ChorusOS man pages section 3RPC: RPC Services

Sun Microsystems, Inc.
901 San Antonio Road
Palo Alto, CA 94303-4900
U.S.A.

Part No: 806-3334
December 10, 1999

Contents

PREFACE

Overview

A man page is provided for both the naive user, and sophisticated user who is familiar with the ChorusOS™ operating system and is in need of on-line information. A man page is intended to answer concisely the question "What does it do?" The man pages in general comprise a reference manual. They are not intended to be a tutorial.

The following is a list of sections in the ChorusOS man pages and the information it references:

- *Section 1CC: User Utilities; Host and Target Utilities*
- *Section 1M: System Management Utilities*
- *Section 2DL: System Calls; Data Link Services*
- *Section 2K: System Calls; Kernel Services*
- *Section 2MON: System Calls; Monitoring Services*
- *Section 2POSIX: System Calls; POSIX System Calls*
- *Section 2RESTART: System Calls; Hot Restart and Persistent Memory*
- *Section 2SEG: System Calls; Virtual Memory Segment Services*
- *Section 3FTPD: Libraries; FTP Daemon*
- *Section 3M: Libraries; Mathematical Libraries*
- *Section 3POSIX: Libraries; POSIX Library Functions*
- *Section 3RPC: Libraries; RPC Services*
- *Section 3STDC: Libraries; Standard C Library Functions*
- *Section 3TELD: Libraries; Telnet Services*
- *Section 4CC: Files*

- *Section 5FEA: ChorusOS Features and APIs*
- *Section 7P: Protocols*
- *Section 7S: Services*
- *Section 9DDI: Device Driver Interfaces*
- *Section 9DKI: Driver to Kernel Interface*
- *Section 9DRV: Driver Implementations*

ChorusOS man pages are grouped in Reference Manuals, with one reference manual per section.

Below is a generic format for man pages. The man pages of each manual section generally follow this order, but include only needed headings. For example, if there are no bugs to report, there is no BUGS section. See the `intro` pages for more information and detail about each section, and `man`(1) for more information about man pages in general.

NAME	This section gives the names of the commands or functions documented, followed by a brief description of what they do.
SYNOPSIS	This section shows the syntax of commands or functions. When a command or file does not exist in the standard path, its full pathname is shown. Options and arguments are alphabetized, with single letter arguments first, and options with arguments next, unless a different argument order is required.

The following special characters are used in this section:

[ ]	The option or argument enclosed in these brackets is optional. If the brackets are omitted, the argument must be specified.
. . .	Ellipses. Several values may be provided for the previous argument, or the previous argument can be specified multiple times, for example, ' "`filename` . . ." .
|	Separator. Only one of the arguments separated by this character can be specified at time.
{ }	Braces. The options and/or arguments enclosed within braces are

	interdependent, such that everything enclosed must be treated as a unit.
FEATURES	This section provides the list of features which offer an interface. An API may be associated with one or more system features. The interface will be available if one of the associated features has been configured.
DESCRIPTION	This section defines the functionality and behavior of the service. Thus it describes concisely what the command does. It does not discuss OPTIONS or cite EXAMPLES.. Interactive commands, subcommands, requests, macros, functions and such, are described under USAGE.
OPTIONS	This lists the command options with a concise summary of what each option does. The options are listed literally and in the order they appear in the SYNOPSIS section. Possible arguments to options are discussed under the option, and where appropriate, default values are supplied.
OPERANDS	This section lists the command operands and describes how they affect the actions of the command.
OUTPUT	This section describes the output - standard output, standard error, or output files - generated by the command.
RETURN VALUES	If the man page documents functions that return values, this section lists these values and describes the conditions under which they are returned. If a function can return only constant values, such as 0 or –1, these values are listed in tagged paragraphs. Otherwise, a single paragraph describes the return values of each function. Functions declared void do not return values, so they are not discussed in RETURN VALUES.
ERRORS	On failure, most functions place an error code in the global variable `errno` indicating why they failed. This section lists alphabetically all error codes a function can generate and describes the conditions that cause each error. When more than one condition can cause the same error, each condition is described in a separate paragraph under the error code.

USAGE	This section is provided as a guidance on use. This section lists special rules, features and commands that require in-depth explanations. The subsections listed below are used to explain built-in functionality: Commands Modifiers Variables Expressions Input Grammar
EXAMPLES	This section provides examples of usage or of how to use a command or function. Wherever possible a complete example including command line entry and machine response is shown. Whenever an example is given, the prompt is shown as `example%` or if the user must be superuser, `example#`. Examples are followed by explanations, variable substitution rules, or returned values. Most examples illustrate concepts from the SYNOPSIS, DESCRIPTION, OPTIONS and USAGE sections.
ENVIRONMENT VARIABLES	This section lists any environment variables that the command or function affects, followed by a brief description of the effect.
EXIT STATUS	This section lists the values the command returns to the calling program or shell and the conditions that cause these values to be returned. Usually, zero is returned for successful completion and values other than zero for various error conditions.
FILES	This section lists all filenames referred to by the man page, files of interest, and files created or required by commands. Each is followed by a descriptive summary or explanation.
SEE ALSO	This section lists references to other man pages, in-house documentation and outside publications.
DIAGNOSTICS	This section lists diagnostic messages with a brief explanation of the condition causing the error.
WARNINGS	This section lists warnings about special conditions which could seriously affect your working conditions. This is not a list of diagnostics.
NOTES	This section lists additional information that does not belong anywhere else on the page. It takes the form of an aside to the user, covering points of special interest. Critical information is never covered here.

BUGS

This section describes known bugs and wherever possible, suggests workarounds.

CHAPTER

RPC Library

10

NAME

bindresvport – bind a socket to a privileged IP port

SYNOPSIS

```
#include <sys/types.h>
#include <netinet/in.h>
int bindresvport(int sd, struct sockaddr_in *sin);
```

FEATURES

POSIX_SOCKETS

DESCRIPTION

The `bindresvport()` function is used to bind a socket descriptor to a privileged IP port, that is, a port number in the range `0–1023`.

Only root can bind to a privileged port. The call fails for any other users.

RETURN VALUES

The routine returns `0` if it is successful, otherwise `-1` is returned and *errno* is set to indicate the error condition.

RESTRICTIONS

This library call does not support multi-threaded applications.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

getrpcent, getrpcbyname, getrpcbynumber – get RPC entry

SYNOPSIS

#include <netdb.h>
struct rpcent * **getrpcent**(void);

struct rpcent * **getrpcbyname**(char * *name*);

struct rpcent * **getrpcbynumber**(int *number*);

setrpcent(int *stayopen*);

endrpcent(void);

FEATURES

POSIX_SOCKETS

DESCRIPTION

The getrpcent( ) , getrpcbyname( ) , and getrpcbynumber( ) functions each return a pointer to an object containing the broken-out fields of a line in the RPC program number database, /etc/rpc . The object has the following structure:

```
struct  rpcent {
        char*   r_name;         /* name of server for this rpc program */
        char**  r_aliases;      /* alias list */
        long    r_number;       /* rpc program number */
};
```

The members of this structure are:

r_name	The name of the server for this rpc program
r_aliases	A zero terminated list of alternate names for the rpc program
r_number	The rpc program number for this service

The getrpcent( ) function reads the next line of the file, opening the file if necessary.

The getrpcent( ) function opens and rewinds the file. If the *stayopen* flag is non-zero, the net database will not be closed after each call to getrpcent( ) (called either directly, or indirectly through one of the other "getrpc*" calls).

The endrpcent( ) function closes the file.

The getrpcbyname( ) and getrpcbynumber( ) functions search sequentially from the beginning of the file until a matching rpc program name or program number is found, or until end-of-file is encountered.

DIAGNOSTICS

A NULL pointer is returned on EOF or error.

BUGS

All information is contained in a static area, and must be copied if it is to be saved.

RESTRICTIONS

These library calls do not support multi-threaded applications.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

getrpcent, getrpcbyname, getrpcbynumber – get RPC entry

SYNOPSIS

#include <netdb.h>
struct rpcent * **getrpcent**(void);

struct rpcent * **getrpcbyname**(char * *name*);

struct rpcent * **getrpcbynumber**(int *number*);

setrpcent(int *stayopen*);

endrpcent(void);

FEATURES

POSIX_SOCKETS

DESCRIPTION

The `getrpcent()` , `getrpcbyname()` , and `getrpcbynumber()` functions each return a pointer to an object containing the broken-out fields of a line in the RPC program number database, `/etc/rpc` . The object has the following structure:

```
struct  rpcent {
        char*   r_name;         /* name of server for this rpc program */
        char**  r_aliases;      /* alias list */
        long    r_number;       /* rpc program number */
};
```

The members of this structure are:

`r_name`	The name of the server for this rpc program
`r_aliases`	A zero terminated list of alternate names for the rpc program
`r_number`	The rpc program number for this service

The `getrpcent()` function reads the next line of the file, opening the file if necessary.

The `getrpcent()` function opens and rewinds the file. If the *stayopen* flag is non-zero, the net database will not be closed after each call to `getrpcent()` (called either directly, or indirectly through one of the other “getrpc*” calls).

The `endrpcent()` function closes the file.

The `getrpcbyname()` and `getrpcbynumber()` functions search sequentially from the beginning of the file until a matching rpc program name or program number is found, or until end-of-file is encountered.

DIAGNOSTICS

A `NULL` pointer is returned on `EOF` or error.

BUGS

All information is contained in a static area, and must be copied if it is to be saved.

RESTRICTIONS

These library calls do not support multi-threaded applications.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

getrpcent, getrpcbyname, getrpcbynumber – get RPC entry

SYNOPSIS

#include <netdb.h>
struct rpcent * **getrpcent**(void);

struct rpcent * **getrpcbyname**(char * *name*);

struct rpcent * **getrpcbynumber**(int *number*);

setrpcent(int *stayopen*);

endrpcent(void);

FEATURES

POSIX_SOCKETS

DESCRIPTION

The `getrpcent()`, `getrpcbyname()`, and `getrpcbynumber()` functions each return a pointer to an object containing the broken-out fields of a line in the RPC program number database, `/etc/rpc`. The object has the following structure:

```
struct  rpcent {
        char*   r_name;         /* name of server for this rpc program */
        char**  r_aliases;      /* alias list */
        long    r_number;       /* rpc program number */
};
```

The members of this structure are:

`r_name`	The name of the server for this rpc program
`r_aliases`	A zero terminated list of alternate names for the rpc program
`r_number`	The rpc program number for this service

The `getrpcent()` function reads the next line of the file, opening the file if necessary.

The `getrpcent()` function opens and rewinds the file. If the *stayopen* flag is non-zero, the net database will not be closed after each call to `getrpcent()` (called either directly, or indirectly through one of the other “getrpc*” calls).

The `endrpcent()` function closes the file.

The `getrpcbyname()` and `getrpcbynumber()` functions search sequentially from the beginning of the file until a matching rpc program name or program number is found, or until end-of-file is encountered.

DIAGNOSTICS

A `NULL` pointer is returned on `EOF` or error.

BUGS

All information is contained in a static area, and must be copied if it is to be saved.

RESTRICTIONS

These library calls do not support multi-threaded applications.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

getrpcport – get RPC port number

SYNOPSIS

int **`getrpcport`**(char **host*, int *prognum*, int *versnum*, int *proto*);

FEATURES

POSIX_SOCKETS

DESCRIPTION

The `getrpcport()` function returns the port number for version *versnum*, of the RPC program *prognum*, running on *host* and using protocol *proto.* Version mismatch is detected upon the first call to the service.

RETURN VALUES

`getrpcport()` returns `0` if it cannot contact the port mapper, or if *prognum* is not registered. If *prognum* is registered but not with version *versnum*, it will still return a port number (for some version of the program) indicating that the program is registered.

RESTRICTIONS

This library call does not support multi-threaded applications.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

rpc – library routines for remote procedure calls

FEATURES

POSIX_SOCKETS

DESCRIPTION

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 sends back a reply. Finally, the procedure call returns to the client.

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


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

The macro above destroys the authentication information associated with *auth*. Destruction usually involves deallocation of private data structures. The effect of using *auth* after calling `auth_destroy( )` is undefined.

```
AUTH *
authnone_create( )
```

The code above creates and returns an RPC authentication handle that passes non usable authentication information with each remote procedure call. This is the default authentication used by RPC.

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

The code above creates and returns an RPC authentication handle that contains authentication information. The *host* parameter 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. Note that it is easy to impersonate a user.

```
AUTH *
authunix_create_default( )
```

The code above calls `authunix_create( )` with the appropriate parameters.

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

The code above calls the remote procedure associated with *prognum*, *versnum*, and *procnum* on the *host* system. The parameter *in* is the address of the procedure arguments, and *out* is the address to which the results are to be returned. The *inproc* argument is used to encode the procedure's parameters, and *outproc* is used to decode the procedure's results. This routine returns zero if sucessful, or the value of *enum clnt_stat* cast to an `int` if it fails. The *clnt_perrno* routine is useful for translating failure statuses into messages.

Caution: 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(prognum, versnum, procnum, inproc, in, outproc, out,
               eachresult)
u_long prognum, versnum, procnum;
char *in, *out;
xdrproc_t inproc, outproc;
resultproc_t eachresult;
```

The code illustrated above is similar to `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:

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

where *out* is similar to the *out* passed to `clnt_broadcast()`, except that the remote procedure's output is decoded there. The *addr* pointer indicates the address of the machine that sent the results. If `eachresult()` returns zero, `clnt_broadcast()` waits for more replies; otherwise, it returns with the appropriate status.

Caution: 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(clnt, procnum, inproc, in, outproc, out, tout)
CLIENT *clnt;
u_long
```

```
procnum;
xdrproc_t inproc, outproc;
char *in, *out;
struct timeval tout;
```

The macro illustrated above 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 parameter *in* is the address of the procedure's argument(s), and *out* is the address to which the results are returned. The *inproc* argument is used to encode the procedure's parameters, and *outproc* is used to decode the procedure's results; *tout* is the time allowed for results to come back.

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

The macro illustrated above destroys the client's RPC handle. Destruction usually involves deallocation of private data structures, including *clnt* itself. The effect of using *clnt* after calling `clnt_destroy()` is undefined. If the RPC library opened the associated socket, it will also close it. Otherwise, the socket remains open.

```
CLIENT *
clnt_create(host, prog, vers, proto)
char *host;
u_long prog, vers;
char *proto;
```

The code above is a generic client creation routine. The *host* parameter identifies the name of the remote host where the server is located. The *proto* parameter indicates which kind of transport protocol to use. The values currently supported for this field are `udp` and `tcp`. Default timeouts are set, but can be modified using *clnt_control.*

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

```
bool_t
clnt_control(cl, req, info)
CLIENT *cl;
char *info;
```

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

```
CLSET_TIMEOUT   struct timeval   set total timeout
CLGET_TIMEOUT   struct timeval   get total timeout
```

If you set the timeout using `clnt_control()`, the timeout parameter passed to `clnt_call()` will be ignored in all future calls.

```
CLGET_SERVER_ADDR  struct sockaddr_in  get server's address
```

The following operations are valid only for UDP:

```
CLSET_RETRY_TIMEOUT   struct timeval   set the retry
CLGET_RETRY_TIMEOUT   struct timeval   get the retry
```

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

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

The macro above frees any data allocated by the RPC/XDR system when it decoded the results of an RPC call. The *out* parameter 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(clnt, errp)
CLIENT *clnt;
struct rpc_err *errp;
```

The macro above copies the error structure from the client handle to the structure at address *errp*.

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

The code above prints a message to standard error indicating why a client RPC handle could not be created. The message is prepended with the string *s* and a colon. Used when a `clnt_create( )`, `clntraw_create( )`, `clnttcp_create( )`, or `clntudp_create( )` call fails.

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

The code above prints a message to standard error corresponding to the condition indicated by *stat*. Used after `callrpc( )`.

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

The code above prints a message to standard error indicating why an RPC call failed; *clnt* is the handle used to perform the call. The message is prepended with the string *s* and a colon. Used after `clnt_call( )`.

```
char *
clnt_spcreateerror
char *s;
```

The code illustrated above performs in a similar way to `clnt_pcreateerror( )`, except that it returns a string instead of printing to the standard error.

Bugs: it returns a pointer to static data which is overwritten on each call.

```
char *
clnt_sperrno(stat)
enum clnt_stat stat;
```

The code illustrated above takes the same arguments as `clnt_perrno( )`, but instead of sending a message to the standard error indicating why an RPC call failed, it returns a pointer to a string which contains the message. The string ends with a `NEWLINE`.

The `clnt_sperrno( )` function is used instead of `clnt_perrno( )` if the program does not have a standard error (programs running as servers quite often do not), or if the programmer does not want the message to be output using `printf( )`, or if a message format different from the one supported by `clnt_perrno( )` is to be used.

Note: unlike `clnt_sperror( )` and `clnt_spcreaterror( )`, `clnt_sperrno( )` returns a pointer to static data, but the result is not overwritten on each call.

```
char *
clnt_sperror(rpch, s)
CLIENT *rpch;
char *s;
```

The code illustrated above performs in a similar way to`clnt_perror( )`, except that (like `clnt_sperrno( )`) it returns a string instead of printing to standard error.

Bugs: It returns a pointer to static data which is overwritten on each call.

```
CLIENT *
clntraw_create(prognum, versnum)
u_long prognum, 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, the corresponding RPC server should therefore be located in the same address space (see `svcraw_create( )`). 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(addr, prognum, versnum, sockp, sendsz, recvsz)
struct sockaddr_in *addr;
u_long prognum, versnum;
int *sockp;
u_int sendsz, 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 the 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 `portmap` service is consulted for this information). The parameter *sockp* is a socket; if it is `RPC_ANYSOCK`, this routine opens a new one and sets *sockp*. As TCP-based RPC uses buffered I/O, the user may specify the size of the send and receive buffers using the *sendsz* and *recvsz* parameters. Zero values set suitable defaults. This routine returns `NULL` if it fails.

```
CLIENT *
clntudp_create(addr, prognum, versnum, wait, sockp)
struct sockaddr_in *addr;
u_long prognum, 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 the 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 `portmap` service is consulted for this information). The *sockp* parameter is a socket; if it is `RPC_ANYSOCK`, this routine opens a new one and sets *sockp*. The UDP transport resends the call message at 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()`.

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

```
CLIENT *
clntudp_bufcreate(addr, prognum, versnum, wait, sockp, sendsize,
                  recosize)
struct sockaddr_in *addr;
u_long prognum, 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 the 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 `portmap` service is consulted for this information). The *sockp* parameter is a socket; if it is `RPC_ANYSOCK`, this routine opens a new one and sets *sockp*. The UDP transport resends the call message at 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 maximun packet size for sending and receiving UDP-based RPC messages.

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

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)`. On top of ChorusOS, this primitive returns 0 when a port number and an IP address are found, and *addr* is filled with a valid *sockaddr_in* structure. Otherwise, a non zero value is returned, and *addr* is left unchanged.

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

This is a user interface to the `portmap` 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 `rpcinfo -p` command uses this routine.

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

This is a user interface to the `portmap` service, which returns the port number on which a service is waiting. The service supports program number *prognum*, version *versnum*, and can use the transport protocol associated with *protocol*. The value of *protocol* will usually be `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 `portmap` service. In the latter case, the global variable *rpc_createerr* contains the RPC status.

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

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

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

This is a user interface to the `portmap` service, which establishes a mapping between the triple [*prognum*,*versnum*,*protocol*] and *port* on the system's `portmap` service. The value of *protocol* will usually be `IPPROTO_UDP` or `IPPROTO_TCP`. This routine returns one if it succeeds, zero otherwise. The mapping is automatically done by `svc_register()`.

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

This is a user interface to the `portmap` service, which destroys all mapping between the triple [*prognum*,*versnum*,*] and `ports` on the system's `portmap` service. This routine returns one if it succeeds, zero otherwise.

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

The above routine registers the 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 parameter(s). The *progname* argument should return a pointer to its static result(s). *inproc* is used to decode the parameters, and *outproc* is used to encode the results. This routine returns zero if the registration succeeded, -1 otherwise.

Caution: Remote procedures registered in this form are accessed using the UDP/IP transport; see *svcudp_create* for restrictions.

```
struct rpc_createerr     rpc_createerr;
```

This 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.

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

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

```
fd_set svc_fdset;
```

This is a global variable which reflects the RPC service side's read file descriptor bit mask. It can be used as a parameter to the `select()` system call. This is only of use if a service implementor does its own asynchronous event processing and does not call `svc_run()` . This variable is read-only (does not pass its address to `select()`!), but it may change after calls to `svc_getreqset()` or any other creation routines.

```
int svc_fds;
```

This interfqce is similar to `svc_fdset()`, but limited to 32 descriptors. This interface is rendered obsolete by `svc_fdset()`.

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

This is 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 0 otherwise.

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

This is a macro that decodes the arguments of an RPC request associated with the RPC service transport handle, *xprt*. The *in* parameter is the address where the arguments will be placed; `inproc()` is the XDR routine used to decode the arguments. This routine returns 1 if decoding succeeds, and 0 otherwise.

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

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

```
svc_getreqset(rdfds)
fd_set *rdfds;
```

This routine is only of use if a service implementor does not call `svc_run( )`, but instead implements custom asynchronous event processing. It is called when the `select( )` system call has determined that an RPC request has arrived on RPC socket(s); *rdfds* is the resultant read file descriptor bit mask. The routine returns when all sockets associated with the value of *rdfds* have been serviced.

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

This interface is similar to `svc_getreqset( )`, but limited to 32 descriptors. This interface is rendered obsolete by `svc_getreqset( )`.

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

This routine associates *prognum* and *versnum* with the service dispatch procedure, *dispatch*. If *protocol* is zero, the service is not registered with the `portmap` service. If *protocol* is non-zero, a mapping of the triple [*prognum*,*versnum*,*protocol*] to *xprt->xp_port* is established with the local `portmap` service (*protocol* is usually zero, `IPPROTO_UDP` or `IPPROTO_TCP`). The *dispatch* procedure has the following form:

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

The `svc_register( )` routine returns 1 if it succeeds, and 0 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( )` system call to return.

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

This is called by an RPC service's dispatch routine to send the results of a remote procedure call. The *xprt* parameter 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 1 if it succeeds, and 0 otherwise.

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

This function removes all mapping of the double [*prognum*,*versnum*] to dispatch routines, and of the triple [*prognum*,*versnum*,*] to port numbers.

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

This is called by a service dispatch routine that cannot perform a remote procedure call due to an authentication error.

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

This is called by a service dispatch routine that cannot decode its parameters successfully. See also *svc_getargs*.

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

This is called by a service dispatch routine that does not implement the procedure number requested by the caller.

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

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

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

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

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

This is 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(xprt)
SVCXPRT *xprt;
```

This is called by a service dispatch routine that cannot perform a remote procedure call due to insufficient authentication parameters. The routine calls svcerr_auth(xprt, AUTH_TOOWEAK).

```
SVCXPRT *
svcraw_create( )
```

This routine creates a dummy RPC service transport, to which it returns a pointer. The transport is really a buffer within the process's address space. The corresponding RPC client should therefore be located 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(sock, send_buf_size, recv_buf_size)
int sock;
u_int send_buf_size, 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, this routine binds it to an arbitrary port. Upon completion, *xprt->xp_sock* is the transport's socket descriptor, and *xprt->xp_port* is the transport's port number. This routine returns `NULL` if it fails. TCP-based RPC uses buffered I/O, therefore users may specify the size of buffers. Zero values set suitable defaults.

```
SVCXPRT *
svcfd_create(fd, sendsize, recvsize)
int fd;
u_int sendsize;
u_int recvsize;
```

This routine creates 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 *recvsize* parameters indicate sizes for the send and receive buffers. If they are zero, a reasonable default is chosen.

```
SVCXPRT *
svcudp_bufcreate(sock, sendsize, recosize)
int sock;
```

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, this routine binds it to an arbitrary port. Upon completion, *xprt->xp_sock* 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 maximun packet size for sending and receiving UDP-based RPC messages.

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

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

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

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

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

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

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

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

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

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

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

This routine is used to describe parameters to various `portmap` procedures externally. It is useful for users who wish to generate these parameters without using the `pmap` interface.

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

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

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

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

```
xdr_replymsg(xdrs, rmsg)
XDR *xdrs;
struct rpc_msg *rmsg;
```

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

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

After RPC service transport handles are created, they should be registered with the RPC service package using this routine, which modifies the global variable `svc_fds()`. Service implementors do not usually need this routine.

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

Before an RPC service transport handle is destroyed, it should be unregistered with the RPC service package using this routine, which modifies the global variable `svc_fds()`. Service implementors do not usually need this routine.

RESTRICTIONS These library calls do not support multi-threaded applications.

ATTRIBUTES See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO `xdr`(3RPC)

NAME

rpcgen – an RPC protocol compiler

SYNOPSIS

rpcgen infile

```
rpcgen [ -a ] [ -A ] [ -b ] [ -C ] \
[ -D name [ = value ] ] [ -i size ] [ -I [ -K seconds ] ] \
[ -L ] [ -M ] [ -N ] [ -T ] [ -Y pathname ] infile
rpcgen [ -c | -h | -l | -m | -t | -Sc | -Ss | -Sm ] \
[ -o outfile ] [ infile ]
rpcgen [ -s nettype ] [ -o outfile ] [ infile ]
rpcgen [ -n netid ] [ -o outfile ] [ infile ]
```

DESCRIPTION

`rpcgen` is a tool that generates C code to implement an RPC protocol. The input to `rpcgen` is a language similar to C, known as RPC Language (Remote Procedure Call Language).

`rpcgen` is normally used as in the first synopsis where it takes an infile and generates three output files. Assuming the infile is named `proto.x`, `rpcgen` will generate:

- a header in `proto.h`
- XDR routines in `proto_xdr.c`
- server-side stubs in `proto_svc.c`
- client-side stubs in `proto_clnt.c`

With the -T option, rpcgen also generates the RPC dispatch table in `proto_tbl.i`.

`rpcgen` can also generate sample client and server files that can be customized to suit a particular application. The -Sc, -Ss, and -Sm options generate a sample client, server and `Makefile`, respectively. The -a option generates all files, including sample files. If the infile is `proto.x`, then the client side sample file is written to the `proto_client.c` file, the server side sample file to the `proto_server.c` file and the sample `Makefile` to the `makefile.proto` file.

The server can be started either by itself or by the port monitors (for example, `inetd` or `listen`) . When it is started by a port monitor, it only creates servers for the transport for which the file descriptor 0 was passed. The name of the transport must be specified by setting up the environment variable `PM_TRANSPORT`. When the server generated by `rpcgen` is executed, it creates server handles for all the transports specified in the `NETPATH` environment variable, or if it is unset, it creates server handles for all the visible transports from the `/etc/netconfig` file.

The transports are chosen at run time and not at compile time. When the server is self-started, by default, it runs in the background. A special define symbol `RPC_SVC_FG` can be used to run the server process in foreground.

The second synopsis provides special features which allow for the creation of more sophisticated RPC servers. These features include support for user-provided #defines and RPC dispatch tables. The entries in the RPC dispatch table contain:

- Pointers to the service routine corresponding to that procedure.
- A pointer to the input and output arguments indicating the size of these routines.

A server can use the dispatch table to check authorization and then to execute the service routine; a client library may use it to deal with the details of storage management and XDR data conversion.

The synopses shown above are used when one does not want to generate all the output files, but only a particular one. See the EXAMPLES section below for examples of `rpcgen` usage. When `rpcgen` is executed with the -s option, it creates servers for that particular class of transports. When executed with the -n option, it creates a server for the transport specified by `netid`. If infile is not specified, `rpcgen` accepts the standard input.

All the options mentioned in the second synopsis can be used with the other three synopses, but the changes will be made only to the specified output file.

The C preprocessor cc -E is run on the input file before it is actually interpreted by `rpcgen`. For each type of output file, `rpcgen` defines a special preprocessor symbol for use by the rpcgen programmer:

- RPC_HDR: defined when compiling into headers
- RPC_XDR: defined when compiling into XDR routines
- RPC_SVC: defined when compiling into server-side stubs
- RPC_CLNT: defined when compiling into client-side stubs
- RPC_TBL: defined when compiling into RPC dispatch tables

Any line beginning with “%” is passed directly into the output file, uninterpreted by `rpcgen`. To specify the path name of the C preprocessor, use the -Y flag.

For every data type referred to in the infile, `rpcgen` assumes that there is a routine with the string `xdr_` associated to the name of the data type. If this routine does not exist in the RPC/XDR library, it must be provided. Providing an undefined data type allows customization of XDR routines.

OPTIONS

The following options are supported:

`–a`	Generate all files, including sample files.
`–A`	Enable the Automatic MT mode in the server main program. In this mode, the RPC library automatically creates threads to service client

	requests. This option generates multithread-safe stubs by implicitly turning on the -M option. Server multithreading modes and parameters can be set using the `rpc_control`(3N) call. `rpcgen` generated code does not change the default values for the Automatic MT mode. See RESTRICTIONS below.
`–b`	Backward compatibility mode. Generate transport-specific RPC code for older versions of the operating system (generates code for SunOS 4.X ChorusOS 4.0).
`–c`	Compile into XDR routines.
`–C`	Generate header and stub files which can be used with ANSI C compilers. Headers generated with this flag can also be used with C++ programs.
`–D` *name[=value]*	Define a symbol name. Equivalent to the #define directive in the source. If no value is given, the value is defined as 1 . This option may be specified more than once.
`–h`	Compile into C data-definitions (a header). The -T option can be used in conjunction to produce a header which supports RPC dispatch tables.
`–i` *size*	Size at which to start generating inline code. This option is useful for optimization. The default size is 5.
`–I`	Compile support for `inetd`(1M) in the server side stubs. Such servers can be self-started or can be started by `inetd`. When the server is selfstarted, it backgrounds itself by default. A special define symbol, `RPC_SVC_FG`, can be used to run the server process in foreground, or the user may simply compile without the -I option. If there are no pending client requests, the inetd servers exit after 120 seconds (default). The default can be changed with the -K option. All of the error messages for inetd servers are always logged with `syslog`(3).

Note - This option is supported for backward compatibility only. It should always be used in conjunction with the -b option which generates backward compatibility code. By default (that is, when -b is not specified), `rpcgen` generates servers that can be invoked through port monitors. See RESTRICTIONS below.

`–K`
: By default, services created using `rpcgen` and invoked through port monitors wait 120 seconds after servicing a request before exiting. That interval can be changed using the -K flag. To create a server that exits immediately upon servicing a request, use -K 0 . To create a server that never exits, the appropriate argument is -K -1 .

  When monitoring for a server, some portmonitors, like `listen`(1M), always spawn a new process in response to a service request. If it is known that a server will be used with this type of monitor, the server should exit immediately on completion. For these servers, `rpcgen` should be used with -K 0 . See RESTRICTIONS.

`–l`
: Compile into client-side stubs.

`–L`
: When the servers are started in foreground, use syslog(3) to log the server errors instead of printing them on the standard error. See RESTRICTIONS.

`–m`
: Compile into server-side stubs, but do not generate a "main" routine. This option is useful for doing callback-routines and for users who need to write their own "main" routine to do initialization.

`–M`
: Generate multithread-safe stubs for passing arguments and results between `rpcgen`-generated code and user written code. This option is useful for users who want to use threads in their code. See RESTRICTIONS.

`–N`	This option allows procedures to have multiple arguments. It also uses a style of parameter passing that closely resembles C. So, when passing an argument to a remote procedure, you do not have to pass a pointer to the argument, but can pass the argument itself. This behavior is different from the old style of `rpcgen`-generated code. To maintain backward compatibility, this option is not the default.
`–n` *netid*	Compile into server-side stubs for the transport specified by `netid`. There should be an entry for `netid` in the `netconfig` database. This option may be specified more than once, to compile a server that serves multiple transports.
`–o` *outfile*	Specify the name of the output file. If none is specified, standard output is used (-c, -h, -l, -m, -n, -s, -Sc, -Sm ,-Ss, and -t modes only).
`–s` *nettype*	Compile into server-side stubs for all the transports belonging to the class `nettype`. The supported classes are `netpath`, `visible`, `circuit_n`, `circuit_v`, `datagram_n`, `datagram_v`, `tcp`, and `udp` (see `rpc`(3N) for the meanings associated with these classes). This option may be specified more than once. Note: the transports are chosen at run time and not at compile time.
`–Sc`	Generate sample client code that uses remote procedure calls.
`–Sm`	Generate a sample `Makefile` which can be used for compiling the application.
`–Ss`	Generate sample server code that uses remote procedure calls.
`–t`	Compile into RPC dispatch table.
`–T`	Generate the code to support RPC dispatch tables. The options -c, -h, -l, -m, -s, -Sc, -Sm ,-Ss, and -t are used exclusively to generate a particular type of file, while the options -D and -T are global and can be used with the other options.

–Y *pathname*	Give the name of the directory where `rpcgen` will start looking for the C preprocessor.

OPERANDS

infile input file

EXAMPLES

The following example,

```
example% rpcgen -T prot.x
```

generates all five files: `prot.h`, `prot_clnt.c`, `prot_svc.c`, `prot_xdr.c`, and `prot_tbl.i`.

The following example sends the C data-definitions (header) to the standard output:

```
example% rpcgen -h prot.x
```

To send the test version of the -DTEST, server side stubs for all the transport belonging to the class datagram_n to standard output, use:

```
example% rpcgen -s datagram_n -DTEST prot.x
```

To create the server side stubs for the transport indicated by `netid` tcp, use:

```
example% rpcgen -n tcp -o prot_svc.c prot.x
```

The following example,

```
example% rpcgen -a -b prot.x
```

generates all the files for ChorusOS.

RESTRICTIONS

- The -b option is mandatory.
- Server background processing is not supported.
- Signals management is not supported.
- These library functions do not support multithreaded applications.
- Setting parameters using the `rpc_control` is not supported.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`xdr`(3RPC) `xdr`(3RPC)

NAME

xdr – library routines for external data representation

FEATURES

POSIX_SOCKETS

DESCRIPTION

These routines allow C programmers to describe arbitrary data structures in a machine-independent fashion. Data for remote procedure calls are transmitted using these routines.

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

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

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

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

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

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

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

This is a filter primitive that translates between C characters and their external representations. This routine returns one if it succeeds, zero otherwise. Note: encoded characters are not packed, and occupy 4 bytes each. For arrays of characters, it is worth considering `xdr_bytes()`, `xdr_opaque()` or `xdr_string()`.

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

This is a macro that invokes the destroy routine associated with the XDR stream, *xdrs*. Destruction generally involves freeing private data structures associated with the stream. The effect of using *xdrs* after invoking `xdr_destroy()` is undefined.

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

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

```
void
xdr_free(proc, objp)
xdrproc_t proc;
char *objp;
```

This is a generic freeing routine. The first argument is the XDR routine for the object being freed. The second argument is a pointer to the object itself. Note: the pointer passed to this routine is *not* freed, but what it points to *is* freed (recursively).

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

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

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

This is a macro that invokes the in-line routine associated with the XDR stream, *xdrs*. The routine returns a pointer to a contiguous piece of the stream's buffer; *len* is the byte length of the required buffer. Note: the pointer is cast to *long **.

Warning: `xdr_inline()` may return `NULL` (`0`) if it cannot allocate a contiguous piece of a buffer. Therefore the behavior may vary among stream instances; it exists for the sake of efficiency.

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

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

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

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

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

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

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

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

```
xdr_pointer(xdrs, objpp, objsize, xdrobj)
XDR *xdrs;
char **objpp;
u_int objsize;
xdrproc_t xdrobj;
```

This function is similar to `xdr_reference( )` except that it serializes `NULL` pointers and `xdr_reference( )` does not. Thus, `xdr_pointer( )` can represent recursive data structures, such as binary trees or linked lists.

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

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

Warning: this XDR stream implements an intermediate record stream. Therefore there are additional bytes in the stream to provide record boundary information.

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

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

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

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

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

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

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

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

Caution: This routine does not understand `NULL` pointers. Use `xdr_pointer()` instead.

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

This is a macro that invokes the set position routine associated with the XDR stream *xdrs*. The *pos* parameter is a position value obtained from `xdr_getpos()`. This routine returns one if the XDR stream could be repositioned, and zero otherwise.

Caution: It is difficult to reposition some types of XDR streams, so this routine may fail with one type of stream and succeed with another.

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

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

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

This routine initializes the XDR stream object pointed to by *xdrs*. The XDR stream data is written to, or read from, the Standard I/O stream *file*. The *op* parameterd etermines the direction of the XDR stream (either `XDR_ENCODE`, `XDR_DECODE`, or `XDR_FREE`).

Warning: the destroy routine associated with these XDR streams calls `fflush()` on the *file* stream, but never `fclose()`.

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

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

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

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

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

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

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

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

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

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

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

This is a filter primitive that translates between a discriminated C `union` and its corresponding external representation. It first translates the discriminant of the union located at *dscmp*. This discriminant is always an `enum_t`. Next, the union located at *unp* is translated. The *choices* parameter 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`. If the discriminant is not found in the *choices* array, then the `defaultarm( )` procedure is called (if it is not `NULL`). This functions returns one if it succeeds, zero otherwise.

```
xdr_vector(xdrs, arrp, size, elsize, elproc)
XDR *xdrs;
char *arrp;
u_int size, elsize;
xdrproc_t elproc;
```

This is a filter primitive that translates between fixed-length arrays and their corresponding external representations. The *arrp* parameter is the address of the pointer to the array, while *size* is the element count of the array. The *elsize* parameter is the `sizeof( )` each of the array's elements, and *elproc* is an XDR filter that translates between the array elements' C form, and their external representation. This routine returns one if it succeeds, zero otherwise.

```
xdr_void( )
```

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

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

This is a primitive that calls xdr_string(xdrs,sp,MAXUN.UNSIGNED), where *MAXUN.UNSIGNED* is the maximum value of an unsigned integer. The `xdr_wrapstring()` primitive is useful because the RPC package passes a maximum of two XDR routines as parameters, and `xdr_string()`, one of the most frequently used primitives, requires three. Returns one if it succeeds, zero otherwise.

RESTRICTIONS These library calls do not support multi-threaded applications.

ATTRIBUTES See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO `rpc`(3RPC)

Index