), and followed by the `n` bytes of the sequence.

Byte `b` of the sequence always precedes byte `b+1` of the sequence, and byte `0` of the sequence always follows the sequence's length (count). The `n` bytes are followed by enough (0 to 3) residual zero bytes, `r`, to make the total byte count a multiple of four.

Declaration

Variable-length opaque data is declared in the following way:

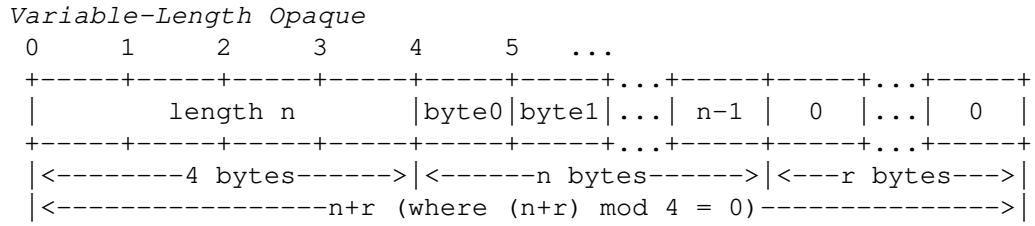
```
opaque identifier<m>;
 or
opaque identifier<>;
```

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. For example, a filing protocol may state that the maximum data transfer size is `8192` bytes, as follows:

```
opaque filedata<8192>;
```

Encoding

Variable-Length Opaque



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

Counted Byte Strings

Description

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 previously), and followed by the `n` bytes of the string. Byte `b` of the string always precedes byte `b+1` of the string, and byte `0` of the string always follows the string's length. The `n` bytes are followed by enough (0 to 3) residual zero bytes, `r`, to make the total byte count a multiple of four.

Declaration

Counted byte strings are declared as follows:

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

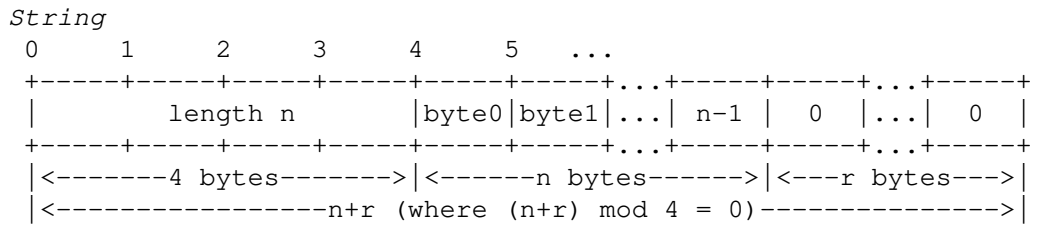
or

```
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>;
```

Encoding



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

Fixed-Length Array

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.

Declaration

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

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

Encoding

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

Variable-Length Array

Description

Counted arrays allow variable-length arrays to be encoded as homogeneous elements: the element count n (an unsigned integer) is followed by each array element, starting with element `0` and progressing through element `n-1`.

Declaration

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, it is assumed to be `(2**32) - 1`.

Encoding

```
Counted Array
 0  1  2  3
 +--+--+--+--+--+--+--+--+--+--+--+...+--+--+--+--+
 |     n     | element 0 | element 1 |...|element n-1|
 +--+--+--+--+--+--+--+--+--+--+--+...+--+--+--+--+
 |<-4 bytes->|<--------------n elements------------->|
```

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

Structure

Description

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.

Declaration

Structures are declared as follows:

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

Encoding

```
Structure
 +-------------+-------------+...
 | component A | component B |...
 +-------------+-------------+...
```

Discriminated Union

Description

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 that implies their encoding.

Declaration

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.

Encoding

```
Discriminated Union
 0   1   2   3
 +---+---+---+---+---+---+---+---+
 | discriminant  |  implied arm  |
 +---+---+---+---+---+---+---+---+
 |<---4 bytes--->|
```

Void

Description

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.

Declaration

The declaration is simply as follows:

```
void;
```

Voids are illustrated as follows:

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

Constant

Description

`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.

The following example defines a symbolic constant `DOZEN`, equal to `12`.

```
const DOZEN = 12;
```

Declaration

The declaration of a constant follows this form:

`const` *name–identifier* `=` *n*`;`

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`. The following example defines a new type called `eggbox` using an existing type called `egg` and the symbolic constant `DOZEN`:

```
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 were 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`, `enum`, or `union` 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
};
```

This syntax is preferred because one does not have to go to the end of a declaration to learn the name of the new type.

Optional-Data

Optional-data is one kind of union that occurs so frequently that it is given 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 useful for describing recursive data-structures, such as linked lists and trees.

C

The XDR Language Specification

Notational Conventions

This specification uses a modified Backus-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 embedded in quotes (").
3. Nonterminal symbols are strings of nonspecial *italic* 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 the 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
```

Lexical Notes

1. Comments begin with /* and end 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 underbars (_). 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 (−).

Code Example C-1 XDR Specification

```
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"
    | "quadruple"
    | "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 *
```

Syntax Notes

1. The following are keywords and cannot be used as identifiers:

Table C-1 XDR Keywords

bool	**const**	**enum**	**int**	**string**	**typedef**	**void**
case	**default**	**float**	**opaque**	**struct**	**union**	
char	**double**	**hyper**	**quadruple**	**switch**	**unsigned**	

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 `enum` type, or any `typedef` that evaluates to one of these. Also, the `case` values must be legal discriminant values. Finally, a `case` value may not be specified more than once within the scope of a `union` declaration.

XDR Data Description

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

Code Example C-2 XDR File Data Structure

```
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>; /*proginterptr*/
};
```

```
/* 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:

Table C-2 XDR Data Description Example

Offset	Hex Bytes	ASCII	Description
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	(qu	File data bytes ...
44	74 29 00 00	t)..	... and 2 zero-bytes of fill

C

RPC Language Reference

The RPC language is an extension of the XDR language. The sole extension is the addition of the `program` and `version` types.

For a description of the RPC extensions to the XDR language, see Appendix B, "RPC Protocol and Language Specification."

The RPC language is similar to C. This section describes the syntax of the RPC language, showing a few examples along the way. It also shows how RPC and XDR type definitions get compiled into C type definitions in the output header file.

An RPC language file consists of a series of definitions.

```
definition-list:
    definition;
    definition; definition-list
```

It recognizes six types of definitions.

```
definition:

        enum-definition
        const-definition
        typedef-definition
        struct-definition
        union-definition
        program-definition
```

Definitions are not the same as declarations. No space is allocated by a definition – only the type definition of a single or series of data elements. This means that variables still must be declared.

Enumerations

RPC/XDR enumerations have similar syntax as C enumerations.

```
enum-definition:
    "enum" enum-ident "{"
        enum-value-list
    "}"

enum-value-list:
    enum-value
    enum-value "," enum-value-list
```

```
enum-value:
    enum-value-ident
    enum-value-ident "=" value
```

Here is an example of an XDR `enum` and the C `enum` to which it gets compiled.

```
enum colortype {                    enum colortype {
    RED = 0,                            RED = 0,
    GREEN = 1,          -->             GREEN = 1,
    BLUE = 2                            BLUE = 2,
};                                  };
                                    typedef enum colortype colortype;
```

Constants

XDR symbolic constants may be used wherever an integer constant is used. For example, in array size specifications:

```
    const-definition:
        const const-ident = integer
```

The following example defines a constant, `DOZEN` as equal to `12`:

```
const DOZEN = 12; --> #define DOZEN 12
```

Type Definitions

XDR `typedefs` have the same syntax as C `typedefs`.

```
typedef-definition:
        typedef declaration
```

This example defines an `fname_type` used for declaring file name strings that have a maximum length of 255 characters.

```
typedef string fname_type<255>; --> typedef char *fname_type;
```

Declarations

In XDR, there are four kinds of declarations. These declarations must be a part of a `struct` or a `typedef`; they cannot stand alone:

```
declaration:
```

```
simple-declaration
fixed-array-declaration
variable-array-declaration
pointer-declaration
```

Simple Declarations

Simple declarations are just like simple C declarations:

```
simple-declaration:
        type-ident variable-ident
```

Example:

```
    colortype color; --> colortype color;
```

Fixed-Length Array Declarations

Fixed-length array declarations are just like C array declarations:

```
fixed-array-declaration:
        type-ident variable-ident [value]
```

Example:

```
colortype palette[8]; --> colortype palette[8];
```

Many programmers confuse variable declarations with type declarations. It is important to note that `rpcgen` does not support variable declarations. This example is a program that will not compile:

```
int data[10];
program P {
    version V {
        int PROC(data) = 1;
    } = 1;
} = 0x200000;
```

The example above will not compile because of the variable declaration:

```
int data[10]
```

Instead, use:

```
typedef int data[10];
```

or

```
struct data {int dummy [10]};
```

Variable-Length Array Declarations

Variable-length array declarations have no explicit syntax in C. The XDR language does have a syntax, using angle brackets:

```
variable-array-declaration:
        type-ident variable-ident <value>
        type-ident variable-ident < >
```

The maximum size is specified between the angle brackets. The size may be omitted, indicating that the array may be of any size:

```
int heights<12>; /* at most 12 items */
int widths<>; /* any number of items */
```

Because variable-length arrays have no explicit syntax in C, these declarations are compiled into `struct` declarations. For example, the `heights` declaration compiled into the following `struct`:

```
struct {
    u_int heights_len;              /* # of items in array */
    int *heights_val;               /* pointer to array */
} heights;
```

The number of items in the array is stored in the `_len` component and the pointer to the array is stored in the `_val` component. The first part of each component name is the same as the name of the declared XDR variable (`heights`).

Pointer Declarations

Pointer declarations are made in XDR exactly as they are in C. Address pointers are not really sent over the network; instead, XDR pointers are useful for sending recursive data types such as lists and trees. The type is called “optional-data,” not “pointer,” in XDR language:

```
pointer-declaration:
    type-ident *variable-ident
```

Example:

```
listitem *next; --> listitem *next;
```

Structures

An RPC/XDR `struct` is declared almost exactly like its C counterpart. It looks like the following:

```
struct-definition:
    struct struct-ident "{"
        declaration-list
    "}"

declaration-list:
    declaration ";"
    declaration ";" declaration-list
```

The following XDR structure is an example of a two-dimensional coordinate and the C structure that it compiles into:

```
struct coord {                  struct coord {
    int x;              -->         int x;
    int y;                          int y;
};                                  };
                                    typedef struct coord coord;
```

The output is identical to the input, except for the added `typedef` at the end of the output. This enables one to use `coord` instead of `struct coord` when declaring items.

Unions

XDR unions are discriminated unions, and do not look like C unions – they are more similar to Pascal variant records:

```
union-definition:
    "union" union-ident "switch" "("simple declaration")" "{"
        case-list
    "}"

case-list:
    "case" value ":" declaration ";"
    "case" value ":" declaration ";" case-list
    "default" ":" declaration ";"
```

The following is an example of a type returned as the result of a “read data” operation: If there is no error, return a block of data; otherwise, don’t return anything.

```
union read_result switch (int errno) {
    case 0:
        opaque data[1024];
    default:
        void;
    };
```

It compiles into the following:

```
struct read_result {
    int errno;
    union {
        char data[1024];
    } read_result_u;
};
typedef struct read_result read_result;
```

Notice that the union component of the output struct has the same name as the type name, except for the trailing _u.

Programs

RPC programs are declared using the following syntax:

```
program-definition:
    "program" program-ident "{"
        version-list
    "}" "=" value;

version-list:
    version ";"
    version ";" version-list

version:
    "version" version-ident "{"
        procedure-list
    "}" "=" value;

procedure-list:
    procedure ";"
    procedure ";" procedure-list

procedure:
    type-ident procedure-ident "(" type-ident ")" "=" value;
```

When the -N option is specified, rpcgen also recognizes the following syntax:

```
procedure:
    type-ident procedure-ident "(" type-ident-list ")" "=" value;
```

```
type-ident-list:
    type-ident
    type-ident "," type-ident-list
```

For example:

```
/*
 * time.x: Get or set the time. Time is represented as seconds
 * since 0:00, January 1, 1970.
 */
program TIMEPROG {
    version TIMEVERS {
        unsigned int TIMEGET(void) = 1;
        void TIMESET(unsigned) = 2;
    } = 1;
} = 0x20000044;
```

Note that the `void` argument type means that no argument is passed.

This file compiles into these `#define` statements in the output header file:

```
#define TIMEPROG 0x20000044
#define TIMEVERS 1
#define TIMEGET 1
#define TIMESET 2
```

Special Cases

There are several exceptions to the RPC language rules.

C-style Mode

In the new features section we talked about the features of the C-style mode of `rpcgen`. These features have implications with regard to the passing of `void` arguments. No arguments need be passed if their value is `void`.

Booleans

C has no built-in boolean type. However, the RPC library uses a boolean type called `bool_t` that is either `TRUE` or `FALSE`. Parameters declared as type `bool` in XDR language are compiled into `bool_t` in the output header file.

Example:

```
bool married; --> bool_t married;
```

Strings

The C language has no built-in string type, but instead uses the null-terminated `char *` convention. In C, strings are usually treated as null-terminated single-dimensional arrays.

In XDR language, strings are declared using the `string` keyword, and compiled into type `char *` in the output header file. The maximum size contained in the angle brackets specifies the maximum number of characters allowed in the strings (not counting the `NULL` character). The maximum size may be omitted, indicating a string of arbitrary length.

Examples:

```
string name<32>;   --> char *name;
string longname<>; --> char *longname;
```

Note – `NULL` strings cannot be passed; however, a zero-length string (that is, just the terminator or `NULL` byte) can be passed.

Opaque Data

Opaque data is used in XDR to describe untyped data, that is, sequences of arbitrary bytes. It may be declared either as a fixed length or variable length array. Examples:

```
opaque diskblock[512]; --> char diskblock[512];

opaque filedata<1024>; --> struct {
                               u_int filedata_len;
                               char *filedata_val;
                           } filedata;
```

Voids

In a void declaration, the variable is not named. The declaration is just `void` and nothing else. Void declarations can only occur in two places: union definitions and program definitions (as the argument or result of a remote procedure, for example no arguments are passed.)

C