External Data Representation: Sun Technical Notes

This chapter contains technical notes on Sun's implementation of the External Data Representation (XDR) standard, a set of library routines that allow a C programmer to describe arbitrary data structures in a machinex-independent fashion. For a formal specification of the XDR standard, see the *External Data Representation Standard: Protocol Specification*. XDR 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.[1]

This chapter contains a short tutorial overview of the XDR library routines, a guide to accessing currently available XDR streams, and information on defining new streams and data types. XDR was designed to work across different languages, operating systems, and machine architectures. Most users (particularly RPC users) will only need the information in the *Number Filters*, *Floating Point Filters*, and *Enumeration Filters* sections. Programmers wishing to implement RPC and XDR on new machines will be interested in the rest of the chapter, as well as the *External Data Representaiton Standard: Protocol Specification*, which will be their primary reference.

Note: *rpcgen can be used to write XDR routines even in cases where no RPC calls are being made.*

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.

```
example% cc program.c
```

1. 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);
          }
     }
     exit(0);
}
```

and *reader*:

[1] For a compete specification of the system External Data Representation routines, see the *xdr(3N)* manual page.

```
#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");
    exit(0);
}
```

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 4.2BSD 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 rpc */

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);
          }
     }
     exit(0);
}
```

and *reader*:

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

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");
     exit(0);
}
```

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%
```

Note: *Integers are 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. And pointers, which are very convenient to use,* *have no meaning outside the machine where they are defined.*

2. A Canonical Standard

XDR's approach to standardizing data representations is *canonical*. That is, XDR defines a single byte order (Big Endian), a single floating-point representation (IEEE), and so on. Any program running on any machine can use XDR to create portable data by translating its local representation to the XDR standard representations; similarly, any program running on any machine can read portable data by translating the XDR standard representaions to its local equivalents. The single standard completely decouples programs that create or send portable data from those that use or receive portable data. The advent of a new machine or a new language has no effect upon the community of existing portable data creators and users. A new machine joins this community by being "taught" how to convert the standard representations and its local representations; the local representations of other machines are irrelevant. Conversely, to existing programs running on other machines, the local representations of the new machine are also irrelevant; such programs can immediately read portable data produced by the new machine because such data conforms to the canonical standards that they already understand.

There are strong precedents for XDR's canonical approach. For example, TCP/IP, UDP/IP, XNS, Ethernet, and, indeed, all protocols below layer five of the ISO model, are canonical protocols. The advantage of any canonical approach is simplicity; in the case of XDR, a single set of conversion routines is written once and is never touched again. The canonical approach has a disadvantage, but it is unimportant in real-world data transfer applications. Suppose two Little-Endian machines are transferring integers according to the XDR standard. The sending machine converts the integers from Little-Endian byte order to XDR (Big-Endian) byte order; the receiving machine performs the reverse conversion. Because both machines observe the same byte order, their conversions are unnecessary. The point, however, is not necessity, but cost as compared to the alternative.

The time spent converting to and from a canonical representation is insignificant, especially in networking applications. Most of the time required to prepare a data structure for transfer is not spent in conversion but in traversing the elements of the data structure. To transmit a tree, for example, each leaf must be visited and each element in a leaf record must be copied to a buffer and aligned there; storage for the leaf may have to be deallocated as well. Similarly, to receive a tree, storage must be allocated for each leaf, data must be moved from the buffer to the leaf and properly aligned, and pointers must be constructed to link the leaves together. Every machine pays the cost of traversing and copying data structures whether or not conversion is required. In networking applications, communications overhead—the time required to move the data down through the sender's protocol layers, across the network and up through the receiver's protocol layers—dwarfs conversion overhead.

3. The XDR Library

The XDR library not only solves data portability problems, it also allows you to write and read arbitrary C constructs in a consistent, specified, well-documented manner. Thus, it can make 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 users never need to create XDR streams; the RPC system itself creates these streams, which are then passed to the users.

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, xp)
     XDR *xdrs;
     xxx *xp;
{
}
```

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, the user can obtain the direction of the XDR operation. See the *XDR Operation Directions* section below 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
```

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.

4. 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*>.

4.1. Number Filters

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

```
[signed, unsigned] * [short, int, long]
```

Specifically, the eight primitives are:

```
bool_t xdr_char(xdrs, cp)
    XDR *xdrs;
    char *cp;

bool_t xdr_u_char(xdrs, ucp)
    XDR *xdrs;
    unsigned char *ucp;

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.

4.2. 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. Both 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.

4.3. 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 *TRUE* (1) or *FALSE* (0).

```
#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*.

4.4. 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 */
```

4.5. 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*.

4.5.1. 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.

Keep *maxlength* small. If it is too big you can blow the heap, since *xdr_string()* will call *malloc()* for space.

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 the string 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 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.

4.5.2. 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.

4.5.3. 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, elementsiz, xdr_element)
    XDR *xdrs;
    char **ap;
    u_int *lp;
    u_int maxlength;
    u_int elementsiz;
    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; *elementsiz* is the byte size of each element of the array (the C function *sizeof()* can be used to obtain this value). The *xdr_element()* routine is called to serialize, deserialize, or free each element of the array.

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 the *gethostname* man page; (b) the user's UID: see the *geteuid* man page; and (c) the group numbers to which the user belongs: see the *getgroups* man page. 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 < 256 chars */
#define NGRPS 20     /* user can't be in > 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 cannot be > 1000 chars */
#define NARGC 100    /* commands cannot have > 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.

4.5.4. 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 *xdr_opaque()* primitive 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.

4.5.5. Fixed Sized Arrays

The XDR library provides a primitive, *xdr_vector*(), for fixed-length arrays.

```
#define NLEN 255      /* machine names must be < 256 chars */
#define NGRPS 20      /* user belongs to exactly 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);
     if (!xdr_vector(xdrs, nup->nu_gids, NGRPS, sizeof(int),
          xdr_int)) {
               return(FALSE);
     }
     return(TRUE);
}
```

4.5.6. 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 ordered 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;    /* 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 above in *The XDR Library* section. *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 may legally 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.

4.5.7. Pointers

In C it is often convenient to put pointers to another structure within a structure. The *xdr_reference()* primitive 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);
}
```

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 serializing data will most likely cause a memory fault and, on the UNIX system, a core dump.

xdr_pointer() correctly handles *NULL* pointers. For more information about its use, see the *Linked Lists* topics below.

Exercise: After reading the section on *Linked Lists*, 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. That is, implement *xdr_pointer*().

4.6. 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.

4.7. 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 the *Linked Lists* topic below, demonstrates the usefulness of the *xdrs->x_op* field.

4.8. 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.

4.8.1. 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 part of rpc  */

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.

4.8.2. 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.

4.8.3. 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 part of rpc */

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 the *Advanced Topics* topic below. 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.

4.9. XDR Stream Implementation

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

4.9.1. 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 added param */
    struct xdr_ops {
        bool_t  (*x_getlong)();     /* get long from stream */
        bool_t  (*x_putlong)();     /* put long to stream */
        bool_t  (*x_getbytes)();    /* get bytes from stream */
        bool_t  (*x_putbytes)();    /* put bytes to stream */
        u_int   (*x_getpostn)();    /* return stream offset */
        bool_t  (*x_setpostn)();    /* reposition offset */
        caddr_t (*x_inline)();      /* ptr to buffered data */
        VOID    (*x_destroy)();     /* free private area */
    } *x_ops;
    caddr_t     x_public;           /* users' data */
    caddr_t     x_private;          /* pointer to private data */
    caddr_t     x_base;             /* private 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. *x_getpostn()*, *x_setpostn()* and *x_destroy()* are macros for accessing operations. 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. 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.

5. 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. The *External Data Representation Standard: Protocol Specification* describes this language in complete detail.

5.1. Linked Lists

The last example in the *Pointers* topic earlier in this chapter presented a C data structure and its associated XDR routines for an individual'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:

```
struct gnumbers_node {
    struct gnumbers gn_numbers;
    struct gnumbers_node *gn_next;
};

typedef struct gnumbers_node *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 *gn_next* field is used to indicate whether or not the object has terminated. Unfortunately, if the object continues, the *gn_next* 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 declaration of *gnumbers_list*:

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

struct gnumbers_node {
    gnumbers gn_numbers;
    gnumbers_node *gn_next;
};
```

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*. Note that the C declaration has no boolean explicitly declared in it (though the *gn_next* field implicitly carries the information), while the XDR data description has no pointer explicitly declared in it.

Hints for writing the XDR routines for a *gnumbers_list* follow easily from the XDR description above. Note how the primitive *xdr_pointer()* is used to implement the XDR union above.

```
bool_t
xdr_gnumbers_node(xdrs, gn)
    XDR *xdrs;
    gnumbers_node *gn;
{
    return(xdr_gnumbers(xdrs, &gn->gn_numbers) &&
        xdr_gnumbers_list(xdrs, &gp->gn_next));
}

bool_t
xdr_gnumbers_list(xdrs, gnp)
    XDR *xdrs;
    gnumbers_list *gnp;
{
    return(xdr_pointer(xdrs, gnp,
        sizeof(struct gnumbers_node),
        xdr_gnumbers_node));
}
```

The unfortunate side effect of XDR'ing a list with these routines is that the C stack grows linearly with respect to the number of node in the list. This is due to the recursion. The following routine collapses the above two mutually recursive into a single, non-recursive one.

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

     for (;;) {
          more_data = (*gnp != NULL);
          if (!xdr_bool(xdrs, &more_data)) {
               return(FALSE);
          }
          if (! more_data) {
               break;
          }
          if (xdrs->x_op == XDR_FREE) {
               nextp = &(*gnp)->gn_next;
          }
          if (!xdr_reference(xdrs, gnp,
               sizeof(struct gnumbers_node), xdr_gnumbers)) {

          return(FALSE);
          }
          gnp = (xdrs->x_op == XDR_FREE) ?
               nextp : &(*gnp)->gn_next;
     }
     *gnp = NULL;
     return(TRUE);
}
```

The first task is to find out whether there is more data or not, so that this boolean information can be serialized. Notice that this statement is unnecessary in the *XDR_DECODE* case, since the value of more_data is not known until we deserialize it in the next statement.

The next statement XDR's the more_data field of the XDR union. Then if there is truly no more data, we set this last pointer to *NULL* to indicate the end of the list, and return *TRUE* because we are done. Note that setting the pointer to *NULL* is only important in the *XDR_DECODE* case, since it is already *NULL* in the *XDR_ENCODE* and XDR_FREE cases.

Next, if the direction is *XDR_FREE*, the value of *nextp* is set to indicate the location of the next pointer in the list. We do this now because we need to dereference gnp to find the location of the next item in the list, and after the next statement the storage pointed to by *gnp* will be freed up and no be longer valid. We can't do this for all directions though, because in the *XDR_DECODE* direction the value of *gnp* won't be set until the next statement.

Next, we XDR the data in the node using the primitive *xdr_reference*(). *xdr_reference()* is like *xdr_pointer()* which we used before, but it does not send over the boolean indicating whether there is more data. We use it instead of *xdr_pointer()* because we have already XDR'd this information ourselves. Notice that the xdr routine passed is not the same type as an element in the list. The routine passed is *xdr_gnumbers*(), for XDR'ing gnumbers, but each element in the list is actually of type *gnumbers_node*. We don't pass *xdr_gnumbers_node()* because it is recursive, and instead use *xdr_gnumbers()* which XDR's all of the non-recursive part. Note that this trick will work only if the *gn_numbers* field is the first item in each element, so that their addresses are identical when passed to *xdr_reference*().

Finally, we update *gnp* to point to the next item in the list. If the direction is *XDR_FREE*, we set it to the previously saved value, otherwise we can dereference *gnp* to get the proper value. Though harder to understand than the recursive version, this non-recursive routine is far less likely to blow the C stack. It will also run more efficiently since a lot of procedure call overhead has been removed. Most lists are small though (in the hundreds of items or less) and the recursive version should be sufficient for them.