Synopsis of RPC and XDR Routines [A]

This appendix summarizes the entry points into the RPC and XDR system.

This macro destroys the authentication information associated with `auth`. Destruction usually involves deallocation of private data structures. The use of `auth` is undefined after `auth_destroy` is called.

`auth_destroy` Format:

```
void
auth_destroy(auth)
    AUTH *auth;
```

`authdes_create` This routine creates and returns an RPC authentication handle that contains the following DES authentication information:

Format:

```
AUTH * authdes_create(netname, window, syncaddr, deskeyp)
    char *netname
    unsigned window;
    struct sockaddr_in *syncaddr;
    des_block *deskeyp;
```

The `netname` parameter is the network name of the server process owner. If the server process is a root process, you can derive the name by using the following declaration and call (argument type is character pointer):

```
char netname [MAXNETNAMELEN];
host2netname(servername,rhostname,NULL);
```

`rhostname` is the host name of the machine on which the server process (`servername`) is running.

`NULL` specifies that the local domain name will be used.

If a user runs the server process, you can derive the name by using the following declaration and call:

```
char netname [MAXNETNAMELEN];
user2netname(servername,uid,NULL);
```

`uid` is the user ID of the user whose server name you are requesting.

The `window` parameter is the lifetime (in seconds) for the credential. You can use a credential only once within the lifetime set by this parameter. The argument type is an unsigned integer.

The `syncaddr` parameter is the network address of the host with which the client must synchronize. Both client and server must be using the same time. If you are sure that the client and server are already synchronized (if, for example, both client and server are running the Network Time Protocol (NTP)), you can specify this argument as `NULL`. The argument type is pointer to the `sockaddr_in` structure (`sockaddr_in*`).

The `deskeyp` parameter is the address of a DES encryption key to use for encrypting time stamps and data. `NULL` indicates that you should choose a random key. The `ah_key` field of the authentication handle contains the encryption key. The argument type is a pointer to the `des_key` structure (`des_key*`).

`authkerb_seccreate`

This client side routine returns an RPC authentication handle that enables the use of the Kerberos authentication system. If the `authkerb_seccreate` routine fails it returns `NULL`. For more information see the `kerberos_rpc`(3) man page.

Format:

```
AUTH *
authunix_seccreate(service, srv_inst, realm
window, timehost, status)
    char *service ;
    char *srv_inst;
    char *realm
    u_int window;
    char *timehost;
    int status;
```

The `service` parameter is the Kerberos principal name of the service to be used.

The `srv_inst` parameter is the instance of the service to be called.

The `window` parameter validates the client credential, with time measured in seconds. The `ntpd`(8) daemon provides this function on a Cray Research machine.

The `timehost` parameter is optional and does nothing.

The `status` parameter is also optional. If you specify `status`, it is used to return a Kerberos error status code if an error occurs.

`authnone_create` This routine creates and returns an RPC authentication handle that passes no usable authentication information with each RCP.

Format:

```
AUTH *
authnone_create()
```

`authunix_create` This routine creates and returns an RPC authentication handle that contains UNICOS authentication information.

Format:

```
AUTH *
authunix_create(host, uid, gid, len, aup_gids)
    char *host;
    int uid, gid, len, *aup_gids;
```

The `host` parameter is the name of the machine on which the information was created. The `uid` parameter is the user's ID. The `gid` parameter is the user's current group ID. The `len` and `aup_gids` parameters are counted arrays of groups to which the user belongs.

`authunix_create_default` This routine calls `authunix_create` with the default parameters.

Format:

```
AUTH *
authunix_create_default()
```

`callrpc` This routine calls the remote procedure associated with the program number (`prognum`), version number (`versnum`), and procedure number (`procnum`) on the machine, `host`.

Format:

```
callrpc(host,prognum,versnum,procnum,inproc,in,outproc,out)
    char *host;
    u_long prognum, versnum, procnum;
    char *in, *out;
    xdrproc_t inproc, outproc;
```

The `inproc` parameter encodes the procedure's parameters, and the `in` parameter is the address of the procedure's arguments. The `outproc` parameter decodes the procedure's results, and the `out` parameter is the address of the destination location for the results.

If it succeeds, this routine returns 0; if it fails, it returns the value of enumeration `clnt_stat`, cast to an integer. The `clnt_perrno` routine is handy for translating failure statuses into messages.

> **Note:** Calling remote procedures with this routine uses UDP/IP as a transport; see `clntudp_create`, page 74, for restrictions.

`clnt_broadcast` This routine is like `callrpc`, except that the call message is broadcast to all locally connected broadcast nets. Each time it receives a response, this routine calls `eachresult()`, which has the following form:

```
eachresult(out, addr)
    char *out;
    struct sockaddr_in *addr;
```

The `out` parameter is the same as `out` passed to `clnt_broadcast`, except that the remote procedure's output is decoded there; `addr` points to the address of the machine that sent the results.

Format:

```
enum clnt_stat
clnt_broadcast(prognum, versnum, procnum, inproc, in, outproc, out,
eachresult)
    u_long prognum, versnum, procnum;
    char *in, *out;
    xdrproc_t inproc, outproc;
    resultproc_t eachresult;
```

If `eachresult()` returns 0, `clnt_broadcast` waits for more replies; otherwise, it returns with appropriate status.

`clnt_call` This macro calls the remote procedure `procnum` associated with the client handle, `clnt`, which is obtained with an RPC client creation routine such as `clntudp_create`.

Format:

```
enum clnt_stat
clnt_call(clnt, procnum, inproc, in, outproc, out, tout)
    CLIENT *clnt; long procnum;
    xdrproc_t inproc, outproc;
    char *in, *out;
    struct timeval tout;
```

`clnt` is the client handle, `procnum` is the procedure number, `inproc` encodes the procedure's parameters, `in` is the address of the procedure's arguments, `outproc` decodes the procedure's results, `out` is the address of the destination location for the results, and `tout` is the time allowed for results to return.

`clnt_create` This routine returns a pointer to a `CLIENT` structure. It allows users to pass the host name and protocol type as parameters of type character pointer.

Format:

```
struct CLIENT *cp;

char *hostname;
unsigned int prog;
unsigned int vers;
char *protocol;
cp=clnt_create (hostname,prog,vers,protocol);
```

`clnt_destroy` This macro destroys the client's RPC handle (`clnt`). Destruction usually involves deallocation of private data structures, including `clnt` itself. Use of `clnt` is undefined after `clnt_destroy` is called. The user must close sockets associated with `clnt`.

Format:

```
clnt_destroy(clnt)
    CLIENT *clnt;
```

`clnt_freeres`

This macro frees any data allocated by the RPC and XDR system when it decoded the results of an RPC.

Format:

```
clnt_freeres(clnt, outproc, out)
   CLIENT *clnt;
   xdrproc_t outproc;
   char *out;
```

`clnt` is the client handle, `outproc` is the XDR routine that describes the results in simple primitives, and `out` is the address of the results. If the results were successfully freed, this routine returns 1; otherwise, it returns 0.

`clnt_geterr`

This macro copies the error structure out of the client handle (`clnt`) to the structure at address `errp`.

Format:

```
void
clnt_geterr(clnt, errp)
    CLIENT *clnt;
    struct rpc_err *errp;
```

`clnt_pcreateerror`

This routine prints a message to standard error that indicates why a client RPC handle could not be created. The message is prepended with string `s` and a colon. This routine is used after a `clntraw_create`, `clnttcp_create`, or `clntudp_create` call.

Format:

```
void
clnt_pcreateerror(s)
    char *s;
```

`clnt_perrno`

This routine prints a message to standard error that corresponds to the condition indicated by `stat`. This routine is used after `callrpc`.

Format:

```
void
clnt_perrno(stat)
    enum clnt_stat stat;
```

`clnt_perror` This routine prints a message to standard error that indicates the reason an RPC failed; `clnt` is the handle used to do the call. The message is prepended with string `s` and a colon. This routine is used after `clnt_call`.

```
clnt_perror(clnt, s)
   CLIENT *clnt;
   char *s;
```

`clntraw_create` This routine creates a trivial RPC client for the remote program `prognum`, version `versnum`.

Format:

```
CLIENT *
clntraw_create(prognum, versnum)
    u_long prognum, versnum;
```

The transport used to pass messages to the service is actually a buffer within the process address space; therefore, the corresponding RPC server should reside in the same address space (see `svcraw_create`, page 82), allowing simulation of RPC and acquisition of RPC overheads, such as round-trip times, without interference from the kernel. If the procedure fails, this routine returns `NULL`.

`clnttcp_create` This routine creates an RPC client for the remote program `prognum`, version `versnum`; the client uses TCP/IP as a transport.

Format:

```
CLIENT *
clnttcp_create(addr,prognum,versnum,sockp,sendsz,recvsz)
    struct sockaddr_in *addr;
    u_long prognum, versnum;
    int *sockp;
    u_int sendsz, recvsz;
```

The remote program is located at Internet address `addr`. If `addr->sin_port` is 0, the portmapper on the host at IP address `addr` is set to the actual port on which the remote program is listening (the remote portmap service is consulted for this information). The `sockp` parameter is a socket; if it is `RPC_ANYSOCK`, this routine opens a new socket and sets `sockp`. Because the RPC that is based on TCP uses buffered I/O, you can specify the size of the send and receive buffers by using the `sendsz` and `recvsz` parameters, respectively; values of 0 indicate that suitable defaults will be chosen. If the procedure fails, this routine returns `NULL`.

`clntudp_create`

This routine creates an RPC client for the remote program `prognum`, version `versnum`; the client uses UDP/IP as a transport.

Note: On systems that limit UDP datagrams to 8 Kbytes of data, you cannot use this transport for procedures that accept large arguments or return large results.

Format:

```
CLIENT *
clntudp_create(addr, prognum, versnum, wait, sockp)
    struct sockaddr_in *addr;
    u_long prognum, versnum;
    struct timeval wait;
    int *sockp;
```

The remote program is located at Internet address `addr`. If `addr->sin_port` is 0, the portmap on the host at IP address `addr` is set to the actual port on which the remote program is listening (the remote portmapper service is consulted for this information). The `sockp` parameter is a socket; if it is `RPC_ANYSOCK`, this routine opens a new socket and sets `sockp`. The UDP transport resends the call message in intervals of `wait` time until a response is received or the call times out. `clnt_call` specifies the total time for the call to time out.

`get_myaddress`

This routine puts the machine's IP address into `addr`, without consulting the library routines that deal with `/etc/hosts`. The port number is always set to `htons(PMAPPORT)`.

Format:

```
void
get_myaddress(addr)
    struct sockaddr_in *addr;
```

`pmap_getmaps` This routine is a user interface to the portmap service. It returns a list of the current RPC program-to-port mappings on the host located at IP address `addr`. This routine can return `NULL`. This routine is used when using the `rpcinfo`(8) command with the `-p` option.

Format:

```
struct pmaplist *
pmap_getmaps(addr)
    struct sockaddr_in *addr;
```

`pmap_getport` This routine is a user interface to the portmap service. It returns the port number of a waiting service that supports the program at Internet address `addr`, with program number `prognum`, version `versnum`, and the transport protocol associated with `protocol`.

A return value of 0 means that the mapping does not exist or that the RPC system failed to contact the remote portmap service. In the latter case, the global variable `rpc_createerr` contains the RPC status.

Format:

```
u_short
pmap_getport(addr, prognum, versnum, protocol)
    struct sockaddr_in *addr;
    u_long prognum, versnum, protocol;
```

`pmap_rmtcall` This routine is a user interface to the portmap service.

Format:

```
enum clnt_stat
pmap_rmtcall(addr, prognum, versnum, procnum, inproc, in, outproc, out,
tout, portp)
    u_long prognum, versnum, procnum;
    char *in, *out;
    xdrproc_t inproc, outproc;
    struct timeval tout;
    u_long *portp;
```

This routine instructs the portmapper on the host at IP address `*addr` to make an RPC on your behalf to a procedure on that host. If the procedure succeeds, the `*portp` parameter is changed to the program's port number. The definitions of other parameters are discussed in descriptions of `callrpc`, page 69, and `clnt_call`, page 71. You should use this procedure only in conjunction with a `ping`(8) command. See also `clnt_broadcast`, page 70.

`pmap_set`

This routine is a user interface to the portmap service.

Format:

```
pmap_set(prognum, versnum, protocol, port)
    u_long prognum, versnum, protocol;
    u_short port;
```

It establishes a mapping between a program's [`prognum`, `versnum`, `protocol`] and a port (`port`) on a machine's portmap service. The value of `protocol` is most likely `IPPROTO_UDP` or `IPPROTO_TCP`. If the program succeeds, routine `svc_register` automatically returns 1; otherwise, it returns 0.

`pmap_unset`

This routine is a user interface to the portmap service.

Format:

```
pmap_unset(prognum, versnum)
    u_long prognum, versnum;
```

This routine destroys all mappings between [`prognum`,`versnum`,`*`] and ports on the machine's portmap service. If the program succeeds, this routine automatically returns 1; otherwise, it returns 0.

`registerrpc`

This routine registers procedure `procname` with the RPC service package.

> **Note:** Remote procedures registered in this form are accessed by using the UDP/IP transport; see `svcudp_create`, page 83.

Format:

```
registerrpc(prognum,versnum,procnum,procname,inproc,outproc)
    u_long prognum, versnum, procnum;
    char *(*procname)();
    xdrproc_t inproc, outproc;
```

If a request arrives for program `prognum`, version `versnum`, and procedure `procnum`, `procname` is called with a pointer to its parameters; `procname` should return a pointer to its static results. `inproc` decodes the parameters; `outproc` encodes the results. If the registration succeeds, this routine automatically returns 0; otherwise, it returns –1.

`rpc_createerr`

This routine is a global variable whose value is set by any RPC client creation routine that does not succeed. Use the `clnt_pcreateerror` routine to print the reason for the failure.

Format:

```
struct rpc_createerr  rpc_createerr;
```

`svc_destroy`

This macro destroys the RPC service transport handle, `xprt`. Destruction usually involves deallocation of private data structures, including `xprt` itself. Use of `xprt` is undefined after this routine is called.

Format:

```
svc_destroy(xprt)
    SVCXPRT *xprt;
```

`svc_freeargs`

This macro frees any data allocated by RPC and XDR when it used `svc_getargs` to decode the arguments to a service procedure. The parameters are those used on the `svc_getargs` macro call. If the results were successfully freed, this routine returns 1; otherwise, it returns 0.

Format:

```
svc_freeargs(xprt, inproc, in)
    SVCXPRT *xprt;
    xdrproc_t inproc;
    char *in;
```

`svc_getargs`

This macro decodes the arguments of an RPC request associated with the RPC service transport handle (`xprt`).

Format:

```
svc_getargs(xprt, inproc, in)
    SVCXPRT *xprt;
    xdrproc_t inproc;
    char *in;
```

`inproc` is the XDR routine used to decode the arguments, and `in` is the address at which the arguments will be placed. If decoding succeeds, this routine returns 1; otherwise, it returns 0.

`svc_getcaller` This routine is the approved way of getting the network address of the caller of a procedure associated with the RPC service transport handle (`xprt`)

Format:

```
struct sockaddr_in
svc_getcaller(xprt)
    SVCXPRT *xprt;
```

`svc_getreq` This routine is similar to `svc_getreqset()`, but it is limited to 64 descriptors.

This routine is similar to `svc_getreqset()`, but it is limited to 64 descriptors.

Format:

```
void
svc_getreq(rdfds)
    int rdfds;
```

`rdfds` is the read file descriptors bit mask.

`svc_getreqset` This routine is of interest only if a service implementer does not call `svc_run`, but instead implements custom asynchronous event processing. It is called when the `select`(2) system call has determined that an RPC request has arrived on some RPC sockets.

Format:

```
svc_getreqset(rdfdsetp)
    fd_set *rdfdsetp;
```

`rdfdsetp` is a pointer to the resultant read file descriptor bit mask. The routine returns when all sockets associated with the value of `rdfdsetp` have been serviced.

The global variable `svc_fdset`, which is of type `fd_set`, reflects the RPC service side's read file descriptor bit mask; it is suitable as a parameter to the `select`(2) system call. This is of interest only if a service implementer does not call `svc_run`, but rather does his or her own asynchronous event processing. This variable is read-only (do not pass its address to `select`(2)), yet it can change after calls to `svc_getreqset` or any creation routines. Its format is as follows:

```
fd_set   svc_fdset;
```

`svc_register` This routine associates `prognum` and `versnum` with the service dispatch procedure, `dispatch()`.

Format:

```
svc_register(xprt, prognum, versnum, dispatch, protocol)
    SVCXPRT *xprt;
    u_long prognum, versnum;
    void (*dispatch)();
    int protocol;
```

If `protocol` is 0, the service is not registered with the portmap service. If `protocol` is a nonzero value, a mapping of [`prognum`,`versnum`,`protocol`] to `xprt->xp_port` is established with the local portmap service (generally protocol is 0, `IPPROTO_UDP`, or `IPPROTO_TCP`). `xprt` is the RPC service transport handle. The `dispatch()` procedure has the following form:

```
dispatch(request, xprt)
     struct svc_req *request;
     SVCXPRT *xprt;
```

If `dispatch()` succeeds, the `svc_register` routine returns 1; otherwise, it returns 0.

`svc_run` This routine never returns. It waits for RPC requests to arrive, and it calls the appropriate service procedure through `svc_getreqset` when one arrives. This procedure is usually waiting for a `select`(2) system call to return.

Format:

```
svc_run()
```

`svc_sendreply` An RPC service's dispatch routine calls this routine to send the results of an RPC.

Format:

```
svc_sendreply(xprt, outproc, out)
    SVCXPRT *xprt;
    xdrproc_t outproc;
    char *out;
```

`xprt` is the caller's associated transport handle, `outproc` is the XDR routine that encodes the results, and `out` is the address of the results. If the procedure succeeds, this routine returns 1; otherwise, it returns 0.

`svc_unregister`

This routine removes all mapping of [`prognum`,`versnum`] to dispatch routines, and all mapping of [`prognum`,`versnum`,*] to port numbers.

Format:

```
void
svc_unregister(prognum, versnum)
    u_long prognum, versnum;
```

`svcerr_auth`

A service dispatch routine that refuses to perform an RPC because of an authentication error calls this routine.

Format:

```
void
svcerr_auth(xprt, why)
    SVCXPRT *xprt;
    enum auth_stat why;
```

`xprt` is the RPC service transport handle. `why` indicates the reason the service dispatch routine is refusing to perform the RPC.

`svcerr_decode`

A service dispatch routine that cannot successfully decode its parameters calls this routine.

Format:

```
void
svcerr_decode(xprt)
    SVCXPRT *xprt;
```

`xprt` is the RPC service transport handle. See also `svc_getargs`, page 77.

`svcerr_noproc`

A service dispatch routine that does not implement the procedure number the caller requests calls this routine.

Format:

```
void
svcerr_noproc(xprt)
    SVCXPRT *xprt;
```

`xprt` is the RPC service transport handle.

`svcerr_noprog`

This routine is called when the specified program is not registered with the RPC package.

Format:

```
void
svcerr_noprog(xprt)
    SVCXPRT *xprt;
```

`xprt` is the RPC service transport handle. Service implementers usually do not need this routine.

`svcerr_progvers`

This routine is called when the desired version of a program is not registered with the RPC package.

Format:

```
void
svcerr_progvers(xprt)
    SVCXPRT *xprt;
```

`xprt` is the RPC service transport handle. Service implementers usually do not need this routine.

`svcerr_systemerr`

A service dispatch routine calls this routine when the service dispatch routine detects a system error not covered by any particular protocol. For example, if a service can no longer allocate storage, it might call this routine.

Format:

```
void
svcerr_systemerr(xprt)
    SVCXPRT *xprt;
```

`xprt` is the RPC service transport handle.

`svcerr_weakauth`

A service dispatch routine that refuses to perform an RPC because of insufficient (but correct) authentication parameters calls this routine.

Format:

```
void
svcerr_weakauth(xprt)
    SVCXPRT *xprt;
```

`xprt` is the RPC service transport handle. The routine calls `svcerr_auth(xprt,AUTH_TOOWEAK)`.

`svcraw_create`

This routine creates a trivial RPC service transport to which it returns a pointer. The transport is really a buffer within the process address space; therefore, the corresponding RPC client should reside in the same address space (see `clntraw_create()`, page 73.

Format:

```
SVCXPRT *
svcraw_create()
```

This routine allows simulation of RPC and acquisition of RPC overheads (such as round-trip times), without any kernel interference. If the procedure fails, this routine returns `NULL`.

`svctcp_create`

This routine creates an RPC service transport based on TCP/IP, to which it returns a pointer.

Format:

```
SVCXPRT *
svctcp_create(sock, send_buf_size, recv_buf_size)
    int sock;
    u_int send_buf_size, recv_buf_size;
```

The transport is associated with the socket `sock`; `sock` can be `RPC_ANYSOCK`, in which case a new socket is created. If the socket is not bound to a local TCP port, this routine binds it to an arbitrary port. Because an RPC that is based on TCP/IP uses buffered I/O, you can specify the size of the send (`send_buf_size`) and receive (`recv_buf_size`) buffers; values of 0 indicate that suitable defaults will be chosen. On completion, the `xp_sock` field of the created `SVCXPRT` structure is the transport's socket number, and the `xp_port` field of the created `SVCXPRT` structure is the transport's port number. If the procedure fails, this routine returns `NULL`.

`svcudp_create`

This routine creates an RPC service transport based on UDP/IP, to which it returns a pointer.

Note: On systems that can hold only up to 8 Kbytes of encoded data, you cannot use this transport for procedures that accept large arguments or return large results.

Format:

```
SVCXPRT *
svcudp_create(sock)
    int sock;
```

The transport is associated with the socket `sock`; `sock` can be `RPC_ANYSOCK`, in which case a new socket is created. If the socket is not bound to a local UDP port, this routine binds it to an arbitrary port. On completion, the `xp_sock` field of the created `SVCXPRT` structure is the transport's socket number, and the `xp_port` field of the created `SVCXPRT` structure is the transport's port number. If the routine fails, it returns `NULL`.

`xdr_accepted_reply`

This routine is used for describing RPC messages externally. It is useful for users who want to generate messages in the RPC style without using the RPC package.

Format:

```
xdr_accepted_reply(xdrs, ar)
    XDR *xdrs;
    struct accepted_reply *ar;
```

`xdrs` is the XDR stream, and `ar` points to the structure that contains the reply structure.

`xdr_array`

This routine is a filter primitive that translates between arrays and their corresponding external representations.

Format:

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

`xdrs` is the XDR stream. `arrp` is the address of the pointer to the array. `sizep` is the address of the element count of the array; this element count cannot exceed `maxsize`. The `elsize` parameter is the size (in bytes) of each of the array's elements, and `elproc` is an XDR filter that translates between the C form of the array elements and their external representation. If the routine succeeds, it returns 1; otherwise, it returns 0.

`xdr_authdes_cred`

This routine serializes/deserializes an `authdes_cred` structure. The client side uses this procedure to serialize a credential structure to be passed to the server. The server side uses this procedure to deserialize an `authdes_cred` structure from a client.

Format:

```
bool_t
xdr_authdes_cred(xdrs, cred)
    XDR *xdrs;
    struct authdes_cred *cred;
```

`xdrs` is the XDR stream. `cred` is the `authdes_cred` structure.

`xdr_authdes_verf`

This routine serializes/deserializes an `authdes_verf` structure. The client side uses it to deserialize a verification structure from the server. The server may use the routine to serialize a verification structure to be passed to the client.

Format:

```
bool_t
xdr_authdes_verf(xdrs, verf)
    register XDR *xdrs;
    register struct authdes_verf *verf;
```

`xdrs` is the XDR stream. `verf` points to a DES authentication verifier.

`xdr_authunix_parms`

This routine describes UNICOS credentials externally. It is useful for users who want to generate these credentials without using the RPC authentication package.

Format:

```
xdr_authunix_parms(xdrs, aupp)
    XDR *xdrs;
    struct authunix_parms *aupp;
```

`xdrs` is the XDR stream, and `aupp` points to the structure that contains the UNICOS authentication parameters.

`xdr_bool` This routine is a filter primitive that translates between Booleans (C integers) specified by `bp` and their external representations (`xdrs`). When encoding data, this filter produces values of either 1 or 0. If the routine succeeds, it returns 1; otherwise, it returns 0.

Format:

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

`xdr_bytes` This routine is a filter primitive that translates between counted byte strings and their external representations (`xdrs`).

Format:

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

`xdrs` is the XDR stream. `sp` is the address of the string pointer. The length of the string is located at address `sizep`; strings cannot be longer than `maxsize`. If the routine succeeds, it returns 1; otherwise, it returns 0.

`xdr_callhdr` This routine describes RPC headers associated with messages externally. It is useful for users who want to generate message headers in the RPC style without using the RPC package.

Format:

```
void
xdr_callhdr(xdrs, chdr)
    XDR *xdrs;
    struct rpc_msg *chdr;
```

`xdrs` is the XDR stream, and `chdr` points to the structure that contains the call header data.

`xdr_callmsg` This routine describes RPC messages externally. It is useful for users who want to generate messages in the RPC style without using the RPC package.

Format:

```
xdr_callmsg(xdrs, cmsg)
    XDR *xdrs;
    struct rpc_msg *cmsg;
```

`xdrs` is the XDR stream, and `cmsg` points to the structure that contains the call message data.

`xdr_char`

This routine is a filter primitive that translates between C characters (`cp`) and their external representations (`xdrs`). If the routine succeeds, it returns 1; otherwise, it returns 0.

Format:

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

`xdr_destroy`

This macro invokes the destroy routine associated with the XDR stream `xdrs`. Destruction usually involves freeing private data structures associated with the stream.

Format:

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

Using `xdrs` after `xdr_destroy` is invoked produces undefined results.

`xdr_double`

This routine is a filter primitive that translates between C double-precision numbers (`dp`) and their external representations (`xdrs`). If the routine succeeds, it returns 1; otherwise, it returns 0.

Format:

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

`xdr_enum`

This routine is a filter primitive that translates between C enumerations (actually integers) specified by `ep` and their external representations (`xdrs`). If the routine succeeds, it returns 1; otherwise, it returns 0.

Format:

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

`xdr_float`

This routine is a filter primitive that translates between C floating-point numbers (`fp`) and their external representations (`xdrs`). If the routine succeeds, it returns 1; otherwise, it returns 0.

Format:

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

`xdr_getpos`

This macro invokes the get-position routine associated with the XDR stream `xdrs`.

Format:

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

The routine returns an unsigned integer that 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 do not ensure this.

`xdr_inline`

This macro invokes the inline routine associated with the XDR stream `xdrs`.

Note: If the `xdr_inline` routine cannot allocate a contiguous piece of a buffer, it might return `NULL` (0). Therefore, the behavior can vary among stream instances; the routine exists for the sake of efficiency.

Format:

```
inline_t*
xdr_inline(xdrs, len)
    XDR *xdrs;
    int_len;
```

The routine returns a pointer to a contiguous piece of the stream's buffer; `len` is the byte length of the desired buffer. The pointer is cast to `inline_t*`, which is `char*` on Cray Research systems. The address returned is cast to `long` *.

`xdr_int`

This routine is a filter primitive that translates between C integers (`ip`) and their external representations (`xdrs`). If the routine succeeds, it returns 1; otherwise, it returns 0.

Format:

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

`xdr_long`

This routine is a filter primitive that translates between C long integers (`lp`) and their external representations (`xdrs`). If the routine succeeds, it returns 1; otherwise, it returns 0.

Format:

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

`xdr_opaque`

This routine is a filter primitive that translates between fixed-size opaque data and its external representation (`xdrs`).

Format:

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

`cp` is the address of the opaque object; `cnt` is its size (in bytes). If the routine succeeds, it returns 1; otherwise, it returns 0.

`xdr_opaque_auth`

This routine describes RPC messages externally. It is useful for users who want to generate messages in the RPC style without using the RPC package.

Format:

```
xdr_opaque_auth(xdrs, ap)
    XDR *xdrs;
    struct opaque_auth *ap;
```

`xdrs` is the XDR stream, and `ap` points to the opaque authentication structure.

`xdr_pmap`

This routine provides an external description of parameters to various portmap procedures. It is useful for users who want to generate these parameters without using the `pmap` interface.

Format:

```
xdr_pmap(xdrs, regs)
    XDR *xdrs;
    struct pmap *regs;
```

`xdrs` is the XDR stream, and `regs` points to the structure that contains registration information.

`xdr_pmaplist`

This routine describes a list of port mappings externally. It is useful for users who want to generate these parameters without using the `pmap` interface.

Format:

```
xdr_pmaplist(xdrs, rp)
    XDR *xdrs;
    struct pmaplist **rp;
```

`xdrs` is the XDR stream, and `rp` is a pointer to the array that will store the `portmap` map entries.

`xdr_pointer`

This routine translates a pointer to a possibly recursive data structure. It differs from `xdr_reference` in that it can serialize and deserialize trees correctly.

Format:

```
xdr_pointer(xdrs, objpp, obj_size, xdr_obj)
    XDR *xdrs;
    char **objpp;
    u_int obj_size;
    xdrproc_t xdr_obj;
```

`xdrs` is the XDR stream, `objpp` is the address of the pointer, `obj_size` is the size of the structure to which `objpp` points, and `xdr_obj` is a pointer to a structure for each data type that is to be encoded or decoded.

`xdr_reference`

This routine is a primitive that provides pointer-dereferencing within structures.

Format:

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

`pp` is the address of the pointer, `size` is the size (in bytes) of the structure to which `*pp` points, and `proc` is an XDR procedure that filters the structure between its C form and its external representation (`xdrs`). If the routine succeeds, it returns 1; otherwise, it returns 0.

`xdr_rejected_reply`

This routine describes RPC reject type messages externally. It is useful for users who want to generate error messages in the RPC style without using the RPC package.

Format:

```
xdr_rejected_reply(xdrs, rr)
    XDR *xdrs;
    struct rejected_reply *rr;
```

`xdrs` is the XDR stream, and `rr` points to the structure that contains the rejected reply information.

`xdr_replymsg`

This routine describes RPC accept type messages externally. It is useful for users who want to generate messages in the RPC style without using the RPC package.

Format:

```
xdr_replymsg(xdrs, rmsg)
    XDR *xdrs;
    struct_rpc_msg *rmsg;
```

`xdrs` is the XDR stream, and `rmsg` points to the structure that contains the reply message information.

`xdr_setpos`

This macro invokes the set position routine associated with the XDR stream `xdrs`.

Note: It is difficult to reposition some types of XDR streams; therefore, this routine might fail with one type of stream and succeed with another.

Format:

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

`pos` is a position value obtained from `xdr_getpos`. If the XDR stream can be repositioned, this routine returns 1; otherwise, it returns 0.

`xdr_short` This routine is a filter primitive that translates between C short integers (`sp`) and their external representations (`xdrs`). If the routine succeeds, it returns 1; otherwise, it returns 0.

Format:

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

`xdr_string` This routine is a filter primitive that translates between C strings and their corresponding external representations (`xdrs`).

Format:

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

Strings cannot be longer than `maxsize`. The `sp` parameter is the address of the string's pointer. If the routine succeeds, it returns 1; otherwise, it returns 0.

`xdr_u_char` This routine is a filter primitive that translates between C unsigned characters (`cp`) and their external representations (`xdrs`). If the routine succeeds, it returns 1; otherwise, it returns 0.

Format:

```
xdr_u_char(xdrs, cp)
    XDR *xdrs
    unsignedchar *cp;
```

`xdr_u_int` This routine is a filter primitive that translates between C unsigned integers (`up`) and their external representations (`xdrs`). If the routine succeeds, it returns 1; otherwise, it returns 0.

Format:

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

`xdr_u_long`

This routine is a filter primitive that translates between C unsigned long integers (`ulp`) and their external representations (`xdrs`). If the routine succeeds, it returns 1; otherwise, it returns 0.

Format:

```
xdr_u_long(xdrs, ulp)
    XDR *xdrs;
    unsigned_long *ulp;
```

`xdr_u_short`

This routine is a filter primitive that translates between C unsigned short integers (`usp`) and their external representations (`xdrs`). If the routine succeeds, it returns 1; otherwise, it returns 0.

Format:

```
xdr_u_short(xdrs, usp)
    XDR *xdrs;
    unsigned_short *usp;
```

`xdr_union`

This routine is a filter primitive that translates between a discriminated C union and its corresponding external representation (`xdrs`).

Format:

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

The `dscmp` parameter is the address of the union's discriminant, and `unp` is the address of the union. If the routine succeeds, it returns 1; otherwise, it returns 0. `choices` points to an array of `xdr_discrim` structures. This array must be terminated with an entry that contains a `NULL` procedure pointer. If the discriminant does not match any entry specified in the `choices` list, `dfault` points to the default `xdr` routine to use. See the `rpc/xdr.h` file for further details.

`xdr_vector`

This routine is a filter primitive that translates between vectors and their corresponding external representations (`xdrs`).

Format:

```
bool_t
xdr_vector(xdrs, basep, nelem, elemsize,
xdr_elem)
    XDR *xdrs;
    char *basep;
    u_int nelem;
    u_int elemsize;
    xdrproc_t xdr_elem;
```

The `basep` parameter is a pointer to the vector. `nelem` is the number of elements in the vector. `elemsize` is the size of each element in the vector. `xdr_elem` is an XDR filter that translates between the vector elements' C form and their external representation (`xdrs`). If the routine succeeds, it returns 1; otherwise, it returns 0.

`xdr_void`

This routine always returns 1.

Format:

```
xdr_void()
```

`xdr_wrapstring`

This routine is a primitive that calls the `xdr_string` (`xdrs,sp,MAXUNSIGNED`) routine; `MAXUNSIGNED` is the maximum value of an unsigned 31-bit integer. `xdr_wrapstring` translates null-terminated strings to or from external representation.

Format:

```
bool_t
xdr_wrapstring(xdrs, cpp)
        XDR *xdrs;
        char **cpp;
```

`xdrs` is the XDR stream, and `cpp` is the address of the pointer to the string.

`xdrmem_create` This routine initializes the XDR stream object to which `xdrs` points.

Format:

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

The stream's data is written to or read from a chunk of memory at location `addr`; the memory length can consist of a maximum of `size` bytes. The `op` parameter determines the direction of the XDR stream (`xdrs`); the direction can be `XDR_ENCODE`, `XDR_DECODE`, or `XDR_FREE`.

`xdrrec_create` This routine initializes the XDR stream object to which `xdrs` points.

Note: This XDR stream implements an intermediate record stream. Therefore, additional bytes in the stream provide record boundary information.

Format:

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

The stream's data is written to a buffer of size `sendsize`; a value of 0 indicates that 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 0 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 that of the UNICOS `read`(2) and `write`(2) system calls, except that `handle` is passed as the first parameter to the UNICOS routines. The caller must set the XDR stream's `op` field.

`xdrrec_endofrecord`

This routine can be invoked only on streams created by `xdrrec_create`.

Format:

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

`xdrs` is the XDR stream. The data in the output buffer is marked as a completed record; if `sendnow` is nonzero, the output buffer is optionally written out. If the routine succeeds, it returns 1; otherwise, it returns 0.

`xdrrec_eof`

This routine can be invoked only on streams (`xdrs`) created by `xdrrec_create`. This routine returns 1 if no more input is in the buffer after the rest of the current record has been consumed.

Format:

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

`xdrrec_skiprecord`

This routine can be invoked only on streams (`xdrs`) 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. If the routine succeeds, it returns 1; otherwise, it returns 0.

Format:

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

`xdrstdio_create`

This routine initializes the XDR stream object to which `xdrs` points.

Note: The destroy routine associated with XDR streams calls `flush` (by using `fflush`, see `fclose`(3)) on the file stream, but never `close` (by using `close`(2)).

Format:

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

The XDR stream data is written to or read from the stream file specified by `file`. The `op` parameter determines the direction of the XDR stream (`XDR_ENCODE`, `XDR_DECODE`, or `XDR_FREE`).

`xprt_register`

After RPC service transport handles (`xprt`) are created, they should be registered with the RPC service package. This routine modifies the global variable `svc_fdset`. Service implementers do not usually need this routine.

Format:

```
void
xprt_register(xprt)
    SVCXPRT *xprt;
```

`xprt_unregister`

Before an RPC service transport handle (`xprt`) is destroyed, it should be unregistered with the RPC service package. This routine modifies the global variable `svc_fdset`. Service implementers do not usually need this routine.

Format:

```
void
xprt_unregister(xprt)
    SVCXPRT *xprt;
```