

**NAME**

`rpc_soc`, `auth_destroy`, `authnone_create`, `authunix_create`, `authunix_create_default`, `callrpc`, `clnt_broadcast`, `clnt_call`, `clnt_control`, `clnt_create`, `clnt_destroy`, `clnt_freeres`, `clnt_geterr`, `clnt_pcreateerror`, `clnt_perrno`, `clnt_perror`, `clnt_screateerror`, `clnt_sperrno`, `clnt_sperror`, `clntraw_create`, `clnttcp_create`, `clntudp_bufcreate`, `clntudp_create`, `clntunix_create`, `get_myaddress`, `pmap_getmaps`, `pmap_getport`, `pmap_rmtcall`, `pmap_set`, `pmap_unset`, `registerrpc`, `rpc_createerr`, `svc_destroy`, `svc_fds`, `svc_fdset`, `svc_getargs`, `svc_getcaller`, `svc_getreq`, `svc_getreqset`, `svc_register`, `svc_run`, `svc_sendreply`, `svc_unregister`, `svcerr_auth`, `svcerr_decode`, `svcerr_noproc`, `svcerr_noprog`, `svcerr_progvers`, `svcerr_systemerr`, `svcerr_weakauth`, `svcfld_create`, `svcunixfd_create`, `svcrw_create`, `svcunix_create`, `xdr_accepted_reply`, `xdr_authunix_parms`, `xdr_callhdr`, `xdr_callmsg`, `xdr_opaque_auth`, `xdr_pmap`, `xdr_pmaplist`, `xdr_rejected_reply`, `xdr_replymsg`, `xprt_register`, `xprt_unregister` - library routines for remote procedure calls

**LIBRARY**

Standard C Library (`libc`, `-lc`)

**SYNOPSIS**

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

See *DESCRIPTION* for function declarations.

**DESCRIPTION**

The `svc_*`() and `clnt_*`() functions described in this page are the old, TS-RPC interface to the XDR and RPC library, and exist for backward compatibility. The new interface is described in the pages referenced from `rpc(3)`.

These routines allow C programs to make procedure calls on other machines across the network. First, the client calls a procedure to send a data packet to the server. Upon receipt of the packet, the server calls a dispatch routine to perform the requested service, and then sends back a reply. Finally, the procedure call returns to the client.

Routines that are used for Secure RPC (DES authentication) are described in `rpc_secure(3)`. Secure RPC can be used only if DES encryption is available.

*void*

```
auth_destroy(AUTH *auth)
```

A macro that destroys the authentication information associated with *auth*. Destruction usually involves deallocation of private data structures. The use of *auth* is undefined after calling **auth\_destroy**().

*AUTH \****authnone\_create()**

Create and return an RPC authentication handle that passes nonusable authentication information with each remote procedure call. This is the default authentication used by RPC.

*AUTH \****authunix\_create**(*char \*host, u\_int uid, u\_int gid, int len, u\_int \*aup\_gids*)

Create and return an RPC authentication handle that contains UNIX authentication information. The *host* argument is the name of the machine on which the information was created; *uid* is the user's user ID; *gid* is the user's current group ID; *len* and *aup\_gids* refer to a counted array of groups to which the user belongs. It is easy to impersonate a user.

*AUTH \****authunix\_create\_default()**

Calls **authunix\_create()** with the appropriate arguments.

*int callrpc*(*char \*host, u\_long prognum, u\_long versnum, u\_long procnum, xdrproc\_t inproc, void \*in, xdrproc\_t outproc, void \*out*)

Call the remote procedure associated with *prognum*, *versnum*, and *procnum* on the machine *host*. The *in* argument is the address of the procedure's argument(s), and *out* is the address of where to place the result(s); *inproc* is used to encode the procedure's arguments, and *outproc* is used to decode the procedure's results. This routine returns zero if it succeeds, or the value of *enum clnt\_stat* cast to an integer if it fails. The routine **clnt\_perrno()** is handy for translating failure statuses into messages.

Warning: calling remote procedures with this routine uses UDP/IP as a transport; see **clntudp\_create()** for restrictions. You do not have control of timeouts or authentication using this routine.

*enum clnt\_stat*

**clnt\_broadcast**(*u\_long prognum, u\_long versnum, u\_long procnum, xdrproc\_t inproc, char \*in, xdrproc\_t outproc, char \*out, bool\_t (\*eachresult)(caddr\_t, struct sockaddr\_in \*)*)

Like **callrpc()**, except the call message is broadcast to all locally connected broadcast nets. Each time it receives a response, this routine calls **eachresult()**, whose form is:

*bool\_t eachresult(caddr\_t out, struct sockaddr\_in \*addr)*

where *out* 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. If **eachresult()** returns zero, **clnt\_broadcast()** waits for more replies; otherwise it returns with appropriate status.

Warning: broadcast sockets are limited in size to the maximum transfer unit of the data link. For ethernet, this value is 1500 bytes.

*enum clnt\_stat*

**clnt\_call**(*CLIENT \*clnt, u\_long procnum, xdrproc\_t inproc, char \*in, xdrproc\_t outproc, char \*out, struct timeval tout*)

A macro that calls the remote procedure *procnum* associated with the client handle, *clnt*, which is obtained with an RPC client creation routine such as **clnt\_create()**. The *in* argument is the address of the procedure's argument(s), and *out* is the address of where to place the result(s); *inproc* is used to encode the procedure's arguments, and *outproc* is used to decode the procedure's results; *tout* is the time allowed for results to come back.

*void clnt\_destroy(CLIENT \*clnt)*

A macro that destroys the client's RPC handle. Destruction usually involves deallocation of private data structures, including *clnt* itself. Use of *clnt* is undefined after calling **clnt\_destroy()**. If the RPC library opened the associated socket, it will close it also. Otherwise, the socket remains open.

*CLIENT \**

**clnt\_create**(*char \*host, u\_long prog, u\_long vers, char \*proto*)

Generic client creation routine. The *host* argument identifies the name of the remote host where the server is located. The *proto* argument indicates which kind of transport protocol to use. The currently supported values for this field are "udp" and "tcp". Default timeouts are set, but can be modified using **clnt\_control()**.

Warning: Using UDP has its shortcomings. Since UDP-based RPC messages can only hold up to 8 Kbytes of encoded data, this transport cannot be used for procedures that take large arguments or return huge results.

*bool\_t*

**clnt\_control**(*CLIENT \*cl, u\_int req, char \*info*)

A macro used to change or retrieve various information about a client object. The *req* argument indicates the type of operation, and *info* is a pointer to the information. For both UDP and TCP, the supported values of *req* and their argument types and what they do are:

CLSET_TIMEOUT	<i>struct timeval</i>	set total timeout
CLGET_TIMEOUT	<i>struct timeval</i>	get total timeout

Note: if you set the timeout using **clnt\_control**(), the timeout argument passed to **clnt\_call**() will be ignored in all future calls.

CLGET_SERVER_ADDR	<i>struct sockaddr_in</i>	get server's address
-------------------	---------------------------	----------------------

The following operations are valid for UDP only:

CLSET_RETRY_TIMEOUT	<i>struct timeval</i>	set the retry timeout
CLGET_RETRY_TIMEOUT	<i>struct timeval</i>	get the retry timeout

The retry timeout is the time that UDP RPC waits for the server to reply before retransmitting the request.

*bool\_t* **clnt\_freeres**(*CLIENT \*clnt, xdrproc\_t outproc, char \*out*)

A macro that frees any data allocated by the RPC/XDR system when it decoded the results of an RPC call. The *out* argument is the address of the results, and *outproc* is the XDR routine describing the results. This routine returns one if the results were successfully freed, and zero otherwise.

*void*

**clnt\_geterr**(*CLIENT \*clnt, struct rpc\_err \*errp*)

A macro that copies the error structure out of the client handle to the structure at address *errp*.

*void*

**clnt\_pcreateerror**(*char \*s*)

prints a message to standard error indicating why a client RPC handle could not be created. The message is prepended with string *s* and a colon. A newline is appended at the end of the message. Used when a **clnt\_create**(), **clntraw\_create**(), **clnttcp\_create**(), or **clntudp\_create**() call

fails.

*void*

**clnt\_perrno**(*enum clnt\_stat stat*)

Print a message to standard error corresponding to the condition indicated by *stat*. A newline is appended at the end of the message. Used after **callrpc**().

*void* **clnt\_perror**(*CLIENT \*clnt, char \*s*)

Print a message to standard error indicating why an RPC call failed; *clnt* is the handle used to do the call. The message is prepended with string *s* and a colon. A newline is appended at the end of the message. Used after **clnt\_call**().

*char \**

**clnt\_spcreateerror**(*char \*s*)

Like **clnt\_pcreateerror**(), except that it returns a string instead of printing to the standard error.

Bugs: returns pointer to static data that is overwritten on each call.

*char \**

**clnt\_sperrno**(*enum clnt\_stat stat*)

Take the same arguments as **clnt\_perrno**(), but instead of sending a message to the standard error indicating why an RPC call failed, return a pointer to a string which contains the message.

The **clnt\_sperrno**() function is used instead of **clnt\_perrno**() if the program does not have a standard error (as a program running as a server quite likely does not), or if the programmer does not want the message to be output with **printf**(), or if a message format different from that supported by **clnt\_perrno**() is to be used.

Note: unlike **clnt\_sperror**() and **clnt\_spcreateerror**(), **clnt\_sperrno**() returns pointer to static data, but the result will not get overwritten on each call.

*char \**

**clnt\_sperror**(*CLIENT \*rpch, char \*s*)

Like **clnt\_perror**(), except that (like **clnt\_sperrno**()) it returns a string instead of printing to standard error.

Bugs: returns pointer to static data that is overwritten on each call.

*CLIENT \**

**clntraw\_create**(*u\_long prognum, u\_long versnum*)

This routine creates a toy RPC client for the remote program *prognum*, version *versnum*. The transport used to pass messages to the service is actually a buffer within the process's address space, so the corresponding RPC server should live in the same address space; see **svccraw\_create**(*u\_long prognum, u\_long versnum, u\_int sendsz, u\_int recvsz*). This allows simulation of RPC and acquisition of RPC overheads, such as round trip times, without any kernel interference. This routine returns NULL if it fails.

*CLIENT \**

**clnttcp\_create**(*struct sockaddr\_in \*addr, u\_long prognum, u\_long versnum, int \*sockp, u\_int sendsz, u\_int recvsz*)

This routine creates an RPC client for the remote program *prognum*, version *versnum*; the client uses TCP/IP as a transport. The remote program is located at Internet address *addr*. If *addr->sin\_port* is zero, then it is set to the actual port that the remote program is listening on (the remote `rpcbind(8)` service is consulted for this information). The *sockp* argument is a socket; if it is `RPC_ANYSOCK`, then this routine opens a new one and sets *sockp*. Since TCP-based RPC uses buffered I/O, the user may specify the size of the send and receive buffers with the *sendsz* and *recvsz* arguments; values of zero choose suitable defaults. This routine returns NULL if it fails.

*CLIENT \**

**clntudp\_create**(*struct sockaddr\_in \*addr, u\_long prognum, u\_long versnum, struct timeval wait, int \*sockp*)

This routine creates an RPC client for the remote program *prognum*, version *versnum*; the client uses UDP/IP as a transport. The remote program is located at Internet address *addr*. If *addr->sin\_port* is zero, then it is set to actual port that the remote program is listening on (the remote `rpcbind(8)` service is consulted for this information). The *sockp* argument is a socket; if it is `RPC_ANYSOCK`, then this routine opens a new one and sets *sockp*. The UDP transport resends the call message in intervals of *wait* time until a response is received or until the call times out. The total time for the call to time out is specified by **clnt\_call**(*u\_long prognum, u\_long versnum, u\_int sendsz, u\_int recvsz, struct timeval wait, int \*sockp*).

Warning: since UDP-based RPC messages can only hold up to 8 Kbytes of encoded data, this transport cannot be used for procedures that take large arguments or return huge results.

*CLIENT \**

**clntudp\_bufcreate**(*struct sockaddr\_in \*addr, u\_long prognum, u\_long versnum, struct timeval wait, int \*sockp, unsigned int sendsize, unsigned int recosize*)

This routine creates an RPC client for the remote program *prognum*, on *versnum*; the client uses UDP/IP as a transport. The remote program is located at Internet address *addr*. If *addr->sin\_port* is zero, then it is set to actual port that the remote program is listening on (the remote `rpcbind(8)` service is consulted for this information). The *sockp* argument is a socket; if it is `RPC_ANYSOCK`, then this routine opens a new one and sets *sockp*. The UDP transport resends the call message in intervals of *wait* time until a response is received or until the call times out. The total time for the call to time out is specified by `clnt_call()`.

This allows the user to specify the maximum packet size for sending and receiving UDP-based RPC messages.

*CLIENT* \*

**clntunix\_create**(*struct sockaddr\_un \*raddr, u\_long prognum, u\_long versnum, int \*sockp, u\_int sendsz, u\_int recvsz*)

This routine creates an RPC client for the local program *prognum*, version *versnum*; the client uses UNIX-domain sockets as a transport. The local program is located at the *raddr*. The *sockp* argument is a socket; if it is `RPC_ANYSOCK`, then this routine opens a new one and sets *sockp*. Since UNIX-based RPC uses buffered I/O, the user may specify the size of the send and receive buffers with the *sendsz* and *recvsz* arguments; values of zero choose suitable defaults. This routine returns NULL if it fails.

*int*

**get\_myaddress**(*struct sockaddr\_in \*addr*)

Stuff 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)`. Returns zero on success, non-zero on failure.

*struct pmaplist* \*

**pmap\_getmaps**(*struct sockaddr\_in \*addr*)

A user interface to the `rpcbind(8)` service, which returns a list of the current RPC program-to-port mappings on the host located at IP address *addr*. This routine can return NULL. The command "`rpcinfo -p`" uses this routine.

*u\_short*

**pmap\_getport**(*struct sockaddr\_in \*addr, u\_long prognum, u\_long versnum, u\_long protocol*)

A user interface to the `rpcbind(8)` service, which returns the port number on which waits a service that supports program number *prognum*, version *versnum*, and speaks the transport protocol associated with *protocol*. The value of *protocol* is most likely `IPPROTO_UDP` or `IPPROTO_TCP`. A return value of zero means that the mapping does not exist or that the RPC system failed to contact the remote `rpcbind(8)` service. In the latter case, the global variable *rpc\_createerr* contains the RPC status.

*enum clnt\_stat*

**pmap\_rmtcall**(*struct sockaddr\_in \*addr, u\_long prognum, u\_long versnum, u\_long procnum, xdrproc\_t inproc, char \*in, xdrproc\_t outproc, char \*out, struct timeval tout, u\_long \*portp*)

A user interface to the `rpcbind(8)` service, which instructs `rpcbind(8)` on the host at IP address *addr* to make an RPC call on your behalf to a procedure on that host. The *portp* argument will be modified to the program's port number if the procedure succeeds. The definitions of other arguments are discussed in `callrpc()` and `clnt_call()`. This procedure should be used for a "ping" and nothing else. See also `clnt_broadcast()`.

*bool\_t* **pmap\_set**(*u\_long prognum, u\_long versnum, u\_long protocol, u\_short port*)

A user interface to the `rpcbind(8)` service, which establishes a mapping between the triple (*prognum, versnum, protocol*) and *port* on the machine's `rpcbind(8)` service. The value of *protocol* is most likely `IPPROTO_UDP` or `IPPROTO_TCP`. This routine returns one if it succeeds, zero otherwise. Automatically done by `svc_register()`.

*bool\_t* **pmap\_unset**(*u\_long prognum, u\_long versnum*)

A user interface to the `rpcbind(8)` service, which destroys all mapping between the triple (*prognum, versnum, \**) and *ports* on the machine's `rpcbind(8)` service. This routine returns one if it succeeds, zero otherwise.

*bool\_t* **registerrpc**(*u\_long prognum, u\_long versnum, u\_long procnum, char \*(\*procname)(void), xdrproc\_t inproc, xdrproc\_t outproc*)

Register procedure *procname* with the RPC service package. If a request arrives for program *prognum*, version *versnum*, and procedure *procnum*, *procname* is called with a pointer to its argument(s); *procname* should return a pointer to its static result(s); *inproc* is used to decode the arguments while *outproc* is used to encode the results. This routine returns zero if the registration succeeded, -1 otherwise.



Warning: remote procedures registered in this form are accessed using the UDP/IP transport; see **svcudp\_create()** for restrictions.

```
struct rpc_createerr rpc_createerr;
```

A global variable whose value is set by any RPC client creation routine that does not succeed. Use the routine **clnt\_pcreateerror()** to print the reason why.

```
bool_t svc_destroy(SVCXPRT *xpirt)
```

A macro that destroys the RPC service transport handle, *xpirt*. Destruction usually involves deallocation of private data structures, including *xpirt* itself. Use of *xpirt* is undefined after calling this routine.

```
fd_set svc_fdset;
```

A global variable reflecting the RPC service side's read file descriptor bit mask; it is suitable as a template argument to the select(2) system call. This is only of interest if a service implementor does not call **svc\_run()**, but rather does his own asynchronous event processing. This variable is read-only (do not pass its address to select(2)!), yet it may change after calls to **svc\_getreqset()** or any creation routines. As well, note that if the process has descriptor limits which are extended beyond FD\_SETSIZE, this variable will only be usable for the first FD\_SETSIZE descriptors.

```
int svc_fds;
```

Similar to *svc\_fdset*, but limited to 32 descriptors. This interface is obsoleted by *svc\_fdset*.

```
bool_t svc_freeargs(SVCXPRT *xpirt, xdrproc_t inproc, char *in)
```

A macro that frees any data allocated by the RPC/XDR system when it decoded the arguments to a service procedure using **svc\_getargs()**. This routine returns 1 if the results were successfully freed, and zero otherwise.

```
bool_t svc_getargs(SVCXPRT *xpirt, xdrproc_t inproc, char *in)
```

A macro that decodes the arguments of an RPC request associated with the RPC service transport handle, *xpirt*. The *in* argument is the address where the arguments will be placed; *inproc* is the XDR routine used to decode the arguments. This routine returns one if decoding succeeds, and zero otherwise.

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

The approved way of getting the network address of the caller of a procedure associated with the RPC service transport handle, *xprt*.

```
void svc_getreqset(fd_set *rdfs)
```

This routine is only of interest if a service implementor 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 socket(s); *rdfs* is the resultant read file descriptor bit mask. The routine returns when all sockets associated with the value of *rdfs* have been serviced.

```
void svc_getreq(int rdfs)
```

Similar to **svc\_getreqset()**, but limited to 32 descriptors. This interface is obsolete by **svc\_getreqset()**.

```
bool_t svc_register(SVCXPRT *xprt, u_long prognum, u_long versnum,
    void (*dispatch)(struct svc_req *, SVCXPRT *), int protocol)
```

Associates *prognum* and *versnum* with the service dispatch procedure, **dispatch()**. If *protocol* is zero, the service is not registered with the `rpcbind(8)` service. If *protocol* is non-zero, then a mapping of the triple (*prognum*, *versnum*, *protocol*) to *xprt->xp\_port* is established with the local `rpcbind(8)` service (generally *protocol* is zero, `IPPROTO_UDP` or `IPPROTO_TCP`). The procedure **dispatch()** has the following form:

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

The **svc\_register()** routine returns one if it succeeds, and zero otherwise.

```
svc_run()
```

This routine never returns. It waits for RPC requests to arrive, and calls the appropriate service procedure using **svc\_getreq()** when one arrives. This procedure is usually waiting for a `select(2)` system call to return.

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

Called by an RPC service's dispatch routine to send the results of a remote procedure call. The *xprt* argument is the request's associated transport handle; *outproc* is the XDR routine which is used to encode the results; and *out* is the address of the results. This routine returns one if it succeeds, zero otherwise.

*void*

**svc\_unregister**(*u\_long prognum, u\_long versnum*)

Remove all mapping of the double (*prognum, versnum*) to dispatch routines, and of the triple (*prognum, versnum, \**) to port number.

*void*

**svcerr\_auth**(*SVCXPRT \*xprt, enum auth\_stat why*)

Called by a service dispatch routine that refuses to perform a remote procedure call due to an authentication error.

*void*

**svcerr\_decode**(*SVCXPRT \*xprt*)

Called by a service dispatch routine that cannot successfully decode its arguments. See also **svc\_getargs**().

*void*

**svcerr\_noproc**(*SVCXPRT \*xprt*)

Called by a service dispatch routine that does not implement the procedure number that the caller requests.

*void*

**svcerr\_noprog**(*SVCXPRT \*xprt*)

Called when the desired program is not registered with the RPC package. Service implementors usually do not need this routine.

*void*

**svcerr\_progvers**(*SVCXPRT \*xprt, u\_long low\_vers, u\_long high\_vers*)

Called when the desired version of a program is not registered with the RPC package. Service implementors usually do not need this routine.

*void*

**svcerr\_systemerr**(*SVCXPRT \*xpirt*)

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

*void*

**svcerr\_weakauth**(*SVCXPRT \*xpirt*)

Called by a service dispatch routine that refuses to perform a remote procedure call due to insufficient authentication arguments. The routine calls **svcerr\_auth**(*xpirt*, *AUTH\_TOOWEAK*).

*SVCXPRT \**

**svccraw\_create**(*void*)

This routine creates a toy RPC service transport, to which it returns a pointer. The transport is really a buffer within the process's address space, so the corresponding RPC client should live in the same address space; see **clntraw\_create**(). This routine allows simulation of RPC and acquisition of RPC overheads (such as round trip times), without any kernel interference. This routine returns NULL if it fails.

*SVCXPRT \**

**svctcp\_create**(*int sock*, *u\_int send\_buf\_size*, *u\_int recv\_buf\_size*)

This routine creates a TCP/IP-based RPC service transport, to which it returns a pointer. The transport is associated with the socket *sock*, which may be *RPC\_ANYSOCK*, in which case a new socket is created. If the socket is not bound to a local TCP port, then this routine binds it to an arbitrary port. Upon completion, *xpirt->xp\_fd* is the transport's socket descriptor, and *xpirt->xp\_port* is the transport's port number. This routine returns NULL if it fails. Since TCP-based RPC uses buffered I/O, users may specify the size of buffers; values of zero choose suitable defaults.

*SVCXPRT \**

**svcunix\_create**(*int sock*, *u\_int send\_buf\_size*, *u\_int recv\_buf\_size*, *char \*path*)

This routine creates a UNIX-based RPC service transport, to which it returns a pointer. The transport is associated with the socket *sock*, which may be *RPC\_ANYSOCK*, in which case a new socket is created. The *\*path* argument is a variable-length file system pathname of at most 104 characters. This file is *not* removed when the socket is closed. The *unlink(2)* system call must be used to remove the file. Upon completion, *xpirt->xp\_fd* is the transport's socket

descriptor. This routine returns NULL if it fails. Since UNIX-based RPC uses buffered I/O, users may specify the size of buffers; values of zero choose suitable defaults.

*SVCXPRT* \*

**svcnixfd\_create**(*int fd, u\_int sendsize, u\_int recvsz*)

Create a service on top of any open descriptor. The *sendsize* and *recvsz* arguments indicate sizes for the send and receive buffers. If they are zero, a reasonable default is chosen.

*SVCXPRT* \*

**svdfd\_create**(*int fd, u\_int sendsize, u\_int recvsz*)

Create a service on top of any open descriptor. Typically, this descriptor is a connected socket for a stream protocol such as TCP. The *sendsize* and *recvsz* arguments indicate sizes for the send and receive buffers. If they are zero, a reasonable default is chosen.

*SVCXPRT* \*

**svcudp\_bufcreate**(*int sock, u\_int sendsize, u\_int recvsz*)

This routine creates a UDP/IP-based RPC service transport, to which it returns a pointer. The transport is associated with the socket *sock*, which may be RPC\_ANYSOCK, in which case a new socket is created. If the socket is not bound to a local UDP port, then this routine binds it to an arbitrary port. Upon completion, *xprt->xp\_fd* is the transport's socket descriptor, and *xprt->xp\_port* is the transport's port number. This routine returns NULL if it fails.

This allows the user to specify the maximum packet size for sending and receiving UDP-based RPC messages.

**bool\_t xdr\_accepted\_reply**(*XDR \*xdrs, struct accepted\_reply \*ar*)

Used for encoding RPC reply messages. This routine is useful for users who wish to generate RPC-style messages without using the RPC package.

**bool\_t xdr\_authunix\_parms**(*XDR \*xdrs, struct authunix\_parms \*aupp*)

Used for describing UNIX credentials. This routine is useful for users who wish to generate these credentials without using the RPC authentication package.

*void*

**bool\_t xdr\_callhdr**(*XDR \*xdrs, struct rpc\_msg \*chdr*)

Used for describing RPC call header messages. This routine is useful for users who wish to generate RPC-style messages without using the RPC package.

*bool\_t* **xdr\_callmsg**(XDR \*xdrs, struct rpc\_msg \*cmsg)

Used for describing RPC call messages. This routine is useful for users who wish to generate RPC-style messages without using the RPC package.

*bool\_t* **xdr\_opaque\_auth**(XDR \*xdrs, struct opaque\_auth \*ap)

Used for describing RPC authentication information messages. This routine is useful for users who wish to generate RPC-style messages without using the RPC package.

*struct pmap*;

*bool\_t* **xdr\_pmap**(XDR \*xdrs, struct pmap \*regs)

Used for describing arguments to various rpcbind(8) procedures, externally. This routine is useful for users who wish to generate these arguments without using the **pmap\_\***() interface.

*bool\_t* **xdr\_pmaplist**(XDR \*xdrs, struct pmaplist \*\*rp)

Used for describing a list of port mappings, externally. This routine is useful for users who wish to generate these arguments without using the **pmap\_\***() interface.

*bool\_t* **xdr\_rejected\_reply**(XDR \*xdrs, struct rejected\_reply \*rr)

Used for describing RPC reply messages. This routine is useful for users who wish to generate RPC-style messages without using the RPC package.

*bool\_t* **xdr\_replymsg**(XDR \*xdrs, struct rpc\_msg \*rmsg)

Used for describing RPC reply messages. This routine is useful for users who wish to generate RPC style messages without using the RPC package.

*void*

**xprt\_register**(SVCXPRT \*xpvt)

After RPC service transport handles are created, they should register themselves with the RPC service package. This routine modifies the global variable *svc\_fds*. Service implementors usually do not need this routine.

*void*

**xprt\_unregister**(*SVCXPRT \*xprt*)

Before an RPC service transport handle is destroyed, it should unregister itself with the RPC service package. This routine modifies the global variable *svc\_fds*. Service implementors usually do not need this routine.

**SEE ALSO**

*rpc\_secure*(3), *xdr*(3)

*Remote Procedure Calls: Protocol Specification.*

*Remote Procedure Call Programming Guide.*

*rpcgen Programming Guide.*

*RPC: Remote Procedure Call Protocol Specification*, Sun Microsystems, Inc., USC-ISI, RFC1050.