External Data Representation Protocol Specification

External Data Representation Protocol Specification

1. Introduction

This manual describes library routines that allow a C programmer to describe arbitrary data structures in a machine-independent fashion. The eXternal Data Representation (XDR) standard is the backbone of Sun's Remote Procedure Call package, in the sense that data for remote procedure calls is transmitted using the standard. XDR library routines should be used to transmit data that is accessed (read or written) by more than one type of machine.

This manual contains a description of XDR library routines, a guide to accessing currently available XDR streams, information on defining new streams and data types, and a formal definition of the XDR standard. XDR was designed to work across different languages, operating systems, and machine architectures. Most users (particularly RPC users) only need the information in sections 2 and 3 of this document. Programmers wishing to implement RPC and XDR on new machines will need the information in sections 4 through 6. Advanced topics, not necessary for all implementations, are covered in section 7.

On Sun systems, C programs that want to use XDR routines must include the file `<rpc/rpc.h>`, which contains all the necessary interfaces to the XDR system. Since the C library `libc.a` contains all the XDR routines, compile as normal.

```
cc program.c
```

2. Justification

Consider the following two programs, `writer`:

```
#include <stdio.h>

main()                  /* writer.c */
{
        long i;

        for (i = 0; i < 8; i++) {
                if (fwrite((char *)&i, sizeof(i), 1, stdout) != 1) {
                        fprintf(stderr, "failed!\n");
                        exit(1);
                }
        }
}
```

and `reader`:

```
#include <stdio.h>

main()                  /* reader.c */
{
        long i, j;

        for (j = 0; j < 8; j++) {
                if (fread((char *)&i, sizeof (i), 1, stdin) != 1) {
                        fprintf(stderr, "failed!\n");
                        exit(1);
                }
                printf("%ld ", i);
        }
        printf("\n");
}
```

The two programs appear to be portable, because (a) they pass `lint` checking, and (b) they exhibit the same behavior when executed on two different hardware architectures, a Sun and a VAX.

Piping the output of the `writer` program to the `reader` program gives identical results on a Sun or a VAX.‡

```
sun% writer | reader
0 1 2 3 4 5 6 7
sun%
---
vax% writer | reader
0 1 2 3 4 5 6 7
vax%
```

With the advent of local area networks and Berkeley's 4.2 BSD UNIX came the concept of ‘‘network pipes’’ — a process produces data on one machine, and a second process consumes data on another machine. A network pipe can be constructed with `writer` and `reader`. Here are the results if the first produces data on a Sun, and the second consumes data on a VAX.

```
sun% writer | rsh vax reader
0 16777216 33554432 50331648 67108864 83886080 100663296 117440512
sun%
```

Identical results can be obtained by executing `writer` on the VAX and `reader` on the Sun. These results occur because the byte ordering of long integers differs between the VAX and the Sun, even though word size is the same. Note that 16777216 is 2^{24} — when four bytes are reversed, the 1 winds up in the 24th bit.

Whenever data is shared by two or more machine types, there is a need for portable data. Programs can be made data-portable by replacing the `read()` and `write()` calls with calls to an XDR library routine `xdr_long()`, a filter that knows the standard representation of a long integer in its external form. Here are the revised versions of `writer`:

```
#include <stdio.h>
#include <rpc/rpc.h>    /* xdr is a sub-library of the rpc library */

main()          /* writer.c */
{
        XDR xdrs;
        long i;

        xdrstdio_create(&xdrs, stdout, XDR_ENCODE);
        for (i = 0; i < 8; i++) {
                if (! xdr_long(&xdrs, &i)) {
                        fprintf(stderr, "failed!\n");
                        exit(1);
                }
        }
}
```

and `reader`:

```
#include <stdio.h>
#include <rpc/rpc.h>    /* xdr is a sub-library of the rpc library */

main()          /* reader.c */
{
        XDR xdrs;
        long i, j;

        xdrstdio_create(&xdrs, stdin, XDR_DECODE);
        for (j = 0; j < 8; j++) {
                if (! xdr_long(&xdrs, &i)) {
                        fprintf(stderr, "failed!\n");
                        exit(1);
                }
                printf("%ld ", i);
        }
        printf("\n");
}
```

The new programs were executed on a Sun, on a VAX, and from a Sun to a VAX; the results are shown below.

```
sun% writer | reader
0 1 2 3 4 5 6 7
sun%
---
vax% writer | reader
0 1 2 3 4 5 6 7
vax%
---
sun% writer | rsh vax reader
0 1 2 3 4 5 6 7
sun%
```

Dealing with integers is just the tip of the portable-data iceberg. Arbitrary data structures present portability problems, particularly with respect to alignment and pointers. Alignment on word boundaries may cause the size of a structure to vary from machine to machine. Pointers are convenient to use, but have no meaning outside the machine where they are defined.

The XDR library package solves data portability problems. It allows you to write and read arbitrary C constructs in a consistent, specified, well-documented manner. Thus, it makes sense to use the library even when the data is not shared among machines on a network.

The XDR library has filter routines for strings (null-terminated arrays of bytes), structures, unions, and arrays, to name a few. Using more primitive routines, you can write your own specific XDR routines to describe arbitrary data structures, including elements of arrays, arms of unions, or objects pointed at from other structures. The structures themselves may contain arrays of arbitrary elements, or pointers to other structures.

Let's examine the two programs more closely. There is a family of XDR stream creation routines in which each member treats the stream of bits differently. In our example, data is manipulated using standard I/O routines, so we use `xdrstdio_create()`. The parameters to XDR stream creation routines vary according to their function. In our example, `xdrstdio_create()` takes a pointer to an XDR structure that it initializes, a pointer to a FILE that the input or output is performed on, and the operation. The operation may be XDR_ENCODE for serializing in the `writer` program, or XDR_DECODE for deserializing in the `reader` program.

Note: RPC clients never need to create XDR streams; the RPC system itself creates these streams, which are then passed to the clients.

The `xdr_long()` primitive is characteristic of most XDR library primitives and all client XDR routines. First, the routine returns FALSE (0) if it fails, and TRUE (1) if it succeeds. Second, for each data type, `xxx`, there is an associated XDR routine of the form:

```
xdr_xxx(xdrs, fp)
        XDR *xdrs;
        xxx *fp;
{
}
```

In our case, `xxx` is long, and the corresponding XDR routine is a primitive, `xdr_long`. The client could also define an arbitrary structure `xxx` in which case the client would also supply the routine `xdr_xxx`, describing each field by calling XDR routines of the appropriate type. In all cases the first parameter, `xdrs` can be treated as an opaque handle, and passed to the primitive routines.

XDR routines are direction independent; that is, the same routines are called to serialize or deserialize data. This feature is critical to software engineering of portable data. The idea is to call the same routine for either operation — this almost guarantees that serialized data can also be deserialized. One routine is used by both producer and consumer of networked data. This is implemented by always passing the address of an object rather than the object itself — only in the case of deserialization is the object modified. This feature is not shown in our trivial example, but its value becomes obvious when nontrivial data structures are passed among machines. If needed, you can obtain the direction of the XDR operation. See section 3.7 for details.

Let’s look at a slightly more complicated example. Assume that a person’s gross assets and liabilities are to be exchanged among processes. Also assume that these values are important enough to warrant their own data type:

```
struct gnumbers {
        long g_assets;
        long g_liabilities;
};
```

The corresponding XDR routine describing this structure would be:

```
bool_t                  /* TRUE is success, FALSE is failure */
xdr_gnumbers(xdrs, gp)
        XDR *xdrs;
        struct gnumbers *gp;
{
        if (xdr_long(xdrs, &gp->g_assets) &&
            xdr_long(xdrs, &gp->g_liabilities))
                return(TRUE);
        return(FALSE);
}
```

Note that the parameter `xdrs` is never inspected or modified; it is only passed on to the subcomponent routines. It is imperative to inspect the return value of each XDR routine call, and to give up immediately and return FALSE if the subroutine fails.

This example also shows that the type `bool_t` is declared as an integer whose only values are TRUE (1) and FALSE (0). This document uses the following definitions:

```
#define bool_t  int
#define TRUE    1
#define FALSE   0

#define enum_t int      /* enum_t’s are used for generic enum’s */
```

Keeping these conventions in mind, `xdr_gnumbers()` can be rewritten as follows:

```
xdr_gnumbers(xdrs, gp)
        XDR *xdrs;
        struct gnumbers *gp;
{
        return (xdr_long(xdrs, &gp->g_assets) &&
                xdr_long(xdrs, &gp->g_liabilities));
}
```

This document uses both coding styles.

3. XDR Library Primitives

This section gives a synopsis of each XDR primitive. It starts with basic data types and moves on to constructed data types. Finally, XDR utilities are discussed. The interface to these primitives and utilities is defined in the include file `<rpc/xdr.h>`, automatically included by `<rpc/rpc.h>`.

3.1. Number Filters

The XDR library provides primitives that translate between C numbers and their corresponding external representations. The primitives cover the set of numbers in:

$$[\textit{signed}, \textit{unsigned}\,]*[\textit{short}, \textit{int}, \textit{long}\,]$$

Specifically, the six primitives are:

```
bool_t xdr_int(xdrs, ip)
        XDR *xdrs;
        int *ip;

bool_t xdr_u_int(xdrs, up)
        XDR *xdrs;
        unsigned *up;

bool_t xdr_long(xdrs, lip)
        XDR *xdrs;
        long *lip;

bool_t xdr_u_long(xdrs, lup)
        XDR *xdrs;
        u_long *lup;

bool_t xdr_short(xdrs, sip)
        XDR *xdrs;
        short *sip;

bool_t xdr_u_short(xdrs, sup)
        XDR *xdrs;
        u_short *sup;
```

The first parameter, `xdrs`, is an XDR stream handle. The second parameter is the address of the number that provides data to the stream or receives data from it. All routines return TRUE if they complete successfully, and FALSE otherwise.

3.2. Floating Point Filters

The XDR library also provides primitive routines for C's floating point types:

```
bool_t xdr_float(xdrs, fp)
        XDR *xdrs;
        float *fp;

bool_t xdr_double(xdrs, dp)
        XDR *xdrs;
        double *dp;
```

The first parameter, `xdrs` is an XDR stream handle. The second parameter is the address of the floating point number that provides data to the stream or receives data from it. All routines return TRUE if they complete successfully, and FALSE otherwise.

Note: Since the numbers are represented in IEEE floating point, routines may fail when decoding a valid IEEE representation into a machine-specific representation, or vice-versa.

3.3. Enumeration Filters

The XDR library provides a primitive for generic enumerations. The primitive assumes that a C `enum` has the same representation inside the machine as a C integer. The boolean type is an important instance of the `enum`. The external representation of a boolean is always one (TRUE) or zero (FALSE).

```
#define bool_t  int
#define FALSE   0
#define TRUE    1

#define enum_t int

bool_t xdr_enum(xdrs, ep)
        XDR *xdrs;
        enum_t *ep;

bool_t xdr_bool(xdrs, bp)
        XDR *xdrs;
        bool_t *bp;
```

The second parameters `ep` and `bp` are addresses of the associated type that provides data to, or receives data from, the stream `xdrs`. The routines return TRUE if they complete successfully, and FALSE otherwise.

3.4. No Data

Occasionally, an XDR routine must be supplied to the RPC system, even when no data is passed or required. The library provides such a routine:

```
bool_t xdr_void();  /* always returns TRUE */
```

3.5. Constructed Data Type Filters

Constructed or compound data type primitives require more parameters and perform more complicated functions then the primitives discussed above. This section includes primitives for strings, arrays, unions, and pointers to structures. Constructed data type primitives may use memory management. In many cases, memory is allocated when deserializing data with XDR_DECODE. Therefore, the XDR package must provide means to deallocate memory. This is done by an XDR operation, XDR_FREE. To review, the three XDR directional operations are XDR_ENCODE, XDR_DECODE, and XDR_FREE.

3.5.1. Strings

In C, a string is defined as a sequence of bytes terminated by a null byte, which is not considered when calculating string length. However, when a string is passed or manipulated, a pointer to it is employed. Therefore, the XDR library defines a string to be a `char *`, and not a sequence of characters. The external representation of a string is drastically different from its internal representation. Externally, strings are represented as sequences of ASCII characters, while internally, they are represented with character pointers. Conversion between the two representations is accomplished with the routine `xdr_string()`:

```
bool_t xdr_string(xdrs, sp, maxlength)
        XDR *xdrs;
        char **sp;
        u_int maxlength;
```

The first parameter `xdrs` is the XDR stream handle. The second parameter `sp` is a pointer to a string (type `char **`). The third parameter `maxlength` specifies the maximum number of bytes allowed during encoding or decoding; its value is usually specified by a protocol. For example, a protocol specification may say that a file name may be no longer than 255 characters. The routine returns FALSE if the number of characters exceeds `maxlength`, and TRUE if it doesn't.

The behavior of `xdr_string()` is similar to the behavior of other routines discussed in this section. The direction XDR_ENCODE is easiest to understand. The parameter `sp` points to a string of a certain length; if it does not exceed `maxlength`, the bytes are serialized.

The effect of deserializing a string is subtle. First the length of the incoming string is determined; it must not exceed `maxlength`. Next `sp` is dereferenced; if the the value is NULL, then a string of the appropriate length is allocated and `*sp` is set to this string. If the original value of `*sp` is non-NULL, then the XDR package assumes that a target area has been allocated, which can hold strings no longer than `maxlength`. In either case, the string is decoded into the target area. The routine then appends a null character to the string.

In the XDR_FREE operation, the string is obtained by dereferencing `sp`. If the string is not NULL, it is freed and `*sp` is set to NULL. In this operation, `xdr_string` ignores the `maxlength` parameter.

3.5.2. Byte Arrays

Often variable-length arrays of bytes are preferable to strings. Byte arrays differ from strings in the following three ways: 1) the length of the array (the byte count) is explicitly located in an unsigned integer, 2) the byte sequence is not terminated by a null character, and 3) the external representation of the bytes is the same as their internal representation. The primitive `xdr_bytes()` converts between the internal and external representations of byte arrays:

```
bool_t xdr_bytes(xdrs, bpp, lp, maxlength)
        XDR *xdrs;
        char **bpp;
        u_int *lp;
        u_int maxlength;
```

The usage of the first, second and fourth parameters are identical to the first, second and third parameters of `xdr_string()`, respectively. The length of the byte area is obtained by dereferencing `lp` when serializing; `*lp` is set to the byte length when deserializing.

3.5.3. Arrays

The XDR library package provides a primitive for handling arrays of arbitrary elements. The `xdr_bytes()` routine treats a subset of generic arrays, in which the size of array elements is known to be 1, and the external description of each element is built-in. The generic array primitive, `xdr_array()` requires parameters identical to those of `xdr_bytes()` plus two more: the size of array elements, and an XDR routine to handle each of the elements. This routine is called to encode or decode each element of the array.

```
bool_t xdr_array(xdrs, ap, lp, maxlength, elementsize, xdr_element)
        XDR *xdrs;
        char **ap;
        u_int *lp;
        u_int maxlength;
        u_int elementsize;
        bool_t (*xdr_element)();
```

The parameter `ap` is the address of the pointer to the array. If `*ap` is NULL when the array is being deserialized, XDR allocates an array of the appropriate size and sets `*ap` to that array. The element count of the array is obtained from `*lp` when the array is serialized; `*lp` is set to the array length when the array is deserialized. The parameter `maxlength` is the maximum number of elements that the array is allowed to have; `elementsize` is the byte size of each element of the array (the C function `sizeof()` can be used to obtain this value). The routine `xdr_element` is called to serialize, deserialize, or free each element of the array.

Examples

Before defining more constructed data types, it is appropriate to present three examples.

Example A

A user on a networked machine can be identified by (a) the machine name, such as `krypton`: see *gethostname* (3); (b) the user's UID: see *geteuid* (2); and (c) the group numbers to which the user belongs: see *getgroups* (2). A structure with this information and its associated XDR routine could be coded like this:

```
struct netuser {
        char    *nu_machinename;
        int     nu_uid;
        u_int   nu_glen;
        int     *nu_gids;
};
#define NLEN 255 /* machine names must be shorter than 256 chars */
#define NGRPS 20 /* user can't be a member of more than 20 groups */

bool_t
xdr_netuser(xdrs, nup)
        XDR *xdrs;
        struct netuser *nup;
{
        return (xdr_string(xdrs, &nup->nu_machinename, NLEN) &&
            xdr_int(xdrs, &nup->nu_uid) &&
            xdr_array(xdrs, &nup->nu_gids, &nup->nu_glen, NGRPS,
                sizeof (int), xdr_int));
}
```

Example B

A party of network users could be implemented as an array of `netuser` structure. The declaration and its associated XDR routines are as follows:

```
struct party {
        u_int p_len;
        struct netuser *p_nusers;
};
#define PLEN 500 /* max number of users in a party */

bool_t
xdr_party(xdrs, pp)
        XDR *xdrs;
        struct party *pp;
{
        return (xdr_array(xdrs, &pp->p_nusers, &pp->p_len, PLEN,
            sizeof (struct netuser), xdr_netuser));
}
```

Example C

The well-known parameters to `main()`, `argc` and `argv` can be combined into a structure. An array of these structures can make up a history of commands. The declarations and XDR routines might look like:

```
struct cmd {
        u_int c_argc;
        char **c_argv;
};
#define ALEN 1000  /* args can be no longer than 1000 chars */
#define NARGC 100  /* commands may have no more than 100 args */

struct history {
        u_int h_len;
        struct cmd *h_cmds;
};
#define NCMDS 75  /* history is no more than 75 commands */

bool_t
xdr_wrap_string(xdrs, sp)
        XDR *xdrs;
        char **sp;
{
        return (xdr_string(xdrs, sp, ALEN));
}

bool_t
xdr_cmd(xdrs, cp)
        XDR *xdrs;
        struct cmd *cp;
{
        return (xdr_array(xdrs, &cp->c_argv, &cp->c_argc, NARGC,
            sizeof (char *), xdr_wrap_string));
}

bool_t
xdr_history(xdrs, hp)
        XDR *xdrs;
        struct history *hp;
{
        return (xdr_array(xdrs, &hp->h_cmds, &hp->h_len, NCMDS,
            sizeof (struct cmd), xdr_cmd));
}
```

The most confusing part of this example is that the routine `xdr_wrap_string()` is needed to package the `xdr_string()` routine, because the implementation of `xdr_array()` only passes two parameters to the array element description routine; `xdr_wrap_string()` supplies the third parameter to `xdr_string()`.

By now the recursive nature of the XDR library should be obvious. Let's continue with more constructed data types.

3.5.4. Opaque Data

In some protocols, handles are passed from a server to client. The client passes the handle back to the server at some later time. Handles are never inspected by clients; they are obtained and submitted. That is to say, handles are opaque. The primitive `xdr_opaque()` is used for describing fixed sized, opaque bytes.

```
bool_t xdr_opaque(xdrs, p, len)
        XDR *xdrs;
        char *p;
        u_int len;
```

The parameter `p` is the location of the bytes; `len` is the number of bytes in the opaque object. By definition, the actual data contained in the opaque object are not machine portable.

3.5.5. Fixed Sized Arrays

The XDR library does not provide a primitive for fixed-length arrays (the primitive `xdr_array()` is for varying-length arrays). Example A could be rewritten to use fixed-sized arrays in the following fashion:

```
#define NLEN 255  /* machine names must be shorter than 256 chars */
#define NGRPS 20  /* user cannot be a member of more than 20 groups */

struct netuser {
        char *nu_machinename;
        int nu_uid;
        int nu_gids[NGRPS];
};

bool_t
xdr_netuser(xdrs, nup)
        XDR *xdrs;
        struct netuser *nup;
{
        int i;

        if (! xdr_string(xdrs, &nup->nu_machinename, NLEN))
                return (FALSE);
        if (! xdr_int(xdrs, &nup->nu_uid))
                return (FALSE);
        for (i = 0; i < NGRPS; i++) {
                if (! xdr_int(xdrs, &nup->nu_gids[i]))
                        return (FALSE);
        }
        return (TRUE);
}
```

Exercise:

Rewrite Example A so that it uses varying-length arrays and so that the `netuser` structure contains the actual `nu_gids` array body as in the example above.

3.5.6. Discriminated Unions

The XDR library supports discriminated unions. A discriminated union is a C union and an `enum_t` value that selects an ‘‘arm’’ of the union.

```
struct xdr_discrim {
        enum_t value;
        bool_t (*proc)();
};

bool_t xdr_union(xdrs, dscmp, unp, arms, defaultarm)
        XDR *xdrs;
        enum_t *dscmp;
        char *unp;
        struct xdr_discrim *arms;
        bool_t (*defaultarm)();  /* may equal NULL */
```

First the routine translates the discriminant of the union located at `*dscmp`. The discriminant is always an `enum_t`. Next the union located at `*unp` is translated. The parameter `arms` is a pointer to an array of `xdr_discrim` structures. Each structure contains an order pair of `[value,proc]`. If the union's discriminant is equal to the associated `value`, then the `proc` is called to translate the union. The end of the `xdr_discrim` structure array is denoted by a routine of value NULL (0). If the discriminant is not found in the `arms` array, then the `defaultarm` procedure is called if it is non-NULL; otherwise the routine returns FALSE.

Example D

Suppose the type of a union may be integer, character pointer (a string), or a `gnumbers` structure. Also, assume the union and its current type are declared in a structure. The declaration is:

```
enum utype { INTEGER=1, STRING=2, GNUMBERS=3 };

struct u_tag {
        enum utype utype;       /* this is the union’s discriminant */
        union {
                int ival;
                char *pval;
                struct gnumbers gn;
        } uval;
};
```

The following constructs and XDR procedure (de)serialize the discriminated union:

```
struct xdr_discrim u_tag_arms[4] = {
        { INTEGER, xdr_int },
        { GNUMBERS, xdr_gnumbers }
        { STRING, xdr_wrap_string },
        { __dontcare__, NULL }
        /* always terminate arms with a NULL xdr_proc */
}

bool_t
xdr_u_tag(xdrs, utp)
        XDR *xdrs;
        struct u_tag *utp;
{
        return (xdr_union(xdrs, &utp->utype, &utp->uval, u_tag_arms,
            NULL));
}
```

The routine `xdr_gnumbers()` was presented in Section 2; `xdr_wrap_string()` was presented in example C. The default arm parameter to `xdr_union()` (the last parameter) is NULL in this example. Therefore the value of the union's discriminant legally may take on only values listed in the `u_tag_arms` array. This example also demonstrates that the elements of the arm's array do not need to be sorted.

It is worth pointing out that the values of the discriminant may be sparse, though in this example they are not. It is always good practice to assign explicitly integer values to each element of the discriminant's type. This practice both documents the external representation of the discriminant and guarantees that different C compilers emit identical discriminant values.

Exercise:

Implement `xdr_union()` using the other primitives in this section.

3.5.7. Pointers

In C it is often convenient to put pointers to another structure within a structure. The primitive `xdr_reference()` makes it easy to serialize, deserialize, and free these referenced structures.

```
bool_t xdr_reference(xdrs, pp, size, proc)
        XDR *xdrs;
        char **pp;
        u_int ssize;
        bool_t (*proc)();
```

Parameter `pp` is the address of the pointer to the structure; parameter `ssize` is the size in bytes of the structure (use the C function `sizeof()` to obtain this value); and `proc` is the XDR routine that describes the structure. When decoding data, storage is allocated if `*pp` is NULL.

There is no need for a primitive `xdr_struct()` to describe structures within structures, because pointers are always sufficient.

Exercise:

Implement `xdr_reference()` using `xdr_array()`. Warning: `xdr_reference()` and `xdr_array()` are NOT interchangeable external representations of data.

Example E

Suppose there is a structure containing a person's name and a pointer to a `gnumbers` structure containing the person's gross assets and liabilities. The construct is:

```
struct pgn {
        char *name;
        struct gnumbers *gnp;
};
```

The corresponding XDR routine for this structure is:

```
bool_t
xdr_pgn(xdrs, pp)
        XDR *xdrs;
        struct pgn *pp;
{
        if (xdr_string(xdrs, &pp->name, NLEN) &&
            xdr_reference(xdrs, &pp->gnp, sizeof(struct gnumbers),
            xdr_gnumbers))
                return(TRUE);
        return(FALSE);
}
```

3.5.7.1. Pointer Semantics and XDR

In many applications, C programmers attach double meaning to the values of a pointer. Typically the value NULL (or zero) means data is not needed, yet some application-specific interpretation applies. In essence, the C programmer is encoding a discriminated union efficiently by overloading the interpretation of the value of a pointer. For instance, in example E a NULL pointer value for `gnp` could indicate that the person's assets and liabilities are unknown. That is, the pointer value encodes two things: whether or not the data is known; and if it is known, where it is located in memory. Linked lists are an extreme example of the use of application-specific pointer interpretation.

The primitive `xdr_reference()` cannot and does not attach any special meaning to a NULL-value pointer during serialization. That is, passing an address of a pointer whose value is NULL to `xdr_reference()` when serialing data will most likely cause a memory fault and, on UNIX, a core dump for debugging.

It is the explicit responsibility of the programmer to expand non-dereferenceable pointers into their specific semantics. This usually involves describing data with a two-armed discriminated union. One arm is used when the pointer is valid; the other is used when the pointer is invalid (NULL). Section 7 has an example (linked lists encoding) that deals with invalid pointer interpretation.

Exercise:

After reading Section 7, return here and extend example E so that it can correctly deal with null pointer values.

Exercise:

Using the `xdr_union()`, `xdr_reference()` and `xdr_void()` primitives, implement a generic pointer handling primitive that implicitly deals with NULL pointers. The XDR library does not provide such a primitive because it does not want to give the illusion that pointers have meaning in the external world.

3.6. Non-filter Primitives

XDR streams can be manipulated with the primitives discussed in this section.

```
u_int xdr_getpos(xdrs)
        XDR *xdrs;

bool_t xdr_setpos(xdrs, pos)
        XDR *xdrs;
        u_int pos;

xdr_destroy(xdrs)
        XDR *xdrs;
```

The routine `xdr_getpos()` returns an unsigned integer that describes the current position in the data stream. Warning: In some XDR streams, the returned value of `xdr_getpos()` is meaningless; the routine returns a –1 in this case (though –1 should be a legitimate value).

The routine `xdr_setpos()` sets a stream position to `pos`. Warning: In some XDR streams, setting a position is impossible; in such cases, `xdr_setpos()` will return FALSE. This routine will also fail if the requested position is out-of-bounds. The definition of bounds varies from stream to stream.

The `xdr_destroy()` primitive destroys the XDR stream. Usage of the stream after calling this routine is undefined.

3.7. XDR Operation Directions

At times you may wish to optimize XDR routines by taking advantage of the direction of the operation (XDR_ENCODE, XDR_DECODE, or XDR_FREE). The value `xdrs->x_op` always contains the direction of the XDR operation. Programmers are not encouraged to take advantage of this information. Therefore, no example is presented here. However, an example in Section 7 demonstrates the usefulness of the `xdrs->x_op` field.

4. XDR Stream Access

An XDR stream is obtained by calling the appropriate creation routine. These creation routines take arguments that are tailored to the specific properties of the stream.

Streams currently exist for (de)serialization of data to or from standard I/O FILE streams, TCP/IP connections and UNIX files, and memory. Section 5 documents the XDR object and how to make new XDR streams when they are required.

4.1. Standard I/O Streams

XDR streams can be interfaced to standard I/O using the `xdrstdio_create()` routine as follows:

```
#include <stdio.h>
#include <rpc/rpc.h>  /* xdr streams are a part of the rpc library */

void
xdrstdio_create(xdrs, fp, x_op)
        XDR *xdrs;
        FILE *fp;
        enum xdr_op x_op;
```

The routine `xdrstdio_create()` initializes an XDR stream pointed to by `xdrs`. The XDR stream interfaces to the standard I/O library. Parameter `fp` is an open file, and `x_op` is an XDR direction.

4.2. Memory Streams

Memory streams allow the streaming of data into or out of a specified area of memory:

```
#include <rpc/rpc.h>

void
xdrmem_create(xdrs, addr, len, x_op)
        XDR *xdrs;
        char *addr;
        u_int len;
        enum xdr_op x_op;
```

The routine `xdrmem_create()` initializes an XDR stream in local memory. The memory is pointed to by parameter `addr`; parameter `len` is the length in bytes of the memory. The parameters `xdrs` and `x_op` are identical to the corresponding parameters of `xdrstdio_create()`. Currently, the UDP/IP implementation of RPC uses `xdrmem_create()`. Complete call or result messages are built in memory before calling the `sendto()` system routine.

4.3. Record (TCP/IP) Streams

A record stream is an XDR stream built on top of a record marking standard that is built on top of the UNIX file or 4.2 BSD connection interface.

```
#include <rpc/rpc.h>  /* xdr streams are a part of the rpc library */

xdrrec_create(xdrs, sendsize, recvsize, iohandle, readproc, writeproc)
        XDR *xdrs;
        u_int sendsize, recvsize;
        char *iohandle;
        int (*readproc)(), (*writeproc)();
```

The routine `xdrrec_create()` provides an XDR stream interface that allows for a bidirectional, arbitrarily long sequence of records. The contents of the records are meant to be data in XDR form. The stream's primary use is for interfacing RPC to TCP connections. However, it can be used to stream data into or out of normal UNIX files.

The parameter `xdrs` is similar to the corresponding parameter described above. The stream does its own data buffering similar to that of standard I/O. The parameters `sendsize` and `recvsize` determine the size in bytes of the output and input buffers, respectively; if their values are zero (0), then predetermined defaults are used. When a buffer needs to be filled or flushed, the routine `readproc` or `writeproc` is called, respectively. The usage and behavior of these routines are similar to the UNIX system calls `read()` and `write()`. However, the first parameter to each of these routines is the opaque parameter `iohandle`. The other two parameters `buf(` and `nbytes`) and the results (byte count) are identical to the system routines. If `xxx` is `readproc` or `writeproc`, then it has the following form:

```
/* returns the actual number of bytes transferred.
 * -1 is an error
 */
int
xxx(iohandle, buf, len)
        char *iohandle;
        char *buf;
        int nbytes;
```

The XDR stream provides means for delimiting records in the byte stream. The implementation details of delimiting records in a stream are discussed in appendix 1. The primitives that are specific to record streams are as follows:

```
bool_t
xdrrec_endofrecord(xdrs, flushnow)
        XDR *xdrs;
        bool_t flushnow;

bool_t
xdrrec_skiprecord(xdrs)
        XDR *xdrs;

bool_t
xdrrec_eof(xdrs)
        XDR *xdrs;
```

The routine `xdrrec_endofrecord()` causes the current outgoing data to be marked as a record. If the parameter `flushnow` is TRUE, then the stream's `writeproc()` will be called; otherwise, `writeproc()` will be called when the output buffer has been filled.

The routine `xdrrec_skiprecord()` causes an input stream's position to be moved past the current record boundary and onto the beginning of the next record in the stream.

If there is no more data in the stream's input buffer, then the routine `xdrrec_eof()` returns TRUE. That is not to say that there is no more data in the underlying file descriptor.

5. XDR Stream Implementation

This section provides the abstract data types needed to implement new instances of XDR streams.

5.1. The XDR Object

The following structure defines the interface to an XDR stream:

```
enum xdr_op { XDR_ENCODE = 0, XDR_DECODE = 1, XDR_FREE = 2 };

typedef struct {
        enum xdr_op     x_op;           /* operation; fast additional param */
        struct xdr_ops {
            bool_t  (*x_getlong)();     /* get a long from underlying stream */
            bool_t  (*x_putlong)();     /* put a long to " */
            bool_t  (*x_getbytes)();    /* get some bytes from " */
            bool_t  (*x_putbytes)();    /* put some bytes to " */
            u_int   (*x_getpostn)();    /* returns byte offset from beginning */
            bool_t  (*x_setpostn)();    /* repositions position in stream */
            caddr_t (*x_inline)();      /* buf quick ptr to buffered data */
            VOID    (*x_destroy)();     /* free privates of this xdr_stream */
        } *x_ops;
        caddr_t         x_public;       /* users' data */
        caddr_t         x_private;      /* pointer to private data */
        caddr_t         x_base;         /* private used for position info */
        int             x_handy;        /* extra private word */
} XDR;
```

The `x_op` field is the current operation being performed on the stream. This field is important to the XDR primitives, but should not affect a stream's implementation. That is, a stream's implementation should not depend on this value. The fields `x_private`, `x_base`, and `x_handy` are private to the particular stream's implementation. The field `x_public` is for the XDR client and should never be used by the XDR stream implementations or the XDR primitives.

Macros for accessing operations `x_getpostn()`, `x_setpostn()`, and `x_destroy()` were defined in Section 3.6. The operation `x_inline()` takes two parameters: an XDR *, and an unsigned integer, which is a byte count. The routine returns a pointer to a piece of the stream's internal buffer. The caller can then use the buffer segment for any purpose. From the stream's point of view, the bytes in the buffer segment have been consumed or put. The routine may return NULL if it cannot return a buffer segment of the requested size. (The `x_inline` routine is for cycle squeezers. Use of the resulting buffer is not data-portable. Users are encouraged not to use this feature.)

The operations `x_getbytes()` and `x_putbytes()` blindly get and put sequences of bytes from or to the underlying stream; they return TRUE if they are successful, and FALSE otherwise. The routines have identical parameters (replace `xxx`):

```
      bool_t
      xxxbytes(xdrs, buf, bytecount)
              XDR *xdrs;
              char *buf;
              u_int bytecount;
```

The operations `x_getlong()` and `x_putlong()` receive and put long numbers from and to the data stream. It is the responsibility of these routines to translate the numbers between the machine representation and the (standard) external representation. The UNIX primitives `htonl()` and `ntohl()` can be helpful in accomplishing this. Section 6 defines the standard representation of numbers. The higher-level XDR implementation assumes that signed and unsigned long integers contain the same number of bits, and that nonnegative integers have the same bit representations as unsigned integers. The routines return TRUE if they succeed, and FALSE otherwise. They have identical parameters:

```
bool_t
xxxlong(xdrs, lp)
        XDR *xdrs;
        long *lp;
```

Implementors of new XDR streams must make an XDR structure (with new operation routines) available to clients, using some kind of create routine.

6. XDR Standard

This section defines the external data representation standard. The standard is independent of languages, operating systems and hardware architectures. Once data is shared among machines, it should not matter that the data was produced on a Sun, but is consumed by a VAX (or vice versa). Similarly the choice of operating systems should have no influence on how the data is represented externally. For programming languages, data produced by a C program should be readable by a FORTRAN or Pascal program.

The external data representation standard depends on the assumption that bytes (or octets) are portable. A byte is defined to be eight bits of data. It is assumed that hardware that encodes bytes onto various media will preserve the bytes' meanings across hardware boundaries. For example, the Ethernet standard suggests that bytes be encoded ‘‘little endian’’ style. Both Sun and VAX hardware implementations adhere to the standard.

The XDR standard also suggests a language used to describe data. The language is a bastardized C; it is a data description language, not a programming language. (The Xerox Courier Standard uses bastardized Mesa as its data description language.)

6.1. 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 $n-1$, where (n mod 4)=0. The bytes are read or written to some byte stream such that byte m always precedes byte $m+1$.

6.2. Integer

An XDR signed integer is a 32-bit datum that encodes an integer in the range `[-2147483648,2147483647]`. The integer is represented in two's complement notation. The most and least significant bytes are 0 and 3, respectively. The data description of integers is `integer`.

6.3. Unsigned Integer

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

6.4. Enumerations

Enumerations have the same representation as integers. Enumerations are handy for describing subsets of the integers. The data description of enumerated data is as follows:

```
typedef enum { name = value, .... } type-name;
```

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

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

6.5. Booleans

Booleans are important enough and occur frequently enough to warrant their own explicit type in the standard. Boolean is an enumeration with the following form:

```
typedef enum { FALSE = 0, TRUE = 1 } boolean;
```

6.6. Hyper Integer and Hyper Unsigned

The standard also defines 64-bit (8-byte) numbers called `hyper integer` and `hyper unsigned`. Their representations are the obvious extensions of the integer and unsigned defined above. The most and least significant bytes are 0 and 7, respectively.

6.7. Floating Point and Double Precision

The standard defines the encoding for the floating point data types `float` (32 bits or 4 bytes) and `double` (64 bits or 8 bytes). The encoding used is the IEEE standard for normalized single- and double-precision floating point numbers. See the IEEE floating point standard for more information. The standard encodes the following three fields, which describe the floating point number:

S	The sign of the number. Values 0 and 1 represent positive and negative, respectively.
E	The exponent of the number, base 2. Floats devote 8 bits to this field, while doubles devote 11 bits. The exponents for float and double are biased by 127 and 1023, respectively.
F	The fractional part of the number's mantissa, base 2. Floats devote 23 bits to this field, while doubles devote 52 bits.

Therefore, the floating point number is described by:

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

Just as the most and least significant bytes of a number are 0 and 3, the most and least significant bits of a single-precision floating point number are 0 and 31. The beginning bit (and most significant bit) offsets of *S*, *E*, and *F* are 0, 1, and 9, respectively.

Doubles have the analogous extensions. The beginning bit (and most significant bit) offsets of *S*, *E*, and *F* are 0, 1, and 12, respectively.

The IEEE specification should be consulted concerning the encoding for signed zero, signed infinity (overflow), and denormalized numbers (underflow). Under IEEE specifications, the ‘‘NaN’’ (not a number) is system dependent and should not be used.

6.8. Opaque Data

At times fixed-sized uninterpreted data needs to be passed among machines. This data is called `opaque` and is described as:

```
typedef opaque type-name[n];
opaque name[n];
```

where `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 (up to 3) zero-valued bytes to make the total byte count of the opaque object a multiple of four.

6.9. Counted Byte Strings

The standard defines a string of *n* (numbered 0 through $n-1$) bytes to be the number *n* encoded as `unsigned`, and followed by the *n* bytes of the string. If *n* is not a multiple of four, then the *n* bytes are followed by enough (up to 3) zero-valued bytes to make the total byte count a multiple of four. The data description of strings is as follows:

```
typedef string type-name<N>;
typedef string type-name<>;
string name<N>;
string name<>;
```

Note that the data description language uses angle brackets (< and >) to denote anything that is varying-length (as opposed to square brackets to denote fixed-length sequences of data).

The constant `N` denotes an upper bound of the number of bytes that a string may contain. If `N` is not specified, it is assumed to be $2^{32}-1$, the maximum length. The constant `N` 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, such as:

```
string filename<255>;
```

The XDR specification does not say what the individual bytes of a string represent; this important information is left to higher-level specifications. A reasonable default is to assume that the bytes encode ASCII characters.

6.10. Fixed Arrays

The data description for fixed-size arrays of homogeneous elements is as follows:

```
typedef elementtype type-name[n];
elementtype name[n];
```

Fixed-size arrays of elements numbered 0 through $n-1$ are encoded by individually encoding the elements of the array in their natural order, 0 through $n-1$.

6.11. Counted Arrays

Counted arrays provide the ability to encode varyiable-length arrays of homogeneous elements. The array is encoded as: the element count *n* (an unsigned integer), followed by the encoding of each of the array's elements, starting with element 0 and progressing through element $n-1$. The data description for counted arrays is similar to that of counted strings:

```
typedef elementtype type-name<N>;
typedef elementtype type-name<>;
elementtype name<N>;
elementtype name<>;
```

Again, the constant `N` specifies the maximum acceptable element count of an array; if `N` is not specified, it is assumed to be $2^{32}-1$.

6.12. Structures

The data description for structures is very similar to that of standard C:

```
typedef struct {
        component-type component-name;
        ...
} type-name;
```

The components of the structure are encoded in the order of their declaration in the structure.

6.13. Discriminated Unions

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 the discriminant is always an enumeration. The component types are called ‘‘arms’’ of the union. The discriminated union is encoded as its discriminant followed by the encoding of the implied arm. The data description for discriminated unions is as follows:

```
typedef union switch (discriminant-type) {
        discriminant-value: arm-type;
        ...
        default: default-arm-type;
} type-name;
```

The default arm is optional. If it is not specified, then a valid encoding of the union cannot take on unspecified discriminant values. Most specifications neither need nor use default arms.

6.14. Missing Specifications

The standard lacks representations for bit fields and bitmaps, since the standard is based on bytes. This is not to say that no specification should be attempted.

6.15. Library Primitive / XDR Standard Cross Reference

The following table describes the association between the C library primitives discussed in Section 3, and the standard data types defined in this section:

C Primitive	XDR Type	Sections

`xdr_int`
`xdr_long`
`xdr_short`
	`integer`	3.1, 6.2

`xdr_u_int`
`xdr_u_long`
`xdr_u_short`
	`unsigned`	3.1, 6.3

-	`hyper integer`
	`hyper unsigned`
		6.6

`xdr_float`	`float`	3.2, 6.7
`xdr_double`	`double`	3.2, 6.7
`xdr_enum`	`enum_t`	3.3, 6.4
`xdr_bool`	`bool_t`	3.3, 6.5
`xdr_string` `xdr_bytes`	`string`	3.5.1, 6.9 3.5.2
`xdr_array`	`(varying arrays)`	3.5.3, 6.11
-	`(fixed arrays)`	3.5.5, 6.10
`xdr_opaque`	`opaque`	3.5.4, 6.8
`xdr_union`	`union`	3.5.6, 6.13
`xdr_reference`	-	3.5.7
-	`struct`	6.6

7. Advanced Topics

This section describes techniques for passing data structures that are not covered in the preceding sections. Such structures include linked lists (of arbitrary lengths). Unlike the simpler examples covered in the earlier sections, the following examples are written using both the XDR C library routines and the XDR data description language. Section 6 describes the XDR data definition language used below.

7.1. Linked Lists

The last example in Section 2 presented a C data structure and its associated XDR routines for a person's gross assets and liabilities. The example is duplicated below:

```
struct gnumbers {
        long g_assets;
        long g_liabilities;
};

bool_t
xdr_gnumbers(xdrs, gp)
        XDR *xdrs;
        struct gnumbers *gp;
{
        if (xdr_long(xdrs, &(gp->g_assets)))
                return (xdr_long(xdrs, &(gp->g_liabilities)));
        return (FALSE);
}
```

Now assume that we wish to implement a linked list of such information. A data structure could be constructed as follows:

```
typedef struct gnnode {
        struct gnumbers gn_numbers;
        struct gnnode *nxt;
};

typedef struct gnnode *gnumbers_list;
```

The head of the linked list can be thought of as the data object; that is, the head is not merely a convenient shorthand for a structure. Similarly the `nxt` field is used to indicate whether or not the object has terminated. Unfortunately, if the object continues, the `nxt` field is also the address of where it continues. The link addresses carry no useful information when the object is serialized.

The XDR data description of this linked list is described by the recursive type declaration of gnumbers_list:

```
struct gnumbers {
        unsigned g_assets;
        unsigned g_liabilities;
};
```

```
typedef union switch (boolean) {
        case TRUE: struct {
                struct gnumbers current_element;
                gnumbers_list rest_of_list;
        };
        case FALSE: struct {};
} gnumbers_list;
```

In this description, the boolean indicates whether there is more data following it. If the boolean is FALSE, then it is the last data field of the structure. If it is TRUE, then it is followed by a `gnumbers` structure and (recursively) by a `gnumbers_list` (the rest of the object). Note that the C declaration has no boolean explicitly declared in it (though the `nxt` field implicitly carries the information), while the XDR data description has no pointer explicitly declared in it.

Hints for writing a set of XDR routines to successfully (de)serialize a linked list of entries can be taken from the XDR description of the pointer-less data. The set consists of the mutually recursive routines `xdr_gnumbers_list`, `xdr_wrap_list`, and `xdr_gnnode`.

```
bool_t
xdr_gnnode(xdrs, gp)
        XDR *xdrs;
        struct gnnode *gp;
{
        return (xdr_gnumbers(xdrs, &(gp->gn_numbers)) &&
                xdr_gnumbers_list(xdrs, &(gp->nxt)) );
}

bool_t
xdr_wrap_list(xdrs, glp)
        XDR *xdrs;
        gnumbers_list *glp;
{
        return (xdr_reference(xdrs, glp, sizeof(struct gnnode),
            xdr_gnnode));
}
```

```
struct xdr_discrim choices[2] = {
        /* called if another node needs (de)serializing */
        { TRUE, xdr_wrap_list },
         /* called when there are no more nodes to be (de)serialized */
        { FALSE, xdr_void }
}

bool_t
xdr_gnumbers_list(xdrs, glp)
        XDR *xdrs;
        gnumbers_list *glp;
{
        bool_t more_data;

        more_data = (*glp != (gnumbers_list)NULL);
        return (xdr_union(xdrs, &more_data, glp, choices, NULL);
}
```

The entry routine is `xdr_gnumbers_list()`; its job is to translate between the boolean value `more_data` and the list pointer values. If there is no more data, the `xdr_union()` primitive calls `xdr_void()` and the recursion is terminated. Otherwise, `xdr_union()` calls `xdr_wrap_list()`, whose job is to dereference the list pointers. The `xdr_gnnode()` routine actually (de)serializes data of the current node of the linked list, and recursively calls `xdr_gnumbers_list()` to handle the remainder of the list.

You should convince yourself that these routines function correctly in all three directions (XDR_ENCODE, XDR_DECODE and XDR_FREE) for linked lists of any length (including zero). Note that the boolean `more_data` is always initialized, but in the XDR_DECODE case it is overwritten by an externally generated value. Also note that the value of the `bool_t` is lost in the stack. The essence of the value is reflected in the list's pointers.

The unfortunate side effect of (de)serializing a list with these routines is that the C stack grows linearly with respect to the number of nodes in the list. This is due to the recursion. The routines are also hard to code (and understand) due to the number and nature of primitives involved (such as `xdr_reference`, `xdr_union`, and `xdr_void`).

The following routine collapses the recursive routines. It also has other optimizations that are discussed below.

```
bool_t
xdr_gnumbers_list(xdrs, glp)
        XDR *xdrs;
        gnumbers_list *glp;
{
        bool_t more_data;

        while (TRUE) {
                more_data = (*glp != (gnumbers_list)NULL);
                if (! xdr_bool(xdrs, &more_data))
                        return (FALSE);
                if (! more_data)
                        return (TRUE);  /* we are done */
                if (! xdr_reference(xdrs, glp, sizeof(struct gnnode),
                    xdr_gnumbers))
                        return (FALSE);
                glp = &((*glp)->nxt);
        }
}
```

The claim is that this one routine is easier to code and understand than the three recursive routines above. (It is also buggy, as discussed below.) The parameter `glp` is treated as the address of the pointer to the head of the remainder of the list to be (de)serialized. Thus, `glp` is set to the address of the current node's `nxt` field at the end of the while loop. The discriminated union is implemented in-line; the variable `more_data` has the same use in this routine as in the routines above. Its value is recomputed and re-(de)serialized each iteration of the loop. Since `*glp` is a pointer to a node, the pointer is dereferenced using `xdr_reference()`. Note that the third parameter is truly the size of a node (data values plus `nxt` pointer), while `xdr_gnumbers()` only (de)serializes the data values. We can get away with this tricky optimization only because the `nxt` data comes after all legitimate external data.

The routine is buggy in the XDR_FREE case. The bug is that `xdr_reference()` will free the node `*glp`. Upon return the assignment `glp = &((*glp)->nxt)` cannot be guaranteed to work since `*glp` is no longer a legitimate node. The following is a rewrite that works in all cases. The hard part is to avoid dereferencing a pointer which has not been initialized or which has been freed.

```
bool_t
xdr_gnumbers_list(xdrs, glp)
        XDR *xdrs;
        gnumbers_list *glp;
{
        bool_t more_data;
        bool_t freeing;
        gnumbers_list *next;  /* the next value of glp */

        freeing = (xdrs->x_op == XDR_FREE);
        while (TRUE) {
                more_data = (*glp != (gnumbers_list)NULL);
                if (! xdr_bool(xdrs, &more_data))
                        return (FALSE);
                if (! more_data)
                        return (TRUE);  /* we are done */
                if (freeing)
                        next = &((*glp)->nxt);
                if (! xdr_reference(xdrs, glp, sizeof(struct gnnode),
                    xdr_gnumbers))
                        return (FALSE);
                glp = (freeing) ? next : &((*glp)->nxt);
        }
}
```

Note that this is the first example in this document that actually inspects the direction of the operation `xdrs->x_op`).(The claim is that the correct iterative implementation is still easier to understand or code than the recursive implementation. It is certainly more efficient with respect to C stack requirements.

Appendix A: The Record Marking Standard

A record is composed of one or more record fragments. A record fragment is a four-byte header followed by 0 to $2^{31}-1$ bytes of fragment data. The bytes encode an unsigned binary number; as with XDR integers, the byte order is from highest to lowest. The number encodes two values — a boolean that indicates whether the fragment is the last fragment of the record (bit value 1 implies the fragment is the last fragment), and a 31-bit unsigned binary value which is the length in bytes of the fragment's data. The boolean value is the high-order bit of the header; the length is the 31 low-order bits.

(Note that this record specification is `not` in XDR standard form and cannot be implemented using XDR primitives!)

Appendix B: Synopsis of XDR Routines

xdr_array()

```
xdr_array(xdrs, arrp, sizep, maxsize, elsize, elproc)
        XDR *xdrs;
        char **arrp;
        u_int *sizep, maxsize, elsize;
        xdrproc_t elproc;
```

A filter primitive that translates between arrays and their corresponding external representations. The parameter `arrp` is the address of the pointer to the array, while `sizep` is the address of the element count of the array; this element count cannot exceed `maxsize`. The parameter `elsize` is the `sizeof()` each of the array's elements, and `elproc` is an XDR filter that translates between the array elements' C form, and their external representation. This routine returns one if it succeeds, zero otherwise.

xdr_bool()

```
xdr_bool(xdrs, bp)
        XDR *xdrs;
        bool_t *bp;
```

A filter primitive that translates between booleans (C integers) and their external representations. When encoding data, this filter produces values of either one or zero. This routine returns one if it succeeds, zero otherwise.

xdr_bytes()

```
xdr_bytes(xdrs, sp, sizep, maxsize)
        XDR *xdrs;
        char **sp;
        u_int *sizep, maxsize;
```

A filter primitive that translates between counted byte strings and their external representations. The parameter `sp` is the address of the string pointer. The length of the string is located at address `sizep`; strings cannot be longer than `maxsize`. This routine returns one if it succeeds, zero otherwise.

xdr_destroy()

```
void
xdr_destroy(xdrs)
        XDR *xdrs;
```

A macro that invokes the destroy routine associated with the XDR stream, `xdrs`. Destruction usually involves freeing private data structures associated with the stream. Using `xdrs` after invoking `xdr_destroy()` is undefined.

xdr_double()

```
xdr_double(xdrs, dp)
        XDR *xdrs;
        double *dp;
```

A filter primitive that translates between C `double` precision numbers and their external representations. This routine returns one if it succeeds, zero otherwise.

xdr_enum()

```
xdr_enum(xdrs, ep)
        XDR *xdrs;
        enum_t *ep;
```

A filter primitive that translates between C `enum`s (actually integers) and their external representations. This routine returns one if it succeeds, zero otherwise.

xdr_float()

```
xdr_float(xdrs, fp)
        XDR *xdrs;
        float *fp;
```

A filter primitive that translates between C `float`s and their external representations. This routine returns one if it succeeds, zero otherwise.

xdr_getpos()

```
u_int
xdr_getpos(xdrs)
        XDR *xdrs;
```

A macro that invokes the get-position routine associated with the XDR stream, `xdrs`. The routine returns an unsigned integer, which indicates the position of the XDR byte stream. A desirable feature of XDR streams is that simple arithmetic works with this number, although the XDR stream instances need not guarantee this.

xdr_inline()

```
long *
xdr_inline(xdrs, len)
        XDR *xdrs;
        int len;
```

A macro that invokes the in-line routine associated with the XDR stream, `xdrs`. The routine returns a pointer to a contiguous piece of the stream's buffer; `len` is the byte length of the desired buffer. Note that the pointer is cast to `long *`. Warning: `xdr_inline()` may return 0 (NULL) if it cannot allocate a contiguous piece of a buffer. Therefore the behavior may vary among stream instances; it exists for the sake of efficiency.

xdr_int()

```
xdr_int(xdrs, ip)
        XDR *xdrs;
        int *ip;
```

A filter primitive that translates between C integers and their external representations. This routine returns one if it succeeds, zero otherwise.

xdr_long()

```
xdr_long(xdrs, lp)
        XDR *xdrs;
        long *lp;
```

A filter primitive that translates between C `long` integers and their external representations. This routine returns one if it succeeds, zero otherwise.

xdr_opaque()

```
xdr_opaque(xdrs, cp, cnt)
        XDR *xdrs;
        char *cp;
        u_int cnt;
```

A filter primitive that translates between fixed size opaque data and its external representation. The parameter `cp` is the address of the opaque object, and `cnt` is its size in bytes. This routine returns one if it succeeds, zero otherwise.

xdr_reference()

```
xdr_reference(xdrs, pp, size, proc)
        XDR *xdrs;
        char **pp;
        u_int size;
        xdrproc_t proc;
```

A primitive that provides pointer chasing within structures. The parameter `pp` is the address of the pointer; `size` is the `sizeof()` the structure that `*pp` points to; and `proc` is an XDR procedure that filters the structure between its C form and its external representation. This routine returns one if it succeeds, zero otherwise.

xdr_setpos()

```
xdr_setpos(xdrs, pos)
        XDR *xdrs;
        u_int pos;
```

A macro that invokes the set position routine associated with the XDR stream `xdrs`. The parameter `pos` is a position value obtained from `xdr_getpos()`. This routine returns one if the XDR stream could be repositioned, and zero otherwise. Warning: it is difficult to reposition some types of XDR streams, so this routine may fail with one type of stream and succeed with another.

xdr_short()

```
xdr_short(xdrs, sp)
        XDR *xdrs;
        short *sp;
```

A filter primitive that translates between C `short` integers and their external representations. This routine returns one if it succeeds, zero otherwise.

xdr_string()

```
xdr_string(xdrs, sp, maxsize)
        XDR *xdrs;
        char **sp;
        u_int maxsize;
```

A filter primitive that translates between C strings and their corresponding external representations. Strings cannot cannot be longer than `maxsize`. Note that `sp` is the address of the string's pointer. This routine returns one if it succeeds, zero otherwise.

xdr_u_int()

```
xdr_u_int(xdrs, up)
        XDR *xdrs;
        unsigned *up;
```

A filter primitive that translates between C `unsigned` integers and their external representations. This routine returns one if it succeeds, zero otherwise.

xdr_u_long()

```
xdr_u_long(xdrs, ulp)
        XDR *xdrs;
        unsigned long *ulp;
```

A filter primitive that translates between C `unsigned long` integers and their external representations. This routine returns one if it succeeds, zero otherwise.

xdr_u_short()

```
xdr_u_short(xdrs, usp)
        XDR *xdrs;
        unsigned short *usp;
```

A filter primitive that translates between C `unsigned short` integers and their external representations. This routine returns one if it succeeds, zero otherwise.

xdr_union()

```
xdr_union(xdrs, dscmp, unp, choices, dfault)
        XDR *xdrs;
        int *dscmp;
        char *unp;
        struct xdr_discrim *choices;
        xdrproc_t dfault;
```

A filter primitive that translates between a discriminated C `union` and its corresponding external representation. The parameter `dscmp` is the address of the union's discriminant, while `unp` in the address of the union. This routine returns one if it succeeds, zero otherwise.

xdr_void()

```
xdr_void()
```

This routine always returns one. It may be passed to RPC routines that require a function parameter, where nothing is to be done.

xdr_wrapstring()

```
xdr_wrapstring(xdrs, sp)
        XDR *xdrs;
        char **sp;
```

A primitive that calls `xdr_string(xdrs,sp,MAXUNSIGNED);` where MAXUNSIGNED is the maximum value of an unsigned integer. This is handy because the RPC package passes only two parameters XDR routines, whereas `xdr_string()`, one of the most frequently used primitives, requires three parameters. This routine returns one if it succeeds, zero otherwise.

xdrmem_create()

```
void
xdrmem_create(xdrs, addr, size, op)
        XDR *xdrs;
        char *addr;
        u_int size;
        enum xdr_op op;
```

This routine initializes the XDR stream object pointed to by `xdrs`. The stream's data is written to, or read from, a chunk of memory at location `addr` whose length is no more than `size` bytes long. The `op` determines the direction of the XDR stream (either XDR_ENCODE, XDR_DECODE, or XDR_FREE).

xdrrec_create()

```
void
xdrrec_create(xdrs, sendsize, recvsize, handle, readit, writeit)
        XDR *xdrs;
        u_int sendsize, recvsize;
        char *handle;
        int (*readit)(), (*writeit)();
```

This routine initializes the XDR stream object pointed to by `xdrs`. The stream's data is written to a buffer of size `sendsize`; a value of zero indicates the system should use a suitable default. The stream's data is read from a buffer of size `recvsize`; it too can be set to a suitable default by passing a zero value. When a stream's output buffer is full, `writeit()` is called. Similarly, when a stream's input buffer is empty, `readit()` is called. The behavior of these two routines is similar to the UNIX system calls `read` and `write`, except that `handle` is passed to the former routines as the first parameter. Note that the XDR stream's `op` field must be set by the caller. Warning: this XDR stream implements an intermediate record stream. Therefore there are additional bytes in the stream to provide record boundary information.

xdrrec_endofrecord()

```
xdrrec_endofrecord(xdrs, sendnow)
        XDR *xdrs;
        int sendnow;
```

This routine can be invoked only on streams created by `xdrrec_create()`. The data in the output buffer is marked as a completed record, and the output buffer is optionally written out if `sendnow` is non-zero. This routine returns one if it succeeds, zero otherwise.

xdrrec_eof()

```
xdrrec_eof(xdrs)
        XDR *xdrs;
        int empty;
```

This routine can be invoked only on streams created by `xdrrec_create()`. After consuming the rest of the current record in the stream, this routine returns one if the stream has no more input, zero otherwise.

xdrrec_skiprecord()

```
xdrrec_skiprecord(xdrs)
        XDR *xdrs;
```

This routine can be invoked only on streams created by `xdrrec_create()`. It tells the XDR implementation that the rest of the current record in the stream's input buffer should be discarded. This routine returns one if it succeeds, zero otherwise.

xdrstdio_create()

```
void
xdrstdio_create(xdrs, file, op)
        XDR *xdrs;
        FILE *file;
        enum xdr_op op;
```

This routine initializes the XDR stream object pointed to by `xdrs`. The XDR stream data is written to, or read from, the Standard I/O stream `file`. The parameter `op` determines the direction of the XDR stream (either XDR_ENCODE, XDR_DECODE, or XDR_FREE). Warning: the destroy routine associated with such XDR streams calls `fflush()` on the `file` stream, but never `fclose()`.