Home

Using Network File System/Remote Procedure Call

Version 4.7

RadiSys.
THE POWER OF WE

www.radisys.com
Revision A • July 2006

Contents

Overview 7

Remote Procedure Call Overview 8

Network File System Overview 8

Remote Procedure Call 9

External Data Representation 9

Stateless Servers 9

NFS Protocol Definition 9

File System Model 9

NFS/RPC Utilities and Daemon Server Programs 11

NFS/RPC Utilities 12

NFS/RPC Daemon Server Programs 13

exportfs 14

mount 15

mountd 16

nfsd 17

nfsstat 18

pcnfsd 19

portmap 20

rpcdbgen 21

rpcdump 23

rpcgen 24

rpchost 26

rpcinfo 27

rstatd 28

rup 29

rusers 30

rusersd 31

showmount 32

Remote Procedure Calls 33

The RPC Protocol 34

RPC Protocol Requirements 34

The RPC Language 34

The RPC Language Specification 35

Syntax Notes 35

Transport Independence 35

RPC Semantics 36

Binding 36

Programs and Procedures 36

RPC's Three Layers 37

The Highest Layer 38

The Middle Layer 39

Lowest Layer of RPC 46

Using RPC's Lowest Layer on the Server Side 46
Memory Allocation with XDR and the Lower Layer 49
The Calling Side of the Lower Layer 50
Other RPC Features 52
Select on the Server Side 52
Broadcast RPC 53
Broadcast RPC Synopsis 53
Batching 54
The RPC Message Protocol 57
Record Marking Standard 60
Authentication 60
Examples 64
Callback Procedures 64

Port Mapper 69

Introduction 70
Port Mapper Operation 70
Port Mapper Protocol Specification
(in RPC Language) 71

External Data Representation 73

Introduction 74
The XDR Library 74
Writer 74
Reader 75
Explaining Writer/Reader Examples 75
Serializing and Deserializing Data 76
XDR Library Primitives 77
Number Filters 77
Floating Point Filters 78
Enumeration Filters 78
No Data 79
Constructed Data Type Filters 79
Strings 79
Byte Arrays 80
Arrays 80
Implementing Arrays Example A 81
Implementing Arrays Example B 82
Implementing Arrays Example C 82
Opaque Data 83
Fixed-Sized Arrays 83
Discriminated Unions 84
Discriminated Union Example 84
Pointers 85
Pointer Example 85
Non-filter Primitives 86
XDR Operation Directions 87
XDR Stream Access 87
Standard I/O Streams 87
Memory Streams 87
Record (TCP/IP) Streams 88

XDR Stream Implementation 89
The XDR Object 89
Linked Lists 90
XDR Data Types 94
Basic Block Size 94
Integer 94
Unsigned Integer 95
Enumeration 95
Boolean 95
Hyper Integer and Unsigned Hyper Integer 95
Floating-Point 96
Double-Precision Floating-Point 96
Fixed-Length Opaque Data 97
Variable-Length Opaque Data 98
String 98
Fixed-Length Array 99
Variable-Length Array 99
Structures 100
Discriminated Union 100
Void 101
Constant 101
Typedef 101
Optional-Data 102
The XDR Language Specification 103
Notational Conventions 103
Lexical Notes 103
Syntax Information 104
Syntax Notes 105
An Example of an XDR Data Description 105
RPCGEN Programming Guide 109
An Overview of rpcgen 110
Converting Local Procedures to Remote Procedures 110
Generating XDR Routines 114
The C Preprocessor 117
RPC Language 119
Definitions 119
Structures 119
Unions 120
Enumerations 120
Typedef 121
Constants 121
Programs 121
Declarations 122
Special Cases 123
Booleans 123
Strings 123
Opaque Data 124
Voids 124
Getting Started With Network File System/Remote Procedure Call 125

Introduction ... 126
System Components ... 126
System Architecture ... 126
Configuring the NFS Client ... 127
Directory Structure ... 127
Configuration Overview ... 127
Step 1: Configure Group and User ID Mapping File for NFS ... 127
Step 2: Build the RPC Data Base Module ... 128
Step 3: Configure the Startup Procedure ... 129
Step 4: Verify the Installation ... 132
Configuring NFS Server and RPC Development System ... 133
Directory Structure ... 133
Configuration Overview ... 133
Step 1: Configure Group and User ID Mapping Files for NFS ... 134
Step 2: Build the RPC Data Base Module ... 134
Step 3: Configure the Startup Procedures ... 135
Step 4: Export Local File Systems ... 138
Step 5: Verify the Installation ... 138

Index

1 Overview

This chapter provides a general overview of Network File System/Remote Procedure Call (NFS/RPC). It provides a brief description of the NFS/RPC components and lists the utilities and server programs provided with this package.

Network File System (NFS) is a software component that provides transparent file access for client applications across local area networks.

Function calls in the RPC C Library and External Data Representation (XDR) C Library are described in the ***OS-9® Networking Programming Reference Manual***.

Remote Procedure Call Overview

Remote Procedure Call (RPC) is a protocol for writing distributed network applications. Each system on the network can provide remote procedures to any number of servers, which can be dynamically called by client programs on other systems. RPC can:

- initiate remote execution of a program
- return system statistics
- look up network processes
- access and control remote file systems.

RPC applications are easy to implement because they avoid low-level primitives such as sockets. They can be written and tested on stand-alone systems. You can later split these applications into client and server procedures.

You call remote procedures in the same manner as a local C language function. When you call remote procedures, RPC sets up a client to server communications link and sends a data packet to the server. When the packet arrives, the server performs the following steps:

1. Calls a dispatch routine
2. Performs service as requested
3. Sends back a reply
4. Returns the procedure call to the client.

RPC can handle arbitrary data structures regardless of different systems' byte orders or structure layout conventions. It does so by converting data to a network standard called eXternal Data Representation (XDR).

RPC provides authentication parameters so that servers, such as NFS, can validate access to remote data. It provides additional hooks for service-specific security.

For more information about XDR, refer to Chapter 5 External Data Representation. For more information about Authentication, refer to .

RPC includes the `rpcgen` compiler, a preprocessor for writing RPC applications. It accepts a remote procedure interface definition and produces C language output including stubs for client, server, and XDR filter routines.

Network File System Overview

The Network File System (NFS) protocol provides transparent remote access to shared file systems over local area networks. Like RPC and XDR, the NFS protocol design is machine, operating system, network architecture, and transport protocol independent.

The mount protocol, in conjunction with the `exportfs` utility, enables the server to restrict the set of client machines that are allowed to access each exported file system. Once mounted, the NFS client and server components convert operating system specific functions into NFS protocol functions. This enables, for example, an OS-9 system to view and access a remote file system as if it were a local OS-9 device.

Remote Procedure Call

OS-9's remote procedure call specification provides a procedure-oriented interface to remote services. Each server supplies a program that is a set of procedures. The combination of host address, program number, and procedure number specifies one remote service procedure. RPC does not depend on services provided by specific protocols. This allows you to use RPC with any underlying transport protocol.

External Data Representation

The eXternal Data Representation (XDR) standard provides a common method of representing a set of data types over a network. OS-9 provides implementations of XDR and RPC.

Stateless Servers

NFS is a stateless server. A stateless server does not need to maintain any extra information about its clients to function correctly. A stateful server maintains this extra information. This distinction is important in the event of a failure. When a stateless server does not respond, a client need only retry a request until the server responds. This enables the client to operate normally when a server is temporarily unavailable.

The client of a stateful server needs to do one of the following:

- Detect a server crash and rebuild its state when it comes back up.
- Cause the client operations to fail.

NFS Protocol Definition

Servers and protocols may change over time. RPC provides a version number with each request. This manual describes version two of the NFS protocol.

File System Model

NFS assumes a hierarchical file system. Each entry in a directory (file, directory, device, etc.) has a string name. Different operating systems may have restrictions on the depth of the tree or the names used. They may also use different syntax to represent the pathname. Therefore, a pathname is the concatenation of all directory and file names in the path. A file system is a tree on a single server, usually a single disk or physical partition, with a specified root. Some operating systems provide a mount operation to make all file systems appear as a single tree, while other systems maintain a "forest" of file systems. Files are unstructured streams of uninterpreted bytes.

NFS looks up one component of a pathname at a time. It does this because pathnames need separators between the directory components, and because different operating systems may not use the same separators.

Although files and directories are similar, different procedures read directories and files. This provides a network standard format for representing directories.

Like other RPC services, NFS includes a client and server side.

The NFS Server Side

The NFS server on OS-9 is implemented as a daemon (`nfsd`) program which handles I/O requests through RBF on behalf of the remote client. This allows a user on a Sun workstation, for example, to mount and manipulate an OS-9 file system with commands such as:

```
mount delta:/h0 /mnt  /* mount the remote OS-9/OS-9000 file system */
ls /mnt               /* display directory of the OS-9/OS-9000 /h0 device */
cat /mnt/sys/motd      /* display the OS-9/OS-9000 motd file */
```

The following daemons must be running on OS-9:

- `portmap`
- `mountd`
- `nfsd`

The following modules must be loaded on OS-9:

- `rpcdb`
- `exportfs` (must be run to export the local hard disk)

The NFS Client Side

The client side of NFS is implemented as a child thread, `nfsc`, and file manager `nfs`. NFS operates with file constructs and network protocols and presents the remote file system as if it were a local OS-9 file system. This allows an OS-9 user to mount and manipulate remote file systems. For example, you can use commands such as:

```
mount mwca:/usr /nf           /* mount remote file system to device nf */
dir /nf                       /* display directory for remote file system */
list /nf/hello.c              /* display a C source file */
copy /nf/hello.c /h0/hello.c  /* copy remote file onto local device */
```

The following modules must be loaded to act as a NFS client:

- `nfs`
- `nfsnul`
- `nfs_devices`
- `rpcdb`
- `mount`
- `nfsc`

2 NFS/RPC Utilities and Daemon Server Programs

This chapter describes the NFS/RPC utilities, client and server programs.

NFS/RPC Utilities

Table 2-1 on page 12 lists the utilities provided with the NFS/RPC software. The utilities are described further on the following pages.

Utilities can be found in `MWOS/<OS>/<CPU>/CMDS`.

Table 2-1. NFS/RPC Utilities

Utility	Description
exportfs	Generate Exports List Specifies which devices and directories can be remotely mounted.
mount	Mount/Unmount Remote File System Mounts a remote file system and makes it available to OS-9/OS-9000 as a local device managed by NFSRBF.
nfsstat	Display RPC and NFS Statistics Accesses the statistics for RPC and NFS clients and servers and displays the information.
rpcdbgen	Generate RPC Database Creates the RPC database module and specifies the optional backup/recovery directory and Group/User ID mapping for NFS.
rpcdump	Display RPC Database Reads the RPC database module and displays its contents.
rpcgen	Generate C Code for RPC Protocols Generates client and server programs from an RPC interface definition.
rpchost	Display Hostname Returns the name of the host as defined in the `inetdb` configuration files.
rpcinfo	Display RPC Information Requests information from an RPC server and displays the results.
rup	Display Status of Remote System Displays a system status for the host.
rusers	Display Network Users Information Displays a list of users logged into a remote system.
showmount	Display Remote Mounts Displays which remote systems are currently mounted to the OS-9 NFS file server.

Table 2-2. NFS Client System Modules

System Module	Description
nfs	NFS File Manager
nfs_devices	NFS Device Descriptor
nfsc	NFS Client Auxiliary Process
nfsnul	NFS Device Driver

See the daemons `mountd`, `nfsd`, and `portmap` for more information about NFS server support.

NFS/RPC Daemon Server Programs

Table 2-3 on page 13 lists the NFS/RPC daemon server programs. The server programs are described further on the following pages.

Table 2-3. NFS/RPC Daemon Server Programs

Daemon	Description
mountd	NFS Mount Server Answers file system `mount` requests and determines which file systems are available to which machines and users.
nfsd	NFS Protocol Server Responds to low-level I/O requests through NFS.
pcnfsd	PCNFS login daemon Runs on a NFS server system to service PC-NFS client authentication.
portmap	Port to RPC Program Number Mapper Provides a mapping between RPC programs and ports.
rstatd	Remote System Statistics Server Returns performance statistics obtained from the kernel.
rusersd	Remote Network Users Server Provides a list of users logged into the local host.

exportfs

Generate Exports List in NFS Database Module

Syntax

```
exportfs [<opts>]{<dev> {-a <machine_list>}}
```

Description

`exportfs` indicates to the NFS server system which devices can be mounted by remote hosts. If invoked without parameters, `exportfs` displays the current exports list.

`exportfs` generates a data module containing the exports list, and active mount table. This allows NFS to operate in a ROM-based (diskless) environment. If a backup/recovery directory is specified with `rpcdbgen`, all of this `exportfs` information is saved across a boot or system reset of the operating system.

A particular device may be exported to a restricted set of machines using the -a option. For example,

```
exportfs /h0 -a blue,green.xyz.com
```

will allow the machines `blue` and `green.xyz.com` to mount the local `/h0` device. Multiple devices and access lists may be specifed on a single command line. For example,

```
exportfs /h0 -a blue /r0 /h1 -a blue, red
```

will export `/h0` to only the machine `blue`, the `/r0` device to all users, and the `/h1` device to only the machines `blue` and `red`.

The machine names specified in an access list must match that returned from `gethostbyaddr()`. Normally this includes a fully qualified domain name if it is resolved using DNS, and just a machine name if it is resolved locally in your `inetdb`.

Options

`-?`	Display the help message.
`<dev>`	Add the device to the **exports** list.
`-a <machine_list>`	Restrict access to a mount point to a specific list of machines. Multiple machines may be listed separated by commas and containing no spaces.
`-s`	Create a new mount table for exported systems.

- OS-9 exports should be at the device level (`/h0` or `/d0`, not `/h0/cmds`).
- File systems up to the RBF maximum of 4 gigabytes are supported.

mount

Mount and Dismount NFS File System

Syntax

```
mount [<opts>] {[<host>:/<path>][/<dev>][<opts>]}
```

Description

`mount` indicates to the system that a file system is to be associated with local device `<dev>` and accessed via NFS. You can also use it to display the current mounted device status. The unmount option, `-u`, indicates that a file system is to be dismouted and no longer accessed.

- The default read and write block size is 8K. A smaller block size can be specified with the `-r` and `-w` options.
- OS-9 NFS clients can only mount OS-9 NFS servers at the device level (`/n0` or `/d0`, not `/n0 /CMDS`).

Options

`-?`	Display the help message.
`-d`	Display the currently mounted devices.
`-m`	Use group/user ID mapping for this mount.
`-r=<rsize>`	Use read block size of `<rsize>`.
`-w=<wsize>`	Use write block size of `<wsize>`.
`-u`	Unmount a specified file system.

Use the `nfs.map` file to specify group/user ID mapping between the OS-9 NFS client system and the remote file server. Refer to Appendix A Getting Started With Network File System/Remote Procedure Call for more information.

mountd

NFS Mount Request Server

Syntax

```
mountd [<opts>]
```

Description

The `mountd` daemon answers file system mount requests. It determines which file systems are available to which machines and users. `mountd` also stores information as to which clients have file systems mounted. You can also use the `showmount` command to display this information. The `mount` client command calls `mountd` to mount the file system.

To place `mountd` in the background, end the command line with an ampersand (`&`). For example, `mountd&`.

On OS-9 NFS server systems, NFS clients can only mount at the device level (`/n0` or `/d0`, not `/h0 /CMDS`).

Options

`-?`	Display the help message.

nfsd

NFS Protocol Server

Syntax

```
nfsd [<opts>]
```

Description

The `nfsd` daemon responds to low-level I/O requests through NFS.

To place `nfsd` in the background, end the command line with an ampersand (`&`). For example, `nfsd&`.

Options

`-?`	Display the help message.

Use the `nfsd.map` file to specify group/user ID mapping between the remote client system and the local OS-9 file server. Refer to Appendix A Getting Started With Network File System/Remote Procedure Call for more information.

nfsstat

Display RPC and NFS Statistics

Syntax

```
nfsstat [<opts>]
```

Description

`nfsstat` displays statistics about NFS and RPC. You can also use `nfsstat` to re-initialize this information. `nfsstat` produces a report similar to that shown in Figure 2-1. Sample `nfsstat` Report.

The `rpcdb` data module must be created with `rpcdgen -s` option in order to turn on statistics

Figure 2-1. Sample `nfsstat` Report

```
Diag:nfsstat
Server rpc:
calls    badcalls nullrecv badlen   xdrcall
0        0        0        0        0
Server nfs:
calls    badcalls
1294     0
null     getattr  setattr  root     lookup   readlink read
0    0%  1    0%  1    0%  0    0%  1196 92%  0    0%  0    0%
wrcache  write    create   remove   rename   link     symlink
0    0%  1    0%  2    0%  1    0%  2    0%  0    0%  0    0%
mkdir    rmdir    readdir  fsstat
1    0%  1    0%  87   6%  1    0%
Client rpc:
calls    badcalls retrans  badxid   timeout  wait     newcred
```

Options

`-?`	Display the help message.
`-r`	Re-initialize RPC and NFS statistics.

pcnfsd
PC NFS Login Daemon

Syntax

```
pcnfsd [<opts>]
```

Description

`pcnfsd` runs on an NFS server system to service PC-NFS client authentication.

Options

`-?`	Display the help message.

portmap
DARPA Port to RPC Program Mapper

Syntax

```
portmap [<opts>]
```

Description

The `portmap` daemon converts RPC program numbers into DARPA protocol port numbers. `portmap` must be running in order to run other RPC servers.

When an RPC server is started, the server tells `portmap` what port number it is listening to and what RPC program numbers the RPC server is prepared to serve.

When a client wishes to make an RPC call to a given program number, it contacts `portmap` on the server machine to determine the port number where RPC packets should be sent.

To place `portmap` in the background, end the command line with an ampersand (`&`). For example, `portmap&`.

If `portmap` is restarted, all servers must be restarted.

Options

`-?`	Display the help message.

rpcdbgen

Generate NFS/RPC Database Module

Syntax

```
rpcdbgen [<opts>]
```

Description

`rpcdbgen` generates an OS-9 data module from host information supplied in the `rpcdbgen` call. This allows RPC to operate in a ROM-based (diskless) environment.

`rpcdbgen` also processes group ID and user ID map files for NFS. This allows users with different group/user IDs between hosts to transparently access their files. There are two mapping files:

`nfs.map`This is a mapping file for the NFS client. It maps local OS-9 group and user IDs to remote group and user IDs.

`nfsd.map`This mapping file is used by the NFS server (`nfsd`) to map remote requests to local group/user IDs.

Any time you make a change to `nfs.map` or `nfsd.map`, use `rpcdbgen` to generate a new data module.

The idbgen utility reads the RPC database file `(MWOS/SRC/ETC/rpc)` and places the contents in the Internet data module.

Options

`-?`	Display the help message.
`-c[=<file>]`	Specify NFS client map file. Default is `nfs.map`.
`-d[=<file>]`	Specify NFS server map file. Default is `nfsd.map`.
`-r=<dir>`	Specify NFS server backup/recovery directory.
`-s`	Specify NFS/RPC to collect internal statistics.
`-w=<str>`	Directory for NFS/RPC database files.
`-n=<num>`	Set module revision to `<num>`.
`-o[=<path>]`	Specify the name of the file if different from module.
`-x`	Place module in execution directory.
`-to[=]<name>`	Specify target operating system.

Target operating systems are shown in Table 2-4.

Table 2-4. Target Operating Systems

<name>	Target Operating System
OSK	OS-9 for 68K
OS9000 or OS9K	OS-9

`-tp[=]<name>`Specify target processor and options.

Target processors are listed in Table 2-5.

Table 2-5. Target Processors

<name>	Target Processor(s)
68K or 68000	Motorola 68000/68010/68070
CPU32	Motorola 683000 family
020 or 68020	Motorola 68020/68030/68040
040 or 68040	Motorola 68040
386 or 80386	Intel 80386/80486/Pentium™
PPC	Generic PowerPC™ Processor
403	PPC 403
601	MPC 601
603	MPC 603
ARM	Generic ARM^{TM} Processor
ARMV3	ARMV3 Processor
ARMV4	ARMV4 Processor

`-z[[=]<file>]` Read additional command line arguments from `<file>`.

rpcdump

Display RPC Database Module

Syntax

```
rpcdump [<opts>]
```

Description

`rpcdump` displays information in the RPC database module `rpcdb`. It produces a report similar to the output shown below.

```
Diag:rpcdump
Dump of NFS/RPC data module [rpcdb]
   recovery dir: MWOS/SRC/ETC
   collect stats: yes
   use nfs client map: yes
   use nfsd server map: yes
NFS Client Mapping
   default client uid: 99
   default client group: 12
      OS-9 uid NFS uid
```

By default, `rpcdump` looks for the `rpcdb` module from disk. Use the `-m` option to dump it from memory.

Options

`-?`	Display the help message.
`-m`	Dump the `rpcdb` module in memory. Default is from a file.

rpcgen
Generate C Programs to Implement RPC Protocol

Syntax

```
rpcgen [<opts>] <inpath>
```

Description

`rpcgen` generates source code to implement an RPC application. `rpcgen` greatly simplifies the development process by producing C language source code for client, server, and XDR filter routines. You can compile and link these to produce the distributed RPC application.

The input to `rpcgen` is the RPC language. In the development cycle, `rpcgen` takes an input file and generates four output files. If the input file is named `proto.x`, `rpcgen` generates the following:

- A header file in `proto.h`
- XDR routines in `proto_xdr.c`
- Server-side stubs in `proto_svc.c`
- Client-side stubs in `proto_clnt.c`

The OS-9 C preprocessor (`cpp`) is run on all input files before the files are actually interpreted by `rpcgen`. This ensures that all `cpp` directives are legal within an `rpcgen` input file. For each type of output file, `rpcgen` defines a special `cpp` symbol for you to use as shown in Table 2-6. `CPP` Symbols.

Table 2-6. `CPP` Symbols

Name	Defined When Compiling Into
RPC_HDR	Header files.
RPC_XDR	XDR routines.
RPC_SVC	Server-side stubs.
RPC_CLNT	Client-side stubs.

Any line beginning with an apostrophe (’) passes directly into the output file. It is not interpreted by `rpcgen`.

Options

`-?`	Display the help message.
`-c`	Compile into XDR routines.
`-h`	Compile into C data definitions (a header file).

`-k`	Use the K&R C preprocessor (`cpp`)
`-l`	Compile into client-side stubs.
`-m`	Compile into server-side stubs, but do not produce a `main()` routine.
`-o <path>`	Specify the name of the output file. If not specified, standard output is assumed.
`-s <tr>`	Compiles into server-side stubs using the given transport, TCP (Transmission Control Protocol) or UDP (User Datagram Protocol). You can invoke this option more than once to compile a server that serves multiple transports.

rpchost
Display Hostname

Syntax

```
rpchost [<opts>]
```

Description

`rpchost` displays the hostname as defined in the `inetdb` configuration files.

Options

-?	Display the help message.

rpcinfo

Display RPC Information

Syntax

```
rpcinfo [<opts>]
```

Description

`rpcinfo` makes a call to an RPC server (`portmap`) and reports the results. The program parameter `<prognum>` can either be a name or a number. If you specify a version, `rpcinfo` attempts to call that version of the specified program. Otherwise, `rpcinfo` attempts to find all registered version numbers for the specified program.

Options

`-?`	Displays the help message.
`-p[<host>]`	Call `portmap` on `<host>` and display list of registered programs. If `<host>` is not specified, it defaults to the value returned by `gethostname()`.
`-t <host> <prognum>[<version>]`	Makes an RPC call to procedure 0 of `<program>` on `<host>` using TCP and reports all versions that are available. If `<version>` is specified, report only if the specific version is available.
`-u <host> <prognum>[<version>]`	The same as the `-t` option only use UDP instead of TCP when querying `<host>`.
`-n <port>`	Use `<port>` as the port number for the `-t` and `-u` options. It must be placed before the `-t` or `-u` option.

rstatd
Remote Systems Statistics Server

Syntax

```
rstatd [<opts>]
```

Description

The `rstatd` RPC daemon returns statistics obtained from the kernel. You can use the `rup` RPC client program to call `rstatd`.

To place `rstatd` in the background, end the command line with an ampersand (`&`). For example, `rstatd&`.

Options

`-?`	Display the help message.

rup
Display Status of Remote System

Syntax

```
rup [<opts>] host
```

Description

`rup` RPC client displays a system status for the specified host. The remote system must be running the `rstatd` RPC server to respond.

`rstatd` is the OS-9 RPC server.

Options

`-?`	Display the help message.

rusers
Display List of Users Logged into Remote System

Syntax

```
rusers [<opts>] <host>
```

Description

`rusers` RPC client displays a list of users logged into the specified host. The remote system must be running the `rusersd` RPC server to respond.

Options

`-?`	Display the help message.
`-a`	Display all processes on `<host>`.
`-n`	Display number of users logged into `<host>`.

rusersd
Rusers Server

Syntax

```
rusersd [<opts>]
```

Description

The `rusersd` RPC server returns a list of users on the system.

To place `rusersd` in the background, end the command line with an ampersand (`&`). For example, `rusersd&`.

`rusers` is the RPC client program.

Options

`-?`	Display the help message.

showmount
Display Remote Mounts

Syntax

```
showmount [<opts>]
```

Description

`showmount` displays the remote hosts and the local OS-9 devices mounted to the OS-9 NFS server.

Options

`-?`	Display the help message.

3 Remote Procedure Calls

This chapter describes the RPC protocol.

The RPC C library functions are described in the ***OS-9 Networking Programming Reference Manual***.

The RPC Protocol

You can use the RPC protocol to write distributed network applications. Each system on the network can provide any number of servers that client programs on other systems can dynamically call. RPC can implement a variety of network services, such as the following:

- Initiating the remote execution of a program
- Returning system statistics
- Looking up network processes
- Accessing and controlling remote file systems

RPC Protocol Requirements

The RPC protocol provides for the following:

- Unique specification of a called procedure
- Provisions for matching response messages to request messages
- Provisions for authenticating the caller to service and the service to caller

Besides these requirements, features that detect the following are also supported:

- RPC protocol mismatches
- Remote program protocol version mismatches
- Protocol errors (such as mis-specification of a procedure's parameters)
- Remote authentication failure
- Any other reasons why the desired procedure was not called

The RPC Language

The RPC language describes the procedures that operate on XDR data-types. RPC is an extension of the XDR language. The following is an example of the specification of a simple message program. It is shown here to familiarize you with the RPC language.

```
/* Simple message program */
program MESSAGEPROG {
   version MESSAGEVERS {
      int PRINTMESSAGE(string) = 1;
   } = 1;
} = 99;
```

`MESSAGEVERS` is the current version of the program. It has one procedure:

- `PRINTMESSAGE`

`PRINTMESSAGE` has one parameter—a string that allows the client to pass the string to the server, which prints the string to standard output.

The example RPC source can be found in `MWOS/SRC/SPF/RPC/DEMO`.

The RPC Language Specification

The RPC language is similar to XDR, except for the added `program-def` definition.

```
program-def:
   "program" identifier "{"
      version-def
      version-def *
   "}" "=" constant ";"
version-def:
   "version" identifier "{"
      procedure-def
      procedure-def *
   "}" "=" constant ";"
procedure-def:
   type-specifier identifier "(" type-specifier ")"
   "=" constant ";"
```

Refer to Chapter 5 External Data Representation for the remaining definitions.

Syntax Notes

The following are syntax notes concerning RPC:

- program and version are keywords. Do not use them as identifiers.
- A version name and/or version number cannot occur more than once within the scope of a program definition.
- A procedure name and/or procedure number cannot occur more than once within the scope of a version definition.
- Program identifiers are located in the same name space as constant and type identifiers.
- Only unsigned constants can be assigned to programs, versions, and procedures.

Transport Independence

The RPC protocol is independent of transport protocols. RPC does not care how a message is passed from one process to another. The protocol deals only with the specification and interpretation of messages.

The application must be aware of the underlying protocol because RPC does not try to implement any kind of reliability (that is, no retransmission or time-out).

- If RPC is running on top of an unreliable transport such as UDP/IP (User Datagram Protocol/Internet Protocol), the application must implement its own retransmission and time-out policy.
- If RPC is running on top of a reliable transport such as TCP/IP (Transmission Control Protocol/Internet Protocol), the transport may handle retransmissions and time-outs.

RPC Semantics

Specific semantics are not attached to the remote procedures or the execution of the remote procedures because RPC is transport independent. Although RPC can infer semantics from the underlying transport protocol, they should be explicitly stated.

For example, if RPC is running on top of an unreliable transport and an application retransmits RPC messages after short time-outs, RPC can only infer that the procedure was executed zero or more times if it received no reply. If RPC does receive a reply, it can infer that the procedure was executed at least once.

A server may not want to regrant a request from a client. Therefore, the server must remember the transaction ID packaged with every RPC request. The client RPC layer uses this transaction ID to match replies with requests. Occasionally, a client application may reuse its previous transaction ID when retransmitting a request. Knowing this, the server may choose to remember the transaction ID after granting a request and not regrant requests with the same ID. The server is not allowed to examine this ID except as a test for equality.

If a reliable transport is used, the application can infer from a reply message that the procedure was executed exactly once. If it receives no reply message, the application cannot assume the remote procedure was not executed.

Even if a connection-oriented protocol like TCP is used, an application still needs a time-out and reconnection to handle server crashes.

Transports do not have to be datagram-oriented or connection-oriented protocols. For OS-9, RPC is currently implemented on top of both TCP/IP and UDP/IP transports.

Binding

The act of binding a client to a service is not part of the RPC specification. The higher-level software that uses RPC must bind a client to a service.

Implementors should think of the RPC protocol as the jump-subroutine instruction (`jsr`) of a network. The linker makes `jsr` useful, and the linker itself uses `jsr` to accomplish a task. Likewise, the network uses RPC to accomplish this task.

Programs and Procedures

Each RPC procedure is uniquely defined by the following:

- A program number
- A version number

- A procedure number

The program number specifies a group of related remote procedures, each of which has a different procedure number. Each program has a version number. When a minor change is made to a remote service, a new program number does not have to be assigned.

Program numbers are administered by a central authority. A remote program can be used when its program number is known. Because most new protocols evolve into stable, mature protocols, a version field of the call message identifies which version of the protocol the caller is using. Version numbers make it possible for old and new protocols to speak through the same server process.

The procedure number identifies the procedure to call. These numbers are documented in the specific program's protocol specification. For example, a file service's protocol specification may state that procedure number 5 is `read` and procedure number 12 is `write`.

The actual RPC message protocol can also change. Therefore, the call message contains the RPC version number, which is always equal to 2 for the version of RPC described here.

Reply Message and Error Conditions

A reply message has enough information to distinguish the following error conditions:

- The remote implementation of RPC does not speak protocol version 2. The lowest and highest supported RPC version numbers are returned.
- The remote program is not available on the remote system.
- The remote program does not support the requested version number. The lowest and highest supported remote program version numbers are returned.
- The requested procedure number does not exist. This is usually a caller side protocol or programming error.
- The parameters to the remote procedure are not usable by the server. Again, this is usually caused by a disagreement about the protocol between the client and the service.

RPC's Three Layers

The RPC interface can be divided into a high, middle, and low layer.

- The highest layer is totally transparent to the hardware and software running RPC.
- The middle layer allows you to make remote procedure calls without considering sockets, OS-9, or low-level implementation mechanisms.
- The lowest layer does deal with sockets, OS-9, and low-level implementation mechanisms to allow the user to override specific defaults used by the middle layer's procedure calls.

Each of these layers is discussed in more detail, and examples are shown to illustrate how you may use them.

The Highest Layer

RPC's highest layer is totally transparent to the operating system, machine, and network upon which it is run. It is probably best to think of this level as a way of using RPC, rather than as a part of RPC.

The following example uses RPC's highest layer to determine how many users are logged into a remote machine. The RPC function `rmsg` is used.

```
#include <stdio.h>
#include <RPC/rpc.h>
#include "msg.h"
main(argc, argv)
   int argc;
   char *argv[];
{
   int num;
   if (argc != 3)
      exit(_errmsg(1,"usage: msg host message\n"));
   if ((num = rmsg(argv[1],argv[2])) <0) {
      fprintf(stderr,"error: rmsg\n");
      exit(-1);
   }
   printf("Message delivered to %s!\n", argv[1]);
   exit (0);
}
```

Following is the code for the `rmsg` function.

```
int rmsg(server,message)
   char *server;
   char *message;
{
   CLIENT *cl;
   int *result;

   cl = clnt_create(server, MESSAGEPROG, MESSAGEVERS, "tcp");
   if (cl == NULL) {
      clnt_pcreateerror(server);
      exit(1);
   }
   result = printmessage_1(&message, cl);
   if (result == NULL) {
      clnt_perror(cl, server);
      exit (1);
   }
   else return(0);
}
```

C programmers can write RPC service routines to implement the high layer RPC implementation.

The Middle Layer

Most applications can use the middle layer. When using the middle layer, sockets, OS-9, or low-level implementation mechanisms need not be considered. A remote procedure call is simply made to routines on other systems. RPC calls are made with the system routines `registerrpc()`, `callrpc()`, and `svc_run()`.

The middle layer does not allow the following:

- time-out specifications
- choice of control
- flexibility in case of errors.

The following program uses RPC's middle layer to send a message.

```
#include <stdio.h>
#include <RPC/rpc.h>
#include "msg.h"

main(argc, argv)
   int argc;
   char *argv[];
{
   CLIENT *cl;
   int result,result2;
   char *server;
   char *message;

   if (argc != 3)
      exit(_errmsg(1,"usage: msg host message\n"));

   server = argv[1];
   message = argv[2];

   if ((result2 = callrpc(argv[1], MESSAGEPROG, MESSAGEVERS,
PRINTMESSAGE,
      xdr_string, &message, xdr_int, &result)))
      {
         fprintf(stderr, "%s: call to message service failed. ",
            argv[0]);
      clnt_perrno(result);
      fprintf(stderr, "\n");
      exit(1);
   }
```

```
    if (result == 0) {
        fprintf(stderr, "%s: %s couldn't print your message\n",
        argv[0], server);
    exit(1);
    }
    printf("Message delivered to %s!\n", server);
    exit (0);
}
```

This program uses the function `callrpc()`. `callrpc()` is the simplest way of making remote procedure calls.

For more information on `callrpc()`, refer to the description in the RPC C library section.

Data types may have different representations on different machines. Therefore, `callrpc()` requires the type of the RPC parameter and a pointer to the parameter itself. `callrpc()` returns `xdr_int` as its first parameter, and `PRINTMESSAGE` returns an integer. Therefore, the result is of type `int`. `&result`, a pointer to where the long result will be placed, is the second parameter returned by `callrpc()`. `PRINTMESSAGE` takes a string as an argument, `callrpc()` is passed `xdr_string`.

If `callrpc()` tries to deliver a message several times without receiving an answer, it returns an error code. Because `callrpc()` uses UDP as its delivery mechanism, methods for adjusting the number of retries or for using a different protocol require the use of RPC's lower layer.

The following remote server procedure takes a pointer to the input of the remote procedure call and returns a pointer to the result.

```
#include <stdio.h>
#include <RPC/rpc.h>
#include "msg.h"

int *
printmessage_1(msg)
    char **msg;
{
    static int result;
    FILE *f;

    f = fopen("/term", "w");
    if (f == NULL) {
        result = 0;
        return (&result);
    }
    fprintf(f, "%s\n", *msg);
    fclose(f);
    result = 1;
```

```
   return (&result);
}
```

Normally, a server goes into an infinite loop while waiting to service requests after registering all of the RPC calls it plans to handle. In this example, only a single procedure needs to be registered:

```
#include <stdio.h>
#include <RPC/rpc.h>
#include "msg.h"
static void messageprog_1();

main()
{
    SVCXPRT *transp;

    registerrpc(MESSAGEPROG,
MESSAGEVERS,PRINTMESSAGE,printmessage_1,
        xdr_wrapstring,xdr_int);
    svc_run();
    fprintf(stderr, "svc_run returned\n");
    exit(1);
}
```

`registerrpc()` registers a C procedure that corresponds to a specific RPC procedure number. Refer to the RPC C library section of this chapter for information concerning the parameters that `registerrpc()` accepts. Multiple parameters or multiple results are passed as structures. You can only use `registerrpc()` with the UDP transport mechanism. Therefore, `registerrpc()` can always be used with calls generated by `callrpc()`.

The UDP transport mechanism can only deal with parameters and results less than 8K in length.

After registering the local procedure, the server program's main procedure calls `svc_run()`, the RPC library's remote procedure dispatcher. `svc_run()` calls the remote procedures in response to RPC call messages. The dispatcher decodes remote procedure parameters and encodes the results, using the XDR filters specified when the remote procedure was registered.

Assigning Program Numbers

Program numbers are assigned in groups of 0x20000000 according to Table 3-1 on page 42.

Table 3-1. Program Numbers

Program Number	Description
0x00000000 - 0x1fffffff	Defined by Sun Microsystems
0x20000000 - 0x3fffffff	Defined by OS-9
0x40000000 - 0x5fffffff	Transient
0x60000000 - 0x7fffffff	Reserved
0x80000000 - 0x9fffffff	Reserved
0xa0000000 - 0xbfffffff	Reserved
0xc0000000 - 0xdfffffff	Reserved
0xe0000000 - 0xffffffff	Reserved

The first group of numbers are assigned by Sun Microsystems and should be the same for all NFS/RPC systems. The second range is available for RPC services developed on OS-9. The third group is reserved for applications that generate program numbers dynamically. Do not use the remaining groups; they are reserved for future use.

Using the Middle Layer to Pass Arbitrary Data Types

RPC can handle arbitrary data structures, regardless of different machines' byte orders or structure layout conventions. RPC converts the data structure to XDR before passing on the structures. The process of converting from a particular machine representation to XDR format is called serializing. The reverse process is called deserializing. The type field parameters of `callrpc()` and `registerrpc()` can be a built-in procedure such as `xdr_u_long()` or a user-supplied procedure.

XDR built-in type routines are shown in Table 3-2 on page 42.

Table 3-2. XDR Built-in Type Routines

Name	Description
xdr_bool()	Translates booleans to/from XDR.
xdr_char()	Translates characters to/from XDR.

Table 3-2. XDR Built-in Type Routines (Continued)

Name	Description
xdr_enum()	Translates enumerated types to/from XDR.
xdr_int()	Translates integers to/from XDR.
xdr_long()	Translates long integers to/from XDR.
xdr_short()	Translates short integers to/from XDR.
xdr_u_char()	Translates unsigned characters to/from XDR.
xdr_u_int()	Translates unsigned integers to/from XDR.
xdr_u_long()	Translates unsigned long integers to/from XDR.
xdr_u_short()	Translates unsigned short integers to/from XDR.
xdr_wrapstring()	Packages an RPC message.

The routine `xdr_string()` exists, but cannot be used with `callrpc()` and `registerrpc()`. `callrpc()` and `registerrpc()` only pass two parameters to their XDR routines. Use `xdr_wrapstring()` which has only two parameters. `xdr_wrapstring()` calls `xdr_string()`.

The following is a user-defined type routine.

```
struct simple {
   int a;
   short b;
} simple;
```

You can send this routine using `callrpc()` by entering:

```
callrpc(hostname, PROGNUM, VERSNUM, PROCNUM,
        xdr_simple, &simple ...);
```

You can define the `xdr_simple` routine as:

```
#include <RPC/rpc.h>
xdr_simple(xdrsp, simplep)
   XDR *xdrsp;
   struct simple *simplep;
{
   if (!xdr_int(xdrsp, &simplep->a))
      return (0);
   if (!xdr_short(xdrsp, &simplep->b))
      return (0);
   return (1);
}
```

If an XDR routine completes successfully, it returns a non-zero value. Otherwise, it returns a zero.

In addition to the built-in primitives, you can use the building blocks shown in Table 3-3 on page 44.

Table 3-3. Building Blocks

Name	Description
xdr_array()	Translates arrays to/from XDR.
xdr_bytes()	Translates counted bytes to/from XDR.
xdr_opaque()	Translates opaque data to/from XDR.
xdr_pointer()	Translates pointer to/from XDR.
xdr_reference()	Translates pointers to/from XDR.
xdr_string()	Translates strings to/from XDR.
xdr_union()	Translates discriminated union to/from XDR.
xdr_vector()	Translates fixed-length arrays to/from XDR.

To send a variable array of integers, create a structure:

```
struct varintarr {
   int *data;
   int arrlnth;
} arr;
```

Then, make an RPC call:

```
callrpc(hostname, PROGNUM, VERSNUM, PROCNUM,
        xdr_varintarr, &arr...);
```

You can define the `xdr_varintarr()` routine as follows:

```
xdr_varintarr(xdrsp, arrp)
   XDR *xdrsp;
   struct varintarr *arrp;
{
   return (xdr_array(xdrsp, &arrp->data, &arrp->arrlnth,
      MAXLEN, sizeof(int), xdr_int));
}
```

`xdr_array()` accepts the following:

- an XDR handle
- a pointer to the array
- a pointer to the size of the array
- the maximum allowable array size
- the size of each array element
- an XDR routine for handling each array element as parameters.

If the size of the array is already known, you can use `xdr_vector()`. `xdr_vector()` serializes fixed-length arrays:

```
int intarr[SIZE];
xdr_intarr(xdrsp, intarr)
   XDR *xdrsp;
   int intarr[];
{
   int i;
   return (xdr_vector(xdrsp, intarr, SIZE, sizeof(int),
      xdr_int));
}
```

XDR always converts quantities to four-byte multiples when deserializing. This means that if either of the previous examples involved characters instead of integers, each character would occupy 32 bits. You can use the XDR routine `xdr_bytes()` to pack the characters. `xdr_string()` can be used to pack null-terminated strings. On serializing, `xdr_string()` gets the string length from **strlen**. On deserializing, `xdr_string()` creates a null-terminated string.

The following program calls the built-in functions `xdr_string()` and `xdr_reference()` and the example defined earlier, `xdr_simple()`:

```
struct finalexample {
   char *string;
   struct simple *simplep;
} finalexample;
xdr_finalexample(xdrsp, finalp)
   XDR *xdrsp;
   struct finalexample *finalp;
{
   if (!xdr_string(xdrsp, &finalp->string, MAXSTRLEN))
      return (0);
   if (!xdr_reference(xdrsp, &finalp->simplep,
     sizeof(struct simple), xdr_simple);
      return (0);
   return (1);
}
```

Lowest Layer of RPC

In the examples presented so far, RPC automatically handles many details. There may be occasions when you need to override the defaults. You can use RPC's lower layer to change time-out specifications, choice of control, etc.

You should be familiar with sockets and the system calls for dealing with sockets before using the lower layer.

Use RPC's lower layer for the following:

- Using TCP to send long streams of data. Both of the higher layers use UDP, which restricts RPC calls to 8K of data.
- Allocating and freeing memory while serializing or deserializing with XDR routines. The higher levels have no calls to explicitly free memory.
- Performing authentication on either the client or server side by supplying or verifying credentials.

Using RPC's Lowest Layer on the Server Side

The following server is for the `rmsg` program. It uses the lower layer of RPC instead of `registerrpc()`.

```
#include <stdio.h>
#include <stdlib.h>
#include <RPC/rpc.h>
#include "msg.h"

static void messageprog_1();

main()
{
   SVCXPRT *transp;

   pmap_unset(MESSAGEPROG, MESSAGEVERS);

   transp = svcudp_create(RPC_ANYSOCK);
   if (transp == NULL) {
      fprintf(stderr, "cannot create udp service.\n");
      exit(1);
   }
   if (!svc_register(transp, MESSAGEPROG, MESSAGEVERS,
messageprog_1,
            IPPROTO_UDP)){
      fprintf(stderr, "unable to register (MESSAGEPROG,
MESSAGEVERS,
            udp).\n");
      exit(1);
   }
   transp = svctcp_create(RPC_ANYSOCK, 0, 0);
   if (transp == NULL) {
      fprintf(stderr, "cannot create tcp service.\n");
      exit(1);
   }
   if (!svc_register(transp, MESSAGEPROG, MESSAGEVERS,
messageprog_1,
            IPPROTO_TCP)) {
      fprintf(stderr, "unable to register (MESSAGEPROG,
MESSAGEVERS,
            tcp).\n");
   exit(1);
   }
   svc_run();
   fprintf(stderr, "svc_run returned\n");
   exit(1);
}
static void
messageprog_1(rqstp, transp)
   struct svc_req *rqstp;
   SVCXPRT *transp;
{
   union {
      char *printmessage_1_arg;
   } argument;
   char *result;
   bool_t (*xdr_argument)(), (*xdr_result)();
   char *(*local)();
   switch (rqstp->rq_proc) {
   case NULLPROC:
      svc_sendreply(transp, xdr_void, NULL);
      return;
   case PRINTMESSAGE:
      xdr_argument = xdr_wrapstring;
      xdr_result = xdr_int;
      local = (char *(*)()) printmessage_1;
      break;
   default:
      svcerr_noproc(transp);
      return;
   }
   bzero(&argument, sizeof(argument));
   if (!svc_getargs(transp, xdr_argument, &argument)) {
      svcerr_decode(transp);
      return;
   }
   result = (*local)(&argument, rqstp);
   if (result != NULL && !svc_sendreply(transp, xdr_result,
result)) {
      svcerr_systemerr(transp);
   }
   if (!svc_freeargs(transp, xdr_argument, &argument)) {
      fprintf(stderr, "unable to free arguments\n");
      exit(1);
   }
}
```

In this program, the server gets a transport handle, which receives RPC messages and replies to these messages. If `registerrpc()` had been used, it would have called `svcudp_create()` by default to get a UDP handle. `svctcp_create()` allows a TCP handle. RPC's lower layer allows a choice between `svcudp_create()` and `svctcp_create()`. If the parameter is `RPC_ANYSOCK`, the RPC library creates a socket on which to receive and reply to RPC calls. Otherwise, `svcudp_create()` or `svctcp_create()` expects the parameter to be a valid socket number. If a user's socket is specified, it can be bound or unbound. If it is bound to a port by the user, the port numbers of `svcudp_create()` or `svctcp_create()` and `clntudp_create()` or `clntcp_create()` (the low-level client routine) must match.

If you specify the `RPC_ANYSOCK` parameter, the RPC library routines open sockets. Otherwise, the routines expect the user to open the sockets. The routines `svcudp_create()` and `clntudp_create()` [or `svctcp_create()` and `clntcp_create()`] cause the RPC library routines to bind their socket if it is not already bound.

A service may choose to register its port number with the local port mapper service. To do this, specify a non-zero protocol number in `svc_register()`.

A client can discover the server's port number by consulting the port mapper on their server's machine. Specifying 0 as the port number in `clntudp_create()` or `clntcp_create()` requests this information.

Before creating a `SVCXPRT`, call `pmap_unset()`. `pmap_unset()` erases any existing entries for `MESSAGEPROG` from the port mapper's tables.

Finally, the program number is associated with the procedure `messageprog_1`. The final parameter to `svc_register()` is normally the protocol to be used; in this case either `IPPROTO_UDP` or `IPPROTO_TCP`. Unlike `registerrpc()`, XDR routines are not involved in the registration process, and registration is performed on the program level rather than the procedure level.

The user routine `messageprog_1` must call and dispatch the appropriate XDR routines based on the procedure number. `registerrpc()` automatically handles two tasks for `messageprog_1`:

- Procedure `NULLPROC` returns with no results. This can detect if a remote program is running.
- `Registerrpc()` checks for invalid procedure numbers. If an invalid procedure number is detected, `svcerr_noproc()` is called to handle the error.

The user service routine serializes the results and returns them to the RPC caller via `svc_sendreply()`. The first parameter is the `SVCXPRT` handle, the second parameter is the `XDR` routine, and the third parameter is a pointer to the data to be returned.

Memory Allocation with XDR and the Lower Layer

XDR routines also perform memory allocation. This is why the second parameter of `xdr_array()` is a pointer to an array, rather than the array itself. If the second parameter is null, `xdr_array()` allocates space for the array and returns a pointer to it, placing the size of the array in the third parameter. As an example, consider the following XDR routine `xdr_chararr1()` which deals with a fixed array of bytes with length `SIZE`.

```
xdr_chararr1(xdrsp, chararr)
   XDR *xdrsp;
   char chararr[];
{
   char *p;
   int len;
   p = chararr;
   len = SIZE;
   return (xdr_bytes(xdrsp, &p, &len, SIZE));
}
```

It might be called from a server:

```
char chararr[SIZE];
svc_getargs(transp, xdr_chararr1, chararr);
```

Space has already been allocated in `chararr`. If you want XDR to allocate the memory, rewrite the routine:

```
xdr_chararr2(xdrsp, chararrp)
   XDR *xdrsp;
   char **chararrp;
{
   int len;
   len = SIZE;
   return (xdr_bytes(xdrsp, charrarrp, &len, SIZE));
}
```

The RPC call might look like this:

```
char *arrptr;
arrptr = NULL;
svc_getargs(transp, xdr_chararr2, &arrptr);
```

```
/* Use the result here */
svc_freeargs(transp, xdr_chararr2, &arrptr);
```

After being used, you can free the character array with `svc_freeargs()`. `svc_freeargs()` does not attempt to free any memory if the variable indicating the memory is null. For example, in the routine `xdr_finalexample()` presented earlier, if `finalp->string` was null, it would not be freed. The same is true for `finalp->simplep`.

To summarize, each XDR routine is responsible for serializing, deserializing, and freeing memory. When an XDR routine is called from `callrpc()`, the serializing portion is used. When called from `svc_getargs()`, the deserializer is used. When called from `svc_freeargs()`, the memory deallocator is used.

The Calling Side of the Lower Layer

When using `callrpc()`, you have no control over the RPC delivery mechanism or the socket used to transport the data. To illustrate how the lower layer of RPC allows you to adjust these parameters, consider the following code to call the `nusers` service:

```
#include <stdio.h>
#include <RPC/rpc.h>
#include "msg.h"

main(argc, argv)
   int argc;
   char *argv[];
{
   CLIENT *cl;
   int *result;
   char *server;
   char *message;

   if (argc != 3)
   exit(_errmsg(1,"usage: msg host message\n"));

   server = argv[1];
   message = argv[2];

   cl = clnt_create(server, MESSAGEPROG, MESSAGEVERS, "tcp");
   if (cl == NULL) {
      clnt_pcreateerror(server);
      exit(1);
   }

   result = printmessage_1(&message, cl);
   if (result == NULL) {
      clnt_perror(cl, server);
      exit (1);
   }

   if (*result == 0) {
      fprintf(stderr, "%s: %s couldn't print your message\n",
         argv[0], server);
      exit(1);
   }

   printf("Message delivered to %s!\n", server);
   exit (0);
}

#include <RPC/rpc.h>
#include <time.h>
#include "msg.h"

static struct timeval TIMEOUT = { 25, 0 };

int *
printmessage_1(argp, clnt)
   char **argp;
   CLIENT *clnt;
{
   static int res;

   bzero(&res, sizeof(res));
   if (clnt_call(clnt, PRINTMESSAGE, xdr_wrapstring, argp,
xdr_int,
         &res, TIMEOUT) != RPC_SUCCESS) {
      return (NULL);
}
   return (&res);
}
```

The low-level version of `callrpc()` is `clnt_call()`. `clnt_call()` accepts a `CLIENT` pointer. The parameters to `clnt_call()` are:

- A `CLIENT` pointer
- The procedure number
- The XDR routine for serializing the parameter
- A pointer to the parameter
- The XDR routine for deserializing the return value

- A pointer to where the return value will be placed
- The time in seconds to wait for a reply.

The `CLIENT` pointer is encoded with the transport mechanism. `callrpc()` uses UDP. Therefore, it calls `clntudp_create()` to get a `CLIENT` pointer. To get TCP, use `clnttcp_create()`.

The parameters to `clntudp_create()` are:

- The server address
- The program number
- The version number
- A time-out value (between tries)
- A pointer to a socket

The final parameter to `clnt_call()` is the total time to wait for a response. Therefore, the number of tries is the `clnt_call()` time-out divided by the `clntudp_create()` time-out.

`clnt_destroy()` deallocates any space associated with the CLIENT handle. `clnt_destroy()` does not close the associated socket, which was passed as a parameter to `clntudp_create()`. This makes it possible, in cases where multiple client handles are using the same socket, to destroy one handle without closing the socket that other handles are using.

To make a stream connection, `clntudp_create()` is replaced with `clnttcp_create()`.

```
clnttcp_create(&server_addr, prognum, versnum, &sock,
               inputsize, outputsize);
```

There is no time-out parameter. Instead, the receive and send buffer sizes are specified. When the `clnttcp_create()` call is made, a TCP connection is established. All RPC calls using that `CLIENT` handle use this connection. The server side of an RPC call using TCP replaces `svcudp_create()` by `svctcp_create()`.

```
transp = svctcp_create(RPC_ANYSOCK, 0, 0);
```

The last two parameters to `svctcp_create()` are send and receive sizes respectively. If 0 is specified for either of these, the system chooses a default.

Other RPC Features

This section discusses some other aspects of RPC that are occasionally useful.

Select on the Server Side

If a server process is processing RPC requests while performing an activity that involves periodically updating a data structure, the process can set an OS-9 alarm before calling `svc_run()`. However, if the activity involves waiting for input from a file not managed by RPC, the `svc_run()` call does not work.

You can bypass `svc_run()` by calling `svc_getreqset()`. You need to know the path numbers of the socket(s) associated with your programs. You can call `select()` on both the RPC sockets and other path numbers. `svc_fds()` is a bit mask of all the path numbers that RPC uses for services. It can change whenever an RPC library routine is called because descriptors are constantly being opened and closed.

Broadcast RPC

In broadcast RPC-based protocols, the client sends a broadcast packet to the network and waits for numerous replies. Broadcast RPC uses unreliable, packet-based protocols such as UDP as its transports. Servers that support broadcast protocols only respond when a request is successfully processed. This means that there is no response to errors.

Broadcast RPC uses the port mapper daemon. The port mapper converts RPC program numbers into DARPA protocol port numbers. The user cannot perform broadcast RPC without the port mapper.

Table 3-4 on page 53 shows the main differences between normal RPC calls and broadcast RPC calls.

Table 3-4. Normal RPC calls/Broadcast RPC calls

Normal RPC	Broadcast RPC
Expects one answer	Expects one answer or more from each responding machine
Supported on both TCP and UDP protocols	Supported only on UDP protocols
Reports unsuccessful responses	Does not report unsuccessful responses
Does not require messages to be sent to the port mapper	All broadcast RPC messages are sent to the port mapper port

Broadcast RPC Synopsis

Following is an example of the `clnt_broadcast()` call.

```
#include <RPC/pmap_clnt.h>
   . . .
enum clnt_statclnt_stat;
   . . .
clnt_stat = clnt_broadcast(prognum, versnum, procnum,
  inproc, in, outproc, out, eachresult)
   u_long    prognum;        /* program number */
   u_long    versnum;        /* version number */
   u_long    procnum;        /* procedure number */
   xdrproc_t inproc;         /* XDR routine for arguments */
   caddr_t   in;             /* pointer to arguments */
   xdrproc_t outproc;        /* XDR routine for results */
   caddr_t   out;             /* pointer to results */
   bool_t    (*eachresult)();/* call with each result gotten */
```

The procedure `eachresult()` is called each time a valid result is obtained. It returns a boolean value that indicates whether or not the user wants more responses:

```
bool_t done;
   . . .
done = eachresult(resultsp, raddr)
   caddr_t resultsp;
   struct sockaddr_in *raddr;       /* Address of the responding
machine */
```

If `done` is TRUE, broadcasting stops and `clnt_broadcast()` returns successfully. Otherwise, the routine waits for another response. The request is rebroadcast after a few seconds of waiting. If no responses come back, the routine returns with `RPC_TIMEDOUT`.

Batching

The RPC architecture is designed so that clients send a call message and wait for servers to reply that the call succeeded. This implies that clients perform no other functions while servers are processing a call. This is inefficient if the client does not want or need an acknowledgment for every message sent. Using RPC batch facilities, clients can continue computing while waiting for a response.

RPC messages can be placed in a "pipeline" of calls to a desired server. This is called *batching*. Batching assumes the following:

- Each RPC call in the pipeline requires no response from the server, and the server does not send a response message.
- The pipeline of calls is transported on a reliable, byte-stream transport such as TCP/IP.

The client can generate new calls in parallel with the server executing previous calls because the server does not respond to every call. Further, the TCP implementation can buffer many call messages and send them to the server in one write system call. This overlapped execution decreases the interprocess communication overhead of the client and server processes and the total elapsed time of a series of calls.

Because the batched calls are buffered, the client should eventually perform a legitimate RPC call to flush the pipeline.

A contrived example of batching follows. Assume a string rendering service has two similar calls; one renders a string and returns void results, while the other renders a string and remains silent. Using TCP, the service may look like the following:

```
#include <stdio.h>
#include <RPC/rpc.h>
void windowdispatch();
main()
{
   SVCXPRT *transp;
   transp = svctcp_create(RPC_ANYSOCK, 0, 0);
   if (transp == NULL){
      fprintf(stderr, "cannot create an RPC server\n");
      exit(1);
   }
   pmap_unset(WINDOWPROG, WINDOWVERS);
   if (!svc_register(transp, WINDOWPROG, WINDOWVERS,
     windowdispatch, IPPROTO_TCP)) {
      fprintf(stderr, "cannot register WINDOW service\n");
      exit(1);
   }
   svc_run();                               /* Never returns */
   fprintf(stderr, "should never reach this point\n");
}
void
windowdispatch(rqstp, transp)
   struct svc_req *rqstp;
   SVCXPRT *transp;
{
   char *s = NULL;
   switch (rqstp->rq_proc) {
   case NULLPROC:
      if (!svc_sendreply(transp, xdr_void, 0))
         fprintf(stderr, "cannot reply to RPC call\n");
      return;
   case RENDERSTRING:
      if (!svc_getargs(transp, xdr_wrapstring, &s)) {
         fprintf(stderr, "cannot decode arguments\n");
         /* Tell caller that there is an error */
         svcerr_decode(transp);
         break;
      }
      /* Call here to render the string s */
      if (!svc_sendreply(transp, xdr_void, NULL))
         fprintf(stderr, "cannot reply to the RPC call\n");
      break;
   case RENDERSTRING_BATCHED:
      if (!svc_getargs(transp, xdr_wrapstring, &s)) {
         fprintf(stderr, "cannot decode arguments\n");
         /* We are silent in the face of protocol errors */
         break;
      }
      /* Call here to render string s, but send no reply! */
      break;
   default:
      svcerr_noproc(transp);
      return;
   }
   /* Now free string allocated while decoding arguments */
   svc_freeargs(transp, xdr_wrapstring, &s);
}
```

The service could have one procedure that takes the string and a boolean to indicate whether or not the procedure should respond.

In order for a client to take advantage of batching, the client must perform RPC calls on a TCP-based transport. The actual calls must have the following attributes:

- The result's XDR routine must be 0 (null).
- The RPC call's time-out must be 0.

Following is an example of a client that uses batching to render strings. The batching is flushed when the client gets a null string.

```
#include <stdio.h>
#include <RPC/rpc.h>
#include <time.h>

main(argc, argv)
   int argc;
   char **argv;
{
   struct hostent *hp;
   struct timeval pertry_timeout, total_timeout;
   struct sockaddr_in server_addr;
   int sock = RPC_ANYSOCK;
   register CLIENT *client;
   enum clnt_stat clnt_stat;
   char buf[1000], *s = buf;
   if ((client = clnttcp_create(&server_addr,
     WINDOWPROG, WINDOWVERS, &sock, 0, 0)) == NULL) {
      perror("clnttcp_create");
      exit(-1);
   }
   total_timeout.tv_sec = 0;
   total_timeout.tv_usec = 0;
   while (scanf("%s", s) != EOF) {
      clnt_stat = clnt_call(client, RENDERSTRING_BATCHED,
         xdr_wrapstring, &s, NULL, NULL, total_timeout);
      if (clnt_stat != RPC_SUCCESS) {
         clnt_perror(client, "batched RPC");
         exit(-1);
      }
   }
   /* Now flush the pipeline */
   total_timeout.tv_sec = 20;
   clnt_stat = clnt_call(client, NULLPROC, xdr_void, NULL,
      xdr_void, NULL, total_timeout);
   if (clnt_stat != RPC_SUCCESS) {
      clnt_perror(client, "rpc");
      exit(-1);
   }
   clnt_destroy(client);
}
```

Because the server sends no message, the clients cannot be notified of any failures. Therefore, clients must handle their own errors.

The RPC Message Protocol

This section defines the RPC message protocol in the XDR data description language. The message is defined in a top-down style.

```
enum msg_type {
   CALL  = 0,
   REPLY = 1
};
/* A reply to a call message can either be accepted or rejected */
enum reply_stat {
   MSG_ACCEPTED = 0,
   MSG_DENIED   = 1
};
/*
* Given that a call message was accepted, the following is the
* status of an attempt to call a remote procedure.
*/
enum accept_stat {
   SUCCESS       = 0,    /* RPC executed successfully */
   PROG_UNAVAIL  = 1,    /* Remote has not exported program */
   PROG_MISMATCH = 2,    /* Remote cannot support version # */
   PROC_UNAVAIL  = 3,    /* Program cannot support procedure */
   GARBAGE_ARGS  = 4     /* Procedure cannot decode params P*/
};
/* Reasons why a call message was rejected: */
enum reject_stat {
   RPC_MISMATCH = 0,    /* RPC version number != 2 */
   AUTH_ERROR = 1       /* Remote cannot authenticate caller */
};
/* Why authentication failed: */
enum auth_stat {
   AUTH_BADCRED      = 1,  /* Bad credentials (seal broken) */
   AUTH_REJECTEDCRED = 2,  /* Client must begin new session */
   AUTH_BADVERF      = 3,  /* Bad verifier (seal broken)  */
   AUTH_REJECTEDVERF = 4,  /* Verifier expired or replayed  */
   AUTH_TOOWEAK      = 5   /* Rejected for security reasons */
};
/*
* The RPC message:
* All messages start with a transaction identifier, xid, followed
by a
* two-armed discriminated union.  The union's discriminant is a
msg_type
* which switches to one of the two types of the message.  The xid
of a
* REPLY message always matches that of the initiating CALL message.
NB:
* The xid field is only used for clients matching reply messages
```

```
with
* call messages or for servers detecting retransmissions; the
service
* side cannot treat this ID as any type of sequence number.
*/
struct rpc_msg {
   unsigned int xid;
   union switch (msg_type mtype) {
      case CALL:
         call_body cbody;
      case REPLY:
         reply_body rbody;
   } body;
};
/*
* Body of an RPC request call: In version 2 of the RPC protocol
specification,
* rpcvers must be equal to 2.  The fields prog, vers, and proc
specify the
* remote program, its version number and the procedure within the
remote
* program to be called.  After these fields are two authentication
parameters:
* cred (authentication credentials) and verf (authentication
verifier).  The
* two authentication parameters are followed by the parameters to
the remote
* procedure, which are specified by the specific program protocol.
*/
struct call_body {
   unsigned int rpcvers;  /* must be equal to two (2) */
   unsigned int prog;
   unsigned int vers;
   unsigned int proc;
   opaque_auth cred;
   opaque_auth verf;
   /* procedure specific parameters start here */
};
/*
* Body of a reply to an RPC request:
* The call message was either accepted or rejected.
*/
union reply_body switch (reply_stat stat) {
   case MSG_ACCEPTED:
      accepted_reply areply;
   case MSG_DENIED:
      rejected_reply rreply;
} reply;
/*
* Reply to an RPC request that was accepted by the server:  There
could be an
* error even though the request was accepted.  The first field is
```

```
an
* authentication verifier that the server generates to validate
itself to the
* caller.  It is followed by a union whose discriminant is an enum
accept_stat.
* The SUCCESS arm of the union is protocol specific. The
PROG_UNAVAIL,
* PROC_UNAVAIL, and GARBAGE_ARGP arms of the union are void.  The
PROG_MISMATCH
* arm specifies the lowest and highest version numbers of the
remote program
* supported by the server.
*/
struct accepted_reply {
   opaque_auth verf;
   union switch (accept_stat stat) {
      case SUCCESS:
         opaque results[0];
         /* procedure-specific results start here */
      case PROG_MISMATCH:
         struct {
            unsigned int low;
            unsigned int high;
         } mismatch_info;
      default:
         /*
         * Void.  Cases include PROG_UNAVAIL, PROC_UNAVAIL,
         * and GARBAGE_ARGS.
         */
         void;
   } reply_data;
};
/*
* Reply to an RPC request that was rejected by the server: The
request can
* be rejected for two reasons: either the server is not running a
compatible
* version of the RPC protocol (RPC_MISMATCH), or the server refuses
to
* authenticate the caller (AUTH_ERROR).  In case of an RPC version
mismatch,
* the server returns the lowest and highest supported RPC version
numbers.
* In case of refused authentication, failure status is returned.
*/
union rejected_reply switch (reject_stat stat) {
   case RPC_MISMATCH:
      struct {
         unsigned int low;
         unsigned int high;
      } mismatch_info;
   case AUTH_ERROR:
        auth_stat stat;
};
```

Record Marking Standard

When RPC messages are passed on top of a byte stream protocol, messages should be delimited from one another to detect and possibly recover from user protocol errors. This is called record marking (RM). OS-9 uses record marking with the TCP/IP transport for passing RPC messages on TCP streams. One RPC message fits into one RM record.

A record is composed of one or more record fragments. A record fragment is a four-byte header followed by 0 to (2**31) - 1 bytes of fragment data. The bytes encode an unsigned binary number. As with XDR integers, the byte order is from highest to lowest. The number encodes a boolean value which indicates whether the fragment is the last fragment of the record (bit value 1 implies the fragment is the last fragment) and a 31-bit unsigned binary value, which is the length in bytes of the fragment's data. The boolean value is the highest-order bit of the header. The length is the 31 low-order bits.

This record specification is not in XDR standard form.

Authentication

In the examples presented so far, the caller never identified itself to the server, and the server never required an ID from the caller. Clearly, some network services, such as a network file system, require stronger security.

In reality, every RPC call is authenticated by the RPC package on the server. Similarly, the RPC client package generates and sends authentication parameters. Different forms of authentication can be associated with RPC clients. A field in the RPC header indicates which protocol is being used. The default authentication is type none.

Provisions for authentication of caller to service and service to caller are provided as part of the RPC protocol. The call message has the following two authentication fields:

- the credentials
- the verifier

The reply message has one authentication field, the response verifier. The RPC protocol specification defines all three fields to be the following opaque type:

```
enum auth_flavor {
    AUTH_NULL        = 0,
    AUTH_UNIX        = 1,
    AUTH_SHORT       = 2,
    /* and more to be defined */
};
struct opaque_auth {
    auth_flavor flavor;
    opaque body<400>;
};
```

Any `opaque_auth` structure is an `auth_flavor` enumeration followed by bytes which are opaque to the RPC protocol implementation.

The interpretation and semantics of the data contained within the authentication fields is specified by individual, independent authentication protocol specifications. If authentication parameters were rejected, the response message contains information stating the reason for the rejection.

The authentication subsystem of the RPC package is open ended. That is, numerous types of authentication are easy to support. The following are the types of authentication currently implemented. You can also create your own authentication types.

Null Authentication

Often calls must be made where the caller does not know who it is or the server does not care who the caller is. In this case, the flavor value (the discriminant of the `opaque_auth`'s union) of the RPC message's credentials, verifier, and response verifier is `AUTH_NULL`. The bytes of the `opaque_auth`'s body are undefined. It is recommended that the opaque length be zero.

OS-9 Authentication

The caller of a remote procedure may wish to have the same identification as on an OS-9 system. OS-9 uses a UNIX-style of RPC authentication. The value of the credential's discriminant of an RPC call message is `AUTH_UNIX`. The credential's opaque body encode the following structure. The verifier accompanying the credentials should be of `AUTH_NULL`.

```
struct auth_unix {
   unsigned int stamp;          /* an arbitrary ID */
   string machinename<255>;     /* name of caller's machine */
   unsigned int uid;            /* caller's user ID */
   unsigned int gid;            /* caller's group ID */
   unsigned int gids<10>;       /* this array is not used */
};
```

The value of the response verifier's discriminant received in the reply message from the server may be `AUTH_NULL` or `AUTH_SHORT`. In the case of `AUTH_SHORT`, the bytes of the response verifier's string encode an opaque structure. You can pass this new opaque structure to the server instead of the original `AUTH_UNIX` flavor credentials. The server keeps a cache which maps shorthand opaque structures (passed back by way of an `AUTH_SHORT` style response verifier) to the original credentials of the caller. The caller can save network bandwidth and server CPU cycles by using the new credentials.

The server may flush the shorthand opaque structure at any time. If this happens, the remote procedure call message is rejected due to an authentication error. The reason for the failure is `AUTH_REJECTEDCRED`. At this point, the caller may wish to try the original `AUTH_UNIX` style of credentials.

A caller creates a new RPC client handle as follows:

```
clnt = clntudp_create(address, prognum, versnum,
            wait, sockp)
```

The appropriate transport instance defaults the associate authentication handle to be:

```
clnt->cl_auth = authnone_create();
```

The RPC client can choose to use the OS-9/UNIX style authentication by setting `clnt->cl_auth` after creating the RPC client handle:

```
clnt->cl_auth = authunix_create_default();
```

This causes each RPC call associated with `clnt` to carry the following authentication credentials structure:

```
/* OS-9/OS-9000/
UNIX style credentials. */
struct authunix_parms {
    u_long  aup_time;        /* credentials creation time */
    char    *aup_machname;   /* host name where client is */
    int     aup_uid;         /* client's OS-9/OS-9000 user ID */
    int     aup_gid;         /* client's OS-9/OS-9000 group ID */
    u_int   aup_len;         /* element length of aup_gids */
    int     *aup_gids;       /* array of groups user is in */
};
```

These fields are set by `authunix_create_default`. Because the RPC user created this new style of authentication, the user is responsible for destroying it to conserve memory:

```
auth_destroy(clnt->cl_auth);
```

Server Side Authentication

Authentication issues are more complex for service implementors because RPC requests passed to the service dispatch routine have an arbitrary authentication style. Consider the fields of a request handle passed to a service dispatch routine:

```
/* An RPC Service request */
struct svc_req {
    u_long    rq_prog;         /* service program number */
    u_long    rq_vers;         /* service protocol version number
*/
    u_long    rq_proc;         /* desired procedure number */
    struct opaque_auth rq_cred;    /* raw credentials */
    caddr_t   rq_clntcred;       /* credentials (read only) */
};
```

The `rq_cred` structure is opaque except for the style of authentication credentials:

```
/* Authentication info.  Mostly opaque to the programmer. */
struct opaque_auth {
    enum_t  oa_flavor;  /* style of credentials */
    caddr_t oa_base;    /* address of more auth stuff */
    u_int   oa_length;  /* not to exceed MAX_AUTH_BYTES */
};
```

The RPC package guarantees that the request's `rq_cred` is well formed. Therefore, the service implementor may inspect the request's `rq_cred.oa_flavor` to determine which style of authentication the caller used. The service implementor may also wish to inspect the other fields of `rq_cred` if the style is not one of the authentication styles supported by the RPC package.

The RPC package also guarantees that the request's `rq_clntcred` field is either null or points to a well-formed structure that corresponds to a supported style of authentication credentials. Only OS-9/UNIX style is currently supported. `rq_clntcred` could be cast to a pointer to an `authunix_parms` structure. If `rq_clntcred` is null, the service implementor may inspect the other opaque fields of `rq_cred` to see if the service knows about a new type of authentication that the RPC package does not know about.

Our remote users service example can be extended so that it computes results for all users except `uid 16`:

```
nuser(rqstp, transp)
   struct svc_req *rqstp;
   SVCXPRT *transp;
{
   struct authunix_parms *unix_cred;
   int uid;
   unsigned long nusers;
   /* we do not care about authentication for NULL proc */
   if (rqstp->rq_proc == NULLPROC) {
      if (!svc_sendreply(transp, xdr_void, 0)) {
         fprintf(stderr, "cannot reply to RPC call\n");
         exit(1);
       }
       return;
   }
   /* now get the uid */
   switch (rqstp->rq_cred.oa_flavor) {
   case AUTH_UNIX:
      unix_cred =
         (struct authunix_parms *)rqstp->rq_clntcred;
      uid = unix_cred->aup_uid;
      break;
   case AUTH_NULL:
   default:
      svcerr_weakauth(transp);
      return;
   }
   switch (rqstp->rq_proc) {
   case RUSERSPROC_NUM:
      /* make sure caller is allowed to call this proc */
      if (uid == 16) {
         svcerr_systemerr(transp);
         return;
      }
      /*
       * code here to compute the number of users
       * and put in variable nusers
```

```
         */
        if (!svc_sendreply(transp, xdr_u_long, &nusers)) {
           fprintf(stderr, "cannot reply to RPC call\n");
           exit(1);
        }
        return;
     default:
        svcerr_noproc(transp);
        return;
     }
  }
```

It is customary not to check the authentication parameters associated with the NULLPROC. If the authentication parameter's type is not suitable for a particular service, call svcerr_weakauth(). The service protocol itself should return status for denied access. In this example, the protocol does not have such a status. The svcerr_systemerr() system primitive is called instead.

RPC deals only with authentication and not with individual services' "access control." The services themselves must implement their own access control policies and reflect these policies as return statuses in their protocols.

Examples

Callback Procedures

Occasionally, a server may become a client and make an RPC call back to the client process. Whenever an RPC call is made back to the client, a program number is required. The program number should be in the transient range (0x40000000 - 05fffffff) because it is dynamically generated.

In the following program, the gettransient() routine returns a valid program number and registers this number with the port mapper located on the same machine as the gettransient() routine. The call to pmap_set() is a test and set operation. It tests whether a program number has already been registered with the port mapper. pmap_set() reserves the program number if it has not been registered. On return, the sockp parameter contains a socket so that the svcudp_create() and svctcp_create() calls can be used as parameters.

```
#include <stdio.h>
#include <RPC/rpc.h>
#include <sys/types.h>
#include <sys/socket.h>
gettransient(proto, vers, sockp)
   int proto, vers, *sockp;
{
   static int prognum = 0x40000000;
   int s, len, socktype;
   struct sockaddr_in addr;
   switch(proto) {
        case IPPROTO_UDP:
            socktype = SOCK_DGRAM;
            break;
        case IPPROTO_TCP:
            socktype = SOCK_STREAM;
            break;
        default:
            fprintf(stderr, "unknown protocol type\n");
            return 0;
    }
    if (*sockp == RPC_ANYSOCK) {
        if ((s = socket(AF_INET, socktype, 0)) < 0) {
            perror("socket");
            return (0);
        }
        *sockp = s;
    }
    else
        s = *sockp;
    addr.sin_addr.s_addr = 0;
    addr.sin_family = AF_INET;
    addr.sin_port = 0;
    len = sizeof(addr);
    /* may be already bound, so do not check for error */
    bind(s, &addr, len);
    if (getsockname(s, &addr, &len)< 0) {
        perror("getsockname");
        return (0);
    }
    while (!pmap_set(prognum++, vers, proto,
        ntohs(addr.sin_port))) continue;
    return (prognum-1);
}
```

The call to `ntohs` ensures that the port number in `addr.sin_port`, which is in network byte order, is passed in host byte order as `pmap_set()` expects.

Remote debugging is an instance where a callback procedure is necessary. For example, a remote debugger's client may be a window system program, and the server may be a debugger running on a remote machine. Normally when a user clicks a mouse button on the debugging window, it is converted to a debugger command and an RPC call is made to the server to execute the user's command. However, when the debugger hits a breakpoint, the server (which in this example is the debugger) makes an RPC call to the client process to inform the user that a breakpoint has been reached.

The following pair of programs illustrate the `gettransient()` routine. The client makes an RPC call to the server, passing the server a transient program number. The client then waits to receive a call back from the server at the program number. The server registers the program `EXAMPLEPROG` to receive the RPC call informing it of the callback program number. In this example, the server sends a callback RPC call using the program number it received earlier when the `ALRM` signal was received.

```
/* client */
#include <stdio.h>
#include <RPC/rpc.h>
int callback();
char hostname[256];
main()
{
   int x, ans, s;
   SVCXPRT *xprt;
   gethostname(hostname, sizeof(hostname));
   s = RPC_ANYSOCK;
   x = gettransient(IPPROTO_UDP, 1, &s);
   fprintf(stderr, "client gets prognum %d\n", x);
   if ((xprt = svcudp_create(s)) == NULL) {
     fprintf(stderr, "rpc_server: svcudp_create\n");
      exit(1);
   }
   /* protocol is 0 - gettransient() does registering */
   (void)svc_register(xprt, x, 1, callback, 0);
   ans = callrpc(hostname, EXAMPLEPROG, EXAMPLEVERS,
      EXAMPLEPROC_CALLBACK, xdr_int, &x, xdr_void, 0);
   if ((enum clnt_stat) ans != RPC_SUCCESS) {
      fprintf(stderr, "call: ");
      clnt_perrno(ans);
      fprintf(stderr, "\n");
   }
   svc_run();
   fprintf(stderr, "Error: svc_run should not return\n");
}
callback(rqstp, transp)
   register struct svc_req *rqstp;
   register SVCXPRT *transp;
{
   switch (rqstp->rq_proc) {
      case 0:
         if (!svc_sendreply(transp, xdr_void, 0)) {
            fprintf(stderr, "err: rusersd\n");
            exit(1);
         }
         exit(0);
      case 1:
         if (!svc_getargs(transp, xdr_void, 0)) {
            svcerr_decode(transp);
            exit(1);
         }
         fprintf(stderr, "client got callback\n");
         if (!svc_sendreply(transp, xdr_void, 0)) {
            fprintf(stderr, "err: rusersd");
            exit(1);
         }
   }
}
/* server */
#include <stdio.h>
#include <RPC/rpc.h>
char *getnewprog();
char hostname[256];
int docallback();
int pnum;                 /* program number for callback routine */
main()
{
   gethostname(hostname, sizeof(hostname));
   registerrpc(EXAMPLEPROG, EXAMPLEVERS,
     EXAMPLEPROC_CALLBACK, getnewprog, xdr_int, xdr_void);
   fprintf(stderr, "server going into svc_run\n");
   intercept(docallback);
   alm_set(sigcode, 10);
   svc_run();
   fprintf(stderr, "Error: svc_run should not return\n");
}
char *
getnewprog(pnump)
   char *pnump;
{
   pnum = *(int *)pnump;
   return NULL;
}
docallback()
{
   int ans;
   ans = callrpc(hostname, pnum, 1, 1, xdr_void, 0,
      xdr_void, 0);
   if (ans != 0) {
      fprintf(stderr, "server: ");
      clnt_perrno(ans);
      fprintf(stderr, "\n");
   }
}
```

4 Port Mapper

This chapter describes the port mapper protocol.

Introduction

The port mapper protocol maps RPC program and version numbers to transport-specific port numbers. This program makes dynamic binding of remote programs possible.

The range of reserved port numbers is small, and the number of potential remote programs is large. By running only the port mapper on a reserved port, the port numbers of other remote programs can be determined by querying the port mapper. The port mapper procedure returns the remote program's port number.

The port mapper procedure sends a response only if the procedure executed successfully. Otherwise, it does not send a response.

The port mapper also aids in broadcast RPC. A given RPC program usually has different port number bindings on different machines, so there is no way to directly broadcast to all of these programs. The port mapper, however, does have a fixed port number. To broadcast to a given program, the client actually sends its message to the port mapper located at the broadcast address. Each port mapper that receives the broadcast then calls the local service specified by the client. When the port mapper gets the reply from the local service, it sends the reply back to the client.

Port Mapper Operation

The port mapper program currently supports UDP and TCP. The port mapper is located on assigned port number 111 on either of these protocols. Following are descriptions of each of the port mapper procedures.

`PMAPPROC_NULL`

This procedure performs no work. By convention, procedure zero of any protocol accepts no parameters and returns no results.

`PMAPPROC_SET`

When a program becomes available on a machine, it registers with the port mapper program on the same machine. The program passes its program number `prog`, version number `vers`, transport protocol number `prot`, and the port `port` on which it awaits service requests. The procedure returns a boolean response with value of TRUE if the procedure successfully established the mapping. Otherwise, it returns FALSE. The procedure refuses to establish a mapping if one already exists for the `prog`, `vers`, and `prot`.

`PMAPPROC_UNSET`

When a program becomes unavailable, it should unregister with the port mapper program on the same machine. The parameters and results have meanings identical to those of `PMAPPROC_SET`. The protocol and port number fields of the parameter are ignored.

`PMAPPROC_GETPORT`

Given a program number `prog`, version number `vers`, and transport protocol number `prot`, this procedure returns the port number on which the program is awaiting call requests. An unregistered program has a 0 filled port value. The `port` field of the parameter is ignored.

`PMAPPROC_DUMP`

This procedure enumerates all entries in the port mapper's database. The procedure takes no parameters and returns a list of program, version, protocol, and port values.

`PMAPPROC_CALLIT`

This procedure allows a caller to call another remote procedure on the same machine without knowing the remote procedure's port number. It is intended for supporting broadcasts to arbitrary remote programs via portmap's well-known port. The parameters `prog`, `vers`, `proc`, and the bytes of `args` are the program number, version number, procedure number, and parameters of the remote procedure.

Port Mapper Protocol Specification (in RPC Language)

```
const PMAP_PORT = 111;      /* portmapper port number */
/* A mapping of (program, version, protocol) to port number */
struct mapping {
   unsigned int prog;
   unsigned int vers;
   unsigned int prot;
   unsigned int port;
};
/* Supported values for the "prot" field */
const IPPROTO_TCP = 6;      /* protocol number for TCP/IP */
const IPPROTO_UDP = 17;     /* protocol number for UDP/IP */
/* A list of mappings */
struct *pmaplist {
   mapping map;
   pmaplist next;
};
/* Arguments to callit */
struct call_args {
   unsigned int prog;
   unsigned int vers;
   unsigned int proc;
   opaque args<>;
};
/* Results of callit */
struct call_result {
   unsigned int port;
   opaque res<>;
};
```

```
/* Port mapper procedures */
program PMAP_PROG {
   version PMAP_VERS {
      void
      PMAPPROC_NULL(void)         = 0;
      bool
      PMAPPROC_SET(mapping)       = 1;
      bool
      PMAPPROC_UNSET(mapping)     = 2;
      unsigned int
      PMAPPROC_GETPORT(mapping)   = 3;
      pmaplist
      PMAPPROC_DUMP(void)         = 4;
      call_result
      PMAPPROC_CALLIT(call_args)  = 5;
   } = 2;
} = 100000;
```

5 External Data Representation

This chapter describes the XDR protocol. XDR is the underlying data-exchange standard used by all RPC services.

The XDR C library functions are described in *OS-9 Networking Programming Reference*.

Introduction

The OS-9 NFS/RPC package uses the External Data Representation (XDR) standard for sending remote procedure calls between such diverse machines as OS-9, UNIX Workstations, VAX, IBM-PCs, large mainframes, and supercomputers. XDR uses a language similar to the C language to concisely describe data formats. XDR can only be used to describe data; it is not a programming language. Use the XDR library routines whenever data is accessed by more than one type of machine.

As long as a machine can translate its data to and from XDR, it can communicate with any other system on the network. When a process on a remote machine wants to look at the data, the remote machine translates the XDR representation of the data to its own local representation. XDR defines a single byte order (big-endian), a single floating-point representation (IEEE), etc. This allows any program running on any machine to use XDR to create portable data. The data is portable because it can be translated from its local representation to XDR.

On OS-9 systems, C programs that use XDR routines must include the file `<RPC/rpc.h>`. This file contains all the necessary interfaces to the XDR system. The C library `rpc.l` contains all the XDR routines.

The XDR Library

The XDR library enables you to read and write arbitrary C constructs in a consistent manner. Therefore, it makes sense to use the library even when the data is not shared among machines on a network.

The XDR library has filter routines for strings, structures, unions, arrays, etc. Using more primitive routines, you can write specific XDR routines to describe arbitrary data structures, including elements of arrays, arms of unions, or objects pointed at from other structures. The structures themselves may contain arrays of arbitrary elements or pointers to other structures.

You can make programs data-portable by replacing the read and write calls with calls to the XDR library routine `xdr_long()`. `xdr_long()` is a filter that knows the standard representation of a long integer in its external form. The following programs use XDR.

Writer

The first program is called `writer`.

```
#include <stdio.h>
#include <RPC/rpc.h>
main()
               /* writer.c */
{
   XDR xdrs;
   long i;
   xdrstdio_create(&xdrs, stdout, XDR_ENCODE);
   for (i = 0; i < 8; i++) {
      if (!xdr_long(&xdrs, &i)) {
            fprintf(stderr, "failed!\n");
            exit(1);
        }
    }
}
```

Reader

The second program is called `reader`.

```
#include <stdio.h>
#include <RPC/rpc.h>
main()
              /* reader.c */
{
    XDR xdrs;
    long i, j;
    xdrstdio_create(&xdrs, stdin, XDR_DECODE);
    for (j = 0; j < 8; j++) {
        if (!xdr_long(&xdrs, &i)) {
            fprintf(stderr, "failed!\n");
            exit(1);
        }
        printf("%ld ", i);
    }
    printf("\n");
}
```

Explaining Writer/Reader Examples

Arbitrary data structures present portability problems, particularly with respect to alignment and pointers. Alignment on word boundaries may cause the size of a structure to vary from machine to machine. Pointers have no meaning outside the machine where they are defined.

Examine the programs. Members of the XDR stream creation routines treat the stream of bits differently. In the example programs, data is manipulated using standard I/O routines. Therefore, `xdrstdio_create()` is used. The parameters to XDR stream creation routines vary according to their function. In the example, `xdrstdio_create()` takes a pointer to an XDR structure that it initializes, a pointer to a file that the input or output is performed on, and the operation.

The above operation may be either:

`XDR_ENCODE` for serializing in the `writer` program.

`XDR_DECODE` for deserializing in the `reader` program.

RPC users never need to create XDR streams. The RPC system creates these streams before passing them to the users.

The `xdr_long()` primitive is characteristic of most XDR library primitives and all client XDR routines. The routine returns FALSE (0) if it fails and TRUE (1) if it succeeds. Also, for each data type, `xxx`, there is an associated XDR routine of the form:

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

In this example, `xxx` is long. The corresponding XDR routine is a primitive, `xdr_long()`. The client could define an arbitrary structure `xxx`. In this case, the client would also supply the routine `xdr_xxx()` which describes each field by calling XDR routines of the appropriate type. In all cases, `xdrs` can be treated as an opaque handle and passed to the primitive routines.

Serializing and Deserializing Data

XDR routines are direction independent. That is, the same routine can be called to either serialize or deserialize data. This almost guarantees that serialized data can also be deserialized. This is possible because the address of an object is passed rather than the object itself. Only in the case of deserialization is the object modified.

Assume that a person's gross assets and liabilities are to be exchanged among processes. Also assume that these values are important enough to warrant their own data type:

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

The corresponding XDR routine describing this structure is:

```
bool_t  /* TRUE is success, FALSE is failure */
xdr_gnumbers(xdrs, gp)
   XDR *xdrs;
   struct gnumbers *gp;
{
   if (xdr_long(xdrs, &gp->g_assets) &&
       xdr_long(xdrs, &gp->g_liabilities))
      return(TRUE);
   return(FALSE);
}
```

The parameter `xdrs` is never inspected or modified. It is only passed to the subcomponent routines. It is imperative to inspect the return value of each XDR routine call and to immediately return FALSE if the subroutine fails.

This example also shows that the type `bool_t` is declared as an integer in which the only values are TRUE (1) and FALSE (0). The following definitions are used throughout this chapter:

```
#define bool_t int
#define TRUE    1
#define FALSE   0
#define enum_t int/* enum_t used for generic enums */
```

Keeping these conventions in mind, you can rewrite `xdr_gnumbers():`

```
xdr_gnumbers(xdrs, gp)
   XDR *xdrs;
   struct gnumbers *gp;
{
   return(xdr_long(xdrs, &gp->g_assets) &&
      xdr_long(xdrs, &gp->g_liabilities));
}
```

Both coding styles are used.

XDR Library Primitives

This section gives a synopsis for each XDR primitive. It starts with the basic data types and moves on to constructed data types. It also examines the XDR utilities. The interface to these primitives and utilities is defined in the include file `<RPC/xdr.h>`, which is automatically included by `<RPC/rpc.h>`.

Number Filters

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

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

Specifically, the eight primitives are:

```
bool_t xdr_char(xdrs, cp)
   XDR *xdrs;            /* an XDR stream handle */
   char *cp;             /* address of character to provide/receive
data */

bool_t xdr_u_char(xdrs, ucp)
   XDR *xdrs;            /* an XDR stream handle */
   unsigned char *ucp;   /* address of unsigned character */
```

```
bool_t xdr_int(xdrs, ip)
   XDR *xdrs;             /* an XDR stream handle */
   int *ip;               /* address of integer */

bool_t xdr_u_int(xdrs, up)
   XDR *xdrs;             /* an XDR stream handle */
   unsigned *up;          /* address of unsigned integer */

bool_t xdr_long(xdrs, lip)
   XDR *xdrs;             /* an XDR stream handle */
   long *lip;             /* address of long integer */

bool_t xdr_u_long(xdrs, lup)
   XDR *xdrs;             /* an XDR stream handle */
   u_long *lup;           /* address of long unsigned integer */

bool_t xdr_short(xdrs, sip)
   XDR *xdrs;             /* an XDR stream handle */
   short *sip;            /* address of short integer */

bool_t xdr_u_short(xdrs, sup)
   XDR *xdrs;             /* an XDR stream handle */
   u_short *sup;             /* address of short unsigned integer */
```

All routines return TRUE if they complete successfully, and FALSE if an error occurs.

Floating Point Filters

The XDR library also provides primitive routines for C's floating point types:

```
bool_t xdr_float(xdrs, fp)
   XDR *xdrs;                  /* XDR stream handle */
   float *fp;                  /* address of floating point number */
bool_t xdr_double(xdrs, dp)
   XDR *xdrs;                  /* XDR stream handle */
   double *dp;                 /* address of double */
```

If successful, all routines return TRUE. Otherwise, they return FALSE.

Enumeration Filters

The XDR library provides a primitive for generic enumerations. The primitive assumes that a C enumerator has the same representation inside the machine as a C integer. The boolean type is an important instance of the enumerator. The external representation of a boolean is always TRUE (1) or FALSE (0).

```
#define bool_tint
#define FALSE0
#define TRUE1
#define enum_t int
```

```
bool_t xdr_enum(xdrs, ep)
   XDR *xdrs;              /* XDR stream handle */
   enum_t *ep;             /* address of enumerator */
bool_t xdr_bool(xdrs, bp)
   XDR *xdrs;              /* XDR stream handle */
   bool_t *bp;             /* address of boolean */
```

FALSE is returned if the number of characters exceeds `maxlength`. Otherwise, TRUE is returned. The value of `maxlength` is usually specified by a protocol.

The boolean type is an important instance of the enumerator. The external representation of a boolean is always TRUE (1) or FALSE (0).

No Data

Occasionally, an XDR routine must be supplied to the RPC system even when no data is passed or required. The library provides such a routine:

```
bool_t xdr_void();  /* always returns TRUE */
```

Constructed Data Type Filters

Constructed or compound data type primitives require more parameters and perform more complicated functions than the primitives already discussed. This section includes primitives for the following:

- Strings
- Arrays
- Unions
- Pointers to structures.

You may use constructed data type primitives to aid memory management. In many cases, memory is allocated when deserializing data with `XDR_DECODE`. The XDR operation `XDR_FREE` provides a way to deallocate memory. The three XDR directional operations are:

- `XDR_ENCODE`
- `XDR_DECODE`
- `XDR_FREE`

Strings

In C, a string is defined as a sequence of bytes terminated by a null byte. The null byte is not considered when calculating string length. However, when a string is passed or manipulated, a pointer to the null byte is used. Therefore, the XDR library defines a string to be a char and not a sequence of characters. The external representation of a string is vastly different from the internal representation. `xdr_string()` converts between the two representations:

```
bool_t xdr_string(xdrs, sp, maxlength)
   XDR *xdrs;          /* XDR stream handle */
   char **sp;         /* pointer to a string */
   u_int maxlength;   /* max. no. bytes allowed for encoding/decoding
*/
```

The value of `maxlength` is usually specified by a protocol. For example, a protocol specification may limit a file name to 255 characters. `xdr_string()` returns FALSE if the number of characters exceeds `maxlength`. Otherwise, it returns TRUE.

`xdr_string()` behaves like the other routines discussed in this section. The direction `XDR_ENCODE` is easiest to understand. The parameter `sp` points to a string of a certain length. If the string does not exceed `maxlength`, the bytes are serialized.

The effect of deserializing a string is subtle. First, the length of the incoming string is determined. It must not exceed `maxlength`. Next, `sp` is dereferenced. If the the value is null, a string of the appropriate length is allocated and `*sp` is set to this string. If the original value of `*sp` is non-null, the XDR package assumes that a target area has been allocated. The target area can hold strings no longer than `maxlength`. In either case, the string is decoded in the target area. The routine appends a null character to the string.

In the `XDR_FREE` operation, the string is obtained by dereferencing `sp`. If the string is not null, it is freed and `*sp` is set to null. In this operation, `xdr_string()` ignores `maxlength`.

Byte Arrays

Often, variable-length arrays of bytes are preferable to strings. Byte arrays differ from strings in the following ways:

- The length of the array (the byte count) is explicitly located in an unsigned integer.
- The byte sequence is not terminated by a null character.
- The external representation of the bytes is the same as the internal representation.

The primitive `xdr_bytes()` converts between the internal and external representations of byte arrays:

```
bool_t xdr_bytes(xdrs, bpp, lp, maxlength)
    XDR *xdrs;            /* XDR stream handle */
    char **bpp;           /* pointer to a string */
    u_int *lp;            /* byte length */
    u_int maxlength;     /* max. no. of bytes for encoding/decoding
*/
```

The length of the byte area is obtained by dereferencing `lp` when serializing. `*lp` is set to the byte length when deserializing.

Arrays

The XDR library package provides a primitive for handling arrays of arbitrary elements. `xdr_bytes()` treats a subset of generic arrays in which the size of the array elements is known to be 1, and the external description of each element is built-in. `xdr_array()` is called to encode or decode each element of the array.

```
bool_t
xdr_array(xdrs, ap, lp, maxlength, elementsiz, xdr_element)
    XDR *xdrs;         /* opaque handle */
    char **ap;         /* the address of the pointer to the array */
    u_int *lp;         /* length of array after deserialization */
    u_int maxlength; /* maximum number of elements allows in an array
*/
    u_int elementsiz;   /* byte size of each element in the array */
    bool_t (*xdr_element)();
```

If `*ap` is null when the array is being deserialized, XDR allocates an array of the appropriate size and sets `*ap` to that array. The element count of the array is obtained from `*lp` when the array is serialized. `*lp` is set to the array length when the array is deserialized. The routine `xdr_element()` serializes, deserializes, or frees each element of the array.

Before defining more constructed data types, three examples are presented.

Implementing Arrays Example A

You can identify a user on a networked machine by the machine name, the user's `uid`, and the group numbers to which the user belongs. You could code a structure with this information and its associated XDR routine like this:

```
struct netuser {
    char    *nu_machinename;
    int     nu_uid;
    u_int   nu_glen;
    int     *nu_gids;
};
#define NLEN 255    /* machine names < 256 chars */
#define NGRPS 20    /* user cannot be in > 20 groups */
bool_t
xdr_netuser(xdrs, nup)
    XDR *xdrs;
    struct netuser *nup;
{
    return(xdr_string(xdrs, &nup->nu_machinename, NLEN) &&
        xdr_int(xdrs, &nup->nu_uid) &&
        xdr_array(xdrs, &nup->nu_gids, &nup->nu_glen,
        NGRPS, sizeof (int), xdr_int));
}
```

Implementing Arrays Example B

You could implement a party of network users as an array of `netuser` structure. The declaration and the associated XDR routines are as follows:

```
struct party {
    u_int p_len;
    struct netuser *p_nusers;
};
#define PLEN 500    /* maximum number of users in a party */
bool_t
xdr_party(xdrs, pp)
    XDR *xdrs;
    struct party *pp;
{
    return(xdr_array(xdrs, &pp->p_nusers, &pp->p_len, PLEN,
        sizeof (struct netuser), xdr_netuser));
}
```

Implementing Arrays Example C

You can combine the well-known parameters to `main`, `argc`, and `argv` into a structure. An array of these structures can make up a history of commands. The declarations and XDR routines might look like:

```
struct cmd {
    u_int c_argc;
    char **c_argv;
};
#define ALEN 1000   /* args cannot be > 1000 chars */
#define NARGC 100   /* commands cannot have > 100 args */
struct history {
    u_int h_len;
    struct cmd *h_cmds;
};
#define NCMDS 75    /* history is no more than 75 commands */
bool_t
xdr_wrap_string(xdrs, sp)
    XDR *xdrs;
    char **sp;
{
    return(xdr_string(xdrs, sp, ALEN));
}
bool_t
xdr_cmd(xdrs, cp)
    XDR *xdrs;
    struct cmd *cp;
{
    return(xdr_array(xdrs, &cp->c_argv, &cp->c_argc, NARGC,
        sizeof (char *), xdr_wrap_string));
}
bool_t
xdr_history(xdrs, hp)
    XDR *xdrs;
    struct history *hp;
{
    return(xdr_array(xdrs, &hp->h_cmds, &hp->h_len, NCMDS,
        sizeof (struct cmd), xdr_cmd));
}
```

The most confusing part of this example is that the routine `xdr_wrap_string()` is needed to package the `xdr_string()` routine. `xdr_array()` only passes two parameters to the array element description routine. `xdr_wrap_string()` supplies the third parameter to `xdr_string()`.

Opaque Data

In some protocols, handles are passed from a server to a client. The client later passes the handle back to the server. Handles are never inspected by clients; they are merely obtained and submitted. That is, handles are **opaque**. `xdr_opaque()` describes fixed-sized, opaque bytes.

```
bool_t xdr_opaque(xdrs, p, len)
    XDR *xdrs;
    char *p;  /* location of the bytes */
    u_int len;/* number of bytes in the opaque object */
```

By definition, the actual data contained in the opaque object are not machine portable.

Fixed-Sized Arrays

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

```
#define NLEN 255    /* machine names must be less than 256 chars */
#define NGRPS 20    /* user belongs to exactly 20 groups */
struct netuser {
    char *nu_machinename;
    int nu_uid;
    int nu_gids[NGRPS];
};
bool_t
xdr_netuser(xdrs, nup)
    XDR *xdrs;
    struct netuser *nup;
{
    int i;
    if (!xdr_string(xdrs, &nup->nu_machinename, NLEN))
        return(FALSE);
    if (!xdr_int(xdrs, &nup->nu_uid))
        return(FALSE);
    if (!xdr_vector(xdrs, nup->nu_gids, NGRPS, sizeof(int),
        xdr_int)) {
            return(FALSE);
    }
    return(TRUE);
}
```

Discriminated Unions

The XDR library supports discriminated unions. A discriminated union is a C union and an `enum_t` value that selects an arm of the union.

```
struct xdr_discrim {
    enum_t value;
    bool_t (*proc)();
};
bool_t xdr_union(xdrs, dscmp, unp, arms, defaultarm)
    XDR *xdrs;
    enum_t *dscmp;
    char *unp;
    struct xdr_discrim *arms;
    bool_t (*defaultarm)();  /* may equal NULL */
```

First, the routine translates the discriminant of the union located at `*dscmp`. The discriminant is always an `enum_t`. Next, the union located at `*unp` is translated. The parameter `arms` is a pointer to an array of `xdr_discrim()` structures. Each structure contains an ordered pair of [`value, proc`]. If the union's discriminant is equal to the associated value, the `proc` is called to translate the union. The end of the `xdr_discrim()` structure array is denoted by a routine of value null (0). If the discriminant is not found in the `arms` array, the `defaultarm` procedure is called if it is non-null. Otherwise, the routine returns FALSE.

Discriminated Union Example

Suppose the type of a union may be an integer, a character pointer (a string), or a `gnumbers` structure. Also, assume the union and its current type are declared in a structure. The declaration is:

```
enum utype { INTEGER=1, STRING=2, GNUMBERS=3 };
struct u_tag {
    enum utype utype;   /* the union's discriminant */
    union {
        int ival;
        char *pval;
        struct gnumbers gn;
    } uval;
};
```

The following constructs and XDR procedure (de)serialize the discriminated union:

```
struct xdr_discrim u_tag_arms[4] = {
    { INTEGER, xdr_int },
    { GNUMBERS, xdr_gnumbers }
    { STRING, xdr_wrap_string },
    { __dontcare__, NULL }
    /* always terminate arms with a NULL xdr_proc */
}
bool_t
xdr_u_tag(xdrs, utp)
    XDR *xdrs;
    struct u_tag *utp;
{
    return(xdr_union(xdrs, &utp->utype, &utp->uval,
        u_tag_arms, NULL));
}
```

`xdr_gnumbers()` was defined in an earlier example. The default `arm` parameter to `xdr_union()` is null in this example. Therefore, the value of the union's discriminant may legally take on only values listed in the `u_tag_arm` array. This example also demonstrates that you do not need to sort the elements of the `arm` array.

The values of the discriminant may be sparse, although in this example they are not. It is always a good practice to assign explicit integer values to each element of the discriminant's type. This practice both documents the external representation of the discriminant and guarantees that different C compilers provide identical discriminant values.

Pointers

In C, placing pointers to a structure within a structure is often convenient. `xdr_reference()` makes it easy to serialize, deserialize, and free these referenced structures.

```
bool_t xdr_reference(xdrs, pp, size, proc)
    XDR *xdrs;        /* opaque handle */
    char **pp;        /* address of the pointer to the structure */
    u_int ssize;      /* size in bytes of the structure */
    bool_t (*proc)();/* the XDR routine that describes the structure
*/
```

When decoding data, storage is allocated if `*pp` is null.

There is no need for a primitive `xdr_struct()` to describe structures within structures because pointers are always sufficient.

Pointer Example

Suppose a structure contains a person's name and a pointer to a `gnumbers` structure containing the person's gross assets and liabilities. The construct is:

```
struct pgn {
    char *name;
    struct gnumbers *gnp;
};
```

The corresponding XDR routine for this structure is:

```
bool_t
xdr_pgn(xdrs, pp)
    XDR *xdrs;
    struct pgn *pp;
{
    if (xdr_string(xdrs, &pp->name, NLEN) &&
      xdr_reference(xdrs, &pp->gnp,
      sizeof(struct gnumbers), xdr_gnumbers))
        return(TRUE);
    return(FALSE);
}
```

In many applications, C programmers attach double meaning to the values of a pointer. Typically, the value null (or zero) means data is not needed, yet some application-specific interpretation applies. In essence, the C programmer is efficiently encoding a discriminated union by overloading the interpretation of the pointer's value.

In the preceding example, a null pointer value for `gnp` could indicate that the person's assets and liabilities are unknown. That is, the pointer value encodes whether or not the data is known and if it is known, where it is located in memory. Linked lists are an extreme example of the use of application-specific pointer interpretation.

`xdr_reference()` cannot and does not attach any special meaning to a null-value pointer during serialization. That is, passing an address of a pointer in which the value is null to `xdr_reference()` when serializing data generally causes a bus error.

`xdr_pointer()` correctly handles null pointers. For more information about its use, see the section on linked lists.

Non-filter Primitives

You can manipulate XDR streams with the primitives discussed in this section.

```
u_int xdr_getpos(xdrs)
    XDR *xdrs;
bool_t xdr_setpos(xdrs, pos)
    XDR *xdrs;
    u_int pos;
xdr_destroy(xdrs)
    XDR *xdrs;
```

`xdr_getpos()` returns an unsigned integer that describes the current position in the data stream.

In some XDR streams, the returned value of `xdr_getpos()` is meaningless. The routine returns a -1 in this case, which should be a legitimate value.

`xdr_setpos()` sets a stream position to `pos`.

In some XDR streams, setting a position is impossible. In such cases, `xdr_setpos()` returns FALSE. This routine also fails if the requested position is out-of-bounds. The definition of bounds varies from stream to stream.

`xdr_destroy()` destroys the XDR stream. Using the stream after calling this routine is undefined.

XDR Operation Directions

You may want to optimize XDR routines by taking advantage of the direction of the operation `XDR_ENCODE`, `XDR_DECODE`, or `XDR_FREE`. The value `xdrs->x_op` always contains the direction of the XDR operation.

XDR Stream Access

You obtain an XDR stream by calling the appropriate creation routine. These creation routines accept parameters that are tailored to the specific properties of the stream. Streams currently exist for (de)serialization of data to or from standard I/O file streams, TCP/IP connections, and OS-9 disk and pipe files and memory.

Standard I/O Streams

You can interface XDR streams to standard I/O using the `xdrstdio_create()` routine as follows:

```
#include <stdio.h>
#include <RPC/rpc.h>    /* XDR streams part of RPC */
void
xdrstdio_create(xdrs, fp, x_op)
    XDR *xdrs;          /* opaque handle */
    FILE *fp;           /* open file */
    enum xdr_op x_op;   /* an XDR direction */
```

`xdrstdio_create()` initializes an XDR stream pointed to by `xdrs`. The XDR stream interfaces with the standard I/O library.

Memory Streams

Memory streams allow the streaming of data into or out of a specified area of memory:

```
#include <RPC/rpc.h>
void
xdrmem_create(xdrs, addr, len, x_op)
```

```
    XDR *xdrs;        /* opaque handle */
    char *addr;       /* pointer to memory location */
    u_int len;        /* length in bytes of the memory */
    enum xdr_op x_op;   /* an XDR direction */
```

`xdrmem_create()` initializes an XDR stream in local memory. The UDP/IP implementation of RPC uses `xdrmem_create()`. Complete call or result messages are built into memory before calling the `sendto` system routine.

Record (TCP/IP) Streams

A record stream is an XDR stream built on top of a record marking standard that is built on top of the TCP/IP connection interface.

```
#include <RPC/rpc.h>        /* xdr streams part of rpc */
xdrrec_create(xdrs,
  sendsize, recvsize, iohandle, readproc, writeproc)
    XDR *xdrs;              /* opaque handle */
    u_int sendsize,         /* size in bytes of output buffer */
          recvsize;         /* size in bytes of input buffer */
    char *iohandle;         /* opaque parameter */
    int (*readproc)(),      /* fills buffer */
        (*writeproc)();     /* flushes buffer */
```

`xdrrec_create()` provides an XDR stream interface that allows for a bi-directional, arbitrarily long sequence of records. The contents of the records are meant to be data in XDR form. The stream primarily interfaces RPC to TCP connections. However, it can stream data into or out of normal OS-9 files.

The stream does its own data buffering similar to that of standard I/O. If the values of `sendsize` and `recvsize` are zero (0), predetermined defaults are used. The function and behavior of these routines are similar to the OS-9 system calls `read` and `write`. However, the first parameter for both routines is the opaque parameter, `iohandle`. The other two parameters `buf` and `nbytes` and the results (byte count) are identical to the system routines. If `xxx` is `readproc` or `writeproc`, it has the following form:

```
/* returns the actual number of bytes transferred. -1 is an error. */
int
xxx(iohandle, buf, len)
    char *iohandle;
    char *buf;
    int nbytes;
```

The XDR stream provides a way to delimit records in the byte stream. The primitives that are specific to record streams are as follows:

```
bool_t
xdrrec_endofrecord(xdrs, flushnow)
    XDR *xdrs;
    bool_t flushnow;
bool_t
```

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

`xdrrec_endofrecord()` causes the current outgoing data to be marked as a record. If the parameter `flushnow` is TRUE, the stream's `writeproc` is called. Otherwise, `writeproc` is called when the output buffer has been filled.

`xdrrec_skiprecord()` causes the position of an input stream to move past the current record boundary and onto the beginning of the next record in the stream.

If the stream's input buffer contains no more data, `xdrrec_eof()` returns TRUE. However, more data may be located beneath the file descriptor.

XDR Stream Implementation

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

The XDR Object

The following structure defines the interface to an XDR stream:

```
enum xdr_op { XDR_ENCODE=0, XDR_DECODE=1, XDR_FREE=2 };
typedef struct {
    enum xdr_op x_op;           /* operation; fast added parameter */
    struct xdr_ops {
        bool_t  (*x_getlong)();  /* get long from stream */
        bool_t  (*x_putlong)();  /* put long to stream */
        bool_t  (*x_getbytes)(); /* get bytes from stream */
        bool_t  (*x_putbytes)(); /* put bytes to stream */
        u_int   (*x_getpostn)(); /* return stream offset */
        bool_t  (*x_setpostn)(); /* reposition offset */
        caddr_t (*x_inline)();   /* pointer to buffered data */
        VOID    (*x_destroy)();  /* free private area */
    } *x_ops;
    caddr_t     x_public;        /* users' data */
    caddr_t     x_private;       /* pointer to private data */
    caddr_t     x_base;         /* private for position information
*/
    int         x_handy;         /* extra private word */
} XDR;
```

The `x_op` field is the current operation being performed on the stream. This field is important to the XDR primitives. It should not affect a stream's implementation, and a stream's implementation should not depend on this value. The fields `x_private`, `x_base`, and `x_handy` are private to the implementation of a particular stream. The field `x_public` is for the XDR client and should never be used by the XDR stream implementations or the XDR primitives.

Macros for accessing the operations `x_getpostn()`, `x_setpostn()`, and `x_destroy()` have already been defined. The operation `x_inline()` takes an `XDR *` and an unsigned integer as parameters. The unsigned integer is a byte count. The routine returns a pointer to a portion of the stream's internal buffer. The caller can use the buffer segment for any purpose. From the stream's point of view, the bytes in the buffer segment have been consumed or placed. `x_inline()` may return null if it cannot return a buffer segment of the requested size. Use of the resulting buffer is not data-portable.

`x_getbytes()` blindly receives sequences of bytes from the underlying stream. `x_putbytes()` is the opposite; it blindly places sequences of bytes into the underlying stream. If successful, these routines return TRUE. Otherwise, they return FALSE. The routines have identical parameters (replace the `xxx`):

```
bool_t
xxxbytes(xdrs, buf, bytecount)
   XDR *xdrs;
   char *buf;
   u_int bytecount;
```

`x_getlong()` receives long numbers from the data stream, and `x_putlong()` places long numbers to the data stream. These routines translate the numbers between the machine representation and the standard external representation. The OS-9 Internet functions `htonl` and `ntohl` can be helpful in accomplishing this.

The higher-level XDR implementation assumes that signed and unsigned long integers contain the same number of bits and that nonnegative integers have the same bit representations as unsigned integers.

If successful, these routines return TRUE. Otherwise, they return FALSE. They have identical parameters (replace the `xxx`):

```
bool_t
xxxlong(xdrs, lp)
   XDR *xdrs;
   long *lp;
```

Implementors of new XDR streams must make an XDR structure with new operation routines available to clients using some kind of create routine.

Linked Lists

The last example in the section on pointers presented a C data structure and its associated XDR routines for an individual's gross assets and liabilities. The example is duplicated below:

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

```
bool_t
xdr_gnumbers(xdrs, gp)
   XDR *xdrs;
   struct gnumbers *gp;
{
   if (xdr_long(xdrs, &(gp->g_assets)))
      return(xdr_long(xdrs, &(gp->g_liabilities)));
   return(FALSE);
}
```

To implement a linked list of such information, you could construct a data structure:

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

Think of the head of the linked list as the data object. That is, the head is not merely a convenient shorthand for a structure. Similarly, the `gn_next` field indicates whether or not the object has terminated. Unfortunately, if the object continues, the `gn_next` field is also the address of where it continues. The link addresses carry no useful information when the object is serialized.

The XDR data description of this linked list is described by the recursive declaration of `gnumbers_list`:

```
struct gnumbers {
   int g_assets;
   int g_liabilities;
};
struct gnumbers_node {
   gnumbers gn_numbers;
   gnumbers_list gn_next;
};



union gnumbers_list switch (bool more_data) {
case TRUE:
   gnumbers_node node;
case FALSE:
   void;
};
```

In this description, the boolean indicates whether more data follows.

- If the boolean is FALSE, it is the last data field of the structure.

- If it is TRUE, it is followed by a `gnumbers` structure and recursively by a `gnumbers_list`.

The C declaration has no boolean explicitly declared in it (although the `gn_next` field implicitly carries the information), while the XDR data description has no pointer explicitly declared in it.

Hints for writing the XDR routines for a `gnumbers_list` follow easily from the previous XDR description.

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

The unfortunate side effect of using XDR on a list with these routines is that the C stack grows linearly with respect to the number of nodes in the list. This is due to recursion. The following routine combines the preceding mutually recursive routines into a single, non-recursive routine.

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

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

This routine performs the following:

1. Determines if there is more data. This indicates whether this boolean information can be serialized. This statement is not needed in the `XDR_DECODE` case because the value of `more_data` is not known until it is deserialized in the next statement.
2. The next statement XDRs the `more_data` field of the XDR union. If there is no more data, the last pointer is set to null to indicate the end of the list. A value of TRUE is returned.

Setting the pointer to null is only important in the `XDR_DECODE` case. It is already null in the `XDR_ENCODE` and `XDR_FREE` cases.

3. If the direction is `XDR_FREE`, `nextp` indicates the location of the next pointer in the list. This is performed now so that `gnp` is not dereferenced to find the location of the next list item. After the next statement, the pointer `gnp` will be freed and no longer valid. This does not work for all directions because `gnp` is not set until the next statement in the `XDR_DECODE` direction.
4. XDR is used on the data in the node via the primitive `xdr_reference()`. `xdr_reference()` is similar to `xdr_pointer()`, but it does not send the boolean indicating whether there is more data. `xdr_reference()` is used because this information is already used by XDR. Notice that the `xdr` routine passed is not the same type as an element in the list. The routine passed is `xdr_gnumbers()` for use by XDR `gnumbers`, but each element in the list is actually of type `gnumbers_node`. `xdr_gnumbers_node` is not passed because it is recursive. Instead, use `xdr_gnumbers`, which uses XDR on all of the non-recursive part.

This works only if the `gn_numbers` field is the first item in each element so that their addresses are identical when passed to `xdr_reference`.

5. `gnp` is updated to point to the next item in the list.
   - If the direction is `XDR_FREE`, set `gnp` to the previously saved value.
   - Otherwise, dereference `gnp` to get the proper value.

Although harder to understand than the recursive version, this non-recursive routine never causes the C stack to overflow. It also runs more efficiently because some of the procedure call overhead has been removed.

XDR Data Types

Each of the following sections describes a data type defined in the XDR standard, shows how it is declared in the language, and includes a graphic illustration of its encoding.

For each data type in the language, a general paradigm declaration is shown.

- Angle brackets (< and >) denote variable length sequences of data.
- Square brackets ([and]) denote fixed-length sequences of data.
- `n`, `m`, and `r` denote integers.

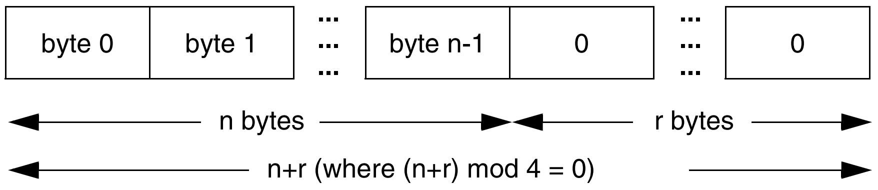
For some data types, more specific examples are included.

Basic Block Size

The representation of all items requires a multiple of four bytes (or 32 bits) of data. The bytes are numbered **0** through `n`-**1**. The bytes are read or written to some byte stream such that byte `m` always precedes byte `m`+**1**. If the `n` bytes needed to contain the data are not a multiple of four, they are followed by enough (0 to 3) residual zero bytes (`r`) to make the total byte count a multiple of 4.

In the following illustrations, each box depicts one byte. Ellipses (...) between boxes show zero or more additional bytes where required. For example, Figure 5-1. illustrates a block.

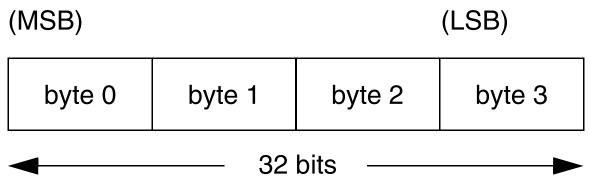
Figure 5-1. Sample Block



Integer

An XDR signed integer is a 32-bit datum that encodes an integer in the range [-2147483648,2147483647]. The integer is represented in two's complement notation. The most significant byte (MSB) and least significant byte (LSB) are 0 and 3, respectively. Integers are declared as shown in Figure 5-2..

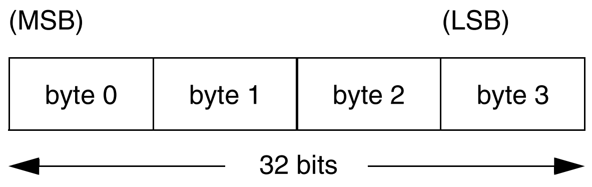
Figure 5-2. Sampe Integer



Unsigned Integer

An XDR unsigned integer is a 32-bit datum that encodes a non-negative integer in the range [0,4294967295]. It is represented by an unsigned binary number whose most and least significant bytes are 0 and 3, respectively. An unsigned integer is declared as shown in Figure 5-3..

Figure 5-3. Sample Unsigned Integer



Enumeration

Enumerations have the same representation as signed integers. You can use enumerations to describe subsets of integers. Declare enumerated data as follows:

```
enum { name-identifier = constant, ... } identifier;
```

For example, three colors (red, yellow, and blue) could be described by an enumerated type:

```
enum { RED = 2, YELLOW = 3, BLUE = 5 } colors;
```

- It is an error to encode as an `enum` any integer other than those that have been given assignments in the `enum` declaration.

Boolean

Booleans occur frequently enough to warrant their own explicit type in the standard. Booleans are declared as follows:

```
bool identifier;
```

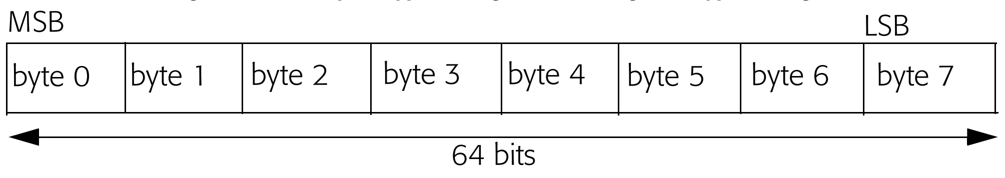
This is equivalent to:

```
enum { FALSE = 0, TRUE = 1 } identifier;
```

Hyper Integer and Unsigned Hyper Integer

The standard also defines 64-bit (8-byte) numbers called hyper integer and unsigned hyper integer. Their representations are extensions of the previously defined integer and unsigned integer. They are represented in two's complement notation. The most and least significant bytes are 0 and 7, respectively. Their declaration is shown in Figure 5-4..

Figure 5-4. Sample Hyper Integer and Unsigned Hyper Integer



Floating-Point

The standard defines the floating-point data type `float` (32 bits or 4 bytes). The encoding used is the IEEE standard for normalized single-precision, floating-point numbers. Table 5-1 on page 96 describes the single-precision, floating-point number.

Table 5-1. Single-Precision, Floating-Point Number

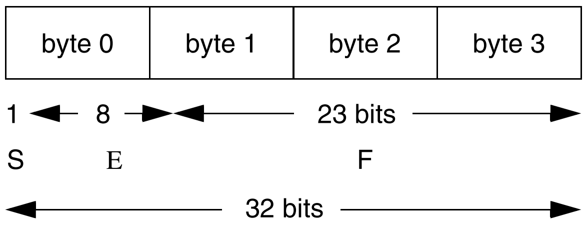
Field	Description
S	The sign of the number. Values 0 and 1 represent positive and negative, respectively. This field contains 1 bit.
E	The exponent of the number, base 2. This field contains 8 bits. The exponent is biased by 127.
F	The fractional part of the number's mantissa, base 2. This field contains 23 bits.

Therefore, the floating-point number is described by:

```
(-1)**S * 2**(E-Bias) * 1.F
```

It is declared as shown in Figure 5-5..

Figure 5-5. Floating-Point Number



The most and least significant bits of a single-precision, floating-point number are 0 and 31, respectively. The beginning, and most significant, bit offsets of `S`, `E`, and `F` are 0, 1, and 9, respectively.

These numbers refer to the mathematical positions of the bits and NOT to their actual physical locations. Their actual physical location varies from medium to medium.

Double-Precision Floating-Point

The XDR standard defines the encoding for the double-precision, floating-point data type **double** (64 bits or 8 bytes). The encoding used is the IEEE standard for normalized double-precision, floating-point numbers. The standard encodes the following fields which describe the double-precision, floating-point number. These fields are shown in Table 5-2 on page 97.

Table 5-2. Double-Precision, Floating-Point Number Fields

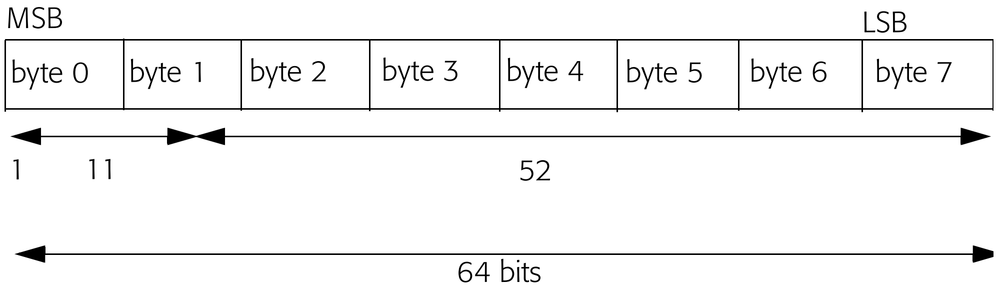
Field	Description
S	The sign of the number. Values 0 and 1 represent positive and negative, respectively. This field contains 1 bit.
E	The exponent of the number, base 2. This field contains 11 bits. The exponent is biased by 1023.
F	The fractional part of the number's mantissa, base 2. This field contains 52 bits.

Therefore, the floating-point number is described by:

```
(-1)**S * 2**(E-Bias) * 1.F
```

It is declared as shown in Figure 5-6..

Figure 5-6.



The most and least significant bits of a double-precision, floating-point number are 0 and 63, respectively. The beginning, and most significant, bit offsets of `S`, `E`, and `F` are 0, 1, and 12, respectively.

Mathematical positions of the bits, NOT their actual physical locations, are represented. The physical locations vary from medium to medium.

Consult the IEEE specifications concerning the encoding for signed 0, signed infinity (overflow), and denormalized numbers (underflow). According to IEEE specifications, the "NaN" (not a number) is system dependent and should not be used externally.

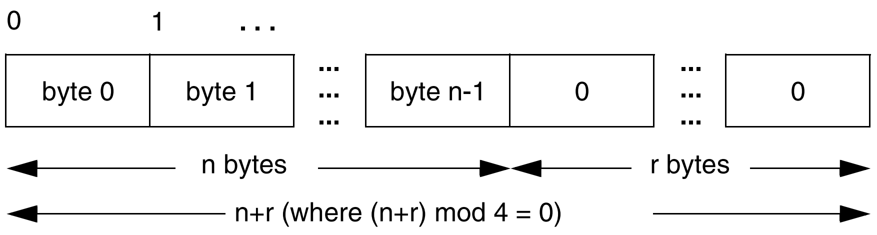
Fixed-Length Opaque Data

At times, fixed-length uninterpreted data needs to be passed between machines. This data is called `opaque` and is declared as follows:

```
opaque identifier[n];
```

The constant `n` is the (static) number of bytes necessary to contain the opaque data. If `n` is not a multiple of four, the `n` bytes are followed by enough (0 to 3) residual 0 bytes, `r`, to make the total byte count of the opaque object a multiple of four. An example of fixed-length opaque data is shown in Figure 5-7..

Figure 5-7. Fixed Length Opaque Data



Variable-Length Opaque Data

The standard also provides for variable-length (counted) opaque data, defined as a sequence of `n` (numbered 0 through `n-1`) arbitrary bytes to be the number `n` encoded as an unsigned integer and followed by the `n` bytes of the sequence.

Byte `m` of the sequence always precedes byte `m+1` of the sequence, and byte 0 of the sequence always follows the sequence's length (count). Enough (0 to 3) residual 0 bytes, `r`, to make the total byte count a multiple of four are added. Declare variable-length opaque data as follows:

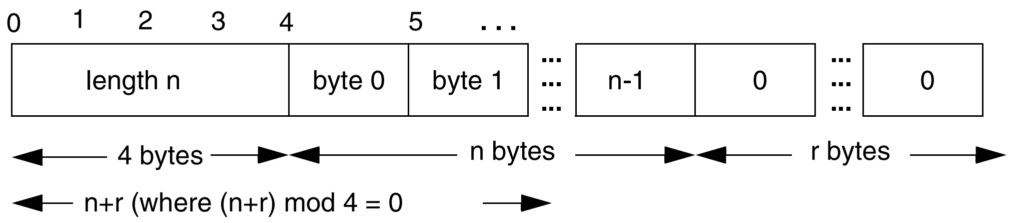
```
opaque identifier<m>;
```

The constant `m` denotes an upper bound of the number of bytes that the sequence may contain. If `m` is not specified, it is assumed to be (2**32)-1, the maximum length. The constant `m` is normally found in a protocol specification. For example, a filing protocol may state that the maximum data transfer size is 8192 bytes:

```
opaque filedata<8192>;
```

This can be illustrated as shown in Figure 5-8..

Figure 5-8. Variable-Length Opaque Data



It is an error to encode a length greater than the maximum described in the specification.

String

The XDR standard defines a string of `n` (numbered 0 through `n-1`) ASCII bytes as the number `n` encoded as an unsigned integer, and followed by the `n` bytes of the string. Byte `m` of the string always precedes byte `m+1` of the string, and byte 0 of the string always follows the string's length. If `n` is not a multiple of four, the `n` bytes are followed by enough (0 to 3) residual 0 bytes (`r`) to make the total byte count a multiple of 4. Declare counted byte strings as follows:

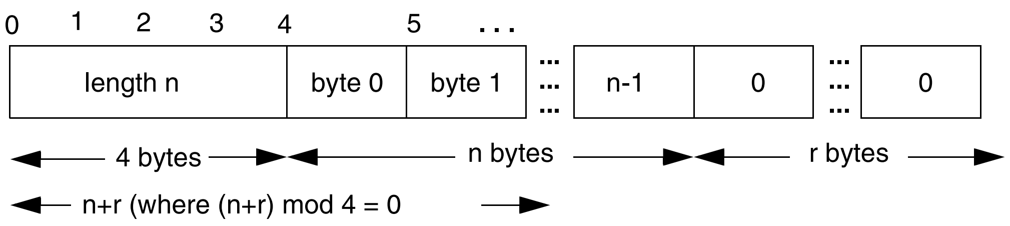
```
string object<m>;
```

The constant `m` denotes an upper bound of the number of bytes that a string may contain. If `m` is not specified, it is assumed to be (2**32)-1, the maximum length. The constant `m` would normally be found in a protocol specification. For example, a filing protocol may state that a file name can be no longer than 255 bytes:

```
string filename<255>;
```

This is illustrated in Figure 5-9..

Figure 5-9. Sample String



It is an error to encode a length greater than the maximum described in the specification.

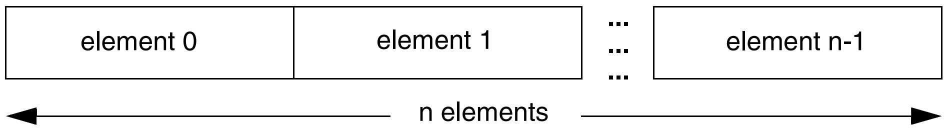
Fixed-Length Array

Declarations for fixed-length arrays of homogeneous elements are in the following form:

```
type-name identifier[n];
```

Fixed-length arrays of elements numbered 0 through `n-1` are encoded by individually encoding the elements of the array in their natural order, 0 through `n-1`. Each element's size is a multiple of four bytes. Although all elements are of the same type, the elements may have different sizes. For example, in a fixed-length array of strings, all elements are of type `string`, yet each element varies in length. This is shown in Figure 5-10.

Figure 5-10. Fixed Length Array of Strings



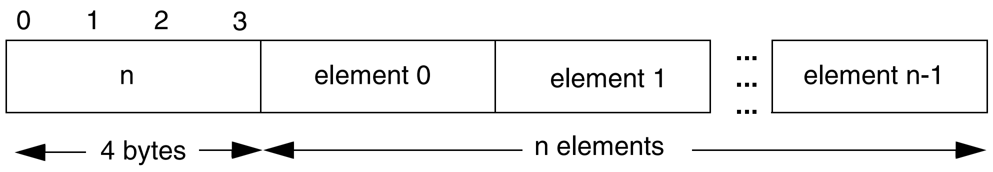
Variable-Length Array

Counted arrays provide the ability to encode variable-length arrays of homogeneous elements. The array is encoded as the element count `n` (an unsigned integer) followed by the encoding of each of the array's elements, starting with element 0 and progressing through element n-1. The declaration for variable-length arrays follows this form:

```
type-name identifier<m>;
```

The constant m specifies the maximum acceptable element count of an array. If m is not specified, it is assumed to be (2**32)-1. This is shown in Figure 5-11..

Figure 5-11. Acceptable Element Count of an Array



It is an error to encode a value of n that is greater than the maximum described in the specification.

Structures

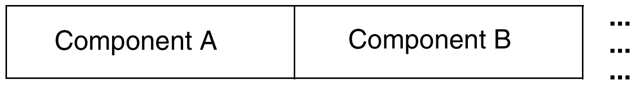
Structures are declared as follows:

```
struct {
   component-declaration-A;
   component-declaration-B;

} identifier;
```

The structure's components are encoded in the order declared in the structure. Each component's size is a multiple of four bytes, although the components may be of different sizes. Structure components are shown in Figure 5-12..

Figure 5-12. Structure Components



Discriminated Union

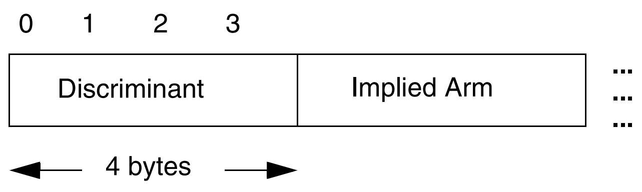
A discriminated union is composed of a discriminant followed by a type selected from a set of prearranged types according to the value of the discriminant. The type of discriminant is either int, unsigned int, or an enumerated type, such as bool. The component types are called arms of the union and are preceded by the value of the discriminant which implies their encoding. Discriminated unions are declared as follows:

```
union switch (discriminant-declaration) {
   case discriminant-value-A:
   arm-declaration-A;
   case discriminant-value-B:
   arm-declaration-B;
   default: default-declaration;
} identifier;
```

Each `case` keyword is followed by a discriminant's legal value. If the default arm is not specified, a valid encoding of the union cannot take on unspecified discriminant values.

The size of the implied arm is always a multiple of 4 bytes. The discriminated union is encoded as its discriminant followed by the encoding of the implied arm, as shown in Figure 5-13.

Figure 5-13. Discriminated Union



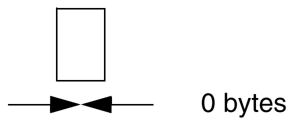
Void

An XDR void is a 0-byte quantity. Voids are useful for describing operations that accept no data as input or output. They are also useful in unions, where some arms may contain data and others may not. The declaration is simply as follows:

```
void;
```

Voids are illustrated in Figure 5-14..

Figure 5-14. Void



Constant

The data declaration for a constant follows this form:

```
const name-identifier = n;
```

`const` defines a symbolic name for a constant; it does not declare any data. Use the symbolic constant anywhere you use a regular constant. This example defines a symbolic constant `DOZEN`, which is equal to 12.

```
const DOZEN = 12;
```

Typedef

`typedef` does not declare any data. It serves to define new identifiers for declaring data. The syntax is:

```
typedef declaration;
```

The new type name is actually the variable name in the declaration portion of the `typedef`. This example defines a new type called `eggbox` using an existing type called `egg`:

```
typedef egg eggbox[DOZEN];
```

Variables declared using the new type name have the same type as the new type name would have in the `typedef`, if it was considered a variable. For example, the following two declarations are equivalent in declaring the variable `fresheggs`:

```
eggbox  fresheggs;
egg     fresheggs[DOZEN];
```

When a `typedef` involves a `struct`, `enum`, or `union` definition, there is another preferred syntax that may define the same type. In general, you may convert a `typedef` of the following form to the alternative form by removing the `typedef` portion and placing the identifier after the `struct`, `union`, or `enum` keyword, instead of at the end.

```
typedef <<struct, union, or enum definition>> identifier;
```

For example, two ways to define the type `bool` are:

```
typedef enum {     /* using typedef */
   FALSE = 0,
   TRUE  = 1
   } bool;
enum bool {        /* preferred alternative */
   FALSE = 0,
   TRUE  = 1
   };
```

The second syntax is preferred because you do not have to wait until the end of a declaration to figure out the name of the new type.

Optional-Data

Optional-data is an example of a frequently occurring union with its own declaration syntax. The declaration is as follows:

```
type-name *identifier;
```

This is equivalent to the following union:

```
union switch (bool opted) {
   case TRUE:
   type-name element;
   case FALSE:
   void;
} identifier;
```

It is also equivalent to the following variable-length array declaration, because the boolean **opted** can be interpreted as the length of the array:

```
type-name identifier<1>;
```

Optional-data describes recursive data-structures such as linked-lists and trees. For example, the following defines a type `stringlist` that encodes lists of arbitrary length strings:

```
struct *stringlist {
   string item<>;
   stringlist next;
};
```

It could have been equivalently declared as the following union:

```
union stringlist switch (bool opted) {
   case TRUE:
      struct {
         string item<>;
         stringlist next;
      } element;
   case FALSE:
      void;
};
```

It could also have been declared as a variable-length array:

```
struct stringlist<1> {
   string item<>;
   stringlist next;
};
```

Both of these declarations obscure the intention of the `stringlist` type. Therefore, the optional-data declaration is preferred. The optional-data type also has a close correlation to how recursive data structures are represented in high-level languages such as Pascal or C by using pointers. In fact, the syntax is the same as that of the C language for pointers.

The XDR Language Specification

Notational Conventions

This specification uses an extended Backus-Naur Form notation for describing the XDR language. Here is a brief description of the notation:

- The characters |, (,), [,], ", and * are special.
- Terminal symbols are strings of any characters surrounded by double quotes (").
- Non-terminal symbols are strings of non-special characters.
- Alternative items are separated by a vertical bar (|).
- Optional items are enclosed in brackets ([ ]).
- Items are grouped together by enclosing the items in parentheses (()).
- An asterisk (*) following an item means 0 or more occurrences of that item.

Lexical Notes

- Comments begin with /* and terminate with */.
- White space serves to separate items and is otherwise ignored.

- An identifier is a letter followed by an optional sequence of letters, digits, or an underscore (_). The case of identifiers is not ignored.
- A constant is a sequence of one or more decimal digits, optionally preceded by a hyphen (-).

Syntax Information

```
declaration:
   type-specifier identifier
   | type-specifier identifier "[" value "]"
   | type-specifier identifier "<" [ value ] ">"
   | "opaque" identifier "[" value "]"
   | "opaque" identifier "<" [ value ] ">"
   | "string" identifier "<" [ value ] ">"
   | type-specifier "*" identifier
   | "void"
value:
   constant
   | identifier
type-specifier:
     [ "unsigned" ] "int"
   | [ "unsigned" ] "hyper"
   | "float"
   | "double"
   | "bool"
   | enum-type-spec
   | struct-type-spec
   | union-type-spec
   | identifier
enum-type-spec:
   "enum" enum-body
enum-body:
   "{"
   ( identifier "=" value )
   ( "," identifier "=" value )*
   "}"
struct-type-spec:
   "struct" struct-body
struct-body:
   "{"
   ( declaration ";" )
   ( declaration ";" )*
   "}"
union-type-spec:
   "union" union-body
union-body:
   "switch" "(" declaration ")" "{"
   ( "case" value ":" declaration ";" )
   ( "case" value ":" declaration ";" )*
   [ "default" ":" declaration ";" ]
   "}"
constant-def:
   "const" identifier "=" constant ";"
type-def:
   "typedef" declaration ";"
   | "enum" identifier enum-body ";"
   | "struct" identifier struct-body ";"
   | "union" identifier union-body ";"
definition:
   type-def
   | constant-def
specification:
   definition *
```

Syntax Notes

- Do not use the following keywords as identifiers: `bool`, `case`, `const`, `default`, `double`, `enum`, `float`, `hyper`, `opaque`, `string`, `struct`, `switch`, `typedef`, `union`, `unsigned`, and `void`.
- Only unsigned constants may be used as size specifications for arrays. If an identifier is used, it must have been declared previously as an unsigned constant in a `const` definition.
- Constant and type identifiers within the scope of a specification are in the same name space and must be declared uniquely within this scope.
- Similarly, variable names must be unique within the scope of `struct` and `union` declarations. Nested `struct` and `union` declarations create new scopes.
- The discriminant of a union must be of a type that evaluates to an integer. That is, `int`, `unsigned int`, `bool`, `enum`, or any `typedef` that evaluates to one of these is legal. Also, the case values must be one of the legal values of the discriminant. Finally, a case value may not be specified more than once within the scope of a union declaration.

An Example of an XDR Data Description

Following is a short XDR data description of a procedure called `file`, which you might use to transfer files from one machine to another.

```
const MAXUSERNAME = 32;      /* max length of a user name */
const MAXFILELEN = 65535;    /* max length of a file      */
const MAXNAMELEN = 255;      /* max length of a file name */
/* Types of files: */
enum filekind {
   TEXT = 0,                /* ascii data */
   DATA = 1,               /* raw data   */
   EXEC = 2                /* executable */
};
/* File information, per kind of file: */
union filetype switch (filekind kind) {
   case TEXT:
      void;                           /* no extra information \*/
   case DATA:
      string creator<MAXNAMELEN>;     /* data creator */
   case EXEC:
      string interpretor<MAXNAMELEN>; /* program interpretor */
};
/* A complete file: */
struct file {
   string filename<MAXNAMELEN>; /* name of file */
   filetype type;               /* info about file */
   string owner<MAXUSERNAME>;   /* owner of file */
   opaque data<MAXFILELEN>;     /* file data  */
};
```

Suppose that a user named `john` wants to store his `lisp` program `sillyprog` that contains just the data `(quit)`. His file would be encoded as shown in Table 5-3 on page 106.

Table 5-3. Sample Program

Offset	Hex Bytes	ASCII	Description
0	00 00 00 09	....	Length of filename = 9
4	73 69 6c 6c	sill	Filename characters
8	79 70 72 6f	ypro	... and more characters ...
12	67 00 00 00	g...	... and 3 zero-bytes of fill
16	00 00 00 02	....	Filekind is EXEC = 2
20	00 00 00 04	....	Length of interpretor = 4
24	6c 69 73 70	lisp	Interpretor characters
28	00 00 00 04	....	Length of owner = 4
32	6a 6f 68 6e	john	Owner characters
36	00 00 00 06	....	Length of file data = 6

Table 5-3. Sample Program (Continued)

Offset	Hex Bytes	ASCII	Description
40	28 71 75 69	(qui	File data bytes ...
44	74 29 00 00	t)..	... and 2 zero-bytes of fill

6 RPCGEN Programming Guide

This chapter describes the `rpcgen` compiler. `rpcgen` greatly simplifies implementing RPC services by automatically generating client and server stub programs as well as all necessary XDR routines.

There is a version of `rpcgen` for Hawk™ and resident development.

An Overview of rpcgen

The `rpcgen` compiler accepts a remote program interface definition written in RPC. `rpcgen` produces a C language output which includes:

- Stub versions of the client routines
- A server skeleton
- XDR filter routines for both parameters and results
- A header file that contains common definitions

The client stubs interface with the RPC library and effectively hide the network from their callers. The server stub similarly hides the network from the server procedures that remote clients invoke. You can compile and link `rpcgen` output files in the usual way. The developer writes server procedures and links them with the server skeleton produced by `rpcgen` to get an executable server program.

To use a remote program, a programmer writes an ordinary C language main program that makes local procedure calls to the client stubs produced by `rpcgen`. Linking this program with `rpcgen` stubs creates an executable program. You can use `rpcgen` options to suppress stub generation and specify the transport to be used by the server stub.

Converting Local Procedures to Remote Procedures

The following program explains how to convert a local procedure to a remote procedure.

```
/* rmsg.c: print a message on the console */
#include <stdio.h>
main(argc, argv)
   int argc;
   char *argv[];
{
   char *message;
   if (argc < 2) {
      fprintf(stderr, "usage: %s <message>\n", argv[0]);
      exit(1);
   }
   message = argv[1];
   if (!printmessage(message)) {
      fprintf(stderr, "%s: couldn’t print your message\n",
         argv[0]);
      exit(1);
   }
   printf("Message delivered!\n");
}
/*
 * Print a message to the console.
 * Return a boolean indicating whether the message was actually
printed.
 */
```

```
printmessage(msg)
   char *msg;
{
   FILE *f;
   f = fopen("/term", "w");
   if (f == NULL) {
      return (0);
   }
   fprintf(f, "%s\n", msg);
   fclose(f);
   return(1);
}
```

By turning `printmessage` into a remote procedure, it can be called from anywhere in the network.

In general, it is necessary to figure out what the types are for all procedure inputs and outputs. In this case, the procedure `printmessage` takes a string as input and returns an integer as output. Knowing this, you can write a protocol specification in RPC language that describes the remote version of `printmessage`:

```
/* msg.x: Remote message printing protocol */
program MESSAGEPROG {
   version MESSAGEVERS {
      int PRINTMESSAGE(string) = 1;
   } = 1;
} = 99;
```

Remote procedures are a part of remote programs. Therefore, this procedure was declared to be in version 1 of the remote program. No null procedure (procedure 0) is necessary because `rpcgen` generates it automatically.

Notice that everything is declared with all capital letters. This is not required, but it is a good convention to follow.

Notice also that the parameter type is `string` and not `char *`. `char *` in C is ambiguous. Programmers usually intend it to mean a null-terminated string of characters. However, it could also represent a pointer to a single character or a pointer to an array of characters. In RPC, a null-terminated string is unambiguously called a `string`.

Two more sections need to be written. First, the remote procedure needs to be written. Here is the definition of a remote procedure to implement the `PRINTMESSAGE` procedure declared above:

```
/* msg_proc.c: implementation of the remote procedure "printmessage"
*/
#include <stdio.h>
#include <RPC/rpc.h>    /* always needed */
#include "msg.h"        /* need this too: msg.h will be generated by
rpcgen */
                  /* Remote verson of "printmessage" */
int *
printmessage_1(msg)
   char **msg;
```

```
{
   static int result;  /* must be static! */
   FILE *f;
   f = fopen("/term", "w");
   if (f == NULL) {
      result = 0;
      return (&result);
   }
   fprintf(f, "%s\n", *msg);
   fclose(f);
   result = 1;
   return (&result);
}
```

Notice that the declaration of the remote procedure `printmessage_1` differs from that of the local procedure `printmessage` in three ways:

1. `printmessage_1` takes a pointer to a string instead of a string itself. All remote procedures take pointers to their parameters rather than the parameters themselves.
2. `printmessage_1` returns a pointer to an integer instead of an integer itself. This is also generally true of remote procedures.
3. `printmessage_1` has an `_1` appended to its name. In general, all remote procedures called by `rpcgen` are named by the following rule:
   - the name in the program definition (here `printmessage`) is converted to lower-case letters.
   - an underscore (_) is appended to it.
   - the version number (here 1) is appended.

Finally, declare the main client program that calls the remote procedure:

```
/* rmsg.c: remote version of "printmsg.c" */
#include <stdio.h>
#include <RPC/rpc.h>     /* always needed */
#include "msg.h"        /* need this too: msg.h will be generated by
rpcgen*/
main(argc, argv)
   int argc;
   char *argv[];
{
   CLIENT *cl;
   int *result;
   char *server;
   char *message;
   if (argc < 3) {
      fprintf(stderr, "usage: %s host message\n", argv[0]);
      exit(1);
   }
   /* Save values of command line arguments  */
   server = argv[1];
   message = argv[2];
```

```
    /*
     * Create client "handle" used for calling MESSAGEPROG on the
     * server designated on the command line. We tell the RPC package
     * to use the "tcp" protocol when contacting the server.
     */
    cl = clnt_create(server, MESSAGEPROG, MESSAGEVERS, "tcp");
    if (cl == NULL) {
        /*
         * Couldn't establish connection with server.
         * Print error message and die.
         */
        clnt_pcreateerror(server);
        exit(1);
    }

    /*  Call the remote procedure "printmessage" on the server */
    result = printmessage_1(&message, cl);
    if (result == NULL) {
        /*
         * An error occurred while calling the server.
         * Print error message and die.
         */
        clnt_perror(cl, server);
        exit(1);
    }
    /* Okay, we successfully called the remote procedure. */
    if (*result == 0) {
        /*
         * Server was unable to print message.
         * Print error message and die.
         */
        fprintf(stderr, "%s: %s couldn't print your message\en",
            argv[0], server);
        exit(1);
    }
    /* The message got printed on the server's console */
    printf("Message delivered to %s!\n", server);
}
```

There are two things to note here:

1. A client handle is created using the RPC library routine `clnt_create()`. This client handle is passed to the stub routines which call the remote procedure.
2. The remote procedure `printmessage_1` is called exactly the same way as it is declared in `msg_proc.c` except for the inserted client handle as the first parameter.

The pieces are put together as follows:

```
$ rpcgen msg.x
```

Refer to MWOS/SRC/SPF/RPC/DEMO/MSG for example source.

The client program printmsg and the server program msg_server were compiled. rpcgen filled in the missing pieces before the programs were compiled.

Here is what rpcgen did with the input file msg.x:

- rpcgen created a header file called msg.h that contained a #define for MESSAGEPROG, MESSAGEVERS, and PRINTMESSAGE for use in the other modules.
- rpcgen created client stub routines in the msg_clnt.c file. In this case there is only one, the printmessage_1 that was referred to from the rmsg client program. The name of the output file for client stub routines is always formed in this way: if the name of the input file is FOO.x, the client stubs output file is called FOO_clnt.c.
- rpcgen created the server program which calls printmessage_1 in msg_proc.c. This server program is named msg_svc.c. The rule for naming the server output file is similar to the previous one—for an input file called FOO.x, the output server file is named FOO_svc.c.

Copy the server to a remote machine and run it. For this example, the machine is called moon. Server processes are run in the background.

```
moon$ msgdd&
```

Then, on the local machine (earth) you can print a message to moon's console.

```
earth$ rmsg moon "Hello, moon."
```

In order for a server to be an RPC server, it must be running portmap.

Generating XDR Routines

The previous example only demonstrated the automatic generation of client and server RPC code. You can also use rpcgen to generate XDR routines, that is, the routines necessary to convert local data structures into network format and vice-versa. This example presents a complete RPC service, a remote directory listing service, which uses rpcgen not only to generate stub routines, but also to generate the XDR routines. Here is the protocol description file:

```
/* dir.x: Remote directory listing protocol */
const MAXNAMELEN = 255;/* maximum length of a directory entry */
typedef string nametype<MAXNAMELEN>;/* a directory entry */
typedef struct namenode *namelist;/* a link in the listing */
/* A node in the directory listing */
struct namenode {
   nametype name;/* name of directory entry */
   namelist next;/* next entry */
};
```

```
/* The result of a READDIR operation. */
union readdir_res switch (int errno) {
case 0:
   namelist list;/* no error: return directory listing */
default:
   void; /* error occurred: nothing else to return */
};
/* The directory program definition */
program DIRPROG {
   version DIRVERS {
      readdir_res
      READDIR(nametype) = 1;
   } = 1;
} = 76;
```

Refer to `MWOS/SRC/SPF/RPC/DEMO/DIR` for source code.

Running `rpcgen` on `dir.x` creates four output files. Three are the same as before: header file, client stub routines, and server skeleton. The fourth contains the XDR routines necessary for converting the data types declared into XDR format and vice-versa. These are output in the file `dir_xdr.c`.

The READDIR Procedure

Following is the implementation of the `READDIR` procedure.

```
/* dir_proc.c: remote readdir implementation */
#include <RPC/rpc.h>
#include <dir.h>
#include "dir.h"
extern int errno;
extern char *malloc();
extern char *strdup();
readdir_res *
readdir_1(dirname)
   nametype *dirname;
{
   DIR *dirp;
   struct direct *d;
   namelist nl;
   namelist *nlp;
   static readdir_res res;              /* must be static! */
   dirp = opendir(*dirname);            /* open directory */
   if (dirp == NULL) {
      res.errno = errno;
      return (&res);
   }
   xdr_free(xdr_readdir_res, &res);     /* free previous result */
   /* Collect directory entries */
   nlp = &res.readdir_res_u.list;
   while (d = readdir(dirp)) {
      nl = *nlp = (namenode *) malloc(sizeof(namenode));
      nl->name = strdup(d->d_name);
      nlp = &nl->next;
   }
   *nlp = NULL;
   /* Return the result */
   res.errno = 0;
   closedir(dirp);
   return (&res);
}
```

Client Calling Server

Finally, the client side program calls the server.

```
/* rdir.c: Remote directory listing client */
#include <stdio.h>
#include <RPC/rpc.h>    /* always need this */
#include "dir.h"            /* need this too: will be generated by
rpcgen */
extern int errno;
main(argc, argv)
    int argc;
    char *argv[];
{
    CLIENT *cl;
    char *server;
    char *dir;
    readdir_res *result;
    namelist nl;

    if (argc != 3) {
        fprintf(stderr, "usage: %s host directory\n", argv[0]);
        exit(1);
    }
    /* Remember what our command line arguments refer to */
    server = argv[1];
    dir = argv[2];
    /*
     * Create client "handle" used for calling MESSAGEPROG on the
     * server designated on the command line. The RPC package is told
     * to use the "tcp" protocol when contacting the server.
     */
    cl = clnt_create(server, DIRPROG, DIRVERS, "tcp");
    if (cl == NULL) {
        /*
         * Could not establish connection with server.
         * Print error message and die.
         */
        clnt_pcreateerror(server);
        exit(1);
    }
    /* Call the remote procedure readdir on the server */
    result = readdir_1(&dir, cl);
    if (result == NULL) {
        /*
         * An error occurred while calling the server.
         * Print error message and die.
         */
        clnt_perror(cl, server);
        exit(1);
    }
    /* Okay, we successfully called the remote procedure. */
    if (result->errno != 0) {
        /* A remote system error occurred. Print error message and
die. */
        errno = result->errno;
        perror(dir);
        exit(1);
    }
    /* Successfully got a directory listing.  Print it. */
    for (nl = result->readdir_res_u.list; nl != NULL;
      nl = nl->next) {
        printf("%s\n", nl->name);
    }
}
```

Compile and run:

```
earth$  rpcgen dir.x
```

A sample makefile is located in `MWOS/SRC/SPF/RPC/DEMO/DIR`.

You may want to comment out calls to RPC library routines and have client-side routines call server routines directly.

The C Preprocessor

The C preprocessor is run on all input files before they are compiled. Therefore, all preprocessor directives are legal within a `.x` file. Four symbols may be defined, depending upon which output file is being generated. The symbols are shown in Table 6-1..

Table 6-1. C Preprocessor Symbols

Symbol	Function
RPC_HDR	For header-file output
RPC_XDR	For XDR routine output
RPC_SVC	For server-skeleton output
RPC_CLNT	For client stub output

`rpcgen` also performs some preprocessing. Any line that begins with a percent sign (`%`) is passed directly into the output file without any interpretation of the line. Here is a simple example that demonstrates the preprocessing features:

```
/* time.x: Remote time protocol */
program TIMEPROG {
        version TIMEVERS {
                unsigned int TIMEGET(void) = 1;
        } = 1;
} = 44;
#ifdef RPC_SVC
%int *
%timeget_1()
%{
%        static int thetime;
%
%        thetime = time(0);
%        return (&thetime);
%}
#endif
```

If you plan to use a line continuation character at the end of a directive, it is recommended that you do not place a `%` symbol on the continued lines.

For example, use the code:

```
%#define MYNUMBER \
   3
```

instead of:

```
%#define MYNUMBER \
%  3
```

If you place a `%` symbol on the continued lines, compile problems will result in the generated C file. This behavior occurs for the implementation of `rpcgen` on multiple operating systems. Moreover, the % feature is not recommended in general, as there is no guarantee that the Compiler will place the output in the intended location.

RPC Language

RPC language is an extension of XDR language. The sole extension is the addition of the program type. However, the XDR language is close to C. The syntax of the RPC language is described here along with a few examples. Various RPC and XDR type definitions get compiled into C type definitions in the output header file.

Definitions

An RPC language file consists of a series of definitions.

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

It recognizes five types of definitions: `enum`, `struct`, `union`, `typedef`, `const`, and `program`.

Structures

An XDR `struct` is declared similarly to its C counterpart. It looks like the following:

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

As an example, the following XDR structure defines a two-dimensional coordinate:

```
struct coord {
     int x;
     int y;
};
```

This structure gets compiled into the following C structure in the output header file:

```
struct coord {
     int x;
     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. They are more analogous to Pascal variant records than 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 no error occurs, a block of data is returned. Otherwise, nothing is returned.

```
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` structure 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`:

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

It is compiled into the following C `enum`:

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

Typedef

An XDR `typedef` has the same syntax as a C `typedef`.

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

Here is an example that defines an `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 an integer constant is used, for example, in array size specifications.

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

For example, 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 the time protocol:

```
/*
 * 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 `#define` in the output header file:

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

Declarations

In XDR, the four kinds of declarations are: `simple`, `fixed-array`, `variable-array`, and `pointer`.

- Simple declarations are just like simple C declarations.

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

  For example:

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

- Fixed-length Array Declarations are just like C array declarations:

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

  For example:

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

- Variable-Length Array Declarations have no explicit syntax in C. 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. You may omit the size, indicating that the array may be of any size.

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

Because variable-length arrays have no explicit syntax in C, these declarations are actually compiled into `structs`. For example, the `heights` declaration gets compiled into the following:

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

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 component names is the same as the name of the declared XDR variable.

- Pointer Declarations are made in XDR exactly as they are in C. Pointers cannot actually be sent over the network, but the user can use XDR pointers for sending recursive data types. The type is actually called `optional-data`, not `pointer`, in XDR language.

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

For example:

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

Special Cases

There are a few exceptions to the rules described above.

Booleans

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

For example:

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

Strings

C has no built-in `string` type, but instead uses the null-terminated `char *` convention. In the XDR language, strings are declared using the `string` keyword and compiled into `char*` 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 omitted, indicating a string of arbitrary length.

For example:

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

Opaque Data

Opaque data is used in RPC and XDR to describe untyped data. Untyped data are sequences of arbitrary bytes. Opaque data may be declared either as a fixed or variable length array.

For example:

```
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 simply `void`. `Void` declarations can only occur in union definitions and program definitions (as the parameter or result of a remote procedure).

A Getting Started With Network File System/Remote Procedure Call

This chapter describes how to configure the Network File System/Remote Procedure Call (NFS/RPC) package on your OS-9 system.

Introduction

Before you start you should understand the OS-9 operating system and have your OS-9 software installed on your host system.

Your CD-ROM insert has instructions for installing your Microware products on your host platform./.

System Components

NFS/RPC consists of development tools, user applications, and system modules that facilitate communication between OS-9 and other Internet systems through the RPC communication protocol or through NFS local and remote directory services.

System Architecture

The following figures show the architecture and organization of the modules provided. RPC and NFS utilities can be found in `MWOS/<OS>/<CPU>/CMDS`. The system modules can be found in `MWOS/<OS>/<CPU>/CMDS/BOOTOBJS/SPF`.

Figure A-1. NFS Architecture

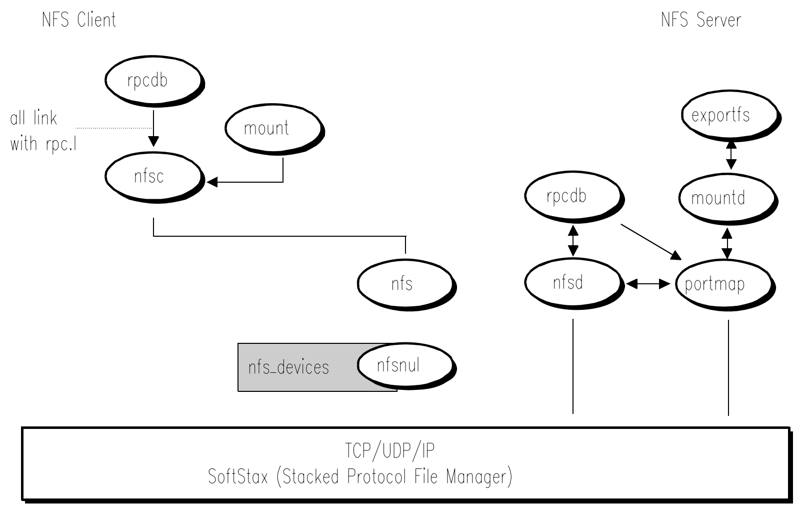
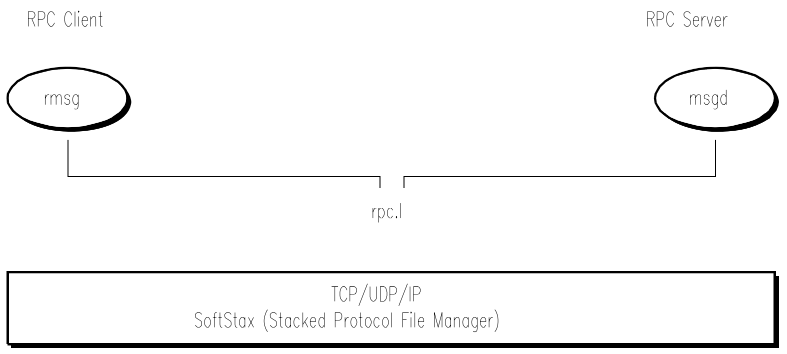


Figure A-2. RPC Architecture



Refer to the *Using LAN Communications* for more information about the TCP/UDP/IP protocol stack.

Configuring the NFS Client

The NFS Client software contains the components to support an OS-9 system as an NFS client, communicating with a remote NFS File Server System across a network.

Directory Structure

The NFS/RPC software package is included within the LAN Communications and is installed automatically.

After successful installation, the NFS directory structure is as shown in Table A-1 on page 127.

Table A-1. NFS Directory Structure

Directory	Contents
`MWOS/<os>/<CPU>/CMDS`	Objects for all NFS and RPC utilities.
`MWOS/<os>/<CPU>/CMDS/BOOTOBJS/SPF`	Objects for file manager, driver, and descriptor.
`MWOS/SRC/ETC`	RPC services file and NFS mapping files.

Configuration Overview

This section describes the steps for configuring an NFS client on your system. Each step is described in the following sections:

1. Configure group and user ID mapping files for NFS.
2. Build the RPC database module.
3. Configure the startup procedures.
4. Verify the installation.

Step 1: Configure Group and User ID Mapping File for NFS

NFS supports user permission mapping files that the NFS file manager uses.

For the client side of NFS, the `nfs.map` file specifies group/user ID mappings between the local system and the remote server systems.

NFS Client Map File (nfs.map)

The `-c` option of `rpcdbgen` specifies the user permission mapping file for the NFS Client. The map file consists of one-line entries specifying the Client group or user number, followed by the remote system group or user ID number. For example, the following entries map the OS-9 group number 10 to the remote system group number 12, and the OS-9 user number 77 to the remote system user ID number 99.

```
g10   12
u77   99
```

The `g` or `u` prefix specifies whether the field is a group or user number. The remote system group or user ID number is not checked for validity.

You can use the asterisk (*) wildcard to specify a generic group or user ID number. For example, the following entries map all local groups and users to group 12 and user 99 respectively:

```
g*    12
u*    99
```

If both specific and generic mapping entries are present, specific entries have precedence. If no entry exists for a specific Client group/user and no generic entries are present, the group and user are not translated.

Not all NFS file server systems allow super user (root) access via NFS. If group/user 0 is not mapped, the remote NFS server configuration determines the effect of super user access. In UNIX, for example, the UNIX `exportfs` utility controls super user access via NFS. Depending on how the server is configured, super user access via NFS to a UNIX server may appear as group -2, 0, or another group as specified on `exportfs`.

Do not access UNIX servers as group 0 or user 0. Files created will be owned by group -2 or user -2, and will require global read and write permissions for subsequent access.

The `nfs.map` file is located in `MWOS/SRC/ETC`.

Mapping is not enforced unless the `-m` option for mount is used.

Step 2: Build the RPC Data Base Module

The RPC data base module contains global information many RPC clients and servers use, including:

- The local RPC hostname (optional)
- The location of the NFS backup/recovery directory (required for server)
- NFS statistics
- Client user and group mappings
- Other global flags

The `rpcdbgen` utility builds the RPC database module. All parameters are optional. A default `rpcdb` can be generated for all systems.

If you want NFS/RPC to collect internal use statistics, use the `-s` option to `rpcdbgen`. You can use `nfsstat` to view the data.

Use the `-c` option for client mapping.

The `inetdb` data module created by `idbgen` includes the `rpc` file contents, which map a version number to a RPC program.

Utilities are described in <links>Chapter 2 NFS/RPC Utilities and Daemon Server Programs

To build the RPC database module, perform the following steps:

1. Change directory to:

`MWOS/SRC/ETC`
or
`MWOS/<OS>/<CPU>/PORTS/<TARGET>/SPF/ETC`

1. Run the following:

os9make

This creates the `inetdb` and `rpcdb` modules in
`MWOS/<OS>/<CPU>/CMDS/BOOTOBJS/SPF`.
or local port directory: `CMDS/BOOTOBJS/SPF`

The `idbdump` and `rpcdump` utilities display the `inetdb` and `rpcdb` data modules. Following is an example of the `rpcdb` data module.

```
Diag:rpcdump
Dump of NFS/RPC data module [rpcdb]
   recovery dir:
   collect stats: yes
   use nfs client map: yes
   use nfsd server map: no
NFS Client Mapping
   default client uid: 99
   default client group: 12
      OS-9 uid NFS uid
         77 99

      OS-9 gid NFS gid
         10 12
```

Step 3: Configure the Startup Procedure

Update your bootlist, or for disk-based systems, use `loadnfs` and `startnfs` scripts in `MWOS/SRC/SYS` to load and initialize the software. LAN Communications and SoftStax® must be loaded and initialized.

An example bootlist follows. Depending on the version of OS software you are using,you may need a relative path of:

`../../../../../../<CPU>` or `../../../<CPU>`

```
*
* NFS protocol file manager, driver and descriptor:
*
../../../../../../<CPU>/CMDS/BOOTOBJS/SPF/nfs
../../../../../../<CPU>/CMDS/BOOTOBJS/SPF/nfsnul
../../../../../../<CPU>/CMDS/BOOTOBJS/SPF/nfs_devices
*
* Local rpcdb data module
*
../../../CMDS/BOOTOBJS/SPF/rpcdb
*
* NFS client process
*
../../../../../../<CPU>/CMDS/nfsc
*
* NFS/RPC utilities:
*
*../../../../../../<CPU>/CMDS/exportfs
../../../../../../<CPU>/CMDS/mount
*../../../../../../<CPU>/CMDS/mountd
*../../../../../../<CPU>/CMDS/nfsd
*../../../../../../<CPU>/CMDS/nfsstat
*../../../../../../<CPU>/CMDS/pcnfsd
*../../../../../../<CPU>/CMDS/portmap
*../../../../../../<CPU>/CMDS/rpcdbgen
*../../../../../../<CPU>/CMDS/rpcdump
*../../../../../../<CPU>/CMDS/rpcgen
*../../../../../../<CPU>/CMDS/rpchost
*../../../../../../<CPU>/CMDS/rpcinfo
*../../../../../../<CPU>/CMDS/rstatd
*../../../../../../<CPU>/CMDS/rup
*../../../../../../<CPU>/CMDS/rusers
*../../../../../../<CPU>/CMDS/rusersd
*../../../../../../<CPU>/CMDS/showmount
*
```

The example `loadnfs` located in `MWOS/SRC/SYS` procedure loads the `nfs_device`, `nfs`, `nfsnul`, `rpcdb`, `mount`, and `nfsc` modules. Review this file to verify that NFS is being loaded if you are using a disk-based system.You can specify, in the `startnfs` file, which remote file systems to mount. The OS-9 `startup` file can call `startnfs` procedures to automatically mount remote file systems.

The following example loads the NFS client modules provided with LAN Communications.

```
*
* loadnfs for NFS modules provided with LAN Communication Package
*
*
* Load NFS Client Modules
*
chd CMDS/BOOTOBJS/SPF
load -d nfs  nfsnul nfs_devices        ;* NFS file manager, driver
                                        * and descriptor
load -d rpcdb                          ;* RPC services module
*
* Load NFS Client Commands
*
chd ../..
*
*load -d nfsc      mount               ;* Client connection handler
*load -d rpcdbgen rpcdump nfsstat      ;* RPC data module utilities
*load -d rpcinfo
*
* Load NFS Server Modules
*
*load -d exportfs portmap nfsd mountd  ;* NFS server required
                                              * utilities/daemons
*load -d showmount
*


* Load RPC Client Modules
*
*load -d rup rusers
*
* Load RPC Server Modules
*
*load -d rstatd rusersd
*
```

If you are using a disk-based system, the `startnfs` script, found in `MWOS/SRC/SYS` can be added to the startup file.

The following `startnfs` script loads and initializes the NFS driver and mounts remote systems.

```
* startnfs for NFS provided with LAN Communication Package
*
* Shell Script to Start NFS Client System and mount file systems
```

```
*
* NOTE:NFS client modules may be loaded into memory using loadnfs
*
chd /h0                    ;* Set default directories for NFS mounts
chx /h0/cmds               ;* Programs are located in CMDS directory
SYS/loadnfs
*
* Start NFS client and mount remote file systems
*
iniz nfs_devices           ;* attach NFS client devices
*
* Example mount commands to connect to server systems remote
* device
*
mount -m peer:/   /peer   ;* mount remote file systems
mount alpha:/h0 /alpha
mount beta:/h0 /beta
*
* Start NFS Server System
*
* Specify file systems to export (Necessary if acting as a NFS
* Server)
*
*exportfs -s /h0            ;* specify remote mountable devices
*
* start rpc services daemons
* Uncomment portmap, mountd and nfsd if acting as a NFS Server
*
*portmap<>>>/nil&           ;* start portmap server      (rpcinfo)
*mountd<>>>/nil&         ;* mount server       (mount,*showmount)
*nfsd<>>>/nil&              ;* nfs server                (..)
*rstatd<>>>/nil&            ;* remote system statisitcs  (rup)
*rusersd<>>>/nil&           ;* network users info        (rusers)
```

Step 4: Verify the Installation

LAN Communications and SoftStax must be loaded and initialized. The modules `rpcdb`, `mount`, `nfsc`, `nfs`, `nfsnul`, and `nfs_devices` must be loaded in memory. LAN Communications and SoftStax must be loaded and initialized. To initialize the NFS Client, run the following:

```
$iniz nfs_devices
```

To verify the installation, perform the following steps:

1. Run `rpcinfo -p <remote host>` specifying a remote system to verify Internet access.
2. Mount a remote file system.

3. Enter `mount -d` to verify mount point.
4. Use `dir` to view mount point.

Configuring NFS Server and RPC Development System

The NFS/RPC Development software contains all of the components to support an OS-9 system as an NFS server, as well as several other standard RPC services on the system. It also contains configuration utilities for the network administrator to use for NFS/RPC configuration, access control, and backup requirements.

Directory Structure

The NFS/RPC software package is included within the LAN Communications and is installed automatically.

After successful installation, the NFS directory structure is as shown in Table A-2 on page 133.

Table A-2. NFS Directory Structure

Directory	Contents
`MWOS/<OS>/<CPU>/CMDS`	Objects for all NFS/RPC utilities.
`MWOS/<OS>/<CPU>/CMDS/BOOTOBJS/SPF`	Objects for file managers, drivers, and descriptors.
`MWOS/SRC/DEFS/SPF/RPC`	NFS/RPC header (include) files.
`/MWOS/<OS>/<CPU>/LIB`	RPC/XDR C library (`rpc.l`).
`/MWOS/SRC/ETC`	RPC services file and NFS mapping files.
`/MWOS/SRC/SPF/RPC/DEMO`	Several example RPC services, including `rdir`, `rmsg`, and `rsort`. Full client and server program source code and examples of using `rpcgen` are provided.

Configuration Overview

This section describes the steps for configuring the NFS/RPC Development software. Each step is described in the following sections.

- Step 1: Configure group and user ID mapping files for NFS
- Step 2: Build the RPC data base module.

- Step 3: Configure the startup procedures.
- Step 4: Export local file systems.
- Step 5: Verify the installation.

Step 1: Configure Group and User ID Mapping Files for NFS

NFS supports user permission mapping files that the NFS file manager uses. For the server side of NFS, the `nfsd.map` file specifies group/user ID mappings between the local system and the remote server systems.

NFS Server Map File (nfsd.map)

The `-d` option of `rpcdbgen` specifies the mapping file the NFS file server uses. The map file maps remote group/user numbers to local server system groups and users. It consists of one-line entries specifying the remote system group or user number, followed by the local server system group or user ID number. For example, the following entries map remote group numbers 12000 and 64099 to the local server system group number 10, and the remote user number 12345 to the local server system user ID number 99.

```
g12000    10

g64099    10

u12345    99
```

The `g` or `u` prefix to the remote system field specifies whether the field is a group or user number.

Use the asterisk (*) wildcard to specify a generic group or user ID number. For example, the following entries map all remote groups and users to group 12 and user 99 on the server system respectively:

```
g*    12

u*    99
```

If both specific and generic mapping entries are present, specific entries have precedence. If no entry exists for a specific remote group/user and no generic entries are present, the group and user are not translated.

The `nfsd.map` file can be found in `MWOS/SRC/ETC`.

Step 2: Build the RPC Data Base Module

The RPC data base module contains global information many RPC clients and servers use, including:

- The local RPC hostname (optional)
- The location of the NFS backup/recovery directory (required for server)
- NFS statistics
- Client user and group mappings

- Other global flags

The `rpcdbgen` utility builds the RPC database module. All parameters are optional. A default `rpcdb` can be generated for all systems.

If you want NFS/RPC to collect internal use statistics, use the `-s` option to `rpcdbgen`. You can use `nfsstat` to view the data.

The `rpcdump` utilities can be used to verify the contents of the `rpcdb` data modules. Following is an example of the `rpcdb` data module. Use the `-d` option for server mapping.

```
Diag:rpcdump
Dump of NFS/RPC data module [rpcdb]
   recovery dir: MWOS/SRC/ETC
   collect stats: yes
   use nfs client map: yes
   use nfsd server map: yes
NFS Client Mapping
   default client uid: 99
   default client group: 12
      OS-9 uid NFS uid
         77 99
      OS-9 gid NFS gid
         10 12


NFS Server Mapping
   default server uid: 99
   default server group: 12
      NFS uid OS-9 uid
      12345 99
      NFS gid OS-9 gid
      64099 10
      12000 10
```

Step 3: Configure the Startup Procedures

Update your bootlist, or for disk-based systems, use `loadnfs` and `startnfs` scripts in `MWOS/SRC/SYS` to load and initialize the software. LAN Communications and SoftStax must be loaded and initialized.

An example bootlist for NFS servers follows. Depending on the OS software version you are using, you may need a relative path of:

`../../../../../../<CPU>` or `../../../<CPU>`

```
*
* NFS protocol file manager, driver and descriptor:
*
../../../../../../<CPU>/CMDS/BOOTOBJS/SPF/nfs
../../../../../../<CPU>/CMDS/BOOTOBJS/SPF/nfsnul
```

```
../../../../../../<CPU>/CMDS/BOOTOBJS/SPF/nfs_devices
*
* Local rpcdb data module
*
../../../CMDS/BOOTOBJS/SPF/rpcdb
*
* NFS client process
*
../../../../../../<CPU>/CMDS/nfsc
*
* NFS/RPC utilities:
*
../../../../../../<CPU>/CMDS/exportfs
../../../../../../<CPU>/CMDS/mount
../../../../../../<CPU>/CMDS/mountd
../../../../../../<CPU>/CMDS/nfsd
*../../../../../../<CPU>/CMDS/nfsstat
*../../../../../../<CPU>/CMDS/on
*../../../../../../<CPU>/CMDS/pcnfsd
../../../../../../<CPU>/CMDS/portmap
*../../../../../../<CPU>/CMDS/rcopy
*../../../../../../<CPU>/CMDS/rpcdbgen
*../../../../../../<CPU>/CMDS/rpcdump
*../../../../../../<CPU>/CMDS/rpcgen
*../../../../../../<CPU>/CMDS/rpchost
*../../../../../../<CPU>/CMDS/rpcinfo
*../../../../../../<CPU>/CMDS/rstatd
*../../../../../../<CPU>/CMDS/rup
*../../../../../../<CPU>/CMDS/rusers
*../../../../../../<CPU>/CMDS/rusersd
*../../../../../../<CPU>/CMDS/showmount
*
```

The example `loadnfs` script found in `MWOS/SRC/SYS` loads the NFS and RPC modules and the example `startnfs` script initiates the NFS/RPC server. The modules `rpcdb`, `mountd, nfsd`, and `portmap` must be loaded into memory. Other modules can either be loaded into memory or found in the execution path on a disk-based system.

On disk-based systems, you can use the `startnfs` file to specify which RPC services you will support on the local system. The `startnfs` procedure may start the server and specify which file systems to export. The OS-9 `startup` file can call `startnfs` to automatically mount remote file systems.

The following example loads the NFS modules provided with LAN Communications.

```
*
* loadnfs for NFS modules provided with LAN Communication Package
*
```

```
* Load NFS Client Modules
*
chd CMDS/BOOTOBJS/SPF
* load -d nfs  nfsnul nfs_devices      ;* NFS file manager, driver
                                        * and descriptor
load -d rpcdb                          ;* RPC services module
*
* Load NFS Client Commands
*
chd ../..
*
*load -d nfsc      mount               ;* Client connection handler
*load -d rpcdbgen rpcdump nfsstat      ;* RPC data module utilities
*load -d rpcinfo
*
* Load NFS Server Modules
*
load -d exportfs portmap nfsd mountd  ;* NFS server required
                                            * utilities/daemons
*load -d showmount
*
* Load RPC Client Modules
*
*load -d rup rusers spray
*
* Load RPC Server Modules
*
*load -d rstatd rusersd sprayd
*
```

The following `startnfs` example exports the `/h0` drive and starts up the `portmap`, `mountd`, and `nfsd` daemons. It also lists the daemons that can be started.

```
*
* startnfs for NFS provided with LAN Communication Package
*
*
* Shell Script to Start NFS Client System and mount file systems
*
* NOTE: NFS client modules may be loaded into memory using loadnfs
*
chd /h0                    ;* Set default directories for NFS mounts
chx /h0/cmds               ;* Programs are located in CMDS directory
SYS/loadnfs
* Start NFS client and mount remote file systems
*
* iniz nfs_devices           ;* attach NFS client devices
```

```
*
* Example mount commands to connect to server systems remote
* device
*
*mount -m peer:/   /peer  ;* mount remote file systems
*mount alpha:/h0 /alpha <>>>/nil&
*mount beta:/h0 /beta<>>>/nil&
*
* Start NFS Server System
*
* Specify file systems to export (Necessary if acting as a NFS
* Server)
*
exportfs -s /h0           ;* specify remote mountable devices
*
* start rpc services daemons
* Uncomment portmap, mountd and nfsd if acting as a NFS Server
*
portmap<>>>/nil&          ;* start portmap server      (rpcinfo)
mountd<>>>/nil&           ;* mount server              (mount,
* showmount)
nfsd<>>>/nil&             ;* nfs server                (..)
*rstatd<>>>/nil&           ;* remote system statisitcs  (rup)
*rusersd<>>>/nil&          ;* network users info        (rusers)
```

Step 4: Export Local File Systems

To run the NFS file server, you must specify which disk devices can be remotely mounted. You can specify any local disk device (such as hard, floppy, or RAMdisk).

For example, to allow remote systems to mount and access the local hard disks `/h0` and `/h1`, floppy disks `/d0` and `/d1`, and RAMdisk `/r0`, enter:

```
exportfs -s /h0 /h1
exportfs /d0  /d1 /r0
```

The first time `exportfs` is run, use the `-s` option to create a new mount table for exported file systems.

`portmap` must execute to run any RPC servers. Running the NFS file server requires `portmap`, `mountd`, and `nfsd`.

OS-9 file systems can only be exported and mounted at the root level: `/h0`, not `/h0/CMDS/BOOTOBJS`.

Step 5: Verify the Installation

After starting the NFS/RPC server, verify the software by performing the following steps:

1. Run `rpcinfo -p` to verify that everything is installed.

```
rpcinfo -p
```

You should see a display showing `portmap` and a number of RPC servers, ready to serve. It should be similar to the following, depending on which services you have installed:

```
program vers proto   port
100000    2   tcp     111  portmapper
100000    2   udp     111  portmapper
100005    1   udp     601  mountd
100002    2   udp     606  rusersd
100002    1   udp     606  rusersd
100003    2   udp    2049  nfs
100001    3   udp     607  rstatd
100001    2   udp     607  rstatd
100001    1   udp     607  rstatd
```

If `rpcinfo` reports that it cannot contact the remote system, or the RPC servers report that they cannot contact `portmap`, you may have named the system incorrectly, or there is a problem with the network. Run `hostname` and verify the `hosts` file used to create the `inetdb` module for consistency. If incorrect, set it with the `hostname` utility provided with LAN Communications.

2. Run `rpcinfo -p` on a remote host specifying your NFS server.
3. Mount the server locally to itself.
4. Mount a file system from a remote system and display the directory.
5. If any of these operations fail, try to use `telnet` and `ftp` between the systems in both directions to verify operation of the network.

Index

N A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Symbols

(TCP/IP) Streams, Record 88

A

Acceptable Element Count of an Array 100
access, super user 128
Arbitrary Data Types, Middle Layer to Pass 42
Architecture
    NFS 126
    RPC 126
    System 126
Array, Acceptable Element Count of an 100
Array, Variable-Length 99
Arrays 80
Arrays, Byte 80
Arrays, Fixed-Sized 83
Assigning Program Numbers 42
Authentication 60
Authentication, Null 61
Authentication, OS-9/OS-9000 61
Authentication, Server Side 62

B

Basic Block Size 94
Batching 54
Binding 36
Block Size, Basic 94
Blocks, Building 44
Boolean 95
Booleans 123
Broadcast RPC 53
Broadcast RPC Synopsis 53
Building Blocks 44
Byte Arrays 80

C

C Preprocessor 117
C Preprocessor Symbols 118
C Programs to Implement RPC Protocol 24
Callback Procedures 64
Calling Server, Client 116
Calling Side of the Lower Layer 50
calls, Normal RPC calls/Broadcast RPC 53
Client Calling Server 116
Client Side, NFS 10
Client-only package
    description 127
Components
    System 126
Components, Structure 100
Configuration
    Overview 127, 133
Configuring
    NFS Server and RPC Development System 133
    the NFS Client 127
Constant 101
Constants 121
Constructed Data Type Filters 79
Conventions, Notational 103
Converting Local Procedures to Remote Procedures 110
CPP Symbols 24
CPP Symbols 24

D

Daemon, PC NFS Login 19
DARPA Port to RPC Program Mapper 20
Data Representation, External 9 , 73
Data Type Filters, Constructed 79
Data, Opaque 83 , 124
Data, Serializing and Deserializing 76
Data, Variable-Length Opaque 98
Database Module, Exports List in NFS 14
Database Module, NFS/RPC 21
Declarations 122
Definitions 119
Directory Structure 127, 133
Directory Structure, NFS 127, 133
Discriminated Union 100 , 101
Discriminated Union Example D 84
Discriminated Unions 84
Dismount NFS File System, Mount and 15
Display
    List of Users Logged into Remote System 30
    Name of Current Host 26
    Remote Mounts 32
    RPC and NFS Statistics 18
    RPC Database Module 23
    RPC Information 27
    Status of Remote System 29
Double-Precision Floating-Point 96
Double-Precision, Floating-Point Number Fields 97

E
Enumeration Filters 78
Enumerations 120
Error Conditions, Reply Message and 37
Example of an XDR Data Description 105
example RPC services
 directory 133
exportfs 14
Exports List in NFS Database Module 14
External Data Representation 9 , 73

F
Features, RPC 52
File System Model 9
File System Overview, Network 8
Filters, Constructed Data Type 79
Filters, Enumeration 78
Filters, Floating Point 78
Filters, Number 77
Fixed Length
 Array of Strings 99
 Opaque Data 98
Fixed-Length
 Array 99
 Opaque Data 97
Fixed-Sized Arrays 83
Floating Point Filters 78
Floating-Point 96
Floating-Point Number 96
Floating-Point, Double-Precision 96

G
group/user
 generic ID 128
 ID mappings 127, 134

H
Highest Layer 38
Hyper Integer and Unsigned Hyper Integer 95

I
I/O Streams, Standard 87
Implementing Arrays
 Example A 81
 Example B 82
 Example C 82
Independence, Transport 35
Integer 94
Integer, Unsigned 95
internal usage statistics 129 , 135
Introduction 126

L
Language Specification, RPC 35
Language Specification, XDR 103
Language, RPC 34
Layer, Highest 38
Layer, Middle 39
Lexical Notes 103
Library, XDR 74
Linked Lists 90
Lists, Linked 90
Local Procedures to Remote Procedures, Converting 110
Login Daemon, PC NFS 19
Lower Layer, Calling Side of the 50
Lowest Layer of RPC 46
Lowest Layer on the Server Side, RPC 46

M
Mapper, DARPA Port to RPC Program 20
mapping file
 specify 134
Marking Standard, Record 60
Memory Allocation
 with XDR and the Lower Layer 49
Memory Streams 87
Message Protocol, RPC 57
Middle Layer 39
Middle Layer to Pass Arbitrary Data Types 42
Model, File System 9
Module, NFS/RPC Database 21
mount 15
Mount and Dismount NFS File System 15
mountd 16

N
Network File System Overview 8
NFS
 Architecture 126
 Client Map File (nfs.map) 128
 client support 133
 Client, Configuring the 127
 Directory Structure 127, 133
 directory structure 127, 133
 Directory Structure 133
 Mount Request Server 16
 Protocol Definition 9
 Protocol Server 17
 Server and RPC Development System, Configuring 133
 Server Map File (nfsd.map) 134
NFS Client Side 10
NFS Database Module, Exports List in 14
NFS File System, Mount and Dismount 15
NFS Server Side 10

nfs.map, NFS Client Map File 128
NFS/RPC
  Daemon Server Programs 13
  Daemon Server Programs. 13
  Utilities 12
  Utilities 12
  Utilities and Daemon Server Programs 11
NFS/RPC Database Module 21
NFS/RPC utilities
  directory 127 , 133
NFS/RPC, Getting Started With 125
nfsd 17
nfsd.map, NFS Server Map File 134
nfsstat 18
Non-filter Primitives 86
Normal RPC calls/Broadcast RPC calls 53
Notational Conventions 103
Notes, Lexical 103
Null Authentication 61
Number Filters 77
Number, Floating-Point 96
Numbers, Program 42

O
Object, XDR 89
Opaque Data 83 , 124
Opaque Data, Variable-Length 98
Optional-Data 102
OS-9 disk devices
  remote mounting 138
OS-9/OS-9000 Authentication 61
Overview of rpcgen 110
Overview, Configuration 133
Overview, Network File System 8
Overview, Remote Procedure Call 8

P
PC NFS Login Daemon 19
pcnfsd 19
PMAPPROC_CALLIT 71
PMAPPROC_DUMP 71
PMAPPROC_GETPORT 71
PMAPPROC_NULL 70
PMAPPROC_SET 70
PMAPPROC_UNSET 70
Pointer Example E 85
Pointers 85
Port Mapper Operation 70
Port Mapper Protocol 69
Port Mapper Protocol Specification (in RPC Language) 71
portmap 20 , 139
Preprocessor Symbols, C 118
Preprocessor, C 117
Primitives, Non-filter 86
Procedure, READDIR 115
Procedures, Callback 64
Procedures, Programs and 36
Program Mapper, DARPA Port to RPC 20
Program Numbers 42
Program Numbers, Assigning 42
Programming Guide, RPCGEN 109
Programs 121
Programs and Procedures 36
Protocol Specification(in RPC Language), Port Mapper 71
Protocol, Port Mapper 69
Protocol, RPC 34
Protocol, RPC Message 57

R
READDIR Procedure 115
Reader 75
Record (TCP/IP) Streams 88
Record Marking Standard 60
Remote Procedure Call 9
Remote Procedure Call Overview 8
Remote Procedure Calls 33
Remote Procedures, Converting Local Procedures to 110
Remote Systems Statistics Server 28
Reply Message and Error Conditions 37
Routines, XDR 114
RPC
  Architecture 126
  build database module 128 , 134
  Development System, Configuring NFS Server and 133
  Language 119
  Protocol Requirements 34
  Semantics 36
  Three Layers 37
RPC Features 52
RPC Language 34
RPC Language Specification 35
RPC Lowest Layer on the Server Side 46
RPC Message Protocol 57
RPC Program Mapper, DARPA Port to 20
RPC Protocol 34
RPC Protocol, C Programs to Implement 24
RPC Synopsis, Broadcast 53
RPC, Broadcast 53
RPC, Lowest Layer of 46
rpcdbgen 21 , 128 , 134
rpcdump 23
rpcgen 24
RPCGEN Programming Guide 109
rpcgen, Overview of 110
rpchost 26

rpcinfo 27
rpr 28
rstatd 28
rup 29
rusers 30
Rusers Server 31
rusersd 31

S
Sampe
    Integer 94
Sample
    Block 94
    Hyper Integer and Unsigned Hyper Integer 95
    nfsstat Report 18
    Program 106
    String 99
    Unsigned Integer 95
Select on the Server Side 52
Serializing and Deserializing Data 76
server
    support 133
Server Side Authentication 62
Server Side, NFS 10
Server Side, RPC Lowest Layer on the 46
Server Side, Select on the 52
Server, Client Calling 116
Server, Remote Systems Statistics 28
Server, Rusers 31
Servers, Stateless 9
showmount 32
Single-Precision, Floating-Point Number 96
Specification, RPC Language 35
Specification, XDR Language 103
sprayd 32
Standard I/O Streams 87
Standard, Record Marking 60
Stateless Servers 9
statistics, internal usage 129 , 135
Streams, Memory 87
Streams, Record (TCP/IP) 88
Streams, Standard I/O 87
String 98
Strings 79 , 123
Structure Components 100
Structures 100 , 119
super user access 128
Symbols , CPP 24
Symbols, C Preprocessor 118
Synopsis, Broadcast RPC 53
Syntax
    Information 104
    Notes 35 , 105
System
    Architecture 126
    Components 126
System Model, File 9

T
Target
    Operating Systems 22
    Processors 22
Transport Independence 35
Typedef 101 , 121

U
Union, Discriminated 100 , 101
Unions 120
Unions, Discriminated 84
Unsigned Hyper Integer, Hyper Integer and 95
Unsigned Integer 95
user access, super 128

V
Variable-Length Array 99
Variable-Length Opaque Data 98
Void 101
Voids 124

W
Writer 74
Writer/Reader Examples 75

X
XDR
    Built-in Type Routines 42
    Data Types 94
    Library Primitives 77
    Operation Directions 87
    Stream Access 87
    Stream Implementation 89
XDR Data Description, Example of an 105
XDR Language Specification 103
XDR Library 74
XDR Object 89
XDR Routines 114