CONVEX NFS Programmer's Reference

Order No. DSW-114

Second Edition
October 1990

CONVEX Computer Corporation
Richardson, Texas USA

CONVEX
NFS Programmer's Reference

Order No. DSW-114

Printed in the United States of America

NFS V9.0
Optional Product

The CONVEX Network File System is based on Version 4.0 SUN NFSSRC. CONVEX NFS includes the following services:

- NFS (Network File System) - provides transparent access to remote file systems in a heterogeneous network.
- YP (Yellow Pages) - provides a distributed network lookup service.
- Network Lock Manager - supports advisory file and record locking on NFS filesystems.
- RPC/XDR (Remote Procedure Call/External Data Representation) - set of libraries that implement network transactions.
- Automounter - provides automatic mounting/unmounting of file systems.
- Secure NFS - encrypts network requests to protect against unauthorized access.

The NFS products are built on the RPC/XDR libraries.

- RPC Protocol Compiler (rpcgen) - tool that generates C code to implement the RPC protocol.
- REX (Remote EXecution) - program that executes a command remotely using NFS.
- NETdisk - allows a diskless workstation to boot from a CONVEX system.

CONVEX Network File System documentation:

CONVEX NFS Reference Set, Second Edition
CONVEX NFS System Manager's Guide, Second Edition
CONVEX NFS User's Guide, Second Edition
CONVEX NFS Concepts, First Edition
CONVEX NFS Programmer's Reference, Second Edition

List of Entries:

INSTALL (8)
automount (8)
bootparam (3r)
bootparamd (8)
bootparams (5)
chkey (1)
create_root (8)
des_crypt (3)
domainname (1)
ether (3r)
ethers (3n)
ethers (5)
exportent (3)
exportfs (2)
exportfs (8)
exports (5)
extracting (8)
fix_bootparams (8)
getfh (2)
getnetgrent (3n)
getrpcent (3n)
getrpcport (3r)
intro (3r)
keyenvoy (8c)
keylogin (1)
keyserv (8c)
make_dirs (8)
makedbm (8)
mkfile (8)
mount (3r)
mountd (8c)
netgroup (5)
newkey (8)
nfs (4)
nfsd (8)
nfsstat (8)
nfssvc (2)
opt_software (8)
pong (8)
portmap (8c)
publickey (3r)
publickey (5)
realpath (3)
rex (1c)
rex (3r)
rexd (8c)
rmtab (5)
rnusers (3)r
rpc (3n)
rpc (5)
rpcgen (1)
rpcinfo (8)
rquota (3r)
rquotad (8c)
rstat (3r)
rstatd (8c)
rtime (3n)
rup (1c)
rusers (1c)
rusersd (8c)
rwall (1)
rwall (3r)
rwalld (8c)
setup_client (8)
setup_exec (8)
showmount (8)
spray (3r)
spray (8c)
sprayd (8c)
sunboot (8s)
tftp (1c)
tftpd (8c)
updaters (5)
verify_tapevol_arch (8)
xcrypt (3r)
xdr (3n)
xdrtoc (8)
ypcat (1)
ypclnt (3n)
ypfiles (5)
ypinit (8)
ypmake (8)
ypmatch (1)
yppasswd (1)
yppasswd (3r)
yppasswdd (8c)
yppoll (8)
yppush (8)
ypserv (8)
ypset (8)
ypupdated (8c)
ypwhich (1)
ypxfr (8)

NAME

chkey – change your encryption key

SYNOPSIS

chkey

DESCRIPTION

chkey prompts for your login password, and uses it to encrypt a new encryption key to be stored in the **publickey**(5) database. A valid encryption key must already exist in the database for the command to succeed; initial keys are set up by the superuser with **newkey(8).**

SEE ALSO

keylogin(1), **publickey**(5), **keyserv**(8C), **newkey**(8)

NOTES

chkey is an optional product included with the NFS package for customers in the U.S. and Canada only. For more information, contact your CONVEX sales representative.

NAME

domainname – set or display name of current domain system

SYNOPSIS

domainname [*nameofdomain*]

DESCRIPTION

Without an argument, *domainname* displays the name of the current domain, which typically encompasses a group of hosts under the same administration. As such, the name of a YP domain is normally also a valid Internet domain name and can be used in conjunction with the *sendmail(8)* and the name server *named(8)*.

Only the superuser can set the domainname by giving an argument; this is usually done in the startup script */etc/rc.local*. Currently, domains are only used by the yellow pages to refer collectively to a group of hosts.

SEE ALSO

ypinit(8)

NOTES

domainname is an optional product included with the NFS package; for more information, contact your CONVEX sales representative.

NAME

keylogin – decrypt and store secret key

SYNOPSIS

keylogin

DESCRIPTION

keylogin prompts for the user login password, and uses it to decrypt the user's secret key stored in the **publickey**(5) database. Once decrypted, the user's key is stored by the local key server process **keyserv**(8C) to be used by any secure network services, such as NFS.

Normally, **login**(1) does this work when the user logs in to the system, but running **keylogin** may be necessary if the user did not type a password to **login**(1).

SEE ALSO

chkey(1), **login**(1), **publickey**(5), **keyserv**(8C), **newkey**(8) *keylogin* is an optional product included with the NFS package for customers in the U.S. and Canada only. For more information, contact your CONVEX sales representative.

NAME

rex – execute a command remotely using NFS

SYNOPSIS

rex [**–i** | **–n**] [**–d**] *host command* [*arguments...*] ...

DESCRIPTION

The *rex* program is used to execute commands on another system, in an environment similar to that invoking the program. All environment variables are passed, and the current working directory is preserved. To preserve the working directory, the working file system must be either already mounted on the host or be exported to it. Relative path names will only work if they are within the current file system; absolute path names may cause problems.

Standard input is connected to standard input of the remote command, and standard output and standard error from the remote command are sent to the corresponding files for the *rex* command.

OPTIONS

–i Interactive mode: use remote echoing and special character processing. This option is needed for programs that expect to be talking to a terminal. All terminal modes and window size changes are propagated.

–n No Input: this option causes the remote program to get end-of-file when it reads from standard input, instead of passing standard input from the standard input of the *rex* program. For example, **–n** is necessary when running commands in the background with job control.

–d Debug mode: print out some messages as work is being done.

SEE ALSO

rexd(8c), exports(5)

DIAGNOSTICS

unknown host	Host name not found
cannot connect to server	Host down or not running the server
can't find .	Problem finding the working directory
can't locate mount point	Problem finding current file system

Other error messages may be passed back from the server.

NAME

rpcgen – an RPC protocol compiler

SYNOPSIS

rpcgen *infile*
rpcgen -h [**-o** *outfile*] [*inputfile*]
rpcgen -c [**-o** *outfile*] [*infile*]
rpcgen -s *transport* [**-o** *outfile/* [*infile*]
rpcgen -l [**-o** *outfile/* [*infile*]
rpcgen -m [**-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 **C** known as RPC Language (Remote Procedure Call Language).

rpcgen is normally used as in the first synopsis, where it takes an input file and generates four output files. If the *infile* is named *proto.x*, then *rpcgen* will generate a header file in *proto.h*, XDR routines in *proto_xdr.c*, server-side stubs in *proto_svc.c* and client-side stubs in *proto_clnt.c*.

The other synopses shown above are used when the user does not want to generate all the output files, but only a particular one. Their usage is described in the **OPTIONS** section below.

The C-preprocessor, *cpp(1)*, is run on all input files before they are interpreted by *rpcgen*. All the cpp directives are legal within an *rpcgen* input file. For each type of output file, *rpcgen* defines a special cpp symbol for use by the *rpcgen* programmer:

RPC_HDR
: defined when compiling into header files

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

In addition, *rpcgen* does a little preprocessing of its own. Any line beginning with '%' is passed directly into the output file, uninterpreted by *rpcgen*.

You can customize some of your XDR routines by leaving those data types undefined. For every data type that is undefined, *rpcgen* will assume that there exists a routine with the name 'xdr_' prepended to the name of the undefined type.

OPTIONS

-c Compile into XDR routines.

-h Compile into **C** data-definitions (a header file).

-l Compile into a client-side stubs.

-s *transport*
: Compile into server-side stubs, using the given transport. The supported transports are **udp** and **tcp**. This option may be invoked more than once to compile a server that serves multiple transports.

-m Compile into server-side stubs, but do not produce a *main()* routine. This option is useful for doing callback routines and for those who need to write their own *main()* routine to do initialization.

-o *outfile*
: Specify the name of the output file. If none is specified, standard output is used (**-c**, **-h**, **-l** and **-s** modes only).

USAGE

RPCL Syntax Summary

This summary of RPCL syntax, which is used for *rpcgen* input, is intended more for aiding comprehension than as an exact statement of the language.

Primitive Data Types

[**unsigned**] **char**
[**unsigned**] **short**
[**unsigned**] **int**
[**unsigned**] **long**
[**unsigned**] **hyper**
unsigned
float
double
void
bool

Except for the added boolean data-type **bool**, RPCL is identical to C. *rpcgen* converts **bool** declarations to **int** declarations in the output header file (literally it is converted to a **bool_t**, which has been **#define**'d to be an **int**). Also, **void** declarations may appear only inside of **union** and **program** definitions. For those averse to typing the prefix **unsigned**, the abbreviations **u_char**, **u_short**, **u_int**, **u_long**, and **u_hyper** are available.

DECLARATIONS

An RPC language file consists of a series of definitions:

```
definition-list:
    definition ";"
    definition ";" definition-list
```

It recognizes five types of definitions, as shown below.

```
definition:
    enum-definition
    struct-definition
    union-definition
    typedef-definition
    const-definition
    program-definition
```

Structures

An XDR struct is declared almost exactly like its C counterpart. It looks like the following:

```
struct-definition:
    "struct" struct-ident "{"
        declaration-list
    "}"

declaration-list:
    declaration ";"
    declaration ";" declaration-list
```

As an example, here is an XDR structure to define a two-dimensional coordinate, and the C structure that it gets compiled into in the output header file.

```
struct coord {              struct coord {
    int x;        -->           int x;
    int y;                      int y;
};                          };
                            typedef struct coord coord;
```

The output is identical to the input, except for the added *typedef* at the end of the output. This allows the user to use "coord" instead of "struct coord" when declaring items.

Unions

XDR unions are discriminated unions and look quite different from C unions. XDR unions are more analogous to Pascal variant records than they are to C unions.

```
union-definition:
    "union" union-ident "switch" "(" declaration ")" "{"
        case-list
    "}"

case-list:
    "case" value ":" declaration ";"
    "default" ":" declaration ";"
    "case" value ":" declaration ";" case-list
```

Here is an example of a type that might be returned as the result of a "read data" operation. If there is no error, it returns a block of data. Otherwise, it doesn't return anything.

```
union read_result switch (int errno) {
case 0:
    opaque data[1024];
default:
    void;
};
```

It gets compiled into the following:

```
struct read_result {
    int errno;
    union {
        char data[1024];
    } read_result_u;
};
typedef struct read_result read_result;
```

Notice that the union component of the output struct has the name as the type name, except for the trailing "_u".

Enumerations

XDR enumerations have the same syntax as C enumerations.

```
enum-definition:
    "enum" enum-ident "{"
        enum-value-list
    "}"

enum-value-list:
    enum-value
    enum-value "," enum-value-list

enum-value:
    enum-value-ident
    enum-value-ident "=" value
```

Here is a short example of an XDR enum, and the C enum that it gets compiled into.

```
enum colortype {        enum colortype {
     RED = 0,                RED = 0,
     GREEN = 1,    -->       GREEN = 1,
     BLUE = 2                BLUE = 2,
};                      };
                        typedef enum colortype colortype;
```

Typedef

XDR typedefs have the same syntax as C typedefs.

```
typedef-definition:
    "typedef" declaration
```

Here is an example that defines *fname_type* used for declaring file name strings that have a maximum length of 255 characters.

```
typedef string fname_type<255>; --> typedef char *fname_type;
```

Constants

XDR constants are symbolic constants that may be used wherever a integer constant is used, for example, in array size specifications.

```
const-definition:
    "const" const-ident "=" integer
```

The following defines a constant *DOZEN* equal to 12:

```
const DOZEN = 12;  -->  #define DOZEN 12
```

Programs

RPC programs are declared using the following syntax:

```
program-definition:
    "program" program-ident "{"
        version-list
    "}" "=" value

version-list:
    version ";"
    version ";" version-list

version:
    "version" version-ident "{"
        procedure-list
    "}" "=" value

procedure-list:
    procedure ";"
    procedure ";" procedure-list

procedure:
    type-ident procedure-ident "(" type-ident ")" "=" value
```

For example, here is a sample program statement:

```
/*
 * time.x: Get or set the time. Time is represented as number of seconds
 * since 0:00, January 1, 1970.
 */
program TIMEPROG {
    version TIMEVERS {
        unsigned int TIMEGET(void) = 1;
        void TIMESET(unsigned) = 2;
    } = 1;
} = 44;
```

This file compiles into #defines in the output header file:

```
#define TIMEPROG 44
#define TIMEVERS 1
#define TIMEGET 1
#define TIMESET 2
```

Declarations

In XDR, there are only four kinds of declarations, as shown below.

```
declaration:
    simple-declaration
    fixed-array-declaration
    variable-array-declaration
    pointer-declaration
```

1) **Simple declarations** are like simple C declarations.

```
simple-declaration:
    type-ident variable-ident
```

Example:

```
colortype color;    --> colortype color;
```

2) Fixed-length Array Declarations are like C array declarations:

```
fixed-array-declaration:
    type-ident variable-ident "[" value "]"
```

Example:

```
colortype palette[8];     --> colortype palette[8];
```

3) Variable-Length Array Declarations have no explicit syntax in C, so XDR invents its own using angle-brackets.

```
variable-array-declaration:
    type-ident variable-ident "<" value ">"
    type-ident variable-ident "<" ">"
```

The maximum size is specified between the angle brackets. The size may be omitted, indicating that the array may be of any size.

```
int heights<12>;      /* at most 12 items */
int widths<>;         /* any number of items */
```

Since variable-length arrays have no explicit syntax in C, these declarations are actually compiled into C structures. For example, the "heights" declaration gets compiled into the following struct:

```
struct {
    u_int heights_len;   /* # of items in array */
    int *heights_val;    /* pointer to array */
} heights;
```

Note that the number of items in the array is stored in the "_len" component and the pointer to the array is stored in the "_val" component. The first part of each of these components' names is the same as the name of the declared XDR variable.

4) Pointer Declarations are made in XDR exactly as they are in C. You can't really send pointers over the network, but you can use XDR pointers for sending recursive data types such as lists and trees. The type is actually called "optional-data," not "pointer," in XDR language.

```
pointer-declaration:
    type-ident "*" variable-ident
```

Example:

```
listitem *next;  -->  listitem *next;
```

Special Cases

There are a few exceptions to the rules previously described:

Booleans: C has no built-in boolean type. However, the RPC library has a boolean type called *bool_t* that is either *TRUE* or *FALSE*. Variables declared as type *bool* in XDR language are compiled into *bool_t* in the output header file.

Example:

```
bool married;  -->  bool_t married;
```

Strings: C has no built-in string type, but instead uses the null-terminated "char *" convention. In XDR language, strings are declared using the "string" keyword, and compiled into "char *" character pointers in the output header file. The maximum size contained in the angle brackets specifies the maximum number of characters allowed in the strings (not counting the *NULL* character). The maximum size may be left off, indicating a string of arbitrary length.

Examples:

```
string name<32>;     -->  char *name;
string longname<>;   -->  char *longname;
```

Opaque Data: Opaque data types are used in RPC and XDR to describe untyped data, which are just sequences of arbitrary bytes. They may be declared either as fixed or variable length arrays.

Examples:

```
opaque diskblock[512];   -->  char diskblock[512];

opaque filedata<1024>;   -->  struct {
                                 u_int filedata_len;
                                 char *filedata_val;
                              } filedata;
```

Voids: In a void declaration, the variable is not named. The declaration is just "void" and nothing else. Void declarations can only occur in two places: union definitions and program definitions (as the argument or result of a remote procedure).

SEE ALSO

Remote Procedure Call: Programming Guide and *External Data Representation: Protocol Specification* in the *CONVEX NFS Reference Set*

BUGS

Nesting is not supported. As a work-around, structures can be declared at top-level, and their name used inside other structures in order to achieve the same effect.

Name clashes can occur when using program definitions, since the apparent scoping does not really apply. Most of these can be avoided by giving unique names for programs, versions, procedures and types.

NOTES

The *hyper* type is really equivalent to the 64-bit CONVEX "long long" integer type and as such is an extension to the XDR Protocol Specification. *hyper* integers may not be supported by other vendors.

rpcgen is an optional product included with the NFS package; for more information, contact your CONVEX sales representative.

NAME

rup – show host status of local machines (RPC version)

SYNOPSIS

rup [**–o**] [**–h** | **–l** | **–t**] [host ...]

DESCRIPTION

rup gives a status similar to *uptime* for remote machines; it broadcasts on the local network, and displays the responses it receives.

Normally, the listing is in the order that responses are received, but this order can be changed by specifying one of the options listed below.

When *host* arguments are given, rather than broadcasting, *rup* will only query the list of specified hosts.

A remote host will only respond if it is running the *rstatd* daemon, which is normally started up from *inetd*(8C).

OPTIONS

–h sort the display alphabetically by host name.

–l sort the display by load average

–t sort the display by up time.

–o use a much larger timeout as well as doing an extra broadcast for very old versions of the *rstatd* daemon.

FILES

/etc/inetd.conf

SEE ALSO

ruptime(1C), inetd(8C), rstatd(8C)

BUGS

Broadcasting does not work through gateways.

NOTES

rup is an optional product included with the NFS package; for more information, contact your CONVEX sales representative.

NAME

rusers – who's logged in on local machines (RPC version)

SYNOPSIS

rusers [**–a**] [**–l**] [**–o**] [**–h** | **–i** | **–u**] [host ...]

DESCRIPTION

The *rusers* command produces output similar to *users*(1) and *who*(1), but for remote machines. It broadcasts on the local network, and prints the responses it receives. Normally, the listing is in the order that responses are received, but this order can be changed by specifying one of the options listed below. When *host* arguments are given, rather than broadcasting *rusers* will only query the list of specified hosts.

The default is to print out a listing in the style of *users*(1) with one line per machine. When the **-l** flag is given, a *rwho*(1C) style listing is used. In addition, if a user hasn't typed to the system for a minute or more, the idle time is reported.

A remote host will only respond if it is running the *rusersd* daemon, which is normally started up from *inetd*(8C).

OPTIONS

–a give a report for a machine even if no users are logged on.

–h sort alphabetically by host name.

–i sort by idle time.

–l give a longer listing in the style of *who(1)*.

–u sort by number of users.

–o use a much larger timeout as well as doing an extra broadcast for very old versions of the *rusersd* daemon.

FILES

/etc/utmp

SEE ALSO

rwho(1C), inetd(8C), rusersd(8C)

BUGS

Broadcasting does not work through gateways.

NOTES

rusers is an optional product included with the NFS package; for more information, contact your CONVEX sales representative.

NAME

rwall – write to all users over a network

SYNOPSIS

/usr/etc/rwall host ... [**–n** netgroup ...] [**–h** host ...]

DESCRIPTION

rwall reads a message from standard input until end-of-file. It then sends this message, preceded by the line "Broadcast Message ...", to all users logged in on the specified host machines. Non-printable characters will be deleted from the message before it is displayed to the users.

With the **-n** option, it sends to the specified network groups, which are defined in *netgroup*(5).

A machine can only receive such a message if it is running *rwalld*(8C), which is normally started up from */etc/inetd.conf* by the daemon *inetd*(8C).

FILES

/etc/inetd.conf

SEE ALSO

wall(1), netgroup(5), rwalld(8C), shutdown(8)

BUGS

The timeout is fairly short in order to be able to send to a large group of machines (some of which may be down) in a reasonable amount of time. Thus the message may not get through to a heavily loaded machine.

NOTES

rwall is an optional product included with the NFS package; for more information, contact your CONVEX sales representative.

NAME

tftp – trivial file transfer program

SYNOPSIS

tftp [*host*]

DESCRIPTION

tftp is the user interface to the Internet TFTP (Trivial File Transfer Protocol), which allows users to transfer files to and from a remote machine. The remote *host* may be specified on the command line, in which case *tftp* uses *host* as the default host for future transfers (see the **connect** command below).

COMMANDS

Once *tftp* is running, it issues the prompt **tftp>** and recognizes the following commands:

connect *host-name* [*port*]
Set the *host* (and optionally *port*) for transfers. Note that the TFTP protocol, unlike the FTP protocol, does not maintain connections between transfers; thus, the *connect* command does not actually create a connection, but merely remembers what host is to be used for transfers. You do not have to use the *connect* command; the remote host can be specified as part of the *get* or *put* commands.

mode *transfer-mode*
Set the mode for transfers; *transfer-mode* may be one of *ascii* or *binary*. The default is *ascii*.

put *file*
put *localfile remotefile*
put *file1 file2 ... fileN remote-directory*
Put a file or set of files to the specified remote file or directory. The destination can be in one of two forms: a filename on the remote host, if the host has already been specified, or a string of the form *host:filename* to specify both a host and filename at the same time. If the latter form is used, the hostname specified becomes the default for future transfers. If the remote-directory form is used, the remote host is assumed to be a *UNIX* machine.

get *filename*
get *remotename localname*
get *file1 file2 ... fileN*
Get a file or set of files from the specified *sources*. *Source* can be in one of two forms: a filename on the remote host, if the host has already been specified, or a string of the form *host:filename* to specify both a host and filename at the same time. If the latter form is used, the last hostname specified becomes the default for future transfers.

quit Exit *tftp*. An end of file also exits.

verbose
Toggle verbose mode.

trace Toggle packet tracing.

status Show current status.

rexmt *retransmission-timeout*
Set the per-packet retransmission timeout, in seconds.

timeout *total-transmission-timeout*
Set the total transmission timeout, in seconds.

ascii Shorthand for "mode ascii"

binary Shorthand for "mode binary"

? [*command-name* ...]
Print help information.

BUGS

Because there is no user-login or validation within the *TFTP* protocol, the remote site will probably have some sort of file-access restrictions in place. The exact methods are specific to each site and therefore difficult to document here.

NOTES

tftp is an optional product included with the NFS package; for more information, contact your CONVEX sales representative.

NAME

ypcat – print values in a YP data base

SYNOPSIS

ypcat [**–k**] [**–t**] [**–d** *domainname*] *mname*
ypcat –x

DESCRIPTION

ypcat prints out values in a yellow pages (YP) map specified by *mname*, which may be either a *mapname* or a map *nickname*. Since *ypcat* uses the YP network services, no YP server is specified.

To look at the network-wide password database, *passwd.byname*, (with the nickname *passwd*). Type in:

ypcat passwd

Refer to *ypfiles*(5) and *ypserv*(8) for an overview of the yellow pages.

OPTIONS

–k Before printing the value of a key, print the key itself, followed by a space.

–t Inhibit translation of *mname* to *mapname*. For example, *ypcat –t passwd* will fail because there is no map named *passwd*, whereas *ypcat passwd* will be translated to *ypcat passwd.byname*.

–d Specify a domain other that the default domain. The default domain is returned by *domainname*.

–x Display the map nickname table. This lists the nicknames (*mnames*) the command knows of, and indicates the *mapname* associated with each nickname.

SEE ALSO

ypfiles(5), ypserv(8), ypmatch(1), domainname(1)

NOTES

ypcat is an optional product included with the NFS package; for more information, contact your CONVEX sales representative.

NAME

ypmatch – print the value of one or more keys from a yp map

SYNOPSIS

ypmatch [**-d** *domain*] [**-k**] [**-t**] *key* ... *mname*
ypmatch -x

DESCRIPTION

ypmatch prints the values associated with one or more keys from the yellow pages (YP) map (database) specified by a *mname*, which may be either a *mapname* or an map *nickname*.

Multiple keys can be specified; the same map will be searched for all. The keys must be exact values insofar as capitalization and length are concerned. No pattern matching is available. If a key is not matched, a diagnostic message is produced.

OPTIONS

-d Specify a domain other that the default domain.

-k Before printing the value of a key, print the key itself, followed by a colon (:). This is useful only if the keys are not duplicated in the values, or you've specified so many keys that the output could be confusing.

-t Inhibit translation of nickname to mapname. For example, *ypmatch -t zippy passwd* will fail because there is no map named *passwd*, while *ypmatch zippy passwd* will be translated to *ypmatch zippy passwd.byname*.

-x Display the map nickname table. This lists the nicknames (*mnames*) the command knows of, and indicates the *mapname* associated with each nickname.

SEE ALSO

ypfiles(5), ypcat(1), domainname(1)

NOTES

ypmatch is an optional product included with the NFS package; for more information, contact your CONVEX sales representative.

NAME

yppasswd – change login password in yellow pages

SYNOPSIS

yppasswd [*name*]

DESCRIPTION

yppasswd changes (or installs) a password associated with the user *name* (your own name by default) in the yellow pages and rebuilds the password database files by invoking *mkpasswd*.

yppasswd prompts for the old yellow pages password and then for the new one. The caller must supply both. The new password must be typed twice, to forestall mistakes.

The first time the new password is entered, *yppasswd* checks to see if any password restrictions are enabled for *username*. Password aging is enabled if there is a non-null age field for *username* in the yellow pages restrictions file; password typing is enabled if a "Y" appears in the type field of the user's entry in the yellow pages restrictions file.

If "aging" is enabled, and the password has not "aged" sufficiently (the minimum number of weeks a password must remain unchanged has not elapsed), the new password is rejected and *yppasswd* terminates (see *passwd*(5)). If password typing is enabled, a check is made to insure that the new password meets construction requirements. These requirements are:

Each password must have at least six characters.

Each password must contain at least two alphabetic characters and at least one numeric or special character. In this case "alphabetic" means upper and lower case letters.

Each password must differ from the user's *login* name and any reverse or circular shift of that name. For comparison purposes, an upper case letter and its corresponding lower case letter are equivalent.

New passwords must differ from the old by at least three characters. As before, an upper case letter and its lower case counterpart are equivalent.

When no restrictions are being applied, new passwords must be at least four characters long if they use a sufficiently rich alphabet and at least six characters long if monocase.

Only the owner of the name or the superuser may change a password; in either case you must prove you know the old password.

SEE ALSO

passwd(1), mkpasswd(5), passwd(5), pwrestrict(5), ypfiles(5), yppasswdd(8C)

BUGS

The update protocol passes all the information to the server in one *rpc* call, without ever looking at it. Thus if you type in your old password incorrectly, you will not be notified until after you have entered your new password.

NAME

ypwhich – which host is the YP server or map master?

SYNOPSIS

ypwhich [**–d** *domain*] [**–V1** | **–V2**] [*hostname*]
ypwhich [**–d** *domain*] [**–t**] **–m** [*mname*]
ypwhich –x

DESCRIPTION

ypwhich tells which YP server supplies yellow pages services to a YP client, or which is the master for a map. If invoked without arguments, it gives the YP server for the local machine. If *hostname* is specified, that machine is queried to find out which YP master it is using.

Refer to *ypfiles*(5) and *ypserv*(8) for an overview of the yellow pages.

If the **–m** switch is used without *mname*, a list of every map in the domain and the master of each will be printed. If *mname* is specified, only the master YP server for that map is printed. *mname* may be a mapname, or a nickname for a mapname. Mapnames and nicknames are described in *ypcat*(1).

OPTIONS

–d Use *domain* instead of the default domain.

–V1 Which server is serving v.1 YP protocol-speaking client processes?

–V2 Which server is serving v.2 YP protocol client processes?

If neither version is specified, *ypwhich* attempts to locate the server that supplies the (current) v.2 services. If there is no v.2 server currently bound, *ypwhich* then attempts to locate the server supplying the v.1 services. Since YP servers and YP clients are both backward compatible, the user need seldom be concerned about which version is currently in use.

–t Inhibit nickname translation; useful if there is a mapname identical to a nickname. This is not true of any Sun-supplied map.

–m Find the master YP server for a map. No *hostname* can be specified with **–m**. *mname* can be a mapname, or a nickname for a map.

–x Display the map nickname table. This lists the nicknames (*mnames*) the command knows of, and indicates the *mapname* associated with each nickname.

SEE ALSO

ypfiles(5), rpcinfo(8), ypset(8), ypserv(8), domainname(1)

NOTES

ypwhich is an optional product included with the NFS package; for more information, contact your CONVEX sales representative.

NAME

exportfs – set export characteristics of file/directory

SYNOPSIS

```
#include <sys/types.h>
#include <sys/vfs.h>
#include <nfs/export.h>

exportfs(path, ex)
char *path;
struct export *ex;
```

DESCRIPTION

exportfs sets characteristics pertaining to NFS exported files and directories. See *exportfs*(8) for more information on the types of options that can be specified.

This call is limited to the superuser.

RETURN VALUE

exportfs returns zero if successful; -1 if not.

ERRORS

[EPERM] The effective uid of the caller is not superuser.

[ELOOP] Too many levels of symbolic links were encounted.

[EINVAL] The *ex_flags* or *ex_auth* specified are invalid.

[ENOENT] *path* does not exist.

[EFAULT] *path* or *ex* evaluate to an address outside the process's allocated address space.

[ENAMETOOLONG] *path* was longer than the maximum file name length.

SEE ALSO

mountd(8C), exportfs(8)

NOTES

exportfs is an optional product included with the NFS package; for more information, contact your CONVEX sales representative.

NAME

getfh – convert file descriptor to file handle

SYNOPSIS

```
#include <errno.h>
#include <rpc/rpc.h>
#include <sys/time.h>
#include <nfs/nfs.h>

getfh(path, fhp)
char *path;
fhandle_t *fhp;
```

DESCRIPTION

getfh returns an NFS file handle for the file *path*. The file handle is stored in a user buffer pointed to by *fhp*. This file handle may be used in the NFS remote file system protocol.

RETURN VALUE

getfh returns zero if successful; -1 if not.

ERRORS

[EPERM]	The effective uid of the caller is not superuser.
[ELOOP]	Too many levels of symbolic links were encounted.
[ENOENT]	*path* does not exist.
[EREMOTE]	*fd* referred to an open file on an NFS partition.
[EFAULT]	*path* or *fhp* evaluate to an address outside the process's allocated address space.
[ENAMETOOLONG]	*path* was longer than the maximum file name length.

SEE ALSO

mountd(8C)

NOTES

getfh is an optional product included with the NFS package; for more information, contact your CONVEX sales representative.

NAME

nfssvc, async_daemon – NFS daemons

SYNOPSIS

nfssvc(sock)
int sock;

async_daemon()

DESCRIPTION

nfssvc starts an NFS daemon listening on socket *sock*. The socket must be AF_INET, and SOCK_DGRAM (protocol UDP/IP). The system call will return only if the process is killed.

async_daemon implements the NFS daemon that handles asynchronous I/O for an NFS client. The system call never returns.

Both system calls result in kernel-only processes with user memory discarded.

BUGS

These two system calls allow kernel processes to have user context. There should be a way to dynamically create kernel-only processes instead of having to make system calls from user level to simulate this.

SEE ALSO

mountd(8C)

NAME

bootparam – bootparam protocol

PROTOCOL

/usr/include/rpcsvc/bootparam_prot.x

DESCRIPTION

The bootparam protocol is used for providing information to the diskless clients necessary for booting.

PROGRAMMING

#include <rpcsvc/bootparam.h>

XDR Routines

The following XDR routines are available in **librpcsvc**:

xdr_bp_whoami_arg
xdr_bp_whoami_res
xdr_bp_getfile_arg
xdr_bp_getfile_res

SEE ALSO

bootparams(5), bootparamd(8)

NOTES

bootparam is an optional product included with the NFS package; for more information, contact your CONVEX sales representative.

NAME

des_crypt, ecb_crypt, cbc_crypt, des_setparity – fast DES encryption

SYNOPSIS

```
#include <des_crypt.h>

int ecb_crypt(key, data, datalen, mode)
char *key;
char *data;
unsigned datalen;
unsigned mode;

int cbc_crypt(key, data, datalen, mode, ivec)
char *key;
char *data;
unsigned datalen;
unsigned mode;
char *ivec;

void des_setparity(key)
char *key;
```

DESCRIPTION

ecb_crypt( ) and **cbc_crypt( )** implement the NBS DES (Data Encryption Standard). These routines are faster and more general purpose than **crypt**(3). They also are able to utilize DES hardware, if available. **ecb_crypt( )** encrypts in ECB (Electronic Code Book) mode, which encrypts blocks of data independently. **cbc_crypt( )** encrypts in CBC (Cipher Block Chaining) mode, which chains together successive blocks. CBC mode protects against insertions, deletions and substitutions of blocks. Also, regularities in the clear text will not appear in the cipher text.

Here is how to use these routines. The first parameter, *key*, is the 8-byte encryption key with parity. To set the key's parity, which for DES is in the low bit of each byte, use *des_setparity*. The second parameter, *data*, contains the data to be encrypted or decrypted. The third parameter, *datalen*, is the length in bytes of *data*, which must be a multiple of 8. The fourth parameter, *mode*, is formed by OR'ing together some things. For the encryption direction 'or' in either DES_ENCRYPT or DES_DECRYPT. For software versus hardware encryption, 'or' in either DES_HW or DES_SW. If DES_HW is specified, and there is no hardware, then the encryption is performed in software and the routine returns DESERR_NOHWDEVICE. For *cbc_crypt*, the parameter *ivec* is the the 8-byte initialization vector for the chaining. It is updated to the next initialization vector upon return.

SEE ALSO

des(1), **crypt**(3)

DIAGNOSTICS

DESERR_NONE	No error.
DESERR_NOHWDEVICE	Encryption succeeded, but done in software instead of the requested hardware.
DESERR_HWERR	An error occurred in the hardware or driver.
DESERR_BADPARAM	Bad parameter to routine.

Given a result status *stat*, the macro DES_FAILED(stat) is false only for the first two statuses.

NOTES

The routines described in this manual page are optional products included with the NFS package for customers in the U.S. and Canada only. For more information, contact your CONVEX sales representative.

NAME

ether – monitor traffic on the Ethernet

SYNPOSIS

```
#include <rpc/rpc.h>
#include <sys/time.h>
#include <rpcsvc/ether.h>
```

RPC INFO

program number:

ETHERSTATPROG

xdr routines:

```
xdr_etherstat(xdrs, es)
        XDR *xdrs;
        struct etherstat *es;
xdr_etheraddrs(xdrs, ea)
        XDR *xdrs;
        struct etheraddrs *ea;
xdr_etherhtable(xdrs, hm)
        XDR *xdrs;
        struct etherhmem **hm;
xdr_etherhmem(xdrs, hm)
        XDR *xdrs;
        struct etherhmem **hm;
xdr_etherhbody(xdrs, hm)
        XDR *xdrs;
        struct etherhmem *hm;
xdr_addrmask(xdrs, am)
        XDR *xdrs;
        struct addrmask *am;
```

xdr_etherhmem processes a single *etherhmem* structure. *xdr_etherhtable* processes an array of HASHSIZE **struct etherhmems*. The ***etherhmem* field of *etheraddrs* is actually a hashtable, that is, it is a pointer to an array of HASHSIZE *hmem* pointers.

procs:

ETHERSTATPROC_GETDATA
: no args, returns struct etherstat

ETHERSTATPROC_ON
: no args or results, puts server in promiscuous mode

ETHERSTATPROC_OFF
: no args or results, puts server in promiscuous mode

ETHERSTATPROC_GETSRCDATA
: no args, returns struct etheraddrs with information about source of packets

ETHERSTATPROC_GETDSTDATA
: no args, returns struct etheraddrs with information about destination of packets

ETHERSTATPROC_SELECTSRC
: takes struct mask as argument, no results sets a mask for source

ETHERSTATPROC_SELECTDST
: takes struct mask as argument, no results sets a mask for dst

ETHERSTATPROC_SELECTPROTO

takes struct mask as argument, no results
sets a mask for proto
ETHERSTATPROC_SELECTLNTH
takes struct mask as argument, no results
sets a mask for lnth

versions:
ETHERSTATVERS_ORIG

structures:

```
/*
 * all ether stat's except src, dst addresses
 */
struct etherstat {
        struct timeval   e_time;
        unsigned long    e_bytes;
        unsigned long    e_packets;
        unsigned long    e_bcast;
        unsigned long    e_size[NBUCKETS];
        unsigned long    e_proto[NPROTOS];
};
/*
 * member of address hash table
 */
struct etherhmem {
        int h_addr;
        unsigned h_cnt;
        struct etherhmem *h_nxt;
};
/*
 * src, dst address info
 */
struct etheraddrs {
        struct timeval   e_time;
        unsigned long    e_bytes;
        unsigned long    e_packets;
        unsigned long    e_bcast;
        struct etherhmem **e_addrs;
};
/*
 * for size, a_addr is lowvalue, a_mask is high value
 */
struct addrmask {
        int a_addr;
        int a_mask;      /* 0 means wild card */
};
```

SEE ALSO

On a Sun, traffic(1), etherfind(8C), etherd(8C)

NOTES

The server daemon is not currently available on a Convex Computer. These library routines are provided so that a Convex Computer can probe a remote Sun, running the *rpc.etherd* program, for the requested ethernet statistics.

ether is an optional product included with the NFS package; for more information, contact your CONVEX sales representative.

NAME

ethers, ether_ntoa, ether_aton, ether_ntohost, ether_hostton, ether_line – Ethernet address mapping operations

SYNOPSIS

```
#include <sys/types.h>
#include <sys/socket.h>
#include <sync/queue.h>
#include <sync/sema.h>
#include <net/if.h>
#include <net/if_arp.h>
#include <netinet/in.h>
#include <netinet/if_ether.h>

char *
ether_ntoa(e)
        struct ether_addr *e;

struct ether_addr *
ether_aton(s)
        char *s;

ether_ntohost(hostname, e)
        char *hostname;
        struct ether_addr *e;

ether_hostton(hostname, e)
        char *hostname;
        struct ether_addr *e;

ether_line(l, e, hostname)
        char *l;
        struct ether_addr *e;
        char *hostname;
```

DESCRIPTION

These routines are useful for mapping 48 bit Ethernet numbers to their ASCII representations or their corresponding host names, and vice versa.

The function *ether_ntoa()* converts a 48 bit Ethernet number pointed to by *e* to its standard ACSII representation; it returns a pointer to the ASCII string. The representation is of the form: *x*:*x*:*x*:*x*:*x*:*x* where *x* is a hexadecimal number between 0 and ff. The function *ether_aton()* converts an ASCII string in the standard representation back to a 48 bit Ethernet number; the function returns NULL if the string cannot be scanned successfully.

The function *ether_ntohost()* maps an Ethernet number (pointed to by *e*) to its associated hostname. The string pointed to by *hostname* must be long enough to hold the hostname and a NULL character. The function returns zero upon success and non-zero upon failure. Inversely, the function *ether_hostton()* maps a hostname string to its corresponding Ethernet number; the function modifies the Ethernet number pointed to by *e*. The function also returns zero upon success and non-zero upon failure.

The function *ether_line()* scans a line (pointed to by *l*) and sets the hostname and the Ethernet number (pointed to by *e*). The string pointed to by *hostname* must be long enough to hold the hostname and a NULL character. The function returns zero upon success and non-zero upon failure. The format of the scanned line is described by *ethers*(5).

FILES

/etc/ethers (or the Yellow Pages maps **ethers.byaddr** and **ethers.byname**)

SEE ALSO

ethers(5)

NOTES

ethers is an optional product included with the NFS package; for more information, contact your CONVEX sales representative.

NAME

exportent, getexportent, setexportent, addexportent, remexportent, endexportent, getexportopt – get exported file system information

SYNOPSIS

```
#include <stdio.h>
#include <exportent.h>
FILE *setexportent()
struct exportent *getexportent(filep)
        FILE *filep;
int addexportent(filep, dirname, options)
        FILE *filep;
        char *dirname;
        char *options;
int remexportent(filep, dirname)
        FILE *filep;
        char *dirname;
char *getexportopt(xent, opt)
        struct exportent *xent;
        char *opt;
void endexportent(filep)
        FILE *filep;
```

DESCRIPTION

These routines access the exported filesystem information in */etc/xtab*.

setexportent() opens the export information file and returns a file pointer to use with *getexportent*, *addexportent*, *remexportent*, and *endexportent*. **getexportent()** reads the next line from *filep* and returns a pointer to an object with the following structure containing the broken-out fields of a line in the file, */etc/xtab* The fields have meanings described in *exports*(5).

```
#define ACCESS_OPT  "access"  /* machines that can mount fs */
#define ROOT_OPT    "root"    /* machines with root access of fs */
#define RO_OPT      "ro"      /* export read-only */
#define RW_OPT      "rw"      /* export read-mostly */
#define ANON_OPT    "anon"    /* uid for anonymous requests */
#define ASYNC_OPT   "async"   /* export for asynchronous writes */
#define SECURE_OPT  "secure"  /* require secure NFS for access */
#define WINDOW_OPT  "window"  /* expiration window for credential */
struct exportent {
        char *xent_dirname;  /* directory (or file) to export */
        char *xent_options;  /* options, as above */
};
```

addexportent() adds the *exportent()* to the end of the open file *filep*. It returns 0 if successful and –1 on failure. *remexportent()* removes the indicated entry from the list. It also returns 0 on success and –1 on failure. *getexportopt()* scans the *xent_options* field of the *exportent()* structure for a substring that matches *opt*. It returns the string value of *opt*, or NULL if the option is not found.

endexportent() closes the file.

FILES

/etc/exports
/etc/xtab

SEE ALSO

exports(5), xtab(5), exportfs(8)

DIAGNOSTICS

NULL pointer (0) returned on EOF or error.

BUGS

The returned *exportent()* structure points to static information that is overwritten in each call.

NOTES

The routines described in this manual page are optional products included with the NFS package; for more information, contact your CONVEX sales representative.

NAME

getnetgrent, setnetgrent, endnetgrent, innetgr – get network group entry

SYNOPSIS

innetgr(netgroup, machine, user, domain)
char *netgroup, *machine, *user, *domain;

setnetgrent(netgroup)
char *netgroup

endnetgrent()

getnetgrent(machinep, userp, domainp)
char **machinep, **userp, **domainp;

DESCRIPTION

Inngetgr returns 1 or 0, depending on whether *netgroup* contains the machine, user, domain triple as a member. Any of the three strings machine, user, or domain can be NULL, in which case it signifies a wild card.

getnetgrent returns the next member of a network group. After the call, machinep will contain a pointer to a string containing the name of the machine part of the network group member, and similarly for userp and domainp. If any of machinep, userp or domainp is returned as a NULL pointer, it signifies a wild card. *getnetgrent* will malloc space for the name. This space is released when a *endnetgrent* call is made. *getnetgrent* returns 1 if it succeeding in obtaining another member of the network group, 0 if it has reached the end of the group.

setnetgrent establishes the network group from which *getnetgrent* will obtain members, and also restarts calls to *getnetgrent* from the beginning of the list. If the previous *setnetgrent* call was to a different network group, a endnetgrent call is implied. *endnetgrent* frees the space allocated during the *getnetgrent* calls.

FILES

/etc/netgroup
/etc/yp/*domain*/netgroup
/etc/yp/*domain*/netgroup.byuser
/etc/yp/*domain*/netgroup.byhost

BUGS

innetgr always returns 0 if the yellow pages aren't running.

NOTES

getnetgrent is an optional product included with the NFS package; for more information, contact your CONVEX sales representative.

NAME

getrpcent, getrpcbyname, getrpcbynumber, setrpcent, endrpcent – get rpc entry

SYNOPSIS

```
#include <netdb.h>

struct rpcent *getrpcent()

struct rpcent *getrpcbyname(name)
char *name;

struct rpcent *getrpcbynumber(number)
int number;

setrpcent(stayopen)
int stayopen

endrpcent()
```

DESCRIPTION

getrpcent, *getrpcbyname*, and *getrpcbynumber* each return a pointer to an object with the following structure containing the broken-out fields of a line in the rpc program number data base, */etc/rpc*.

```
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.

getrpcent reads the next line of the file, opening the file if necessary.

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

endrpcent closes the file.

getrpcbyname and *getrpcbynumber* sequentially search from the beginning of the file until a matching rpc program name or program number is found, or until EOF is encountered.

FILES

/etc/rpc
/etc/yp/*domainname*/rpc.byname
/etc/yp/*domainname*/rpc.bynumber

SEE ALSO

rpc(5), rpcinfo(8), ypserv(8)

DIAGNOSTICS

Null pointer (0) returned on EOF or error.

BUGS

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

NAME

getrpcport – get RPC port number

SYNOPSIS

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

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

DESCRIPTION

getrpcport returns the port number for version *versnum* of the RPC program *prognum* running on *host* and using protocol *proto*. It returns 0 if it cannot contact the portmapper, 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 indeed registered. The version mismatch will be detected upon the first call to the service.

proto is typically **IPPROTO_UDP** or **IPPROTO_TCP**.

NOTES

getrpcport is an optional product included with the NFS package; for more information, contact your CONVEX sales representative.

NAME

intro – introduction to RPC service library functions and protocols

DESCRIPTION

These functions constitute the RPC service library. Most of these describe RPC protocols. The PROTOCOL section describes how to access the protocol description file. This file may be compiled with *rpcgen*(1) to produce data definitions and XDR routines. Procompiled versions of header files sometimes exist as *<rpcsvc/*.h>* and precompiled XDR routines and programming interfaces to the protocols sometimes exist in *librpcsvc*. Warning: some of these header files and XDR routines were hand-written because they existed before *rpcgen*. They do not correspond to their protocol description file. In order to get the link editor to load this library, use the *-lrpcsvc* option of *cc*. Information about the availability of programming interfaces to these protocols is available under PROGRAMMING section of each manual page.

Some routines in the *librpcsvc* library do not correspond to protocols, but are useful utilities for RPC programming. These are distinguished by the presence of the SYNOPSIS section instead of the usual PROTOCOL section.

LIST OF STANDARD RPC SERVICES

routine	*on page*	*description*
bootparam( )	**bootparam**(3R)	bootparam protocol
ether( )	**ether**(3R)	monitor traffic on the Ethernet
get( )	**publickey**(3R)	get secret key
getrpcport( )	**getrpcport**(3R)	get RPC port number
getsecretkey( )	**publickey**(3R)	get secret key
key( )	**publickey**(3R)	get secret key
klm_prot( )	**klm_prot**(3R)	protocol between kernel and local lock manager
mount( )	**mount**(3R)	keep track of remotely mounted filesystems
nlm_prot( )	**nlm_prot**(3R)	protocol between local and remote network lock manager
public( )	**publickey**(3R)	get secret key
rex( )	**rex**(3R)	remote execution protocol
rnusers( )	**rnusers**(3R)	return information about users on remote machines
rquota( )	**rquota**(3R)	implement quotas on remote machines
rstat( )	**rstat**(3R)	get performance data from remote kernel
rusers( )	**rnusers**(3R)	return information about users on remote machines
rwall( )	**rwall**(3R)	write to specified remote machines
secret( )	**publickey**(3R)	get secret key
sm_inter( )	**sm_inter**(3R)	status monitor protocol
spray( )	**spray**(3R)	scatter data in order to check the network
xcrypt( )	**xcrypt**(3R)	hex encryption and utility routines
yp( )	**yp**(3R)	Yellow Pages protocol
yppasswd( )	**yppasswd**(3R)	update user password in Yellow Pages

NOTES

The routines described in this manual page are optional products included with the NFS package; for more information, contact your CONVEX sales representative.

NAME

mount – keep track of remotely mounted filesystems

DESCRIPTION

The mount protocol is separate from, but related to, the NFS protocol. It provides all of the operating system specific services to get the NFS off the ground — looking up path names, validating user identity, and checking access permissions. Clients use the mount protocol to get the first file handle, which allows them entry into a remote filesystem.

The mount protocol is kept separate from the NFS protocol to make it easy to plug in new access checking and validation methods without changing the NFS server protocol.

Note: the protocol definition implies stateful servers because the server maintains a list of client's mount requests. The mount list information is not critical for the correct functioning of either the client or the server. It is intended for advisory use only, for example, to warn people when a server is going down.

SYNOPSIS

```
#include <rpc/rpc.h>
#include <errno.h>
#include <sys/time.h>
#include <nfs/nfs.h>
#include <rpcsvc/mount.h>
```

RPC INFO

program number:

MOUNTPROG

xdr routines:

```
xdr_exportbody(xdrs, ex)
        XDR *xdrs;
        struct exports *ex;
xdr_exports(xdrs, ex);
        XDR *xdrs;
        struct exports **ex;
xdr_fhandle(xdrs, fh);
        XDR *xdrs;
        fhandle_t *fp;
xdr_fhstatus(xdrs, fhs);
        XDR *xdrs;
        struct fhstatus *fhs;
xdr_groups(xdrs, gr);
        XDR *xdrs;
        struct groups *gr;
xdr_mountbody(xdrs, ml)
        XDR *xdrs;
        struct mountlist *ml;
xdr_mountlist(xdrs, ml);
        XDR *xdrs;
        struct mountlist **ml;
xdr_path(xdrs, path);
        XDR *xdrs;
        char **path;
```

procs:

MOUNTPROC_MNT

argument of xdr_path, returns fhstatus.
Requires unix authentication.

MOUNTPROC_DUMP
no args, returns struct mountlist

MOUNTPROC_UMNT
argument of xdr_path, no results.
requires unix authentication.

MOUNTPROC_UMNTALL
no arguments, no results.
requires unix authentication.
umounts all remote mounts of sender.

MOUNTPROC_EXPORT
MOUNTPROC_EXPORTALL
no args, returns struct exports

versions:
MOUNTVERS_ORIG

structures:

```
struct mountlist {              /* what is mounted */
        char *ml_name;
        char *ml_path;
        struct mountlist *ml_nxt;
};
struct fhstatus {
        int fhs_status;
        fhandle_t fhs_fh;
};
/*
 * List of exported directories
 * An export entry with ex_groups
 * NULL indicates an entry which is exported to the world.
 */
struct exports {
        dev_t           ex_dev;         /* dev of directory */
        char            *ex_name;       /* name of directory */
        struct groups   *ex_groups;     /* groups allowed to mount this entry */
        struct exports  *ex_next;
        short           ex_rootmap;     /* id to map root requests to */
        short           ex_flags;       /* bits to mask off file mode */
};
struct groups {
        char            *g_name;
        struct groups   *g_next;
};
```

SEE ALSO

mount(8), showmount(8), mountd(8C), *NFS Protocol Spec*

NOTES

mount is an optional product included with the NFS package; for more information, contact your CONVEX sales representative.

NAME

publickey, getpublickey, getsecretkey – get public or secret key

SYNOPSIS

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

getpublickey(netname, publickey)
        char netname[MAXNETNAMELEN+1];
        char publickey[HEXKEYBYTES+1];

getsecretkey(netname, secretkey, passwd)
        char netname[MAXNETNAMELEN+1];
        char secretkey[HEXKEYBYTES+1];
        char *passwd;
```

DESCRIPTION

These routines are used to get public and secret keys from the YP database. **getsecretkey()** has an extra argument, *passwd*, which is used to decrypt the encrypted secret key stored in the database. Both routines return 1 if they are successful in finding the key, 0 otherwise. The keys are returned as NULL-terminated, hexadecimal strings. If the password supplied to **getsecretkey()** fails to decrypt the secret key, the routine will return 1 but the *secretkey* argument will be a NULL string ("").

SEE ALSO

publickey(5)

RPC Programmer's Manual in

NOTES

The routines described in this manual page are optional products included with the NFS package for customers in the U.S. and Canada only. For more information, contact your CONVEX sales representative.

NAME

realpath – returns the real file name.

SYNOPSIS

#include <sys/param.h>

char *realpath(file_name, resolved_name)
char *file_name;
char resolved_name[MAXPATHLEN];

DESCRIPTION

realpath() resolves all links and references to "." and ".." in *file_name* and stores it in *resolved_name.*

It can handle both relative and absolute path names. For absolute path names and the relative names whose resolved name cannot be expressed relatively (e.g. *../../foobar*), it returns the *resolved absolute* name. For the other relative path names, it returns the *resolved relative* name.

RETURN VALUE

If there is no error, it returns a pointer to the *resolved_name.* Otherwise it returns a NULL pointer and places the name of the offending file in *resolved_name.* The global variable *errno* is set to indicate the error.

SEE ALSO

getwd(3)

WARNINGS

It operates on null-terminated strings.

One should have the execute permission on all the directories in the given and the resolved path.

BUGS

realpath() may fail to return to the current directory if an error occurs.

NAME

rex – RPC-based remote execution protocol

SYNOPSIS

#include <sys/ioctl.h>
#include <rpcsvc/rex.h>

DESCRIPTION

This server will execute commands remotely. the working directory and environment of the command can be specified, and the standard input and output of the command can be arbitrarily redirected. An option is provided for interactive I/O for programs that expect to be running on terminals. Note that this service is only provided with the TCP transport.

RPC INFO

program number:

REXPROG

xdr routines:

```
int xdr_rex_start(xdrs, start);
        XDR *xdrs;
        struct rex_start *start;
int  xdr_rex_result(xdrs, result);
        XDR *xdrs;
        struct rex_result *result;
int xdr_rex_ttymode(xdrs, mode);
        XDR *xdrs;
        struct rex_ttymode *mode;
int xdr_rex_ttysize(xdrs, size);
        XDR *xdrs;
        struct ttysize *size;
```

procs:

REXPROC_START
: Takes rex_start structure, starts a command executing, and returns a rex_result structure.

REXPROC_WAIT
: Takes no arguments, waits for a command to finish executing, and returns a rex_result structure.

REXPROC_MODES
: Takes a rex_ttymode structure, and sends the tty modes.

REXPROC_WINCH
: Takes a ttysize structure, and sends window size information.

versions:

REXVERS_ORIG
: Original version

structures:

```
#define REX_INTERACTIVE         1       /* Interactive mode */
struct rex_start {
        char **rst_cmd;                 /* list of command and args */
        char *rst_host;                 /* working directory host name */
        char *rst_fsname;               /* working directory file system name */
        char *rst_dirwithin;            /* working directory within file system */
        char **rst_env;                 /* list of environment */
        u_short rst_port0;              /* port for stdin */
        u_short rst_port1;              /* port for stdin */
        u_short rst_port2;              /* port for stdin */
```

```
        u_long rst_flags;               /* options - see #defines above */
};

struct rex_result {
        int rlt_stat;                   /* integer status code */
        char *rlt_message;              /* string message for human consumption */
};

struct rex_ttymode {
        struct sgttyb basic;            /* standard unix tty flags */
        struct tchars more;             /* interrupt, kill characters, etc. */
        struct ltchars yetmore;         /* special Berkeley characters */
        u_long andmore;                 /* and Berkeley modes */
};
```

SEE ALSO

rex(1C), rexd(8C)

NAME

rnusers, rusers – return information about users on remote machines

SYNOPSIS

```
#include <utmp.h>
#include <rpcsvc/rusers.h>

rnusers(host)
        char *host

rusers(host, up)
        char *host
        struct utmpidlearr *up;
```

DESCRIPTION

rnusers returns the number of users logged on to *host* (–1 if it cannot determine that number). *rusers* fills the *utmpidlearr* structure with data about *host*, and returns 0 if successful.

RPC INFO

program number:

RUSERSPROG

xdr routines:

```
int xdr_utmp(xdrs, up)
        XDR *xdrs;
        struct utmp *up;
int xdr_utmpidle(xdrs, ui);
        XDR *xdrs;
        struct utmpidle *ui;
int xdr_utmpptr(xdrs, up);
        XDR *xdrs;
        struct utmp **up;
int xdr_utmpidleptr(xdrs, up);
        XDR *xdrs;
        struct utmpidle **up;
int xdr_utmparr(xdrs, up);
        XDR *xdrs;
        struct utmparr *up;
int xdr_utmpidlearr(xdrs, up);
        XDR *xdrs;
        struct utmpidlearr *up;
```

procs:

RUSERSPROC_NUM
: No arguments, returns number of users as an *unsigned long*.

RUSERSPROC_NAMES
: No arguments, returns *utmparr* or *utmpidlearr*, depending on the version number.

RUSERSPROC_ALLNAMES
: No arguments, returns *utmparr* or *utmpidlearr*, depending on the version number. Returns listing even for *utmp* entries satisfying *nonuser()* in *utmp.h*.

versions:

RUSERSVERS_ORIG
RUSERSVERS_IDLE

structures:

```
struct utmparr {                    /* RUSERSVERS_ORIG */
        struct utmp **uta_arr;
        int uta_cnt
};
.sp
struct utmpidle {
        struct utmp ui_utmp;
        unsigned ui_idle;
};
.sp
struct utmpidlearr {                /* RUSERSVERS_IDLE */
        struct utmpidle **uia_arr;
        int uia_cnt
};
```

SEE ALSO

rusers(1), rusersd(8c)

NOTES

rnusers is an optional product included with the NFS package; for more information, contact your CONVEX sales representative.

NAME

auth_destroy, authnone_create, authdes_create, authdes_getucred, 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_spcreateerror, clnt_sperrno, clnt_sperror, clnt_syslog, clntraw_create, clnttcp_create, clntudp_create, get_myaddress, getnetname, host2netname, key_decryptsession, key_encryptsession, key_gendes, key_setsecret, netname2host, netname2user, pmap_getmaps, pmap_getport, pmap_rmtcall, pmap_set, pmap_unset, registerrpc, rpc_createerr, svc_destroy, svc_fdset, svc_fds, svc_freeargs, svc_getargs, svc_getcaller, svc_getreqset, svc_getreq, svc_register, svc_run, svc_sendreply, svc_unregister, svcerr_auth, svcerr_decode, svcerr_noproc, svcerr_noprog, svcerr_progvers, svcerr_systemerr, svcerr_weakauth, svcraw_create, svctcp_create, svcfd_create, svcudp_create, user2netname, 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

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

#include <rpc/rpc.h>

void
auth_destroy(auth)
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()

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

AUTH *
authdes_create(name, window, syncaddr, ckey)
char *name;
unsigned window;
struct sockaddr *addr;
des_block *ckey;

authdes_create( ) is the first of two routines which interface to the RPC secure authentication system, known as DES authentication. The second is **authdes_getucred( )**, below. Note: the keyserver daemon **keyserv**(8) must be running for the DES authentication system to work.

authdes_create( ), used on the client side, returns an authentication handle that will enable the use of the secure authentication system. The first parameter *name* is the network name, or *netname*, of the owner of the server process. This field usually represents a *hostname* derived from the utility routine **host2netname**, but could also represent a user name using **user2netname**. The second field is window on the validity of the client credential, given in seconds. A small window is more secure than a large one, but choosing too small of a window will increase the frequency of resynchronizations because of clock drift. The third parameter *syncaddr* is optional. If it is NULL, then the authentication system will assume that the local clock is always in sync with the server's clock, and

will not attempt resynchronizations. If an address is supplied, however, then the system will use the address for consulting the remote time service whenever resynchronization is required. This parameter is usually the address of the RPC server itself. The final parameter *ckey* is also optional. If it is NULL, then the authentication system will generate a random DES key to be used for the encryption of credentials. If it is supplied, however, then it will be used instead.

authdes_getucred(adc, uid, gid, grouplen, groups)
struct authdes_cred *adc;
short *uid;
short *gid;
short *grouplen;
int *groups;

authdes_getucred( ), the second of the two DES authentication routines, is used on the server side for converting a DES credential, which is operating system independent, into a credential. This routine differs from utility routine **netname2user** in that **authdes_getucred( )** pulls its information from a cache, and does not have to do a Yellow Pages lookup everytime it is called to get its information.

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

Creates and returns an RPC authentication handle that contains authentication information. The parameter **host** 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 parameters.

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

Calls the remote procedure associated with **prognum**, **versnum**, and **procnum** on the machine, **host**. The parameter **in** 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 parameters, 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(prognum, versnum, procnum, inproc, in, outproc, out, eachresult)
u_long prognum, versnum, procnum;
char *in, *out;
xdrproc_t inproc, outproc;
resultproc_t eachresult;

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:

eachresult(out, addr)
char *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(clnt, procnum, inproc, in, outproc, out, tout)
CLIENT ***clnt; u_long procnum;**
xdrproc_t inproc, outproc;
char *in, *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 parameter **in** 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 parameters, and **outproc** is used to decode the procedure's results; **tout** is the time allowed for results to come back.

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

A macro used to change or retrieve various information about a client object. **req** 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	struct timeval	set total timeout
CLGET_TIMEOUT	struct timeval	get total timeout

Note: 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	get server's address

The following operations are valid for UDP only:

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

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

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

Generic client creation routine. **host** identifies the name of the remote host where the server is located. **proto** 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.

```
clnt_destroy(clnt)
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.

```
clnt_freeres(clnt, outproc, out)
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 parameter **out** is the address of the results, and **outproc** is the XDR routine describing the results in simple primitives. This routine returns one if the results were successfully freed, and zero otherwise.

```
void
clnt_geterr(clnt, errp)
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(s)
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. Used when a **clnt_create**, **clntraw_create**, **clnttcp_create**, or **clntudp_create** call fails.

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

Prints a message to standard error corresponding to the condition indicated by **stat.** Used after **callrpc.**

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

Prints 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. Used after **clnt_call**.

char *
clnt_spcreateerror(s);
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(stat)
enum clnt_stat stat;

Takes the same arguments as **clnt_perrno**, but instread 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.

clnt_sperrno is used instead of **clnt_perrno** if the program doesn't have a standard error (as a program running as a server quite likely doesn't), or if the programmer doesn't want the message to be output with **printf**, or if a message format different than that supported by **clnt_perrno** is to be used. Note: unlike **clnt_sperror** and **clnt_spcreaterror**, **clnt_sperrno** does not return pointer to static data so the result will not get overwritten on each call.

char *
clnt_sperror(rpch, s)
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.

clnt_syslog(rpch, s)
CLIENT ***rpch;**
char *s;

Like **clnt_perror**, except that it prints the error string to the standard *syslog*(8) file instead of printing to the standard error.

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, so the corresponding RPC server should live 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 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, 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 parameters **sendsz** and **recvsz**; values of zero choose 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 use 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 **portmap** service is consulted for this information). The parameter **sockp** 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**.

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.

host2netname(name, host, domain)
char *name;
char *host;
char *domain;

Convert from a domain-specific hostname to an operating-system independent netname. Return TRUE if it succeeds and FALSE if it fails. Inverse of **netname2host( )**.

key_decryptsession(remotename, deskey)
char *remotename;
des_block *deskey;

key_decryptsession( ) is an interface to the keyserver daemon, which is associated with RPC's secure authentication system (DES authentication). User programs rarely need to call it, or its associated routines **key_encryptsession( )**, **key_gendes( )** and **key_setsecret( )**. System commands such as **login** and the RPC library are the main clients of these four routines.

key_decryptsession( ) takes a server netname and a des key, and decrypts the key by using the the public key of the the server and the secret key associated with the effective uid of the calling process. It is the inverse of **key_encryptsession( )**.

key_encryptsession(remotename, deskey)
char *remotename;
des_block *deskey;

key_encryptsession( ) is a keyserver interface routine. It takes a server netname and a des key, and encrypts it using the public key of the the server and the secret key associated with the effective uid of the calling process. It is the inverse of **key_decryptsession( )**.

key_gendes(deskey)
des_block *deskey;

key_gendes( ) is a keyserver interface routine. It is used to ask the keyserver for a secure conversation key. Choosing one at random is usually not good enough, because the common ways of choosing random numbers, such as using the current time, are very easy to guess.

key_setsecret(key)
char *key;

key_setsecret( ) is a keyserver interface routine. It is used to set the key for the effective *uid* of the calling process.

void
get_myaddress(addr)
struct sockaddr_in *addr;

Stuffs 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).

getnetname(name)
char name[MAXNETNAMELEN];

getnetname( ) installs the unique, operating-system independent netname of the caller in the fixed-length array *name*. Returns TRUE if it succeeds and FALSE if it fails.

netname2host(name, host, hostlen)
char *name;
char *host;
int hostlen;

Convert from an operating-system independent netname to a domain-specific hostname. Returns TRUE if it succeeds and FALSE if it fails. Inverse of **host2netname( )**.

netname2user(name, uidp, gidp, gidlenp, gidlist)
char *name;
int *uidp;
int *gidp;
int *gidlenp;
int *gidlist;

Convert from an operating-system independent netname to a domain-specific user ID. Returns TRUE if it succeeds and FALSE if it fails. Inverse of **user2netname( )**.

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

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 command **rpcinfo -p** uses this routine.

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

A user interface to the **portmap** 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 **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;

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 parameter ***portp** 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;

A user interface to the **portmap** service, which establishes a mapping between the triple [**prognum,versnum,protocol**] and **port** on the machine's **portmap** 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**.

pmap_unset(prognum, versnum)
u_long prognum, versnum;

A user interface to the **portmap** service, which destroys all mapping between the triple [**prognum, versnum, ***] and **ports** on the machine'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;

Registers 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); **progname** should return a pointer to its static result(s); **inproc** is used to decode the parameters 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.

svc_destroy(xprt)
SVCXPRT * xprt;

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;

A global variable reflecting the RPC service side's read file descriptor bit mask; it is suitable as a parameter to the **select** 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**!), yet it may change after calls to **svc_getreqset** or any creation routines.

int svc_fds;

Similar to **svc_fedset**, but limited to 32 descriptors. This interface is obsoleted by **svc_fdset**.

svc_freeargs(xprt, inproc, in)
SVCXPRT *xprt;
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 one if the results were successfully freed, and zero otherwise.

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

A macro that decodes the arguments of an RPC request associated with the RPC service transport handle, **xprt** . The parameter **in** 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(xprt)
SVCXPRT ***xprt;**

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 interest 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 some 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;

Similar to **svc_getreqset**, but limited to 32 descriptors. This interface is obsoleted by **svc_getreqset**.

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

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, then a mapping of the triple [**prognum, versnum, protocol**] to **xprt->xp_port** is established with the local **portmap** service (generally **protocol** is zero, IPPROTO_UDP or IPPROTO_TCP). The procedure **dispatch** has the following form:

dispatch(request, xprt)
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** system call to return.

svc_sendreply(xprt, outproc, out)
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 parameter **xprt** 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(prognum, versnum)
u_long prognum, versnum;

Removes all mapping of the double [**prognum, versnum**] to dispatch routines, and of the triple [**prognum, versnum, ***] to port number.

void
svcerr_auth(xprt, why)
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(xprt)
SVCXPRT *xprt;

Called by a service dispatch routine that can't successfully decode its parameters. See also **svc_getargs**.

void
svcerr_noproc(xprt)
SVCXPRT *xprt;

Called by a service dispatch routine that doesn't implement the procedure number that the caller requests.

void
svcerr_noprog(xprt)
SVCXPRT *xprt;

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

void
svcerr_progvers(xprt)
SVCXPRT *xprt;

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

void
svcerr_systemerr(xprt)
SVCXPRT *xprt;

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;

Called by a service dispatch routine that refuses to perform a remote procedure call due to insufficient (but correct) authentication parameters. The routine calls **svcerr_auth(xprt, AUTH_TOOWEAK)** .

SVCXPRT *
svcraw_create()

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(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, then this routine binds it to an arbitrary port. Upon completion, **xprt->xp_sock** is the transport's socket number, and **xprt->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 *
svcfd_create(fd, sensize, recvsize)
int fd;
u_int sendsize;
u_int recvsize;

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

SVCXPRT *
svcudp_create(sock)
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, then this routine binds it to an arbitrary port. Upon completion, **xprt->xp_sock** is the transport's socket number, and **xprt->xp_port** is the transport's port number. This routine returns NULL if it fails.

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.

user2netname(name, uid, domain)
char *name;
int uid;
char *domain;

Convert from a domain-specific username to an operating-system independent netname. Returns TRUE if it succeeds and FALSE if it fails. Inverse of **netname2user( )**.

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

Used for describing RPC messages, externally. This routine 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;

Used for describing UNIX credentials, externally. This routine 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;

Used for describing RPC messages, externally. This routine 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;

Used for describing RPC messages, externally. This routine 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;

Used for describing RPC messages, externally. This routine 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;

Used for describing parameters to various **portmap** procedures, externally. This routine is useful for users who wish to generate these parameters without using the **pmap** interface.

xdr_pmaplist(xdrs, rp)
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 parameters without using the **pmap** interface.

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

Used for describing RPC messages, externally. This routine 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;

Used for describing RPC messages, externally. This routine 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 register themselves with the RPC service package. This routine modifies the global variable **svc_fds**. Service implementors usually don't need this routine.

void
xprt_unregister(xprt)
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 don't need this routine.

SEE ALSO

Remote Procedure Call Programming Guide, and *RPC Protocol Specification*, xdr(3N), rpcgen(1), keyserv(8)

NOTES

The routines described in this manual page are optional products included with the NFS package; for more information, contact your CONVEX sales representative.

NAME

rquota – implement quotas on remote machines

DESCRIPTION

The **rquota( )** protocol inquires about quotas on remote machines. It is used in conjunction with NFS, since NFS itself does not implement quotas.

SYNPOSIS

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

RPC INFO

program number:

RQUOTAPROG

xdr routines:

```
xdr_getquota_args(xdrs, gqa);
        XDR *xdrs;
        struct getquota_args *gqa;
xdr_getquota_rslt(xdrs, gqr);
        XDR *xdrs;
        struct getquota_rslt *gqr;
xdr_rquota(xdrs, rq);
        XDR *xdrs;
        struct rquota *rq;
```

procs:

RQUOTAPROC_GETQUOTA
RQUOTAPROC_GETACTIVEQUOTA

Arguments of *struct getquota_args.*
Returns *struct getquota_rslt.*
Uses UNIX authentication.
Returns quota only on filesystems with quota active.

versions:

RQUOTAVERS_ORIG

structures:

```
struct getquota_args {
        char *gqa_pathp;        /* path to filesystem of interest */
        int gqa_uid;            /* inquire about quota for uid */
};
/*
 * remote quota structure
 */
struct rquota {
        int rq_bsize;           /* block size for block counts */
        bool_t rq_active;       /* indicates whether quota is active */
        u_long rq_bhardlimit;   /* absolute limit on disk blks alloc */
        u_long rq_bsoftlimit;   /* preferred limit on disk blks */
        u_long rq_curblocks;    /* current block count */
        u_long rq_fhardlimit;   /* absolute limit on allocated files */
        u_long rq_fsoftlimit;   /* preferred file limit */
        u_long rq_curfiles;     /* current # allocated files */
        u_long rq_btimeleft;    /* time left for excessive disk use */
        u_long rq_ftimeleft;    /* time left for excessive files */
```

```
};
enum gqr_status {
        Q_OK = 1,               /* quota returned */
        Q_NOQUOTA = 2,          /* noquota for uid */
        Q_EPERM = 3             /* no permission to access quota */
};
struct getquota_rslt {
        enum gqr_status gqr_status;     /* discriminant */
        struct rquota gqr_rquota;       /* valid if status == Q_OK */
};
```

SEE ALSO

quota(1), quotactl(2)

NOTES

rquota is an optional product included with the NFS package; for more information, contact your CONVEX sales representative.

NAME

havedisk, rstat – get performance data from remote kernel

SYNOPSIS

```
#include <rpcsvc/rstat.h>

havedisk(host)
        char *host;

rstat(host, statp)
        char *host;
        struct statstime *statp;
```

DESCRIPTION

The **rstat( )** protocol is used to gather statistics from remote kernel. Statistics are available on items such as paging, swapping and cpu utilization.

PROGRAMMING

havedisk returns 1 if *host* has a disk, 0 if it does not, and –1 if this cannot be determined. *rstat* fills in the *statstime* structure for *host*, and returns 0 if it was successful.

RPC INFO

program number:

RSTATPROG

xdr routines:

```
int xdr_stats(xdrs, stat)
        XDR *xdrs;
        struct stats *stat;
int xdr_statsswtch(xdrs, stat)
        XDR *xdrs;
        struct statsswtch *stat;
int xdr_statstime(xdrs, stat)
        XDR *xdrs;
        struct statstime *stat;
int xdr_timeval(xdrs, tv)
        XDR *xdrs;
        struct timeval *tv;
```

procs:

RSTATPROC_HAVEDISK
: Takes no arguments, returns *long* which is true if remote host has a disk.

RSTATPROC_STATS
: Takes no arguments, return *struct statsxxx*, depending on version.

versions:

RSTATVERS_ORIG
RSTATVERS_SWTCH
RSTATVERS_TIME

structures:

```
struct stats {                          /* RSTATVERS_ORIG */
        int cp_time[CPUSTATES];
        int dk_xfer[DK_NDRIVE];
        unsigned v_pgpgin;      /* these are cumulative sum */
        unsigned v_pgpgout;
        unsigned v_pswpin;
        unsigned v_pswpout;
```

```
        unsigned v_intr;
        int if_ipackets;
        int if_ierrors;
        int if_opackets;
        int if_oerrors;
        int if_collisions;
};
struct statsswtch {                         /* RSTATVERS_SWTCH */
        int cp_time[CPUSTATES];
        int dk_xfer[DK_NDRIVE];
        unsigned v_pgpgin;      /* these are cumulative sum */
        unsigned v_pgpgout;
        unsigned v_pswpin;
        unsigned v_pswpout;
        unsigned v_intr;
        int if_ipackets;
        int if_ierrors;
        int if_opackets;
        int if_oerrors;
        int if_collisions;
        unsigned v_swtch;
        long avenrun[3];
        struct timeval boottime
};
struct statstime {                          /* RSTATVERS_TIME */
        int cp_time[CPUSTATES];
        int dk_xfer[DK_NDRIVE];
        unsigned v_pgpgin;      /* these are cumulative sum */
        unsigned v_pgpgout;
        unsigned v_pswpin;
        unsigned v_pswpout;
        unsigned v_intr;
        int if_ipackets;
        int if_ierrors;
        int if_opackets;
        int if_oerrors;
        int if_collisions;
        unsigned v_swtch;
        long avenrun[3];
        struct timeval boottime;
        struct timeval curtime;
};
```

SEE ALSO

rup(1), rstatd(8C)

NOTES

The routines described in this manual page are optional products included with the NFS package; for more information, contact your CONVEX sales representative.

NAME

rtime – get remote time

SYNOPSIS

```
#include <sys/types.h>
#include <sys/time.h>
#include <netinet/in.h>

int rtime(addrp, timep, timeout)
struct sockaddr_in *addrp;
struct timeval *timep;
struct timeval *timeout;
```

DESCRIPTION

rtime() consults the Internet Time Server at the address pointed to by *addrp* and returns the remote time in the **timeval** struct pointed to by *timep*. Normally, the UDP protocol is used when consulting the Time Server. The *timeout* parameter specifies how long the routine should wait for a reply before giving up. If *timeout* is specified as NULL, however, the routine will instead use TCP and block until a reply is received from the time server.

The routine returns 0 if it is successful. Otherwise, it returns –1 and **errno** is set to reflect the cause of the error.

SEE ALSO

timed(8c)

NOTES

rtime is an optional product included with the NFS package for customers in the U.S. and Canada only. For more information, contact your CONVEX sales representative.

NAME

rwall – write to specified remote machines

SYNOPSIS

#include <rpcsvc/rwall.h>

rwall(host, msg);
char *host, *msg;

DESCRIPTION

rwall causes *host* to print the string *msg* to all its users. It returns 0 if successful.

RPC INFO

program number:
WALLPROG

procs:
WALLPROC_WALL
Takes string as argument (wrapstring), returns no arguments.
Executes *wall* on remote host with string.

versions:
RSTATVERS_ORIG

SEE ALSO

rwall(1), shutdown(8), rwalld(8C)

NOTES

rwall is an optional product included with the NFS package; for more information, contact your CONVEX sales representative.

NAME

spray – scatter data in order to check the network

DESCRIPTION

The spray protocol sends packets to a given machine to test the speed and reliability of it.

SYNOPSIS

```
#include <sys/time.h>
#include <rpcsvc/spray.h>
```

RPC INFO

program number:

SPRAYPROG

xdr routines:

```
xdr_sprayarr(xdrs, arr);
        XDR *xdrs;
        struct sprayarr *arr;
xdr_spraycumul(xdrs, cumul);
        XDR *xdrs;
        sturct spraycumul *cumul;
```

procs:

SPRAYPROC_SPRAY

Takes a *struct sprayarr* as argument, returns no value.
Increments a counter in server daemon.
The server does not return this call, so the caller should have a timeout of 0.

SPRAYPROC_GET

Takes no arguments, returns *struct spraycumul* with value of counter and clock.

SPRAYPROC_CLEAR

Takes no arguments and returns no value.
Zeros out counter and clock.

versions:

SPRAYVERS_ORIG

structures:

```
struct spraycumul {
        unsigned counter;
        struct timeval clock;
};
struct sprayarr {
        int *data,
        int lnth
};
```

SEE ALSO

spray(8), sprayd(8C)

NOTES

spray is an optional product included with the NFS package; for more information, contact your CONVEX sales representative.

NAME

xcrypt, xdecrypt, passwd2des – hex encryption and utility routines

SYNOPSIS

```
xencrypt(data, key)
        char *data;
        char *key;

xdecrypt(data, key)
        char *data;
        char *key;

passwd2des(pass, key)
        char *pass;
        char *key;
```

DESCRIPTION

The routines **xencrypt** and **xdecrypt** take NULL-terminated hexadecimal strings as arguments, and encrypt them using the 8-byte *key* as input to the DES algorithm. The input strings must have a length that is a multiple of 16 hex digits (64 bits is the DES block size).

passwd2des converts a password, of arbitrary length, into an 8-byte DES key, with odd-parity set in the low bit of each byte. The high-order bit of each input byte is ignored.

These routines are used by the DES authentication subsystem for encrypting and decrypting the secret keys stored in the publickey database.

SEE ALSO

des_crypt(3), **publickey**(5)

NOTES

The routines described in this manual page are optional products included with the NFS package for customers in the U.S. and Canada only. For more information, contact your CONVEX sales representative.

NAME

xdr_array, xdr_bool, xdr_bytes, xdr_char, xdr_destroy, xdr_double, xdr_enum, xdr_float, xdr_free, xdr_getpos, xdr_hyper, xdr_inline, xdr_int, xdr_long, xdrmem_create, xdr_opaque, xdr_pointer, xdrrec_create, xdrrec_endofrecord, xdrrec_eof, xdrrec_skiprecord, xdr_reference, xdr_setpos, xdr_short, xdrstdio_create, xdr_string, xdr_u_char, xdr_u_hyper, xdr_u_int, xdr_u_long, xdr_u_short, xdr_union, xdr_vector, xdr_void, xdr_wrapstring – library routines for external data representation

SYNOPSIS AND 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(xdrs, arrp, sizep, maxsize, elsize, elproc)
XDR *xdrs;
char **arrp;
u_int *sizep, maxsize, elsize;
xdrproc_t elproc;

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

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

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

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

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

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

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 worthwhile to consider **xdr_bytes**, **xdr_opaque**or **xdr_string**.

void
xdr_destroy(xdrs)
XDR *xdrs;

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

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

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

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

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

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

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

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

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;

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

xdr_hyper(xdrs, hp)
XDR *xdrs;
hyper *hp;

A filter primitive that translates between C *hyper* (really CONVEX "long long") integers and their external representations. This routine returns one if it succeeds, zero otherwise.

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

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

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;

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;

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

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

Like **xdr_reference** in that it XDR **xdr_pointer** serializes NULL pointers, whereas **xdr_reference** does not. Thus **xdr_pointer** can XDR recursive data structures, such as binary trees or linked lists, correctly, whereas **xdr_reference** will fail.

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

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

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

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

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

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

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

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

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

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

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

Warning: this routine does not understand NULL pointers. Use **xdr_pointer** instead.

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

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

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

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

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

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

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

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

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

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

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

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

xdr_u_hyper(xdrs, hp)
XDR *xdrs;
u_hyper *hp;

A filter primitive that translates between C **unsigned** *hyper* (really CONVEX "unsigned long long") integers and their external representations. This routine returns one if it succeeds, zero otherwise.

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

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

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

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

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

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

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

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 parameter **choices** 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's not NULL). 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;

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

xdr_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;

A primitive that calls **xdr_string(xdrs, sp,** MAXUN.UNSIGNED); where MAXUN.UNSIGNED is the maximum value of an unsigned integer. **xdr_wrapstring** is handy 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.

SEE ALSO

External Data Representation Protocol Specification, rpc(3N), rpcgen(1)

NOTES

The *hyper* type is really equivalent to the 64-bit CONVEX "long long" integer type and as such is an extension to the XDR Protocol Specification. *hyper* integers may not be supported by other vendors.

xdr is an optional product included with the NFS package; for more information, contact your CONVEX sales representative.

NAME

yp_get_default_domain, yp_bind, yp_unbind, yp_match, yp_first, yp_next, yp_all, yp_order, yp_master, yperr_string, ypprot_err – yellow pages client interface

SYNOPSIS

```
#include <rpcsvc/ypclnt.h>

yp_bind(indomain);
char *indomain;

void yp_unbind(indomain)
char *indomain;

yp_get_default_domain(outdomain);
char **outdomain;

yp_match(indomain, inmap, inkey, inkeylen, outval, outvallen)
char *indomain;
char *inmap;
char *inkey;
int inkeylen;
char **outval;
int *outvallen;

yp_first(indomain, inmap, outkey, outkeylen, outval, outvallen)
char *indomain;
char *inmap;
char **outkey;
int *outkeylen;
char **outval;
int *outvallen;

yp_next(indomain, inmap, inkey, inkeylen, outkey, outkeylen, outval, outvallen);
char *indomain;
char *inmap;
char *inkey;
int inkeylen;
char **outkey;
int *outkeylen;
char **outval;
int *outvallen;

yp_all(indomain, inmap, incallback);
char *indomain;
char *inmap;
struct ypall_callback *incallback;

yp_order(indomain, inmap, outorder);
char *indomain;
char *inmap;
int *outorder;

yp_master(indomain, inmap, outname);
char *indomain;
```

char *inmap;
char **outname;

char *yperr_string(incode)
int incode;

ypprot_err(incode)
unsigned int incode;

DESCRIPTION

This package of functions provides an interface to the yellow pages (YP) network lookup service. The package can be loaded from the standard library, */lib/libc.a*. Refer to *ypfiles*(5) and *ypserv*(8) for an overview of the yellow pages, including the definitions of *map* and *domain*, and a description of the various servers, databases, and commands that comprise the YP.

All input parameters names begin with **in**. Output parameters begin with **out**. Output parameters of type *char ** should be addresses of uninitialized character pointers. Memory is allocated by the YP client package using *malloc*(3), and may be freed if the user code has no continuing need for it. For each *outkey* and *outval*, two extra bytes of memory are allocated at the end that contain NEWLINE and NULL, respectively, but these two bytes are not reflected in *outkeylen* or *outvallen*. *indomain* and *inmap* strings must be non-null and null-terminated. String parameters which are accompanied by a count parameter may not be null, but may point to null strings, with the count parameter indicating this. Counted strings need not be null-terminated.

All functions in this package of type **int** return 0 if they succeed, and a failure code (YPERR_*xxxx*) otherwise. Failure codes are described under **DIAGNOSTICS** below.

The YP lookup calls require a map name and a domain name, at minimum. It is assumed that the client process knows the name of the map of interest. Client processes should fetch the node's default domain by calling **yp_get_default_domain()**, and use the returned *outdomain* as the *indomain* parameter to successive YP calls.

To use the YP services, the client process must be "bound" to a YP server that serves the appropriate domain using *yp_bind*. Binding need not be done explicitly by user code; this is done automatically whenever a YP lookup function is called. *yp_bind* can be called directly for processes that make use of a backup strategy (e.g., a local file) in cases when YP services are not available.

Each binding allocates (uses up) one client process socket descriptor; each bound domain costs one socket descriptor. However, multiple requests to the same domain use that same descriptor. *yp_unbind()* is available at the client interface for processes that explicitly manage their socket descriptors while accessing multiple domains. The call to *yp_unbind()* make the domain *unbound*, and free all per-process and per-node resources used to bind it.

If an RPC failure results upon use of a binding, that domain will be unbound automatically. At that point, the *ypclnt* layer will retry forever or until the operation succeeds, provided that *ypbind* is running, and either

a) the client process can't bind a server for the proper domain, or

b) RPC requests to the server fail.

If an error is not RPC-related, or if *ypbind* is not running, or if a bound *ypserv* process returns any answer (success or failure), the *ypclnt* layer will return control to the user code, either with an error code, or a success code and any results.

yp_match returns the value associated with a passed key. This key must be exact; no pattern matching is available.

yp_first returns the first key-value pair from the named map in the named domain.

yp_next() returns the next key-value pair in a named map. The *inkey* parameter should be the *outkey* returned from an initial call to *yp_first()* (to get the second key-value pair) or the one returned from the nth call to *yp_next()* (to get the nth + second key-value pair).

The concept of first (and, for that matter, of next) is particular to the structure of the YP map being processing; there is no relation in retrieval order to either the lexical order within any original (non-YP) data base, or to any obvious numerical sorting order on the keys, values, or key-value pairs. The only ordering guarantee made is that if the *yp_first()* function is called on a particular map, and then the *yp_next()* function is repeatedly called on the same map at the same server until the call fails with a reason of YPERR_NOMORE, every entry in the data base will be seen exactly once. Further, if the same sequence of operations is performed on the same map at the same server, the entries will be seen in the same order.

Under conditions of heavy server load or server failure, it is possible for the domain to become unbound, then bound once again (perhaps to a different server) while a client is running. This can cause a break in one of the enumeration rules; specific entries may be seen twice by the client, or not at all. This approach protects the client from error messages that would otherwise be returned in the midst of the enumeration. The next paragraph describes a better solution to enumerating all entries in a map.

yp_all provides a way to transfer an entire map from server to client in a single request using TCP (rather than UDP as with other functions in this package). The entire transaction take place as a single RPC request and response. You can use *yp_all* just like any other YP procedure, identify the map in the normal manner, and supply the name of a function which will be called to process each key-value pair within the map. You return from the call to *yp_all* only when the transaction is completed (successfully or unsuccessfully), or your *"foreach"* function decides that it doesn't want to see any more key-value pairs.

The third parameter to *yp_all* is

```
struct ypall_callback *incallback {
        int (*foreach)();
        char *data;
};
```

The function *foreach* is called

```
foreach(instatus, inkey, inkeylen, inval, invallen, indata);
int instatus;
char *inkey;
int inkeylen;
char *inval;
int invalllen;
char *indata;
```

The *instatus* parameter will hold one of the return status values defined in <rpcsvc/yp_prot.h> — *either YP_TRUE* or an error code. (See *ypprot_err*, below, for a function that converts a YP protocol error code to a *ypclnt* layer error code.)

The key and value parameters are somewhat different than defined in the synopsis section above. First, the memory pointed to by the *inkey* and *inval* parameters is private to the *yp_all* function, and is overwritten with the arrival of each new key-value pair. It is the responsibility of the *foreach* function to do something useful with the contents of that memory, but it does not own the memory itself. Key and value objects presented to the *foreach* function look exactly as they do in the server's map — if they were not newline-terminated or null-terminated in the map, they won't be here either.

The *indata* parameter is the contents of the *incallback->data* element passed to *yp_all*. The *data* element of the callback structure may be used to share state information between the *foreach* function and the mainline code. Its use is optional, and no part of the YP client package inspects its contents — cast it to something useful, or ignore it as you see fit.

The *foreach* function is a Boolean. It should return zero to indicate that it wants to be called again for further received key-value pairs, or non-zero to stop the flow of key-value pairs. If *foreach* returns a non-zero value, it is not called again; the functional value of *yp_all* is then 0.

yp_order returns the order number for a map.

yp_master returns the machine name of the master YP server for a map.

yperr_string returns a pointer to an error message string that is null-terminated but contains no period or newline.

ypprot_err takes a YP protocol error code as input, and returns a *ypclnt* layer error code, which may be used in turn as an input to *yperr_string*.

FILES

/usr/include/rpcsvc/ypclnt.h
/usr/include/rpcsvc/yp_prot.h

SEE ALSO

ypfiles(5), ypserv(8),

DIAGNOSTICS

All integer functions return 0 if the requested operation is successful, or one of the following errors if the operation fails.

```
#define YPERR_BADARGS   1  /* args to function are bad */
#define YPERR_RPC       2  /* RPC failure - domain has been unbound */
#define YPERR_DOMAIN    3  /* can't bind to server on this domain */
#define YPERR_MAP       4  /* no such map in server's domain */
#define YPERR_KEY       5  /* no such key in map */
#define YPERR_YPERR     6  /* internal yp server or client error */
#define YPERR_RESRC     7  /* resource allocation failure */
#define YPERR_NOMORE    8  /* no more records in map database */
#define YPERR_PMAP      9  /* can't communicate with portmapper */
#define YPERR_YPBIND    10 /* can't communicate with ypbind */
#define YPERR_YPSERV    11 /* can't communicate with ypserv */
#define YPERR_NODOM     12 /* local domain name not set */
#define YPERR_BADDB     13 /* yp data base is bad */
#define YPERR_VERS      14 /* YP version mismatch */
```

NOTES

ypclnt is an optional product included with the NFS package; for more information, contact your CONVEX sales representative.

NAME

yppasswd – update user password in yellow pages

DESCRIPTION

The yppasswd protocol is used to change a user's password entry in the YP password database.

SYNPOSIS

```
#include <pwd.h>
#include <rpcsvc/yppasswd.h>

yppasswd(oldpass, newpw)
        char *oldpass
        struct passwd *newpw;
```

DESCRIPTION

If *oldpass* is indeed the old user password, this routine replaces the password entry with *newpw*. It returns 0 if successful.

RPC INFO

program number:

YPPASSWDPROG

xdr routines:

```
xdr_ppasswd(xdrs, yp)
        XDR *xdrs;
        struct yppasswd *yp;
xdr_yppasswd(xdrs, pw)
        XDR *xdrs;
        struct passwd *pw;
```

procs:

YPPASSWDPROC_UPDATE

Takes *struct yppasswd* as argument, returns integer.
Same behavior as *yppasswd()* wrapper.
Uses UNIX authentication.

versions:

YPPASSWDVERS_ORIG

structures:

```
struct yppasswd {
        char *oldpass;   /* old (unencrypted) password */
        struct passwd newpw;    /* new pw structure */
};
```

SEE ALSO

yppasswd(1), yppasswdd(8C)

NOTES

yppasswd is an optional product included with the NFS package; for more information, contact your CONVEX sales representative.

NAME

nfs, NFS – Network File System

SYNOPSIS

pseudo-device nfs

options NFS

DESCRIPTION

NFS is a heterogeneous network file system implementation originally developed at Sun Microsystems. It allows a client machine to perform transparent file access over the network. Using it, a client machine can operate on files that reside on a variety of servers, server architectures and across a variety of operating systems. Client file access calls are converted to NFS protocol requests, and are sent to the server system over the network. The server receives the request, performs the actual file system operation, and sends a response back to the client. The Network File System operates in a stateless fashion using remote procedure (RPC) calls built on top of external data representation (XDR) protocol.

Several other vendors support or may be planning support for NFS, including Alliant, Arete, Celerity, DEC, Data General, Gould, Pyramid, and Unisoft. The CONVEX implementation provides both client and server operation.

A client is a machine that mounts a file system physically located on a remote machine into its local directory tree. A client gains access to that filesystem with the **mount**(2) system call, or the **mount**(8) command, which requests a file handle for the filesystem itself. For example:

mount convex:/usr/bin /usr/bin

A server is a machine that exports file systems for other clients to mount. Servers control which file systems are accessible to other machines in the file */etc/exports*, which has one entry for each file system followed by a list of machines or netgroups that are permissible clients. The CONVEX NFS implementation also allows for exporting filesystems to be read-only to all clients, or to change the default root-over-the-net UID mapping (see below) on a per-filesystem basis; see *exports*(5) for more information.

A server may also be a client with respect to filesystems it has mounted over the network, but its clients cannot gain access to those filesystems. Instead, the client must mount a filesystem directly from the server on which it resides.

Once the filesystem is mounted by the client, the server issues a file handle to the client for each file (or directory) the client accesses. If the file is somehow removed on the server side, the file handle becomes stale (dissociated with a known file).

File systems can be mounted *hard* or *soft*. *Hard* mounted file systems retry forever in the event the server is down or unreachable; *soft* mounting causes an error to be returned after a suitable timeout and number of retransmissions. There are two types of *hard* mounts available: *interruptible* and *non-interruptible*, with the default being *interruptible*. This allows for the user to kill a process that is hung waiting for the server to respond if he so desires, rather than being forced to wait for the transaction to complete when the server reboots on *hard* type filesystems.

NFS communicates using RPC/XDR (Sun Remote Procedure Call and Sun External Data Representation) over UDP as a transport layer. Although the use of other transport layers is possible, they aren't currently implemented. All the CONVEX network drivers that support UDP, including Ethernet, Hyperchannel, and DR-11. may be used for NFS operation.

NFS assumes that the user ID space is homogeneous across the network. This means that user ID 21 is the same user on all machines.

NFS uses three types of daemons. *nfsd* is a server daemon that provides user context for a client request. *biod* is a client daemon that allows asynchronous I/O to a client. Several of each kind are normally present on an NFS system. A single *mountd* is found on each server. It is

responsible for the server's part of the mount protocol.

NFS attempts to preserve UNIX file system semantics transparently, but there are some incompatibilities. Here is a short list. For security reasons, *root* (user ID 0) is propagated across the network as *nobody* (user ID -2). (This may be changed as a sysgen option to affect the entire system or changed on a per-filesystem basis in the */etc/exports* file.) File locking using *flock*(2) is not supported across NFS; that is to say a client machine can use *flock(2)* to successfully obtain an exclusive file lock which will prevent other cooperating process on that same client from acquiring an exclusive lock, but will not affect the ability for the server or other client machine to successfully obtain a file lock of that file. See *lockd*(8C) for network-wide record and file locking supported with NFS. Unlinking a file still open by other processes on the client will cause the client to rename the file to a name of the form
last client process closes the file.

NFS operation is normally stateless. This implies that clients should not be able to tell the difference between very slow servers or servers that go down and come back up. The penalty paid is that some operations (particularly write) are slower, because all the information must be committed to permanent storage before an NFS service request can be acknowledged by the server. (This may be changed as a sysgen option on the server.)

DIAGNOSTICS

NFS server xxx not responding still trying
The server named *xxx* is not responding or is down. Seen only for *hard* mounted NFS filesystems. The client continues (forever) to resend the request until it receives an acknowledgement from the server. This means the server can crash or power down, and come back up, without any special action required by the client.

NFS www failed for server xxx: RPC: Timed out
The client was attempting to do a *www* type of NFS operation, and the server *xxx* did not respond after a certain number of retry requests were sent. The system call failed with errno set to ETIMEDOUT. Seen only for *soft* mounted NFS filesystems.

NFS write error ddd on host xxx fh yyyy
The client was attempting to do an NFS write operation, and the server *xxx* rejected the request and returned the decimal reason code *ddd*. The reason code is a standard error number from errno.h (see /usr/include/errno.h).

SEE ALSO

CONVEX NFS User's Guide
CONVEX NFS Reference Set
CONVEX NFS System Manager's Guide
mount(2), getfh(2), exportfs(2),
exports(5), mountd(8c), nfsd(8), showmount(8), mount(8)

BUGS

NFS over ethernet uses considerable CPU cycles on both the client and server to break the I/O data into packets and generate the UDP protocol. Therefore, operation will never be as efficient as local disks. Applications that demand high I/O performance should use local disk.

NFS is an optional product; for procurement information, contact your CONVEX sales representative.

NAME

bootparams – boot parameter data base

SYNOPSIS

/etc/bootparams

DESCRIPTION

The *bootparams* file contains the list of client entries that diskless clients use for booting. For each diskless client the entry should contain the following information:

name of client
a list of keys, names of servers, and pathnames.

The first item of each entry is the name of the diskless client. The subsequent item is a list of keys, names of servers, and pathnames.

Items are separated by TAB characters.

EXAMPLE

Here is an example of the */etc/bootparams* taken from a SunOS system.

```
myclient        root=myserver:/nfsroot/myclient \
        swap=myserver:/nfsswap/myclient \
        dump=myserver:/nfsdump/myclient
```

FILES

/etc/bootparams

SEE ALSO

bootparamd(8)

NAME

ethers – Ethernet address to hostname database or YP domain

DESCRIPTION

The *ethers* file contains information regarding the known (48 bit) Ethernet addresses of hosts on the Internet. For each host on an Ethernet, a single line should be present with the following information:

Ethernet address
official host name

Items are separated by any number of blanks and/or TAB characters. A '#' indicates the beginning of a comment extending to the end of line.

The standard form for Ethernet addresses is "*x:x:x:x:x:x*" where *x* is a hexadecimal number between 0 and ff, representing one byte. The address bytes are always in network order. Host names may contain any printable character other than a SPACE, TAB, NEWLINE, or comment character. It is intended that host names in the *ethers* file correspond to the host names in the *hosts*(5) file.

The *ether_line*( ) routine from the Ethernet address manipulation library, *ethers*(3N) may be used to scan lines of the *ethers* file.

FILES

/etc/ethers

SEE ALSO

ethers(3N), hosts(5)

NOTES

ethers is an optional product included with the NFS package; for more information, contact your CONVEX sales representative.

NAME

exports, xtab – directories to export to NFS clients

SYNOPSIS

/etc/exports

/etc/xtab

DESCRIPTION

The **/etc/exports** file contains entries for directories that can be exported to NFS clients. This file is read automatically by the *exportfs*(8) command. If you change this file, you must run *exportfs*(8) for the changes to affect the daemon's operation.

Only when this file is present at boot time does the */etc/rc.local* script execute *exportfs*(8) and start the NFS file-system daemon, *nfsd*(8).

The **/etc/xtab** file contains entries for directories that are *currently* exported. This file should only be accessed by programs using *getexportent* (see *exportent*(3)). (Use the **–u** option of *exportfs* to remove entries from this file).

An entry for a directory consists of a line of the following form:

directory *–option*[, *option*] ...

directory is the pathname of a directory (or file).

option is one of

ro Export the directory read-only. If not specified, the directory is exported read-write.

async Export the directory asynchronously. If not specified, the directory is exported synchronously. Setting this option provides for significantly better write performance to NFS files residing in the directory. However, this performance gain is offset by the fact that the data written is not guaranteed to be on secondary storage before the server acknowledges the write completion. Thus, this option must be used with caution.

rw=*hostnames*[:*hostname*] ...
Export the directory read-mostly. Read-mostly means read-only to most machines, but read-write to those specified. If not specified, the directory is exported read-write to all.

anon=*uid*
If a request comes from an unknown user, use *uid* as the effective user ID. Note: root users (uid 0) are always considered unknown by the NFS server, unless they are included in the root option below. The default value for this option is –2. Setting anon to –1 disables anonymous access. Note: by default secure NFS will accept insecure requests as anonymous, and those wishing for extra security can disable this feature by setting anon to –1.

root=*hostnames*[:*hostname*] ...
Give root access only to the root users from a specified *hostname*. The default is for no hosts to be granted root access.

access=*client*[:*client*] ...
Give mount access to each *client* listed. A *client* can either be a hostname, or a netgroup (see **netgroup**(5)). Each *client* in the list is checked for first in the hosts database, and then the netgroup database. The default value allows any machine to mount the

given directory.

secure Require clients to use a more secure protocol when accessing the directory.

A '**#**' (pound-sign) anywhere in the file indicates a comment that extends to the end of the line.

EXAMPLE

```
/usr          -access=clients                 # export to my clients
/usr/local    # export to the world
/usr2         -access=hermes:zip:tutorial     # export to only these machines
/usr/sun      -root=hermes:zip                # give root access only to these
/usr/new      -anon=0                         # give all machines root access
/usr/bin      -ro                             # export read-only to everyone
/usr/stuff    -access=zip,anon=-3,ro          # several options on one line
```

FILES

/etc/exports
/etc/xtab
/etc/hosts
/etc/netgroup
rc.local

SEE ALSO

exportent(3), hosts(5), netgroup(5), exportfs(8), nfsd(8)

WARNINGS

You cannot export either a parent directory or a subdirectory of an exported directory that is *within the same filesystem*. It would be illegal, for instance, to export both **/usr** and **/usr/local** if both directories resided on the same disk partition.

NOTES

exports is an optional product included with the NFS package; for more information, contact your CONVEX sales representative.

NAME

netgroup – list of network groups

DESCRIPTION

netgroup defines network wide groups, used for permission checking when doing remote mounts, remote logins, and remote shells. For remote mounts, the information in *netgroup* is used to classify machines; for remote logins and remote shells, it is used to classify users. Each line of the *netgroup* file defines a group and has the format

groupname member1 member2 ....

where member*i* is either another group name, or a triple:

(hostname, username, domainname)

Any of three fields can be empty, in which case it signifies a wild card. Thus

universal (,,)

defines a group to which everyone belongs.

A gateway machine should be listed under all possible hostnames by which it may be recognized:

wan (gateway,,) (gateway-ebb,,)

Field names that begin with something other than a letter, digit or underscore (such as "-") work in precisely the opposite fashion. For example, consider the following entries:

justmachines (analytica,-,convex)
justpeople (-,babbage,convex)

The machine *analytica* belongs to the group *justmachines* in the domain *convex*, but no users belong to it. Similarly, the user *babbage* belongs to the group *justpeople* in the domain *convex*, but no machines belong to it.

The *domainname* field refers to the domain *n* in which the triple is valid, not the name containing the trusted host.

Network groups are contained in the yellow pages, and are accessed through these files:

/usr/etc/yp/*domainname*/netgroup.dir
/usr/etc/yp/*domainname*/netgroup.pag
/usr/etc/yp/*domainname*/netgroup.byuser.dir
/usr/etc/yp/*domainname*/netgroup.byuser.pag
/usr/etc/yp/*domainname*/netgroup.byhost.dir
/usr/etc/yp/*domainname*/netgroup.byhost.pag

These files can be created from */etc/netgroup* using *makedbm*(8).

FILES

/etc/netgroup
/usr/etc/yp/*domainname*/netgroup.dir
/usr/etc/yp/*domainname*/netgroup.pag
/usr/etc/yp/*domainname*/netgroup.byuser.dir
/usr/etc/yp/*domainname*/netgroup.byuser.pag
/usr/etc/yp/*domainname*/netgroup.byhost.dir
/usr/etc/yp/*domainname*/netgroup.byhost.pag

SEE ALSO

getnetgrent(3), exports(5), makedbm(8), ypmake(8), ypserv(8)

NOTES

netgroup is an optional product included with the NFS package; for more information, contact your CONVEX sales representative.

NAME

publickey – public key database

SYNOPSIS

/etc/publickey

DESCRIPTION

/etc/publickey is the public key database used for secure networking. Each entry in the database consists of a network user name (which may refer either to a user or a hostname), followed by the user's public key (in hex notation), a colon, and then the user's secret key encrypted with its login password (also in hex notation).

This file is altered either by the user through the **chkey**(1) command or by the system administrator through the **newkey**(8) command. The file **/etc/publickey** should only contain data on the Yellow Pages master machine, where it is converted into the YP database **publickey.byname**.

SEE ALSO

chkey(1), **publickey**(3R), **newkey**(8), **ypupdated**(8C)

NOTES

publickey is an optional product included with the NFS package for customers in the U.S. and Canada only. For more information, contact your CONVEX sales representative.

NAME

rmtab – remotely mounted file system table

DESCRIPTION

rmtab resides in directory */etc* and contains a record of all clients that have done remote mounts of file systems from this machine. Whenever a remote *mount* is done, an entry is made in the *rmtab* file of the machine serving up that file system. *umount* removes entries, if of a remotely mounted file system. *umount -a* broadcasts to all servers, and informs them that they should remove all entries from *rmtab* created by the sender of the broadcast message. By placing a */etc/umount -at nfs* command in */etc/rc.local* followed immediately by a */etc/mount -at nfs* command, *rmtab* tables can be purged of entries made by a crashed host, which upon rebooting did not remount the same file systems it had before. The table is a series of lines of the form

hostname:directory

This table is used only to preserve information between crashes, and is read only by *mountd*(8) when it starts up. *mountd* keeps an in-core table, which it uses to handle requests from programs like *showmount*(8) and *shutdown*(8).

FILES

/etc/rmtab

SEE ALSO

showmount(8), mountd(8c), mount(8), umount(8), shutdown(8)

BUGS

Although the *rmtab* table is close to the truth, it is not always 100% accurate.

NOTES

The */etc/umount -at nfs* and */etc/mount -at nfs* commands should be placed in the */etc/rc.local* file after the *biod*(8) daemons are started up.

rmtab is an optional product included with the NFS package; for more information, contact your CONVEX sales representative.

NAME

rpc – rpc program number data base

DESCRIPTION

The *rpc* file contains user-readable names that can be used in place of rpc program numbers. Each line has the following information:

name of server for the rpc program
rpc program number
aliases

Items are separated by any number of blanks and/or tab characters. A "#" indicates the beginning of a comment; characters up to the end of the line are not interpreted by routines which search the file.

Here is an example of the */etc/rpc* file from the Convex UNIX System.

```
#
portmapper  100000      portmap sunrpc
rstatd                  100001      rstat rup perfmeter
rusersd                 100002      rusers
nfs                     100003      nfsprog nfsd
ypserv                  100004      ypprog
mountd                  100005      mount showmount
ypbind                  100007
walld                   100008      rwall shutdown rwalld
yppasswdd   100009      yppasswd
etherstatd  100010      etherstat
rquotad                 100011      rquotaprog quota rquota
sprayd                  100012      spray
3270_mapper 100013
rje_mapper  100014
selection_svc           100015      selnsvc
database_svc            100016
rexd                    100017      rex
alis                    100018
sched                   100019
llockmgr    100020
nlockmgr    100021
x25.inr                 100022
statmon                 100023
status                  100024      statd
bootparam   100026      bootparamd
ypupdated   100028      ypupdate
keyserv                 100029      keyserver
tfsd                    100037
nsed                    100038
nsemntd                 100039
pcnfsd                  150001      pcnfs
```

FILES

/etc/rpc

SEE ALSO

getrpcent(3N)

NOTES

rpc is an optional product included with the NFS package; for more information, contact your CONVEX sales representative.

NAME

updaters – configuration file for YP updating

SYNOPSIS

/usr/etc/yp/updaters

DESCRIPTION

The file **/usr/etc/yp/updaters** is a makefile (see **make**(1)) which is used for updating YP databases. The YP update mechanism can only be used for databases in a secure network, that is, one that has a **publickey**(5) database. Each entry in the file is a make target for a particular YP database. For example, if there is a YP database named **passwd.byname** that can be updated, there should be a **make** target named **passwd.byname** in the **updaters** file with the command to update the file. If you are providing the update command, it must follow the conventions described below.

The information necessary to make the update is passed to the update command through standard input. The information passed is described below; all items except the fourth and sixth are followed by a NEWLINE.

- Network name of client wishing to make the update (a string)
- Kind of update (an integer)
- Number of bytes in key (an integer)
- Actual bytes of key
- Number of bytes in data (an integer)
- Actual bytes of data

After getting this information through standard input, the command to update the particular database should decide whether the user is allowed to make the change. If not, it should exit with the status YPERR_ACCESS. If the user is allowed to make the change, the command should make the change and exit with a status of zero. If there are any errors that may prevent the updater from making the change, it should exit with the status that matches a valid YP error code described in **<rpcsvc/ypclnt.h>**.

FILES

/usr/etc/yp/updaters

SEE ALSO

make(1), **ypupdate**(3N), **publickey**(5), **ypupdated**(8C)

NOTES

updaters is an optional product included with the NFS package for customers in the U.S. and Canada only. For more information, contact your CONVEX sales representative.

NAME

ypfiles – the yellowpages database and directory structure

DESCRIPTION

The yellow pages (YP) network lookup service uses a distributed, replicated database of *dbm* files in the directory hierarchy at */usr/etc/yp* on each YP server. A *dbm* database consists of two files, created by calls to the *dbm*(3X) library package. One has the filename extension *.pag* and the other has the filename extension *.dir*. For instance, the database named *hosts.byname*, is implemented by the pair of files *hosts.byname.pag* and *hosts.byname.dir*. A *dbm* database served by the YP is called a YP *map*. A YP *domain* is a subdirectory of */usr/etc/yp* containing a set of YP maps. Any number of YP domains can exist. Each may contain any number of maps.

No maps are required by the YP lookup service itself, although they may be required for the normal operation of other parts of the system. There is no list of maps which YP serves - if the map exists in a given domain, and a client asks about it, the YP will serve it. For a map to be accessible consistently, it must exist on all YP servers that serve the domain. To provide data consistency between the replicated maps, an entry to run *ypxfr* periodically should be made in */usr/lib/crontab* on each server. More information on this topic is in ypxfr(8).

YP maps should contain two distinguished key-value pairs. The first is the key YP_LAST_MODIFIED, having as a value a ten-character ASCII order number. The order number should be the UNIX time in seconds when the map was built. The second key is YP_MASTER_NAME, with the name of the YP master server as a value. *makedbm* generates both key-value pairs automatically. A map that does not contain both key-value pairs can be served by the YP, but the *ypserv* process will not be able to return values for "Get order number" or "Get master name" requests. In addition, values of these two keys are used by *ypxfr* when it transfers a map from a master YP server to a slave. If *ypxfr* cannot figure out where to get the map, or if it is unable to determine whether the local copy is more recent than the copy at the master, you must set extra command line switches when you run it.

YP maps must be generated and modified only at the master server. They are copied to the slaves using *ypxfr*(8) to avoid potential byte-ordering problems among YP servers running on machines with different architectures, and to minimize the amount of disk space required for the dbm files. The YP database can be initially set up for both masters and slaves by using *ypinit*(8).

After the server databases are set up, it is probable that the contents of some maps will change. In general, some ASCII source version of the database exists on the master, and it is changed with a standard text editor. The update is incorporated into the YP map and is propagated from the master to the slaves by running */usr/etc/yp/Makefile* . All Convex-supplied maps have entries in */usr/etc/yp/Makefile ;* if you add a YP map, edit this file to support the new map. The makefile uses *makedbm* to generate the YP map on the master, and *yppush* to propagate the changed map to the slaves. *yppush* is a client of the map *ypservers* , which lists all the YP servers. For more information on this topic, see yppush(8).

STANDARD MAPS

Convex supplies the following YP maps:

YP map name	Data contained	Key	Source file
bootparams	NETdisk boot data	client name	/etc/bootparams
ethers.byaddr	Ethernet addrs	address	/etc/ethers
ethers.byname	Ethernet addrs	hostname	/etc/ethers
group.bygid	group info	group id	/etc/group
group.byname	group info	group name	/etc/group
hosts.byaddr	Internet addrs	address	/etc/hosts
hosts.byname	Internet addrs	hostname	/etc/hosts
mail.aliases	sendmail aliases	alias name	/usr/lib/aliases
netgroup	network groups	group name	/etc/netgroups
netgroup.byhost	network groups	host name	/etc/netgroups
netgroup.byuser	network groups	user name	/etc/netgroups
netid.byname	Secure RPC netids	netname	/etc/netid
networks.byaddr	Internet subnets	address	/etc/networks
networks.byname	Internet subnets	name	/etc/networks
passwd.byname	passwd info	user name	/etc/passwd
passwd.byuid	passwd info	user id	/etc/passwd
protocols.byname	DARPA protocols	proto name	/etc/protocols
protocols.bynumber	DARPA protocols	proto num	/etc/protocols
publickey.byname	Secure RPC keys	netname	/etc/publickey
pwrestrict.byname	pw restrictions	user name	/etc/pwrestrict
pwrestrict.byuid	pw restrictions	user id	/etc/pwrestrict
rpc.byname	rpc programs	prog name	/etc/rpc
rpc.bynumber	rpc programs	prog num	/etc/rpc
services	Internet services	serv num	/etc/services
services.byname	Internet services	serv name	/etc/services
ypservers	YP servers	N/A	/etc/ypservers

SEE ALSO

makedbm(8), ypinit(8), ypmake(8), ypxfr(8), yppush(8), yppoll(8), ypserv(8), rpcinfo(8),

NOTES

YP is an optional product included with the NFS package; for more information, contact your CONVEX sales representative.

NAME

INSTALL – installs NETdisk architecture and clients

DESCRIPTION

The *INSTALL* script guides the super-user through installation procedures for a server of NETdisk clients. It allows the user to select and install client architectural support on the server from the client manufacturer's software distribution tape, and to create the environment on the server for specific client workstations. *INSTALL* is invoked with no arguments, but will ask the user to supply various parameters to be used during the installation process. Suitable answers are provided for most prompts. Notes in this manual page should be read before running the script.

Before starting, you should take a few minutes to consider the server disk space requirements and layout. You will need room for these trees:

roots Client root trees, one for each NETdisk client served. Initial space used is approximately 1.3MB per client, but room must be left for *tmp* and *spool* space for each client.

swaps Client swap files, one for each NETdisk client served. A good initial value is 16MB per client.

dumps Client crash dump files. Initially created empty, so no space is used at first. These files are used when a client workstation has crashed and a memory dump is saved by the *savecore* program. If no dump file is specified for a client, *savecore* will instead use the swap file for that client.

execs Architecture-dependent */usr* filesystem used by the NETdisk clients. Space required is approximately 30-50MB per architecture type, depending on optional software selected.

home A general work area set up to be mounted by all NETdisk clients. The subdirectory */home/<servername>* will be created for each server. The intent is that the individual user home directories are located in these trees, but other common (local) software may be placed here as well. The *home* path may be set to **none** if this setup is not desired.

If individual NETdisk clients are to be installed at the same time as installing architectural support, then the operator must have information available about each client before running *INSTALL*. Information required for each client is architecture type, ethernet address, and IP address. The network addresses must already be entered into the appropriate administration files before running *INSTALL*.

It is also possible to use *INSTALL* to add new NETdisk clients after the base architecture has been installed. The script will detect this fact and present the administrator with several options to choose from, as described below.

PARAMETER VALUES

The tape drive may either be local to the server, or remote on another machine on the network. If a remote tape drive is selected, you will be prompted for the machine name of that remote host. This machine name is verified in the yellow pages (via ypmatch), or in */etc/hosts* if the yellow pages is not running on the server. This is detected by checking the domainname returned by the *domainname* command. You should set the domainname to either *noname* or leave it NULL if you don't want the yellow pages to be used during any NETdisk client installation steps.

Once the tape location is specified, you will be asked for the tape type. This can either be one of the default driver designations listed, or you may specify a full pathname of a non-rewinding raw tape device (such as /dev/nrmt0). Note that only one tape device may be specified per invocation of *INSTALL*. If you have different tape formats for different architectures, you will need to run *INSTALL* separately for each tape type.

The next step is to specify the various client architectures being loaded during this process. The architecture names used should be the same as the designations on the software distribution tapes from the client workstation manufacturer. For each architecture entered, a pathname will be requested where the software is to be loaded on the server. The architecture name will be

automatically appended to the entered path. An example entry to this prompt is */export/exec*. For a sun3 distribution tape for SunOS 4.0, for example, this would make the actual install location for the software be */export/exec/sun3*. Once the pathname is entered for one architecture, this may be used as a default for following architectures by simply hitting **<RETURN>** to those prompts.

After the exec pathname is specified, the script determines if the executables for that architecture have already been loaded. If so, the user may elect to ignore the entered architecture, remove the existing tree and reload it, load additional packages for the architecture, upgrade the architecture with new files from an upgrade tape, or leave the executable tree alone and proceed. In all cases but the first, the user will be allowed to enter information for adding new clients of that architecture.

When selecting new software to load, the tape's table of contents will be searched to obtain the software choices available for loading from that distribution. The operator must select options to load by answering *yes* or *no* to the prompts as they are presented. These prompts will include the size of the particular software option, together with an indication as to the importance of loading the option. A load list is constructed in /tmp/EXTRACTLIST.<architecture> to be used in *setup_exec*.

Once the software selections have been made for the given architecture, the user will be asked whether shared objects should be loaded in such a way that they can be shared by clients of different architectures. Shared objects refer to such items as manual pages and common library support packages such as timezone information and System V terminfo databases. The benefit of sharing is that disk space is saved when two or more architectures are loaded onto the same server. Sharing is not necessary (and should not be selected) if the server is to support only one NETdisk client architecture.

When all desired architectures are specified, enter **done** to the *next architecture* prompt. At this point, you will be asked to specify the individual clients for each architecture. It is required that the ethernet addresses *(/etc/ethers* or the *ethers* yellow pages map) and IP addresses *(/etc/hosts* or the *hosts* yellow pages map) have both been updated with correct information for all new clients being added. This information is verified during the installation process similarly to the remote tape hostname mentioned above. Note that the special character ^ may be used as input to any prompt requesting client information to cause that client information to be discarded, and a new client may be specified. This can be used to correct mistakes after they have already been entered (but before reaching the end of specification requests for that client). Also, information entered in a particular category for one client may be used for following clients by entering only a **RETURN** in response to the following prompts.

Additional information required for clients is the yellow pages type (master, slave, client, or none). Most likely choices here are **client** or **none.** Swap size for the client can be specified in a number of different units. Enter anything not starting with a digit to the prompt to get examples. Typical value is **16M** (for 16 megabytes).

The various pathnames that are specified (for root, swap, dump) are the locations on the server where these functions will be installed. The entered pathname should be the generic path; the client's hostname will be appended to make the actual location. Example input pathname is */export/root* (which would turn into */export/root/<clientname>* during that client's installation. The **home** path is where the user home directories are installed. The convention is that users on workstations being served by **server** will have home directories on */home/<servername>/<username>*. Typical answer to this prompt is */home*. Defaults provided in the script for *root, swap, dump,* and *home* are **/export/root, /export/swap, none,** and **/home,** respectively.

When all client information for a given architecture is complete, enter **done** to the prompt for the next client name. When all architectures are complete, you will be asked to proceed. If you enter *no* to this question, the current state is left in configuration files in /tmp. You may

continue from this point by re-running *INSTALL* and entering **continue** to the first prompt asking for architecture names. When installation begins, each architecture and its clients will be installed in turn. *setup_exec* will first be invoked to load the software from the distribution tapes. Then *setup_client* will be invoked for each client of that architecture.

DIAGNOSTICS

There are various diagnostic messages output while running this script. They should all be self-explanatory.

FILES

/tmp/*.<architecture>

SEE ALSO

opt_software(8), setup_client(8), setup_exec(8), hosts(5), ethers(5)

NOTES

INSTALL is an optional product included with the NFS package; for more information, contact your CONVEX sales representative.

NAME

automount – automatically mount NFS file systems

SYNOPSIS

automount [**–mnTv**] [**–f** *master_map*] [**–D** *env_var=value*] [**–M** *tempdir*] [**–tl** *duration*] [**–tm** *interval*] [**–tw** *interval*]
[*directory mapname* [*-mount-options*]] ...

DESCRIPTION

automount is a daemon that will automatically and transparently mount an NFS file system whenever a file or directory within in that system is opened. **automount** forks a daemon, which appears to the kernel to be an NFS server; lookups on the specified *directory* are intercepted by this daemon, which uses the map contained in *mapname* to determine a server, exported file system, and appropriate mount options for a given file system. The named map can either be a file on the local system, or a Yellow Pages map. *directory* is a full pathname starting with a '/'.

When supplied, *-mount-options* consists of the leading – and a comma-separate list of **mount**(8) options; if mount options are specified in the map, however, those in the map take precedence.

Once mounted, members of the *directory* are made available using a symbolic link to the real mount point within a temporary directory.

If *directory* does not exist, the daemon creates it, and then removes it automatically when the daemon exits.

Since the name-to-location binding is dynamic, updates to a Yellow Pages map are transparent to the user. This obviates the need to "pre-mount" shared file systems for applications that have "hard coded" references to files. It also obviates the need to maintain records of which hosts must be mounted for what applications.

Maps

automount looks first for the indicated *mapname* in a file by that name. If there is no such file, it looks for a YP map by that name.

An automount map is composed of a list of mappings, with one mapping per line. Each mapping is composed of the following fields:

basename [*–mount-options*] *location* [...]

where *basename* is the name of a subdirectory within the *directory* specified in the **automount** command line (not a relative pathname). The *location* field consists of an entry of the form:

host:*directory*[:*subdir*]

where *host* is the name of the host from which to mount the file system, *directory* is the pathname of the directory to mount, and *subdir*, when supplied, is the name of a subdirectory to which the symbolic link is made. This can be used to prevent duplicate mounts in cases where multiple directories in the same remote file system are accessed.

The contents of a YP map can be included within a map by adding an entry of the form:

+*mapname*

A mapping can be continued across line breaks using a \ as the last character before the NEWLINE. Comments begin with a # and end at the subsequent NEWLINE.

If more than one *location* is supplied, there is no guarantee as to which location will be used; the first location to respond to the mount request gets mounted. The *mount-options* field can be used to supply options to the **mount**(8) command for the mounted file system.

Special Maps

There are two special maps currently available. The **–hosts** map uses the Yellow Pages **hosts.byname** map to locate a remote host when the hostname is specified as a subdirectory of *directory*. This map specifies mounts of all exported file systems from any host. For instance, if the following **automount** command is already in effect:

automount /net –hosts

then a reference to **/net/hermes/usr** would initiate an automatic mount of all file systems from **hermes** that **automount** can mount; references to a directory under **/net/hermes** will refer to the corresponding directory on **hermes**. The **–passwd** map uses the **passwd**(5) database to attempt to locate the home directory of a user. For instance, if the following **automount** command is already in effect:

automount /homes –passwd

then if the home directory shown in the passwd entry for the user *username* has the form */dir/server/username*, and **server** matches the host system on which that directory resides, references to files in **/homes/***username* result in the file system containing that directory being mounted, if necessary. All such references will refer to that user's home directory. Note that the use of the **–passwd** map requires home directories to be set up according to the */dir/server/username* naming scheme on the user's home machine and all participating clients which are to be able to mount the home directory.

Configuration

automount normally consults the **auto.master** Yellow Pages configuration database for a list of initial *directory-to-mapname* pairs, and sets up automatic mounts for them in addition to those given on the command line; if there are duplications, the command-line arguments take precedence. (Note that this database contains arguments to the **automount** command, rather than mappings, and that **automount** does *not* look for an **auto.master** file on the local host.)

OPTIONS

–m Suppress initialization of *directory-mapname* pairs listed in the **auto.master** Yellow Pages database.

–n Disable dynamic mounts. With this option, references through the **automount** daemon only succeed when the target filesystem has been previously mounted. This can be used to prevent NFS servers from cross-mounting each other.

-T Trace. Have automount log its actions to standard error stream. May be specified more than once to increase detail.

-v Verbose. Have automount issue verbose error messages via syslog(3) when is has difficulties. Errors are logged to the LOG_DAEMON facility at LOG_ERR priority.

–f master_map
Load *directory-mapname* pairs from the named master map file before consulting the **auto.master** Yellow Pages database.

–D env_var=value
Define an environment variable to be used by the automounter daemon in expansion of map entries.

–M tmpdir
Use named directory for all new mounts, rather than the default /tmp_mnt.

–tl *duration*
Specify a *duration*, in seconds, that a looked up name remains cached when not in use. The default is 5 minutes.

–tm *interval*

Specify an *interval*, in seconds, between attempts to mount a filesystem. The default is 30 seconds.

–tw *interval*

Specify an *interval*, in seconds, between attempts to dismount filesystems that have exceeded their cached times. The default is 1 minute.

EXAMPLE

tutorial# automount –m /net –hosts

Provide **automount** access to the exported file systems of any host in the Yellow Pages **hosts.byname** database, by prefixing the pathname with **/net/***hostname***/** :

tutorial% ls /net/hermes/usr/src ...

FILES

/tmp_mnt	directory under which filesystems are dynamically mounted

SEE ALSO

mount(8)

BUGS

Shell filename expansion does not apply to objects not currently mounted or cached. For instance, in the above example, the command **ls /net/*** might not list **hermes** as a subdirectory of **/net**.

NAME

bootparamd, rpc.bootparamd – boot parameter server

SYNOPSIS

/usr/etc/rpc.bootparamd [**–d**]

DESCRIPTION

bootparamd is a server process that provides information to diskless clients necessary for booting. It consults the **bootparams** database. If the client is not found there, or if the Yellow Pages service is not running, then the */etc/bootparams* file is consulted.

bootparamd can be invoked either by *inetd*(8C) or by the user.

OPTIONS

–d Display the debugging information.

FILES

/etc/bootparams

SEE ALSO

bootparams(5), inetd(8C)

NOTES

bootparamd is an optional product included with the NFS package; for more information, contact your CONVEX sales representative.

NAME

create_root – generate root partition for a diskless client

SYNOPSIS

create_root *hostname yptype rootpath homepath execpath sharepath server domain*

DESCRIPTION

create_root does the necessary manipulations to customize the prototype diskless client root tree for a particular client. This includes copying the prototype tree into the client's rootpath, making the appropriate devices in the client's */dev* directory, and customizing the administration files in the client's */etc* directory.

ARGUMENTS

Running the command with no arguments given will list the available options.

hostname
Name of the client whose root we are creating.

yptype Yellow pages mode of operation for the client and is one of **master, slave, client,** or **none.** Typical values are **client** or **none.**

rootpath
Full pathname on the server where the client's root tree will be installed. Example value is */export/root/<clientname>*.

homepath
Path being used for user home directories and is typically */home*. If another server is providing this service, then the path may have the servername and colon prepended, as in *servername:/home*. May also be specified as **none** if this feature is not used.

execpath
Full pathname on the server where the executable files for the client's architecture are located. Typical value is */export/exec/sun3*.

sharepath
Full pathname on the server where the sharable files are located. Typical value is */export/exec/share*. The client will be set up to mount this path on its local /usr/share directory. If this mount is not desired, then the value **none** may be given instead of a path.

server Hostname of the server machine.

domain Yellow pages domainname for the client.

DIAGNOSTICS

Error output should be self explanatory.

SEE ALSO

INSTALL(8), setup_client(8)

NOTES

create_root is an optional product included with the NFS package; for more information, contact your CONVEX sales representative.

NAME

exportfs – export and unexport directories to NFS clients

SYNOPSIS

/usr/etc/exportfs [**–avu**] [**–o** *options*] [**directory**]

DESCRIPTION

exportfs makes a local directory (or file) available for mounting over the network by NFS clients. It is normally invoked at boot time by the **/etc/rc.local** script, and uses information contained in the **/etc/exports** file to export a *directory* (which must be specified as a full pathname). The super-user can run **exportfs** at any time to alter the list or characteristics of exported directories. Directories that are currently exported are listed in the file **/etc/xtab**.

With no options or arguments, **exportfs** prints out the list of directories currently exported.

OPTIONS

–a All. Export all directories listed in **/etc/exports**, or if **–u** is specified, unexport all of the currently exported directories.

–v Verbose. Print each directory as it is exported or unexported.

–u Unexport the indicated directories.

–i Ignore the options in **/etc/exports.** Normally, **exportfs** will consult **/etc/exports** for the options associated with the exported directory.

–o *options*

Specify a comma-separated list of optional characteristics for the directory being exported. *options* can be selected from among:

ro Export the directory read-only. If not specified, the directory is exported read-write.

async Export the directory asynchronously. If not specified, the directory is exported synchronously. Setting this option provides for significantly better write performance to NFS files residing in the directory. However, this performance gain is offset by the fact that the data written is not guaranteed to be on secondary storage before the server acknowledges the write completion. Thus, this option must be used with caution.

rw=*hostname*[**:***hostname*] . . .

Export the directory read-mostly. Read-mostly means exported read-only to most machines, but read-write to those specified. If not specified, the directory is exported read-write to all.

anon=*uid*

If a request comes from an unknown user, use *uid* as the effective user ID. Note: root users (uid 0) are always considered "unknown" by the NFS server, unless they are included in the "root" option below. The default value for this option is –2. Setting the value of "anon" to -1 disables anonymous access. Note that by default secure NFS accepts insecure requests as anonymous, and those wishing for extra security can disable this feature by setting "anon" to -1.

root=*hostname*[**:***hostname*] . . .

Give root access only to the root users from a specified *hostname*. The default is for no hosts to be granted root access.

access=*client*[**:***client*] . . .

Give mount access to each *client* listed. A *client* can either be a hostname, or a netgroup (see *netgroup*(5)). Each *client* in the list is first checked for in the **/etc/netgroup** database, and then the **/etc/hosts** database. The default value allows any machine to mount the given directory.

secure Require clients to use a more secure protocol when accessing the directory.

FILES

/etc/exports	static export information
/etc/xtab	current state of exported directories
/etc/netgroup	

SEE ALSO

exports(5), netgroup(5), mount(8)

WARNINGS

You cannot export a directory that is either a parent- or a sub-directory of one that is currently exported and *within the same filesystem.* It would be illegal, for example, to export both **/usr** and **/usr/local** if both directories resided in the same disk partition.

A filesystem should not be mounted or unmounted while one of its directories is exported. The proper sequence is to unexport a directory first, unmount the filesystem, re-mount the same or a different filesystem, then re-export the directory. If this procedure is not followed, it is possible to have a directory which *exportfs* claims is exported but which cannot be remotely mounted. Re-exporting the directory fixes this inconsistency.

NOTES

exportfs is an optional product included with the NFS package; for more information, contact your CONVEX sales representative.

NAME

extracting – extract from a tar formatted software distribution tape

SYNOPSIS

extracting *tapedev skip blockfactor keywords* [*tapeserver*]

DESCRIPTION

extracting is used to extract a particular software release file from a distribution tape. It is intended to be called from *setup_exec* as part of the installation procedure for diskless/NFS client support. The release file is assumed to be in **tar** format and will be read from the tape into the current working directory.

ARGUMENTS

These arguments must be specified in the order given in the synopsis above.

tapedev Full pathname of the non-rewinding raw tape device on which the distribution tape is mounted. This device may be on a remote machine. If so, then the optional *tapeserver* argument is the hostname of the machine where *tapedev* is located.

skip Number of tape files to skip over before reading the tape. It is the argument passed to the command *mt* to position the tape.

blockfactor

Blocking factor to use with *tar* while reading the file.

keywords

Quoted string which is used as part of the output to the user to identify what is being loaded from tape.

SEE ALSO

mt(1), tar(1), rsh(1), INSTALL(8), setup_exec(8), verify_tapevol_arch(8)

NOTES

extracting is an optional product included with the NFS package; for more information, contact your CONVEX sales representative.

NAME

fix_bootparams – add or delete entries from /etc/bootparams

SYNOPSIS

fix_bootparams *op clientname* [*rootpath swappath dumppath*]

DESCRIPTION

fix_bootparams is used to add or remove a diskless/NFS client's entries from the server's */etc/bootparams* file. This file specifies the NFS pathname of the client's root partition and swap and dump files. *fix_bootparams* is intended to be run by *setup_client* as part of the administration of diskless/NFS clients.

ARGUMENTS

op Operation to be performed and is either **add** or **remove.**

clientname

Hostname of the client to be added or removed.

The pathname arguments are only required for the **add** operation and are specified in standard NFS format: *servername:pathname-on-server.*

rootpath

NFS pathname of the client's root partition.

swappath

NFS pathname of the client's swap file.

dumppath

NFS pathname of the client's crash dump file. May be specified as **none.**

FILES

/etc/bootparams

SEE ALSO

bootparams(5), INSTALL(8), bootparamd(8), setup_client(8)

NOTES

fix_bootparams is an optional product included with the NFS package; for more information, contact your CONVEX sales representative.

NAME

keyenvoy – talk to keyserver

SYNOPSIS

keyenvoy

DESCRIPTION

keyenvoy is used by some RPC programs to talk to the key server, **keyserv**(8C). The key server will not talk to anything but a root process, and **keyenvoy** is a set-uid root process that acts as an intermediary between a user process that wishes to talk to the key server and the key server itself.

This program cannot be run interactively.

SEE ALSO

keyserv(8C)

NOTES

keyenvoy is an optional product included with the NFS package for customers in the U.S. and Canada only. For more information, contact your CONVEX sales representative.

NAME

keyserv – server for storing public and private keys

SYNOPSIS

keyserv [**–n**]

DESCRIPTION

keyserv is a daemon that is used for storing the private encryption keys of each user logged into the system. These encryption keys are using for accessing secure network services such as secure NFS. When a user logs in to the system, the **login(1)** program uses the login password to decrypt the user's encryption key stored in the Yellow Pages, and then gives the decrypted key to the **keyserv** daemon to store away.

Normally, root's key is read from the file **/etc/.rootkey** when the daemon starts up. This is useful during power-fail reboots when no one is around to type a password, yet you still want the secure network services to operate normally.

OPTIONS

–n Do not read root's key from **/etc/.rootkey**. Instead, prompt the user for the password to decrypt **root 's** key stored in the Yellow Pages and then store the decrypted key in **/etc/.rootkey** for future use. This option is useful if the **/etc/.rootkey** file ever gets out of date or corrupted.

FILES

/etc/.rootkey

SEE ALSO

login(1), **publickey**(5)

NOTES

keyserv is an optional product included with the NFS package for customers in the U.S. and Canada only. For more information, contact your CONVEX sales representative.

NAME

make_dirs – make a directory and its parents

SYNOPSIS

make_dirs *dirname* [*mode*]

DESCRIPTION

This *csh* script will make the directory *dirname* and all necessary parent directories leading to it. If *mode* is specified, then all created directories are set to the mode given; otherwise the newly created directories will have modes subject to the user's **umask** value.

SEE ALSO

chmod(1), mkdir(1), umask(2)

NOTES

make_dirs is an optional product included with the NFS package; for more information, contact your CONVEX sales representative.

NAME

makedbm – make a yellow pages dbm file

SYNOPSIS

/usr/etc/yp/makedbm [**–i** *yp_input_file*] [**–o** *yp_output_name*] [**–d** *yp_domain_name*] [**–m** *yp_master_name*] *infile outfile*
/usr/etc/yp/makedbm [**–u** *dbmfilename*]

DESCRIPTION

makedbm takes *infile* and converts it to a pair of files in *dbm*(3X) format, namely *outfile*.pag and *outfile*.dir. Each line of the input file is converted to a single *dbm* record. All characters up to the first tab or space form the key, and the rest of the line is the data. If a line ends with \, then the data for that record is continued on to the next line. It is left for the clients of the yellow pages to interpret #; *makedbm* does not itself treat it as a comment character. *infile* can be –, in which case standard input is read.

makedbm is meant to be used in generating *dbm* files for the yellow pages, and it generates a special entry with the key *yp_last_modified*, which is the date of *infile* (or the current time, if *infile* is –).

OPTIONS

–i Create a special entry with the key *yp_input_file*.

–o Create a special entry with the key *yp_output_name*.

–d Create a special entry with the key *yp_domain_name*.

–m Create a special entry with the key *yp_master_name*. If no master host name is specified, *yp_master_name* will be set to the local host name.

–u Undo a *dbm* file. That is, print out a *dbm* file one entry per line, with a single space separating keys from values.

EXAMPLE

It is easy to write shell scripts to convert standard files such as */etc/passwd* to the key value form used by *makedbm*. For example,

```
#!/bin/awk -f
BEGIN { FS = ":"; OFS = "\t"; }
{ print $1, $0 }
```

takes the */etc/passwd* file and converts it to a form that can be read by *makedbm* to make the yellow pages file *passwd.byname*. That is, the key is a username, and the value is the remaining line in the */etc/passwd* file.

SEE ALSO

dbm(3X), yppasswd(1), domainname(1)

NOTES

makedbm is an optional product included with the NFS package; for more information, contact your CONVEX sales representative.

NAME

mkfile – create a file

SYNOPSIS

mkfile [**-nv**] *size*[**b**|**k**|**m**] *filename* ...

DESCRIPTION

mkfile creates one or more files that are suitable for use as NFS-mounted swap areas. The file is padded with zeroes by default. The default size unit is bytes, but the following suffixes may be used to multiply by the given factor: **b** (512), **k** (1024), and **m** (1048576).

OPTIONS

-n Create an empty *filename*. The size is noted, but disk blocks aren't allocated until data is written to them. The *filename* produced essentially has a gap or "hole" which occupies no physical space and reads as zeroes.

-v Verbose. Report the names and sizes of created files.

WARNING

If a client's swap file is removed and recreated, it must be re-exported before the client will be able to access it. This action may only be done when the client is not running.

SEE ALSO

chmod(2), lseek(2), stat(2), exportfs(8)

NOTES

mkfile is an optional product included with the NFS package; for more information, contact your CONVEX sales representative.

NAME

mountd, rpc.mountd – NFS mount request server

SYNOPSIS

/usr/etc/rpc.mountd [**–n**]

DESCRIPTION

mountd is an RPC server that answers file system mount requests. It reads the file */etc/exports*, described in *exports*(5), to determine which file systems are available for mounting by which machines. It also provides information as to what filesystems are mounted by which clients. This information can be printed using the *showmount*(8) command.

The *mountd* daemon is normally invoked by *inetd*(8C).

OPTIONS

–n Do not check that the clients are root users. Though this options make things slightly less secure, it does allow older versions (pre-3.0) of client NFS to work.

SEE ALSO

nfs(4), rpc(3N), mount(3R), exports(5), services(5), inetd(8C), showmount(8)

NOTES

mountd is an optional product included with the NFS package; for more information, contact your CONVEX sales representative.

NAME

newkey – create a new key in the publickey database

SYNOPSIS

newkey [**–h** *hostname*] [**–u** *username*]

DESCRIPTION

newkey is normally run by the network administrator on the YP master machine in order to establish public keys for users and super-users on the network. These keys are needed for using secure RPC or secure NFS.

newkey will prompt for the login password of the given username and then create a new public/secret key pair in **/etc/publickey** encrypted with the login password of the given user.

Use of this program is not required: users may create their own keys using **chkey (1).**

OPTIONS

–u *username* Create a new public key for the given username. Prompts for the Yellow Pages password of the given username.

–h *hostname* Create a new public key for the super-user at the given hostname. Prompts for the root password of the given hostname.

SEE ALSO

keylogin(1), chkey(1), publickey(5), keyserv(8)

NOTES

newkey is an optional product included with the NFS package for customers in the U.S. and Canada only. For more information, contact your CONVEX sales representative.

NAME

nfsd, biod – NFS daemons

SYNOPSIS

/etc/nfsd [nservers]

/etc/biod [nservers]

DESCRIPTION

nfsd starts the *NFS*(4) server daemons that handle client filesystem requests. *nservers* is the number of file system request daemons to start. This number should be based on the load expected on this server. Four seems to be a good number.

biod starts *nservers* asynchronous block I/O daemons. This command is used on a NFS client to buffer cache handle read-ahead and write-behind. The magic number for *nservers* in here is also four.

SEE ALSO

mountd(8C), exports(5), nfssvc(2), mount(8)

NOTES

nfsd is an optional product included with the NFS package; for more information, contact your CONVEX sales representative.

NAME

nfsstat – Network File System statistics

SYNOPSIS

/usr/etc/nfsstat [**–csnrz**] [**vmunix** [**core**]]

DESCRIPTION

nfsstat displays statistical information about the Network File System (NFS) and Remote Procedure Call (RPC) interfaces to the kernel. It can also be used to reinitialize this information. If no options are given the default is

nfsstat –csnr

That is, print everything and reinitialize nothing.

OPTIONS

–c Display client information. Only the client side NFS and RPC information will be printed. Can be combined with the **–n** and **–r** options to print client NFS or client RPC information only.

–s Display server information. Works like the **–c** option above.

–n Display NFS information. NFS information for both the client and server side will be printed. Can be combined with the **–c** and **–s** options to print client or server NFS information only.

–r Display RPC information. Works like the **–n** option above.

–z Zero (reinitialize) statistics. Can be combined with any of the above options to zero particular sets of statistics after printing them. The user must have write permission on */dev/kmem* for this option to work.

DISPLAYS

The server RPC display includes the following fields:

calls total number of RPC calls received

badcalls total number of calls rejected

nullrecv number of times no RPC packet was available when trying to receive

badlen number of packets that were too short

xdrcall number of packets that had a malformed header

The server NFS display shows the number of NFS calls received (**calls**) and rejected (**badcalls**), and the counts and percentages for the various calls that were made.

The client RPC display includes the following fields:

calls total number of RPC calls sent
badcalls total of calls rejected by a server
retrans number of times a call had to be retransmitted
badxid number of times a reply did not match the call
timeout number of times a call timed out
wait number of times a call had to wait on a busy handle
newcred number of times authentication information had to be refreshed

The client NFS display shows the number of calls sent and rejected, as well as the number of times a handle was received (**nclget**), the number of times a call had to sleep while awaiting a handle (**nclsleep**), as well as a count of the various calls and their respective percentages.

FILES

/vmunix Kernel name list
/dev/mem Kernel data values

*/lib/kernsyms/symdata_** Kernel symbol addresses

SEE ALSO

nfs(4)

NAME

opt_software – select optional software from a distribution tape

SYNOPSIS

opt_software *arch tapedev* [*tapehost*]

DESCRIPTION

opt_software reads the XDR-format table of contents found on software distribution tapes intended for use by diskless/NFS servers and allows the user to construct an extraction list of optional software. After the table of contents is read, the user will be presented with a series of prompts asking whether to select a particular software package from the distribution tape. In addition to the name, the user will be told the size of the package in bytes along with an indication of the importance of the selection (required, desirable, common, or optional). The user selects the software by entering *yes* or *no* to each of the prompts as they are presented. The selection list is left in the file */tmp/EXTRACTLIST.<architecture>* for use by *setup_exec*.

ARGUMENTS

arch Name of the software architecture, such as **sun2, sun3,** or **sun4.**

tapedev Full pathname of the tape drive device where the tape is mounted. This program assumes that the table of contents is located in the second file on the tape. Alternatively, *tapedev* may be the pathname of an arbitrary file which contains the XDR format table of contents. This is assumed if the pathname does not start with */dev*.

tapehost

Given if the tape device is located on a remote system. The presence of this option requires that *tapedev* be the pathname of an actual tape device.

FILES

/tmp/EXTRACTLIST.<architecture>

SEE ALSO

INSTALL(8), setup_exec(8)

NOTES

opt_software is an optional product included with the NFS package; for more information, contact your CONVEX sales representative.

NAME

pong – network debugging

SYNOPSIS

/usr/etc/pong host [**timeout**]

DESCRIPTION

pong repeatedly sends an icmp echo packet to *host* and reports whether or not a reply was received. It keeps trying until *timeout* seconds have elapsed, or an answer is received. The default timeout is 20 seconds. The *timeout* must not be negative or zero. The *host* argument can be a name or an internet address.

SEE ALSO

icmp(4P)

NOTES

pong is an optional product included with the NFS package; for more information, contact your CONVEX sales representative.

NAME

portmap – DARPA port to RPC program number mapper

SYNOPSIS

/etc/portmap

DESCRIPTION

portmap is a server that converts RPC program numbers into DARPA protocol port numbers. It must be running in order to make RPC calls.

When an RPC server is started, it will tell *portmap* what port number it is listening to, and what RPC program numbers it is prepared to serve. When a client wishes to make an RPC call to a given program number, it will first contact *portmap* on the server machine to determine the port number where RPC packets should be sent.

Normally, standard RPC servers are started by *inetd*(8c), so *portmap* must be started before *inetd* is invoked.

SEE ALSO

rpcinfo(8), inetd(8C)

BUGS

If *portmap* crashes, all servers must be restarted.

NOTES

portmap is an optional product included with the NFS package; for more information, contact your CONVEX sales representative.

NAME

rexd, rpc.rexd – RPC-based remote execution server

SYNOPSIS

/usr/etc/rpc.rexd

DESCRIPTION

rexd is an RPC server for remote program execution. This daemon is started by *inetd*(8C) whenever a remote execution request is made, if the following line is placed in */etc/inetd.conf*:

```
rex   stream  tcp     wait    root    1 /usr/etc/rpc.rexd     rexd
```

For non-interactive programs standard file descriptors are connected directly to TCP connections. Interactive programs involve pseudo-terminals, similar to the login sessions provided by *rlogin*(1). This daemon may use the NFS to mount file systems specified in the remote execution request.

FILES

/dev/tty[pqrs]*n*	pseudo-terminals used for interactive mode
/etc/passwd	authorized users
/tmp_rex/rexd??????	temporary mount points for remote file systems.
/etc/hosts.equiv	authorized hosts
~/.rhosts	authorized hosts

SEE ALSO

rex(1C), rex(3R), exports(5), inetd(8C)

DIAGNOSTICS

Diagnostic messages are normally printed on the console, and returned to the requestor.

BUGS

Should be better access control.

RESTRICTIONS

Root cannot execute commands using **rexd** client programs such as **rex(1C).**

NAME

rpcinfo – report RPC information

SYNOPSIS

/usr/etc/rpcinfo –p [*host*]

/usr/etc/rpcinfo [**–n** *portnum*] **–u** *host program* [*version*]

/usr/etc/rpcinfo [**–n** *portnum*] **–t** *host program* [*version*]

/usr/etc/rpcinfo –b *program version*

rpcinfo –d *program version*

DESCRIPTION

rpcinfo makes an RPC call to an RPC server and reports what it finds.

OPTIONS

–p Probe the portmapper on *host*, and print a list of all registered RPC programs. If *host* is not specified, it defaults to the value returned by *hostname*(1).

–u Make an RPC call to procedure 0 of *program* on the specified *host* using UDP, and report whether a response was received.

–t Make an RPC call to procedure 0 of *program* on the specified *host* using TCP, and report whether a response was received.

–n Use *portnum* as the port number for the *–t* and *–u* options instead of the port number given by the portmapper.

–b Make an RPC broadcast to procedure 0 of the specified *program* and *version* using UDP and report all hosts that respond. This option may return multiple responses from a given server due to UDP retries, so it is recommended that the output be piped through *sort(1)* and *uniq(1)* if this is a problem.

–d Delete registration for the RPC service of the specified *program* and *version*. This option can be exercised only by the super-user.

The *program* argument can be either a name or a number.

If a *version* is specified, *rpcinfo* attempts to call that version of the specified *program*. Otherwise, *rpcinfo* attempts to find all the registered version numbers for the specified *program* by calling version 0 (which is presumed not to exist; if it does exist, *rpcinfo* attempts to obtain this information by calling an extremely high version number instead) and attempts to call each registered version. Note: the version number is required for the **–b** and **–d** options.

EXAMPLES

To show all of the RPC services registered on the local machine use:

example% rpcinfo -p

To show all of the RPC services registered on the machine named **klaxon** use:

example% rpcinfo -p klaxon

To show all machines on the local net that are running the Yellow Pages service use:

example% rpcinfo -b ypserv 'version' | sort | uniq

where 'version' is the current Yellow Pages version obtained from the results of the **–p** switch above.

To delete the registration for version 1 of the **walld** service use:

example% rpcinfo -d walld 1

SEE ALSO

RPC Programming Guide, rpc(5), portmap(8C)

NOTES

rpcinfo is an optional product included with the NFS package; for more information, contact your CONVEX sales representative.

NAME

rquotad, rpc.rquotad – remote quota server

SYNOPSIS

/usr/etc/rpc.rquotad

DESCRIPTION

rquotad is an RPC server which returns quotas for a user of a local file system which is mounted by a remote machine over the NFS. The results are used by *quota*(1) to display user quotas for remote file systems. The *rquotad* daemon is normally invoked by *inetd*(8C).

FILES

quotas	quota file at the file system root

SEE ALSO

quota(1), rquota(3R), services(5), inetd(8C), rpc(3n), nfs(4)

NOTES

rquotad is an optional product included with the NFS package; for more information, contact your CONVEX sales representative.

NAME

rstatd, rpc.rstatd – kernel statistics server

SYNOPSIS

/usr/etc/rpc.rstatd

DESCRIPTION

rstatd is a server which returns performance statistics obtained from the kernel. The *rstatd* daemon is normally invoked by *inetd*(8C).

FILES

/vmunix	Kernel name list
/dev/mem	Kernel data values
*/lib/kernsyms/symdata_**	Kernel symbol addresses

SEE ALSO

services(5), rup(1C), rstat(3R), inetd(8C)

NOTES

rstatd is an optional product included with the NFS package; for more information, contact your CONVEX sales representative.

NAME

rusersd, rpc.rusersd – network rusers server

SYNOPSIS

/usr/etc/rpc.rusersd

DESCRIPTION

rusersd is a server that handles *rusers*(1C) requests. It returns information about the currently logged in users. The *rusersd* daemon is normally invoked by *inetd*(8C).

SEE ALSO

rusers(1C), rnusers(3R), services(5), inetd(8C)

NOTES

rusersd is an optional product included with the NFS package; for more information, contact your CONVEX sales representative.

NAME

rwalld, rpc.rwalld – network rwall server

SYNOPSIS

/usr/etc/rpc.rwalld

DESCRIPTION

rwalld is a server that handles *rwall*(1) and *shutdown*(8) requests. It is implemented by calling *wall*(1) to all the appropriate network machines. The *rwalld* daemon is normally invoked by *inetd*(8C).

SEE ALSO

rwall(1), wall(1), rwall(3R), services(5), inetd(8C), shutdown(8)

NOTES

The default behaviour of *rwalld* is to run as the super-user. This causes it to write on terminals with messages turned off (see *msgs*(1)), even if *rwall* is invoked by someone other than the super-user. The workaround for this is to edit the */etc/inetd.conf* file and change the user name under which the *rpc.rwalld* program is run. This is column 5 of the */etc/inetd.conf* file.

rwalld is an optional product included with the NFS package; for more information, contact your CONVEX sales representative.

NAME

setup_client – add or remove a diskless client to/from a server

SYNOPSIS

setup_client *op name yptype swapsize rootpath swappath dumppath homepath execpath sharepath arch*

DESCRIPTION

This script will install or remove the diskless client's environment on the server. Work done for installation includes making necessary directories, updating configuration files, and actual installation of the client's file space on the server. These items are removed if the client is being deleted.

ARGUMENTS

Running the command with no arguments given will list the available options.

op Operation type and is either **add** or **remove.**

name Hostname of the client being added or removed.

yptype Yellow pages mode of operation for the client, and is one of **master, slave, client,** or **none.**

swapsize

Size of the swap file to make for this client. This value is specified in bytes by default, or the letters **M** (x 1048576), **K** (x 1024), or **B** (x 512) may be appended to the value to multiply by the value given.

rootpath, swappath, dumppath

Full pathnames to be used for each of these functions (client root filesystem, swap file for client, and file to use for crash dumps). **none** may be specified for *dumppath.* In this case, the client kernel will use *swappath* when generating crash dumps. *rootpath* and *swappath* may also be specified as **none** for the **remove** operation. Any pathname so entered will not be removed during the operation.

homepath

Location of user home directories and is typically */home.* If another server is providing this service, then the path may have the servername and colon prepended, as in *servername:/home.* May also be specified as **none** if this feature is not used.

execpath

Full pathname of the location on the server where the /usr filesystem is located for this client's architecture type. Example value is */export/exec/sun3.*

sharepath

Full pathname of the location on the server where the shared /usr/share filesystem is located. Example values are */export/exec/share* or *none. setup_exec* takes no action with this value, but passes it along to be used by create_root(8).

arch Architecture name for this client. Typical values are **sun2, sun3,** and **sun4.**

DIAGNOSTICS

Error output should be self explanatory.

FILES

/tftpboot/*

SEE ALSO

INSTALL(8), sunboot(8s), create_root(8), fix_bootparams(8), exports(5)

BUGS

NOTES

setup_client is an optional product included with the NFS package; for more information, contact your CONVEX sales representative.

NAME

setup_exec – install diskless client software on a server

SYNOPSIS

setup_exec *arch execpath tapetype* [*tapehost*] *tapedev upgrade* [*sharepath*]

DESCRIPTION

This script loads software for a particular diskless client architecture from manufacturer's software distribution tapes. It is generally called by *INSTALL*, although it may be invoked manually to install a particular architecture type.

ARGUMENTS

Running the command with no arguments given will list the available options.

arch Architecture name to be loaded. For Sun distributions this value will be one of **sun2**, **sun3**, or **sun4**.

execpath

Full pathname of the location on the server to load the software. It is typically a value such as */export/exec/sun3*.

tapetype

Either **local** or **remote** depending on where the tape drive being used is located. If remote, then the *tapehost* argument will be taken to be the hostname of the machine to which the tape drive is attached. This hostname is verified via yellow pages (if domainname is not NULL or set to **noname**) or */etc/hosts*.

tapedev Tape driver type being used (such as mt, st, or ar), or else the full pathname of the non-rewinding raw tape drive.

upgrade

yes if an upgrade tape is to be applied to an existing installation, or **no** if a normal installation is to be done. This option must be **yes** to upgrade from SunOS 4.0 to SunOS 4.0.3.

sharepath

Full pathname of the location on the server to load that part of the software which can be shared among different architectures (typically the /usr/share tree). Example value is */export/exec/share*. Shared software can be disabled by specifying a value of **none** in this field.

DIAGNOSTICS

Error output should be self-explanatory.

FILES

/tmp/*.<architecture>

SEE ALSO

INSTALL(8), opt_software(8), extracting(8), verify_tapevol_arch(8), exports(5)

BUGS

There is a bug in some tape devices which randomly causes unexpected results during a seek to a tape file if the tape is not positioned at beginning of tape. To handle this case reliably requires that the tape be rewound after every file read and before seeking to the next file to be read. As can be expected, this is much slower than attempting to perform seeks from where the tape was last left.

Sun has changed tape formats for the Sun4 architecture used by the new SparcStations, and details of the new format are not yet available. It is very likely that setup_exec will be unable to handle a Sun4 release tape without changes; please contact the CONVEX Technical Assistance Center for the latest details.

The default action of *setup_exec* is to perform a rewind between every tape file read. There is an internal variable at the beginning of the script called **DOVERIFY** which may be changed to take advantage of midtape seeks. See the comments in the script to take advantage of this, but change it at your own risk.

NOTES

setup_exec is an optional product included with the NFS package; for more information, contact your CONVEX sales representative.

NAME

showmount – show all remote mounts

SYNOPSIS

/usr/etc/showmount [**–a**] [**–d**] [**–e**] [host]

DESCRIPTION

showmount lists all the clients that have remotely mounted a filesystem from *host*. This information is maintained by the *mountd*(8C) server on *host*, and is saved across crashes in the file */etc/rmtab*. The default value for *host* is the value returned by *hostname*(1).

OPTIONS

–d List directories that have been remotely mounted by clients.

–a Print all remote mounts in the format

hostname:directory

where *hostname* is the name of the client, and *directory* is the root of the file system that has been mounted.

–e Print the list of exported file systems.

SEE ALSO

rmtab(5), mountd(8C), exports(5)

BUGS

If a client crashes, its entry will not be removed from the list until it reboots and executes *umount –a*.

NOTES

showmount is an optional product included with the NFS package; for more information, contact your CONVEX sales representative.

NAME

spray – spray packets

SYNOPSIS

/usr/etc/spray [**–c count**] [**–d delay**] [**–i**] [**–l length**] **host**

DESCRIPTION

spray sends a one-way stream of packets to *host* using *rpc*, and then reports how many were received by *host* and what the transfer rate was. The host name can be either a name or an internet address.

OPTIONS

–c count

Specifies how many packets to send. The default value of *count* is the numbers of packets required to make the total stream size 100000 bytes.

–d delay

Specifies how may microseconds to pause between sending each packet. The default is 0.

–i Use ICMP echo packets rather than *rpc*. Since ICMP automatically echos, this creates a two way stream. You must be the super-user to use this option.

–l length

The *length* parameter is the numbers of bytes in the ethernet packet that holds the *rpc* call message. Since the data is encoded using *xdr*, and *xdr* only deals with 32 bit quantities, not all values of *length* are possible, and *spray* rounds up to the nearest possible value. When *length* is greater than 1514, then the *rpc* call can no longer be encapsulated in one ethernet packet, so the *length* field no longer has a simple correspondence to ethernet packet size. The default value of *length* is 86 bytes (the size of the *rpc* and UDP headers). The *length* must be between 86 and 8842.

SEE ALSO

icmp(4P), ping(8C), pong(8C), sprayd(8C)

NAME

sprayd, rpc.sprayd – spray server

SYNOPSIS

/usr/etc/rpc.sprayd

DESCRIPTION

sprayd is a server which records the packets sent by *spray*(8). The *sprayd* daemon is normally invoked by *inetd*(8C).

SEE ALSO

spray(3R), spray(8), inetd(8C)

NOTES

sprayd is an optional product included with the NFS package; for more information, contact your CONVEX sales representative.

NAME

sunboot, NETdisk – start the Sun system kernel, or a standalone program

SYNOPSIS

>b [*device* [(*c*, *u*, *p*)]] [*filename*] [**–a**] *boot-flags*
>b?
>b!

DESCRIPTION

The boot program is started by the PROM monitor and loads the kernel, or another executable program, into memory.

The form **b?** displays all boot devices and their device arguments.

The form **b!** boots, but does not perform a RESET.

USAGE

Booting Standalone

When booting standalone, the boot program (**/boot**) is brought in by the PROM from the file system. This program contains drivers for all devices.

Booting a Sun-3 System Over the Network

When booting over the network, the Sun-3 system PROM obtains a version of the boot program from a server using the Trivial File Transfer Protocol (TFTP). The client broadcasts a RARP request containing its Ethernet address. A server responds with the client's Internet address. The client then sends a TFTP request for its boot program to that server (or if that fails, it broadcasts the request). The filename requested (unqualified — not a pathname) is the hexadecimal, uppercase representation of the client's Internet address, for example:

Using IP Address 192.9.1.17 = C0090111

When the Sun server receives the request, it looks in the directory **/tftpboot** for *filename*. That file is typically a symbolic link to the client's boot program, normally **boot.sun3** in the same directory. The server invokes the TFTP server, **tftpd**(8C), to transfer the file to the client.

When the file is successfully read in by the client, the boot program jumps to the load-point and loads **vmunix** (or a standalone program). In order to do this, the boot program makes a broadcast RARP request to find the client's IP address, and then makes a second broadcast request to a **bootparamd**(8) bootparams daemon, for information necessary to boot the client. The bootparams daemon obtains this information either from a local **/etc/bootparams** database file, or from a Yellow Pages (YP) map. The boot program sends two requests to the bootparams daemon, the first, **whoami**, to obtain its hostname, and the second, **getfile**, to obtain the name of the client's server and the pathname of the client's root partition.

The boot program then performs a **mount**(8) operation to mount the client's root partition, after which it can read in and execute any program within that partition by pathname (including a symbolic link to another file within that same partition). Typically, it reads in the file **/vmunix**. If the program is not read in successfully, **boot** responds with a short diagnostic message.

Booting a Sun-2 or Sun-4 System Over the Network

Sun-2 and Sun-4 systems boot over the network in a similar fashion. However, the filename requested from a server must have a suffix that reflects the system architecture of the machine being booted. For these systems, the requested filename has the form:

ip-address.arch

where *ip-address* is the machine's Internet Protocol (IP) address in hex, and *arch* is a suffix representing its architecture. (Only Sun-3 systems may omit the *arch* suffix.) These filenames are restricted to 14 characters for compatibility with System V and other operating systems. Therefore, the architecture suffix is limited to 5 characters; it must be in upper case. At present, the

following suffixes are recognized: **SUN2** for Sun-2 system, **SUN3** for Sun-3 system, **SUN4** for Sun-4 system, and **PCNFS** for PC-NFS.

Note: a Sun-2 system boots from its server using one extra step. It broadcasts an ND request which is intercepted by the user-level **ndbootd**(8) server. This server sends back a standalone program that carries out the same TFTP request sequence as is done for all the other systems.

System Startup

Once the system is loaded and running, the kernel performs some internal housekeeping, configures its device drivers, and allocates its internal tables and buffers. The kernel then starts process number 1 to run **init**(8), which performs file system housekeeping, starts system daemons, initializes the system console, and begins multiuser operation. Some of these activities are omitted when **init** is invoked with certain *boot-flags*. These are typically entered as arguments to the boot command, and passed along by the kernel to **init**.

OPTIONS

device One of:

ie	Intel Ethernet
ec	3Com Ethernet
le	Lance Ethernet (Sun 3-50 system)
sd	SCSI disk
st	SCSI 1/4" tape
mt	Tape Master 9-track 1/2" tape
xt	Xylogics 1/2" tape
xy	Xylogics 440/450 disk

c Controller number, **0** if there is only one controller for the indicated type of device.

u Unit number, **0** if only there is only one driver.

filename Name of a standalone program in the selected partition, such as **stand/diag** or **vmunix**. Note: *filename* is relative to the root of the selected device and partition. It never begins with '/' (backslash). If *filename* is not given, the boot program uses a default value (normally **vmunix**). This is stored in the **vmunix** variable in the **boot** executable file supplied by Sun, but can be patched to indicate another standalone program loaded using **adb**(1).

–a Prompt interactively for the device and name of the file to boot. For more information on how to boot from a specific device, refer to **INSTALL(8).**

boot-flags The boot program passes all *boot-flags* to the kernel or standalone program. They are typically arguments to that program or, as with those listed below, arguments to progams that it invokes.

–b Pass the **–b** flag through the kernel to **init**(8) to skip execution of the **/etc/rc.local** script.

–h Halt after loading the system.

–s Pass the **–s** flag through the kernel to **init**(8) for single-user operation.

–i *initname*
Pass the **–i** *initname* to the kernel to tell it to run *initname* as the first program rather than the default **/sbin/init**.

FILES

/boot	standalone boot program
/tftpboot/????????	symbolic link to the boot program for a client
/tftpboot/boot.sun3	programs to boot from the client's root partition
/usr/etc/in.tftpd	TFTP server

/vmunix
/etc/bootparams

SEE ALSO

adb(1), **tftp**(1), **bootparamd**(8), **init**(8), **mount**(8), **ndbootd**(8C), **rc**(8), **reboot**(8), **tftpd**(8C), **INSTALL**(8)

BUGS

On the Sun-2 system, the PROM passes in the default name **vmunix**, overriding the boot program's patchable default.

NOTES

NETdisk is an optional product included with the NFS package; for more information, contact your CONVEX sales representative.

NAME

tftpd, in.tftpd – DARPA Trivial File Transfer Protocol server

SYNOPSIS

/usr/etc/in.tftpd [**–s**] [*homedir*]

DESCRIPTION

tftpd is a server that supports the DARPA Trivial File Transfer Protocol (TFTP). This server is normally started by **inetd**(8C) and operates at the port indicated in the **tftp** Internet service description in the **/etc/inetd.conf** file.

Before responding to a request, the server attempts to change its current directory to *homedir*; the default value is **/tftpboot**.

The use of *tftp* does not require an account or password on the remote system. Due to the lack of authentication information, *tftpd* will allow only publicly readable files to be accessed. Files may be written only if they already exist and are publicly writable. Note that this extends the concept of public to include all users on all hosts that can be reached through the network; this may not be appropriate on all systems, and its implications should be considered before enabling this service.

tftpd runs with the user ID and group ID set to **–2**, under the assumption that no files exist with that owner or group. However, nothing checks this assumption or enforces this restriction.

OPTIONS

–s Secure. When specified, the directory change must succeed; and the daemon also changes its root directory to *homedir*.

FILES

/tftpboot/* filenames are IP addresses

SEE ALSO

inetd(8C), **tftp**(1C)

Sollins, K.R., *The TFTP Protocol (Revision 2)*, RFC 783, Network Information Center, SRI International, Menlo Park, Calif., June 1981.

NOTES

tftpd is an optional product included with the NFS package; for more information, contact your CONVEX sales representative.

NAME

verify_tapevol_arch – verify mounted distribution tape is correct

SYNOPSIS

verify_tapevol_arch *arch volume tapedev* [*tapeserver*]

DESCRIPTION

verify_tapevol_arch is used to verify that the software distribution tape loaded is the proper one. It is called by *setup_exec* as part of the loading of diskless/NFS architecture support to verify that the requested tape is indeed loaded. If the mounted tape is not correct, then the script prompts the user to load the correct tape.

This script first rewinds the loaded tape. The second tape file (the tape's table of contents in XDR format) is then read and piped through **xdrtoc** to decode into ascii. This output is left in the file */tmp/TOC*. The tape's volume number and architecture type is determined from the file and compared to the requested values. If not correct, a message is output to the user to mount the correct tape.

ARGUMENTS

arch Architecture name of the desired distribution tape. It is typically one of the values **sun2, sun3,** or **sun4.**

volume Desired volume (tape) number of the distribution.

tapedev Full pathname of the non-rewinding raw tape device on which the tape is loaded. This device may be on a remote machine. If this is the case, then *tapeserver* is specified and is the hostname of the remote machine with the tape device.

FILES

/tmp/TOC

SEE ALSO

INSTALL(8), extracting(8), setup_exec(8), xdrtoc(8)

NOTES

verify_tapevol_arch is an optional product included with the NFS package; for more information, contact your CONVEX sales representative.

NAME

xdrtoc – convert distribution table of contents into ascii format

SYNOPSIS

xdrtoc [*file*]

DESCRIPTION

xdrtoc translates the XDR-format table of contents found on software distribution tapes into a human readable form. This table of contents information will only usually be found on vendor's software distribution tapes that support booting their machine using the NETdisk protocols.

This information is written to the standard output.

ARGUMENTS

If no arguments are given, then *xdrtoc* reads the XDR table of contents from its standard input. Otherwise, it reads from *file*.

FILES

/tmp/TOC

SEE ALSO

INSTALL(8), setup_exec(8), verify_tapevol_arch(8)

NOTES

xdrtoc is an optional product included with the NFS package; for more information, contact your CONVEX sales representative.

NAME

ypinit – build and install yellow pages database

SYNOPSIS

/usr/etc/yp/ypinit –m
/usr/etc/yp/ypinit –s *master_name*

DESCRIPTION

ypinit sets up a yellow pages database on a YP server. It can be used to set up a master or a slave server. You must be the superuser to run it. It asks a few, self-explanatory questions, and reports success or failure to the terminal.

It sets up a master server using the simple model in which that server is master to all maps in the data base. This is the way to bootstrap the YP system; later if you want you can change the association of maps to masters. All databases are built from scratch, either from information available to the program at runtime, or from the ASCII data base files in */etc*. These files are listed below under **FILES**. All such files should be in their "traditional" form, rather than the abbreviated form used on client machines.

A YP database on a slave server is set up by copying an existing database from a running server. The *master_name* argument should be the hostname of YP server (either the master server for all the maps, or a server on which the data base is up-to-date and stable).

Refer to *ypfiles*(5) and *ypserv*(8) for an overview of the yellow pages.

OPTIONS

–m Indicates that the local host is to be the YP master.

–s Set up a slave database.

FILES

/etc/passwd
/etc/pwrestrict
/etc/group
/etc/hosts
/etc/networks
/etc/services
/etc/protocols
/etc/netgroup
/etc/ethers

SEE ALSO

makedbm(8), ypfiles(5), yppush(8), ypxfr(8), ypmake(8), ypserv(8)

NOTES

ypinit is an optional product included with the NFS package; for more information, contact your CONVEX sales representative.

NAME

ypmake – rebuild yellow pages database

SYNOPSIS

cd /usr/etc/yp ; make [*map*]
cd /usr/etc/yp ; make clean

DESCRIPTION

The file called *Makefile* in */usr/etc/yp* is used by *make* to build the yellow pages database. With no arguments, *make* creates *dbm* databases for any YP maps that are out-of-date, and then executes *yppush* to notify slave databases that there has been a change.

If you supply a *map* on the command line, *make* will update that map only. Typing *make passwd* will create and *yppush* the password database (assuming it is out of date). Likewise, *make pwrestrict* will create and *yppush* the password restrictions database; *make hosts* and *make networks* will create and *yppush* the host and network files, */etc/hosts* and */etc/networks*.

There are three special variables used by *make*: DIR, which gives the directory of the source files; NOPUSH, which when non-null inhibits doing a *yppush* of the new database files; and DOM, used to construct a domain other than the master's default domain. The default for DIR is */etc*, and the default for NOPUSH is the null string.

Typing *make clean* causes all of the makefile timestamp files to be removed (*/usr/etc/yp/*.time*), so that a subsequent *make* will rebuild the entire database.

Refer to ypfiles(5) and ypserv(8) for an overview of the yellow pages.

SEE ALSO

make(1), makedbm(8), ypserv(8)

NOTES

ypmake is an optional product included with the NFS package; for more information, contact your CONVEX sales representative.

NAME

yppasswdd, rpc.yppasswdd – server for modifying yellow pages password and password restrictions files

SYNOPSIS

/usr/etc/rpc.yppasswdd *file1* [*file2*] [**–m** *arg1 arg2* ...]

DESCRIPTION

yppasswdd is a server that handles password change requests from *yppasswd*(1). It changes a password entry in *file1*, which is assumed to be in the format of *passwd*(5). If password restrictions are enabled, yppasswdd also changes the the corresponding restrictions entry in *file2*. *File2* is assumed to be in the format of *pwrestrict*(5).

Entries in *file1* and *file2* will only be changed if the password presented by *yppasswd*(1) matches the encrypted password of that entry.

If the **–m** option is given, then after *file1* and *file2* are modified, a *make*(1) will be performed in */usr/etc/yp*. Any arguments following the flag will be passed to *make*.

This server is not run by default, nor can it be started up from *inetd*(8C). If it is desired to enable remote password updating for the yellow pages, then an entry for *yppasswdd* should be put in the */etc/rc.local* file of the host serving as the master for the yellow pages *passwd* file.

EXAMPLES

If the yellow pages password file is stored as */etc/src/yp/passwd*, then to have all password changes propagated immediately, the server should be invoked as

```
/usr/etc/rpc.yppasswdd /etc/src/yp/passwd -m passwd DIR=/etc/src/yp
```

If password restrictions are to be maintained by the yellow pages password server, then the server should be invoked as

```
/usr/etc/rpc.yppasswdd /etc/passwd /etc/pwrestrict -m passwd pwrestrict DIR=/etc
```

In this example, */etc/passwd* is the yellow pages password file, and */etc/pwrestrict* is the yellow pages password restrictions file. Both the passwd, and pwrestrict maps will be rebuilt after every password change.

FILES

/usr/etc/yp/Makefile

SEE ALSO

yppasswd(1), passwd(5), pwrestrict(5), ypfiles(5), ypmake(8)

CAVEAT

This server will eventually be replaced with a more general service for modifying any map in the yellow pages.

NOTES

yppasswdd is an optional product included with the NFS package; for more information, contact your CONVEX sales representative.

NAME

yppoll – what version of a YP map is at a YP server host

SYNOPSIS

/etc/yp/yppoll [**-h** *host*] [**-d** *domain*] *mapname*

DESCRIPTION

yppoll asks a *ypserv* process what the order number is, and which host is the master YP server for the named map. If the server is a v.1 YP protocol server, *yppoll* uses the older protocol to communicate with it. In this case, it also uses the older diagnostic messages in case of failure.

OPTIONS

-h *host* Ask the *ypserv* process at *host* about the map parameters. If *host* isn't specified, the YP server for the local host is used. That is, the default host is the one returned by *ypwhich*(8).

-d *domain*
Use *domain* instead of the default domain.

SEE ALSO

ypserv(8), ypfiles(5), domainname(1)

NOTES

yppoll is an optional product included with the NFS package; for more information, contact your CONVEX sales representative.

NAME

yppush – force propagation of a changed YP map

SYNOPSIS

/etc/yp/yppush [**–d** *domain*] [**–v**] *mapname*

DESCRIPTION

yppush copies a new version of a Yellow Pages (YP) map from the master YP server to the slave YP servers. It is normally run only on the master YP server by the *Makefile* in */usr/etc/yp/* after the master databases are changed. It first constructs a list of YP server hosts by reading the YP map *ypservers* within the *domain*. Keys within the map *ypservers* are the ASCII names of the machines on which the YP servers run.

A "transfer map" request is sent to the YP server at each host, along with the information needed by the transfer agent (the program which actually moves the map) to call back the *yppush*. When the attempt has completed (successfully or not), and the transfer agent has sent *yppush* a status message, the results may be printed to stdout. Messages are also printed when a transfer is not possible; for instance when the request message is undeliverable, or when the timeout period on responses has expired.

Refer to *ypfiles*(5) and *ypserv*(8) for an overview of the yellow pages.

OPTIONS

–d Specify a *domain*.

–v Verbose. This causes messages to be printed when each server is called, and for each response. If this flag is omitted, only error messages are printed.

FILES

/etc/yp/*domainname*/ypservers.{dir, pag}

SEE ALSO

ypserv(8), ypxfr(8), ypfiles(5), domainname(1), YP protocol specification

BUGS

In the current implementation (version 2 YP protocol), the transfer agent is *ypxfr*, which is started by the *ypserv* program. If *yppush* detects that it is speaking to a version 1 YP protocol server, it uses the older protocol, sending a version 1 YPPROC_GET request and issues a message to that effect. Unfortunately, there is no way of knowing if or when the map transfer is performed for version 1 servers. *yppush* prints a message saying that an "old-style" message has been sent. The system administrator should later check to see that the transfer has actually taken place.

NOTES

yppush is an optional product included with the NFS package; for more information, contact your CONVEX sales representative.

NAME

ypserv, ypbind – yellow pages server and binder processes

SYNOPSIS

/usr/etc/ypserv
/usr/etc/ypbind

DESCRIPTION

The yellow pages (YP) provides a simple network lookup service consisting of databases and processes. The databases are *dbm*(3) files in a directory tree rooted at */usr/etc/yp*. These files are described in ypfiles(5). The processes are */usr/etc/ypserv*, the YP database lookup server, and */usr/etc/ypbind*, the YP binder. The programming interface to YP is described in ypclnt(3N). Administrative tools are described in yppush(8), ypxfr(8), yppoll(8), ypset(8), and ypwhich(1). Tools to see the contents of YP maps are described in ypcat(8), and ypmatch(1). Database generation and maintenance tools are described in ypinit(8), ypmake(8), and makedbm(8).

Both *ypserv* and *ypbind* are daemon processes typically activated at system startup time from */etc/rc.local*. *ypserv* runs only on YP server machines with a complete YP database. *ypbind* runs on all machines using YP services, both YP servers and clients.

The *ypserv* daemon's primary function is to look up information in its local database of YP maps. The operations performed by *ypserv* are defined for the implementor by the *YP protocol specification*, and for the programmer by the header file *<rpcsvc/yp_prot.h>*. Communication to and from *ypserv* is by means of RPC calls. Lookup functions are described in ypclnt(3N), and are supplied as C-callable functions in */lib/libc*. There are four lookup functions, all of which are performed on a specified map within some YP domain: *Match*, *Get_first*, *Get_next*, and *Get_all*. The *Match* operation takes a key, and returns the associated value. The *Get_first* operation returns the first key-value pair from the map, and *Get_next* can be used to enumerate the remainder. *Get_all* ships the entire map to the requester as the response to a single RPC request.

Two other functions supply information about the map, rather than map entries: *Get_order_number*, and *Get_master_name*. In fact, both order number and master name exist in the map as key-value pairs, but the server will not return either through the normal lookup functions. (If you examine the map with *makedbm*(8), however, they will be visible.) Other functions are used within the YP subsystem itself, and are not of general interest to YP clients. They include *Do_you_serve_this_domain?*, *Transfer_map*, and *Reinitialize_internal_state*.

The function of *ypbind* is to remember information that lets client processes on a single node communicate with some *ypserv* process. *ypbind* must run on every machine which has YP client processes; *ypserv* may or may not be running on the same node, but must be running somewhere on the network.

The information *ypbind* remembers is called a *binding* — the association of a domain name with the internet address of the YP server, and the port on that host at which the *ypserv* process is listening for service requests. This information is cached in the directory **/usr/etc/yp/binding** using a filename of **domainname.version**.

The process of binding is driven by client requests. As a request for an unbound domain comes in, the *ypbind* process broadcasts on the net trying to find a *ypserv* process that serves maps within that domain. Since the binding is established by broadcasting, there must be at least one *ypserv* process on every net. Once a domain is bound by a particular *ypbind*, that same binding is given to every client process on the node. The *ypbind* process on the local node or a remote node may be queried for the binding of a particular domain by using the *ypwhich*(1) command.

Bindings and rebindings are handled transparently by the C library routines. If *ypbind* is unable to speak to the *ypserv* process it's bound to, it marks the domain as unbound, tells the client process that the domain is unbound, and tries to bind the domain once again. Requests received for an unbound domain will wait until the domain requested is bound. In general, a bound domain is

marked as unbound when the node running *ypserv* crashes or gets overloaded. In such a case, *ypbind* will attempt to bind to any YP server (typically one that is less-heavily loaded) available on the net.

ypbind also accepts requests to set its binding for a particular domain. The request is usually generated by the YP subsystem itself. *ypset*(8) is a command to access the *Set_domain* facility. It is for unsnarling messes, not for casual use. Note that the *Set Domain* procedure only accepts requests from processes running as root.

FILES

If the file */usr/etc/yp/ypserv.log* exists when *ypserv* starts up, log information will be written to this file when error conditions arise.

The file(s) **/usr/etc/yp/binding/domainname.version** will be created to speed up the binding process. These files cache the last successful binding created for the given domain, when a binding is requested these files are checked for validity and then used.

SEE ALSO

ypclnt(3N), ypfiles(5), ypcat(8), ypmatch(8), yppush(8), ypwhich(1), ypxfr(8), ypset(8), YP protocol specification

NOTES

Both **ypbind** and **ypserv** support multiple domains. The **ypserv** process determines the domains it serves by looking for directories of the same name in the directory **/usr/etc/yp**. It will reply to all broadcasts requesting yp service for that domain. Additionally, the **ypbind** process can maintain bindings to several domains and their servers; the default domain however, is the one specified by the **domainname**(1) command at startup time.

ypserv is an optional product included with the NFS package; for more information, contact your CONVEX sales representative.

NAME

ypset – point ypbind at a particular server

SYNOPSIS

/etc/yp/ypset [**–V1**|**–V2**] [**-h** *host*] [**-d** *domain*] *server*

DESCRIPTION

ypset tells *ypbind* to get YP services for the specified *domain* from the *ypserv* process running on *server*. If *server* is down, or isn't running *ypserv*, this is not discovered until a YP client process tries to get a binding for the domain. At this point, the binding set by *ypset* will be tested by *ypbind*. If the binding is invalid, *ypbind* will attempt to rebind for the same domain.

ypset is useful for binding a client node which is not on a broadcast net, or is on a broadcast net which isn't running a YP server host. It also is useful for debugging YP client applications, for instance where a YP map only exists at a single YP server host.

In cases where several hosts on the local net are supplying YP services, it is possible for *ypbind* to rebind to another host even while you attempt to find out if the *ypset* operation succeeded. That is, you can type "ypset host1", and then "ypwhich", which replies: "host2", which can be confusing. This is a function of the YP subsystem's attempt to load-balance among the available YP servers, and occurs when *host1* does not respond to *ypbind* because it is not running *ypserv* (or is overloaded), and *host2*, running *ypserv*, gets the binding.

server indicates the YP server to bind to, and can be specified as a name or an IP address. If specified as a name, *ypset* will attempt to use YP services to resolve the name to an IP address. This will work only if the node has a current valid binding for the domain in question. In most cases, *server* should be specified as an IP address.

Refer to *ypfiles*(5) and *ypserv*(8) for an overview of the yellow pages.

OPTIONS

–V1 Bind *server* for the (old) v.1 YP protocol.

–V2 Bind *server* for the (current) v.2 YP protocol.

If no version is supplied, *ypset*, first attempts to set the domain for the (current) v.2 protocol. If this attempt fails, *ypset*, then attempts to set the domain for the (old) v.1 protocol.

–h *host* Set *ypbind*'s binding on *host*, instead of locally. *host* can be specified as a name or as an IP address.

–d *domain* Use *domain*, instead of the default domain.

SEE ALSO

ypwhich(1), ypserv(8), ypfiles(5), domainname(1)

NOTES

ypset is an optional product included with the NFS package; for more information, contact your CONVEX sales representative.

NAME

ypupdated, rpc.ypupdated – server for changing YP information

SYNOPSIS

rpc.ypupdated [**–is**]

DESCRIPTION

ypupdated is a daemon that updates information in the Yellow Pages, normally started up by **inetd**(8C). **ypupdated** consults the file **updaters**(5) in the directory **/usr/etc/yp** to determine which YP maps should be updated and how to change them.

By default, the daemon requires the most secure method of authentication available to it, either DES (secure) or UNIX (insecure).

OPTIONS

–s accept only calls authenticated using the secure RPC mechanism (AUTH_DES authentication). This disables programmatic updating of YP maps unless the network supports these calls.

–i also accept RPC calls with the insecure AUTH_UNIX credentials. This allows programmatic updating of YP maps in all networks.

FILES

/usr/etc/yp/updaters

SEE ALSO

updaters(5), **inetd(8C)**, **keyserv**(8C)

NOTES

ypupdated is an optional product included with the NFS package for customers in the U.S. and Canada only. For more information, contact your CONVEX sales representative.

NAME

ypxfr, ypxfr_1perhour, ypxfr_2perday, ypxfr_1perday – transfer a YP map from some YP server to here

SYNOPSIS

/usr/etc/yp/ypxfr [**–f**] [**–c**] [**-d** *domain*] [**-h** *host*] [**-s** *domain*] [**-C** *tid prog ipadd port*] [**–S**] *mapname*

DESCRIPTION

ypxfr moves a YP map to the local host by making use of normal YP services. It creates a temporary map in the directory */usr/etc/yp/domain* (which must already exist), fills it by enumerating the map's entries, fetches the map parameters (master and order number) and loads them. It then deletes any old versions of the map and moves the temporary map to the real mapname.

If *ypxfr* is run interactively, it writes its output to the terminal. However, if it's invoked without a controlling terminal, and if the log file */usr/etc/yp/ypxfr.log* exists, it will append all its output to that file. Since *ypxfr* is most often run from */usr/lib/crontab* , or by *ypserv* , you can use the log file to retain a record of what was attempted and what the results were.

For consistency between servers, *ypxfr* should be run periodically for every map in the YP data base. Different maps change at different rates: the *services.byname* map may not change for months at a time, for instance, and may therefore be checked only once a day in the wee hours. You may know that *mail.aliases* or *hosts.byname* changes several times per day. In such a case, you may want to check hourly for updates. A *crontab*(5) entry can be used to perform periodic updates automatically. Rather than having a separate *crontab* entry for each map, you can group commands to update several maps in a shell script. Examples (mnemonically named) are in */usr/etc/yp*: *ypxfr_1perday*, *ypxfr_2perday*, and *ypxfr_1perhour*. They can serve as reasonable first cuts.

Refer to *ypfiles*(5) and *ypserv*(8) for an overview of the yellow pages.

OPTIONS

–f Force the transfer to occur even if the version at the master is not more recent than the local version.

–c Don't send a "Clear current map" request to the local *ypserv* process. Use this flag if *ypserv* is not running locally at the time you are running *ypxfr*. Otherwise, *ypxfr* will complain that it can't talk to the local *ypserv*, and the transfer will fail.

–d *domain* Specify a domain other than the default domain.

–h *host* Get the map from *host* , regardless of what the map says the master is. If *host* is not specified, *ypxfr* will ask the YP service for the name of the master, and try to get the map from there. *host* may be a name or an internet address in the form *a.b.c.d.*

–s *domain* Specify a source domain from which to transfer a map that should be the same across domains (such as the *services.byname* map).

–C *tid prog ipadd port*

This option is **only** for use by *ypserv*. When *ypserv* invokes *ypxfr*, it specifies that *ypxfr* should call back a *yppush* process at the host with IP address *ipaddr*, registered as program number *prog*, listening on port *port*, and waiting for a response to transaction *tid*. **–S** This option causes **ypxfr** to require that the **ypserv** server, from which it will obtain the maps to be transferred, is using "privileged" IP ports. Since only super-user processes are typically allowed to use privileged ports, this feature adds an extra measure of security to the transfer. If the map being transferred is a secure map, **ypxfr** sets the permissions on the map to 0600.

FILES

/usr/etc/yp/ypxfr.log, /usr/etc/yp/ypxfr_1perday, /usr/etc/yp/ypxfr_2perday, /usr/etc/yp/ypxfr_1perhour, /usr/lib/crontab

SEE ALSO

ypfiles(5), ypserv(8), yppush(8), domainname(1), cron(8), YP protocol spec

NOTES

ypxfr is an optional product included with the NFS package; for more information, contact your CONVEX sales representative.