HP AdvanceNet

Programming and Protocols for NFS Services

Programming and Protocols for NFS Services

HP 9000 Computers

HP Part No. B1013-90010
Printed in England February 1991

First Edition
E0291

Notice

Hewlett-Packard makes no warranty of any kind with regard to this material, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose.

Hewlett-Packard assumes no responsibility for the use or reliability of its software on equipment that is not furnished by Hewlett-Packard.

Hewlett-Packard shall not be liable for errors contained herein or for incidental or consequential damages in connection with the furnishing, performance, or use of this material.

Restricted Rights Legend

MS-DOS [R] is a registered trademark of Microsoft Corp.

UNIX [R] is a U.S. registered trademark of AT&T in the U.S.A. and other countries.

Note The Network Information Service (NIS) was formerly known as Yellow Pages (YP). The functionality of the two remains the same, only the name has changed. The name Yellow Pages is a registered trademark in the United Kingdom of British Telecommunications plc.

Hewlett-Packard Company
19420 Homestead Road
Cupertino, CA 95014 U.S.A.

Print History

First Edition.....February 1991

Contents

1

Documentation Overview

This manual was developed for programmers who write applications using NIS (Network Information Service), RPC (Remote Procedure Call), RPCGEN (Remote Procedure Call Protocol Compiler), and XDR (eXternal Data Representation).

If you are using NFS Services but not writing applications, refer to the *Installing and Administering NFS Services* manual for system administration information. For day–to–day use of NFS, refer to the "Common Commands" chapter of the *Using NFS Services* manual.

Before using this manual you should be familiar with the C programming language and the HP-UX operating system. You should also have access to the *HP-UX Reference* manuals.

Note The information in this manual applies to all HP 9000 computer systems. Exceptions are specifically noted.

Contents

Refer to the following list for a brief description of the information contained in each chapter and appendix.

Chapter 1: Documentation Overview

This chapter describes who should use this manual, what is in this manual, and where to find more information.

Chapter 2: NFS Services Overview

This chapter provides a brief overview of the NFS Services product, including RPC, RPCGEN, XDR, and NIS facilities.

Chapter 3: RPC Programming Guide

This chapter provides instructions and examples for writing applications using the RPC services. It also provides a synopsis of RPC routines to describe the RPC functional interface.

Chapter 4: RPCGEN Programming Guide

This chapter describes the RPC Protocol Compiler. It provides instructions and examples for writing RPC applications using the RPCGEN compiler.

Chapter 5: XDR Protocol Specification

This chapter describes the XDR protocols. It also provides a synopsis of XDR routines to describe the XDR functional interface.

Chapter 6: RPC Protocol Specification

This chapter describes the RPC and portmap protocols.

Chapter 7: NIS Protocol Specification

This chapter describes the NIS protocols.

Index

The index provides a page reference to the subjects contained within this manual.

Conventions

The table below explains the conventions used in this manual.

Conventions

Notation	Description
Boldface	Boldface type is used when a term is defined.
`Computer Text`	Computer type is used for commands and other keyboard entries that must be typed exactly as shown. It is also used for on-screen prompts and messages.
italics	Italic type is used for emphasis and titles of manuals. It is also used to represent a syntax name or a variable.
[Key]	This font is used to indicate a key on the computer's keyboard. When two or more [keys] appear together with a dash between them, such as [CTRL]-[S], press those keys simultaneously to execute the command.
`Softkey`	This font is used to represent function keys. It may refer to keyboard softkeys, such as `f1`, or labels that appear on the bottom of your screen.
[ ]	An element inside brackets in a syntax statement is optional. Several elements stacked inside brackets means you may select any one or none of these elements. For example: `[A]` `[B]` You may select `A`, `B`, or neither.

Conventions (continued)

Notation	Description

{ }	When several elements are stacked within braces in a syntax statement, the user must select one of those elements. For example:
	`{A}`
	`{B}`
	`{C}` You must select `A`, `B`, or `C`.

. . .	A horizontal ellipsis in a syntax statement indicates that a previous element may be repeated. For example: [*option*] [*option*] . . . In addition, vertical and horizontal ellipses may be used in examples to indicate that portions of the example have been omitted.

Documentation Guide

For More Information	Read
ARPA Services: Daily Use	*Using ARPA Services*
ARPA Services: System Administration	*Installing and Administering ARPA Services*
C Programming Language	*C Programming Guide,* Jack Purdum, Que Corporation, Indianapolis, Indiana *The C Programming Language,* Brian W. Kernighan, Dennis M. Ritchie; Prentice-Hall, Inc.
Commands and System Calls Section 1: User Commands Section 1M: System Maintenance Section 2: System Calls Section 3: Subroutines Section 4: Special Files Section 5: File Formats Section 7: Miscellaneous Facilities Section 9: HP-UX Glossary	*HP-UX Reference* manuals
HP-UX: Installation	*Installing and Updating HP-UX*
HP-UX: Operating System (HP 9000)	*How HP-UX Works* *Concepts for the System Administrator* *Installing and Updating HP-UX* *HP-UX Reference* manuals *HP-UX System Administration Tasks* *A Beginner's Guide to HP-UX*

For More Information	Read
HP-UX: System Administration	*HP-UX System Administration Tasks* *Installing and Updating HP-UX*
Networking: General Information	*Networking Overview*
NFS Services: Common Commands	*Using NFS Services*, "Common Commands" chapter
NFS Services: Programming and Protocols	*Programming and Protocols for NFS Services*
NFS Services: System Administration ■ Configuration ■ Installation ■ Maintenance ■ Migrating from NFS to RFA ■ NFS in an HP-UX Cluster Environment ■ NFS Services vs. Local HP-UX ■ Troubleshooting	*Installing and Administering NFS Services*
NS System Administration	*Installing and Administering NS Services*
ARPA System Administration	*Installing and Administering ARPA Services*

2

NFS Services Overview

The NFS (Network File System) Services product provides remote access to shared file systems over local- area networks. Nodes running NFS and sharing files can range from personal computers and minicomputers to high performance workstations and supercomputers. Almost any user command (e.g., `list`, `remove`, `copy`) that can be performed locally will operate on an attached remote NFS file system.

NFS nodes can access remote databases containing drawings, schematics, netlists, models, or source code. This access method eliminates the needs to maintain consistency between multiple file copies and to store information locally.

NFS features include the following:

- An NFS server can provide remote access privileges to a restricted set of clients. Clients can attach a remote directory tree to any point on a local file system.
- NFS is stateless: a server does not need to maintain state information about any of its clients to function correctly. With stateless servers, a client need only retry a request until the server responds. It does not need to know if a server is not working.
- Clients access server information and processes by using RPC (Remote Procedure Call). RPC allows a client process to execute functions on a server via a server process. Although these processes can reside on different network hosts, the client process does not need to know about the networking implementations.
- RPC uses the XDR (eXternal Data Representation) functionality to translate machine-dependent data formats (i.e., internal representations) to a universal format used by all network hosts using RPC/XDR.

- NFS also provides an optional Network Information Service (NIS) that provides read access to replicated databases. NIS also uses RPC and XDR library routines.

Remote Procedure Call (RPC)

Clients make an RPC for these reasons:

- To access server information.
- To request action from servers.

The RPC protocol allows a client process to request that a function be performed by a server process. These processes can reside on different hosts on the network, though server processes appear to be running on the client node.

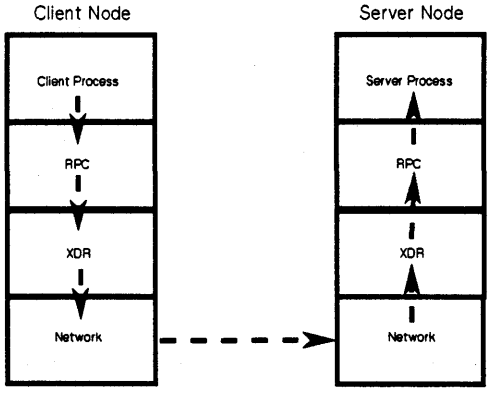
The client first calls an RPC function to initiate the RPC transaction. The client system then sends an encoded message to the server. This message includes all the data needed to identify the service and user authentication information. If the message is valid (i.e., calls an existing service and the authentication is accepted), the server performs the requested service and sends a result message back to the client.

The RPC protocol is a high-level protocol built on top of low-level transports. HP supports both the UDP/IP (user level and kernel level) and TCP/IP (user level only) transport protocols for RPC.

The RPC protocol includes space for authentication parameters on every call. The contents of the authentication parameters are determined by the **flavor** (type of authentication used by the server and client). One server may support several different flavors of authentication at once.

The pre-defined authentication flavors are `AUTH_NULL` and `AUTH_UNIX`. `AUTH_NULL` (the default) passes no authentication information (null authentication). `AUTH_UNIX` passes the UNIX UID, GID, and groups with each call.

RPC provides a version number with each RPC request. Thus, one server can simultaneously service requests for several different versions of the protocol.



RPC and XDR Data Transfer

Note To ensure proper operation on the 8.0 release of the HP-UX operating system, you may need to recompile applications that use RPC. For more information, see Appendix E of the *Installing and Administering NFS Services* manual.

Remote Procedure Call Protocol Compiler (RPCGEN)

RPCGEN is a Remote Procedure Call compiler. It simplifies the creation of RPC applications by eliminating the time-consuming and difficult task of writing XDR routines. You have more time to debug your applications without having to debug network interface code.

RPCGEN compiles your remote program interface definitions, and produces C output files which you may use to produce remote versions of applications.

External Data Representation (XDR)

RPC uses an XDR to translate machine-dependent data formats (i.e., internal representations) to a universal format used by other network hosts using XDR. Therefore, XDR enables heterogeneous nodes and operating systems to talk with each other over the network.

The common way in which XDR represents a set of data types over a network takes care of problems such as different byte ordering on different communicating nodes. XDR also defines the size of each data type so that nodes with different structural alignment can share a common format over the network.

The XDR data definition language specifies the parameters and results of each RPC service procedure that a server provides. The XDR data definition language reads similarly to C language, although it contains a few new constructs.

Network Information Service (NIS)

NIS is an optional distributed network lookup service that provides read access to replicated databases.

Lookup Service: NIS maintains a set of databases for querying. Programs can ask for the value associated with a particular key or keys in a database.

Network Service: Programs do not need to know the location of data or how it is stored. Instead, they use a network protocol to communicate with a database server that knows those details.

Distributed: NIS is a collection of cooperating server processes that provide NIS clients access to data. One NIS **master** server propagates data across the network to other servers. Thus, it does not matter which server answers a request because the answer is the same everywhere.

Since the NIS interface uses RPC and XDR, the service may be available to non-UNIX operating systems and machines from other vendors.

NIS ASCII Source Files

NIS databases are constructed from ASCII files usually found in **/etc**. HP provides some standard functions for accessing the ASCII files' information. For example, the functions **getgrent** and **getpwent** are available to retrieve entries from the **/etc/group** and **/etc/passwd** files, respectively. These functions may also obtain data from NIS databases, if the databases exist.

By using the standard programmatic interfaces, you do not need to know where and how the data is stored.

If you write your own routines to retrieve data from these ASCII files rather than using the standard functions, you may receive results that are different from what the standard functions return. Note that HP does *not* support access other than through the standard HP-UX library routines. Therefore, we advise that you use the standard functions to access the ASCII files from which the standard NIS maps are built.

Refer to **ypclnt** (section 3C) and **yppasswd** (section 3N) of the *HP-UX Reference* for more information.

3

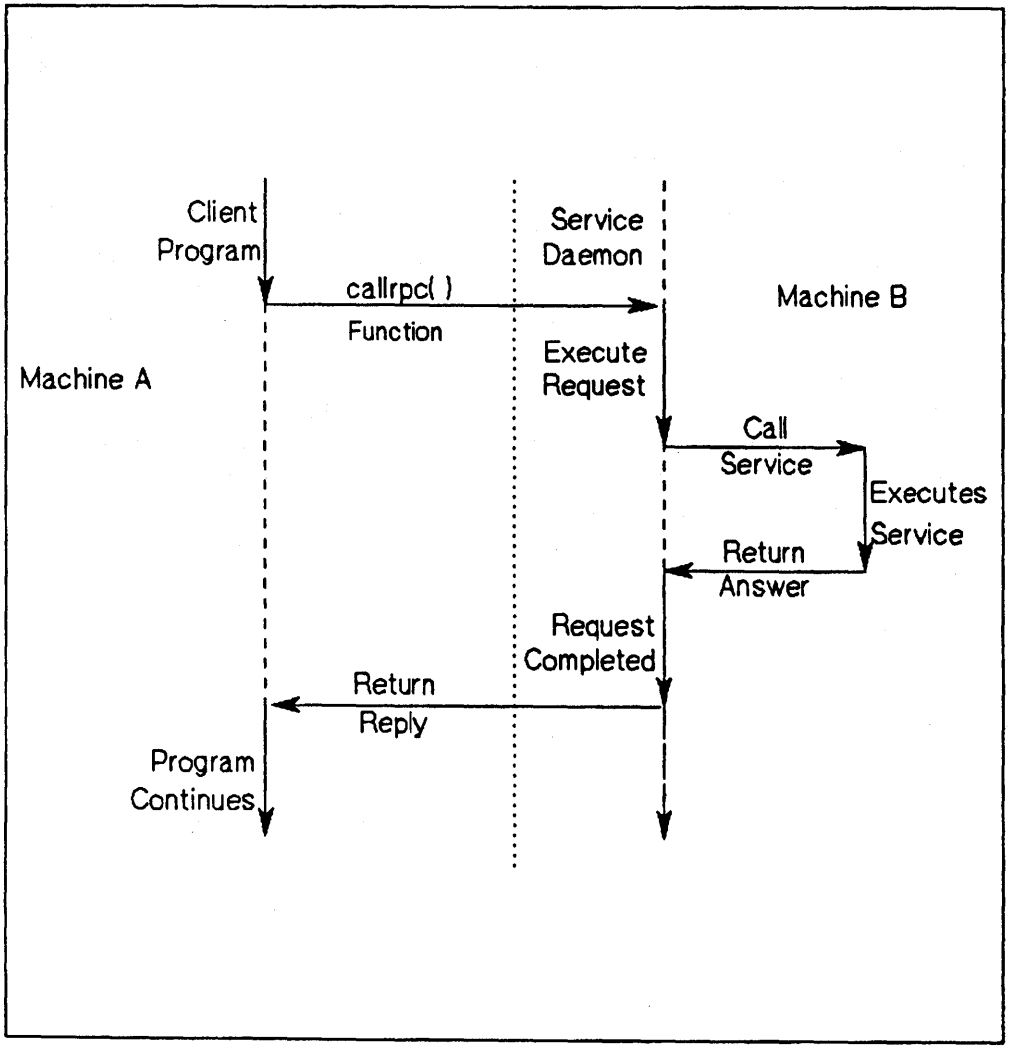
RPC Programming Guide

This chapter will help you write network applications using RPCs (Remote Procedure Calls), thus avoiding low-level system primitives based on sockets. You must be familiar with the C programming language and should have a working knowledge of networking.

Programs communicating over a network need a paradigm for communication. A low-level mechanism might send a signal on the arrival of incoming packets, causing a network signal handler to execute. A high-level mechanism would be the Ada rendezvous. This method is the RPC paradigm in which a client communicates with a server. The client first calls a procedure to send a data packet to the server. When the packet arrives, the server does the following:

- Extracts the procedure's parameters.
- Computes the results.
- Sends a reply message.
- Waits for the next call message.

You can use RPC to communicate between processes on the same node or on different nodes. This chapter discusses the C interface only.



Network Communication with the Remote Procedure Call

Layers of RPC

The RPC interface has three layers.

Highest Layer	The highest layer uses the network and is transparent to the programmer. For example, at this level a program can contain a call to **rnusers()** to return the number of users on a remote node. You do not have to know that RPC is being used since you simply make the call in a program (just as you would call **malloc()** to allocate memory).
Intermediate Layer	The middle-layer routines are for common applications; you do not need to know about sockets. To make RPC calls, use the **registerrpc()** and **callrpc()** routines. The **registerrpc()** routine obtains a unique system-wide number on the server; **callrpc()** executes a remote procedure from the client. For example, these routines are used to implement **rnusers()**.
Lowest Layer	The lowest layer is for more sophisticated applications that require altering the routine defaults. You can explicitly manipulate sockets that transmit RPC messages. HP recommends that you avoid this layer unless the upper two layers are not adequate.

Highest RPC Layer

The following table lists the RPC service library routines available to C programmers. (Refer to the *HP-UX Reference* for detailed information.)

RPC Library Routine	Description
`rnusers()`	Return the number of users on a remote node
`rusers()`	Return information about users on a remote node
`havedisk()`	Determine if a remote node has a disc
`rstat()`	Obtain performance data from a remote node
`rwall()`	Write to the specified remote nodes
`getmaster()`	Obtain the name of an NIS master server
`getrpcport()`	Obtain an RPC port number
`yppasswd()`	Update the user password in NIS

The other RPC services (`mount` and `spray`) are not available as library routines. They do, however, have RPC program numbers so you can invoke them with `callrpc()` as discussed in the next section.

EXAMPLE: To determine how many users logged on to a remote node, call the library routine **rnusers()**.

```
#include <stdio.h>

main(argc, argv)
     int argc;
     char *argv[ ];
{
     int num, rnusers();

     if (argc < 2) {
             fprintf(stderr, "usage: rnusers hostname\n");
             exit(1);
     }
     if ((num = rnusers(argv[1])) < 0) {
             fprintf(stderr, "error: rnusers\n");
             exit(1);
     }
     printf("%d users on %s\n", num, argv[1]);
     exit(0);
}
```

RPC library routines like **rnusers()** are in the RPC services library **librpcsvc.a**. Thus, you should compile the above program to create the **rnusers** program as follows.

```
% cc program.c -o rnusers -lrpcsvc
```

Intermediate RPC Layer

The intermediate RPC layer is the simplest interface that explicitly makes RPC calls using the functions `callrpc()` and `registerrpc()`.

A program number, version number, and procedure number define each RPC procedure. The program number defines a group of related remote procedures, each of which has a different procedure number. Each program also has a version number, so when a minor change is made to a remote service (e.g., adding a new procedure), you do not have to assign a new program number. When you want to call a remote procedure (e.g., to find the number of remote users) you look up the appropriate program, version, and procedure numbers similar to when you look up the name of the memory allocator when wanting to allocate memory.

EXAMPLE: This example shows you a way of using the intermediate RPC layer to obtain the number of remote users.

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

main(argc, argv)
     int argc;
     char *argv[ ];
{
     unsigned long nusers;

     if (argc < 2) {
             fprintf(stderr, "usage: nusers hostname\n");
             exit(1);
     }
     if (callrpc(argv[1],
             RUSERSPROG, RUSERSVERS, RUSERSPROC_NUM,
             xdr_void, 0, xdr_u_long, &nusers) != 0) {
                     fprintf(stderr, "error: callrpc\n");
                     exit(1);
     }
     printf("%d users on %s\n", nusers, argv[1]);
     exit(0);
}
```

callrpc()

The simplest routine in the RPC library for making remote procedure calls is `callrpc()`, which has eight parameters:

- The first parameter is the name of the remote node.
- The second through fourth parameters are the program, version, and procedure numbers.
- The fifth and sixth parameters define the argument of the RPC call.
- The final two parameters define the results of the call.

The `callrpc()` function returns zero if it completes successfully, or nonzero if it does not.

The meaning of the return values is an `enum clnt_stat` cast into an integer. You can find the `enum clnt_stat` definition in `<rpc/clnt.h>`.

Since data types may be represented differently on different nodes, `callrpc()` needs both the type of the RPC argument and a pointer to the argument. (Note, `callrpc()` needs similar information for the result.)

For `RUSERSPROC_NUM`, the return value is an unsigned long. Therefore, `callrpc()` has `xdr_u_long` as its seventh parameter, which means the result is of type `unsigned long`. The final parameter is `&nusers`, which is a pointer to where the unsigned long result will be placed. Since `RUSERSPROC_NUM` takes no argument, the parameters defining the `callrpc()` argument are zero (0) and `xdr_void`.

If `callrpc()` does not receive an answer after trying several times to deliver a message, it returns with an error code. The delivery mechanism is UDP (User Datagram Protocol). Methods for adjusting the number of retries or for using a different protocol require you to use the RPC library lowest layer. The remote server procedure that would reply to the call in the above program might look like the procedure that follows.

EXAMPLE:

```
char *
nuser(indata)
char *indata;
{
static int nusers;

/*
 * code here to compute the number of users
 * and place result in variable nusers
 */
return((char *)&nusers);
}
```

This procedure takes one argument, which is a pointer to the input of the RPC (ignored in the example). It also returns a pointer to the result. In C, character pointers are the generic pointers, so both the input argument and the return value are cast to `char *`.

A server usually registers all the RPC procedures it plans to handle and then goes into an infinite loop waiting to service requests. If there is only a single procedure to register, the main body of the server would look as follows.

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

char *nuser();

main()
{
     registerrpc(RUSERSPROG, RUSERSVERS, RUSERSPROC_NUM,
             nuser, xdr_void, xdr_u_long);
     svc_run();      /* never returns */
     fprintf(stderr, "Error: svc_run returned!\n");
     exit(1);
}
```

registerrpc()

The `registerrpc()` routine establishes which C procedure corresponds to each RPC procedure number.

- The first three parameters, `RUSERPROG`, `RUSERSVERS`, and `RUSERSPROC_NUM` are the program, version, and procedure numbers of the remote procedure to be registered. In the previous example, `nuser` argument is the name of the C procedure implementing the remote procedure.
- The `xdr_void` and `xdr_u_long` types are the type of input to and output from the procedure.

Only the UDP transport mechanism is used by `registerrpc()`; thus, it is always safe to use `registerrpc()` in conjunction with calls generated by `callrpc()`.

Note The UDP transport mechanism can only deal with arguments and results that are less than 8K bytes in length.

Program Numbers

Program numbers are assigned in groups of 0x20000000 as follows.

```
       0 - 1fffffff defined by Sun
20000000 - 3fffffff defined by user
40000000 - 5fffffff transient
60000000 - 7fffffff reserved
80000000 - 9fffffff reserved
a0000000 - bfffffff reserved
c0000000 - dfffffff reserved
e0000000 - ffffffff reserved
```

```
0 - 1fffffff defined by Sun
```

Sun Microsystems, Inc. administers the first group of numbers which should be identical for all systems. If you develop an application of general interest, that application should receive an assigned number in the first range.

```
20000000 - 3fffffff defined by user
```

The second group of numbers is reserved for specific customer applications. This range is primarily for debugging new programs.

```
40000000 - 5fffffff transient
```

The third group is reserved for applications that generate program numbers dynamically.

```
60000000 - 7fffffff reserved
80000000 - 9fffffff reserved
a0000000 - bfffffff reserved
c0000000 - dfffffff reserved
e0000000 - ffffffff reserved
```

The final groups are reserved for future use and should not be used.

To register a protocol specification, send a request to the following location. Please include a complete protocol specification, similar to those in this manual. In return, you will receive a unique program number.

Network Administration Office, Dept. NET
Information Networks Division
19420 Homestead Road
Cupertino, California 95014
408-447-3444

Pass Arbitrary Data Types

RPC can handle arbitrary data structures, regardless of different nodes' byte orders or structure layout conventions. RPC handles these structures by converting them to a network standard form called XDR (**eXternal Data Representation**) before sending them over the network. The process of converting from a particular node representation to XDR format is **serializing,** and the reverse process is **deserializing.** The type field parameters of `callrpc()` and `registerrpc()` can be a built-in procedure (like `xdr_u_long()` in the previous example) or a user supplied one. XDR has the following built-in type routines:

- `xdr_bool()`
- `xdr_char()`
- `xdr_short()`
- `xdr_enum()`
- `xdr_float()`
- `xdr_int()`
- `xdr_long()`
- `xdr_opaque()`
- `xdr_double()`
- `xdr_u_char()`
- `xdr_u_int()`
- `xdr_u_long()`
- `xdr_u_short()`
- `xdr_void()`

EXAMPLE: This example describes a user-defined type routine.

1. Send the following structure.

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

2. Call `callrpc()` as follows.

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

3. Write `xdr_simple()` as follows.

```
#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);
}
```

An XDR routine returns nonzero (true for C) if it completes successfully, or zero (false) if it does not. (Refer to the "XDR Protocol Specification" chapter for more XDR implementation examples.)

In addition to the built-in primitives, there are the following prefabricated building blocks:

- `xdr_array( )`
- `xdr_bytes( )`
- `xdr_pointer( )`
- `xdr_vector( )`
- `xdr_string( )`
- `xdr_union( )`
- `xdr_reference( )`
- `xdr_free( )`

EXAMPLE:

1. To send a variable array of integers, you might package them as a structure.

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

2. Make an RPC call.

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

3. Define the `xdr_varintarr()`.

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

The previous **xdr_array()** routine takes the following as parameters:

- The 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.

EXAMPLE: If both the client and server know the array size in advance, you could use the following function to send out an array of length *SIZE.*

```
int intarr[SIZE];

xdr_intarr(xdrsp,intarr)
     XDR *xdrsp;
     int intarr[ ];
{
     int i;

     for (i = 0; i < SIZE; i++) {
            if (!xdr_int(xdrsp,&intarr[i]))
                   return (0);
     }
     return (1);
}
```

XDR always converts objects such that their lengths are each a multiple of 4-bytes. Thus, if either of the above examples involved characters instead of integers, each character would occupy 32 bits. The XDR routine **xdr_bytes()** is like **xdr_array()** except that it packs characters; **xdr_bytes()** has four parameters, similar to the first four parameters of **xdr_array()**. For null-terminated strings, the **xdr_string()** routine is the same as **xdr_bytes()** without the length parameter. When serializing, it obtains the string length using **strlen()**; when deserializing, it creates a null-terminated string.

EXAMPLE: This example calls the previously written **xdr_simple()** and the built-in functions **xdr_string()** and **xdr_reference()**. The **xdr_reference()** function dereferences pointers.

```
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 RPC Layer

In the previous examples RPC automatically takes care of many details for you. Refer to this section to change the defaults by using the RPC library lowest layer. You should be familiar with sockets and system calls before attempting to use them.

You may have several occasions to use RPC lower layers:

- You may need to use TCP. The higher layers use UDP, which restricts RPC calls to 8K bytes of data. Using TCP permits calls to send longer streams of data. (See the "Additional RPC Examples, TCP" section.)
- You may want to allocate and free memory while serializing or deserializing with XDR routines. The higher layer does not contain a call to let you free memory explicitly. (See the "Memory Allocation with XDR" section.)
- You may need to perform authentication on either the client or server side by supplying credentials or verifying them. (See the "Additional RPC Features, Authentication" section.)

RPC Server Side

The server for the `nusers` program shown below performs the same function as the one using `registerrpc()`, except it uses a lower RPC layer.

```
#include <stdio.h>
#include <rpc/rpc.h>
#include <rpcsvc/rusers.h>

main( )
{
    SVCXPRT *transp;
    int nuser();
    transp = svcudp_create(RPC_ANYSOCK);
    if (transp == NULL){
            fprintf(stderr, "cannot create an RPC server\n");
             exit(1);
     }
     pmap_unset(RUSERSPROG, RUSERSVERS);
     if (!svc_register(transp, RUSERSPROG, RUSERSVERS,
                     nuser, IPPROTO_UDP)) {
             fprintf(stderr, "cannot register RUSERS service\n");
             exit(1);
     }
     svc_run();      /* never returns */
     fprintf(stderr, "should never reach this point\n");
}

nuser(rqstp, transp)
     struct svc_req *rqstp;
     SVCXPRT *transp;
{
     unsigned long nusers;

     switch (rqstp->rq_proc) {
     case  NULLPROC:
             if (!svc_sendreply(transp, xdr_void, 0)) {
                     fprintf(stderr, "cannot reply to RPC call\n");
                     exit(1);
             }
             return;
```

```
        case  RUSERSPROC_NUM:
                /*
                 * 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;
        }
}
```

First, the server receives a transport handle for sending RPC messages. The `registerrpc()` function uses `svcudp_create()` to obtain a UDP handle. If you require a reliable protocol, call `svctcp_create()` instead. If the argument to `svcudp_create()` is `RPC_ANYSOCK`, the RPC library creates a socket on which to send RPC calls. Otherwise, `svcudp_create()` expects its argument to be a valid socket number. If specifying your own socket, it can be bound or unbound. If it is bound, the port numbers of `svcudp_create()` and `clntudp_create()` (the low-level client routine) must match.

When you specify `RPC_ANYSOCK` for a socket or give an unbound socket, the system determines port numbers in the following way:

- The server selects a port number for the RPC procedure if the socket specified to `svcudp_create()` is not already bound.
- When a server starts, it registers that port number with the portmapper daemon on its local node.
- When the `clntudp_create()` call is made with an unbound socket, the system queries the portmapper on the node to which the call is being made and obtains the appropriate port number.
- The RPC call fails if the portmapper is not running or has no port corresponding to the RPC call.

You can make RPC calls directly to the portmapper using the appropriate procedure numbers defined in the include file **<rpc/pmap_prot.h>**.

After creating a service transport, call **pmap_unset()** so if the **nusers** server crashed earlier, any previous trace of it is erased before restarting. The **pmap_unset()** call erases the entry for **RUSERSPROG** from the portmapper's tables.

Associate the program number for **nusers** with the procedure **nuser()**. The final argument to **svc_register()** is the protocol being used; in this case, it is **IPPROTO_UDP**. Notice that unlike **registerrpc()**, no XDR routines are involved in the registration process. The registration occurs on the program, rather than procedure level.

The user routine **nuser()** must call and dispatch the appropriate XDR routines based on the procedure number. Note that **nuser()** handles two items that **registerrpc()** handles automatically:

- First, the procedure **NULLPROC** (currently zero) returns with no arguments. You can use **NULLPROC** as a simple test for detecting if a remote program is running.
- Second, a check occurs for invalid procedure numbers. If one 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( )**. Its first parameter is the service transport handle, the second is the XDR routine, and the third is a pointer to the data to be returned.

A server can handle an RPC program that passes data.

EXAMPLE: This example adds a procedure RUSERSPROC_BOOL that has an argument nusers. The procedure returns TRUE or FALSE, depending on whether *nusers* users are logged on to the node.

```
case RUSERSPROC_BOOL: {
     int bool;
     unsigned long nuserquery;
     if (!svc_getargs(transp, xdr_u_long, &nuserquery)) {

             svcerr_decode(transp);
             return;
     }
     /*
      * code to set nusers = number of users
      */
     if (nuserquery == nusers)
             bool = TRUE;
     else
             bool = FALSE;
     if (!svc_sendreply(transp, xdr_bool, &bool){
             fprintf(stderr, "cannot reply to RPC call\n");
             exit(1);
     }
     return;
}
```

The relevant routine is **svc_getargs()**, which takes the following arguments: a service transport handle, the XDR routine, and a pointer to where the input is to be placed.

Memory Allocation with XDR

XDR routines not only perform input and output, they may also perform memory allocation. For this reason the second parameter of **xdr_array()** is a pointer to an array, rather than the actual array. If it is NULL when deserializing, **xdr_array()** allocates space for the array and returns a pointer to it, putting the size of the array in the third argument.

EXAMPLE: The following XDR routine `xdr_chararr1()` has 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));
}
```

The routine may be called from a server as follows.

```
char chararr[SIZE];

     svc_getargs(transp, xdr_chararr1, chararr);
```

The `chararr` has already allocated space. If you want XDR to do the allocation, you would have to rewrite this routine in the following way.

```
xdr_chararr2(xdrsp, chararrp)
XDR *xdrsp;
char **chararrp;
{
     int len;

     len = SIZE;
     return (xdr_bytes(xdrsp, charrarrp, &len, SIZE));
}
```

Then the RPC call might look as follows.

```
char *arrptr;
arrptr = NULL;
svc_getargs(transp, xdr_chararr2, &arrptr);
/*
 * use the result here
 */
svc_freeargs(transp, xdr_chararr2, &arrptr);
```

After using the character array, it can be freed with **svc_freeargs()**. In the routine **xdr_finalexample()** given earlier, if **finalp->string** was **NULL** in the call

```
svc_getargs(transp, xdr_finalexample, &finalp);
```

then,

```
svc_freeargs(xdrsp, xdr_finalexample, &finalp);
```

frees the array allocated to hold **finalp->string**; otherwise, it frees nothing. The same is true for **finalp->simplep**.

Each XDR routine is responsible for serializing, deserializing, and allocating memory:

- When an XDR routine is called from **callrpc()**, the serializer part is used.
- When an XDR routine is called from **svc_getargs()**, the deserializer is used.
- When an XDR routine is called from **svc_freeargs()** the memory deallocator is used.

When building simple programs like the examples in this section, you do not have to worry about the three modes. Refer to the "XDR Protocol Specification" chapter for examples of more sophisticated XDR routines that determine which of the three modes to use.

RPC Calling Side

When using `callrpc()` you have no control over the RPC delivery mechanism or the socket used to transport the data. To illustrate the RPC layer that lets you adjust these parameters, consider the following code that calls the `nusers` service.

EXAMPLE:

```
#include      <stdio.h>
#include      <rpc/rpc.h>
#include      <rpcsvc/rusers.h>
#include      <sys/socket.h>
#include      <time.h>
#include      <netdb.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;
     unsigned long nusers;

     if (argc < 2) {
             fprintf(stderr, "usage: nusers hostname\n");
             exit(1);
     }
     if ((hp = gethostbyname(argv[1])) == NULL) {
             fprintf(stderr, "cannot get addr for %s\n",argv[1]);
             exit(1);
     }
     pertry_timeout.tv_sec = 3;
     pertry_timeout.tv_usec = 0;
     memcpy((caddr_t)&server->addr.sin_addr, hp->h_addr, hp->h_length);
     server_addr.sin_family = AF_INET;
     server_addr.sin_port =  0;
     if ((client = clntudp_create(&server_addr, RUSERSPROG,
             RUSERSVERS, pertry_timeout, &sock)) == NULL) {
```

```
                clnt_pcreateerror("clntudp_create");
                exit(1);
        }
        total_timeout.tv_sec = 20;
        total_timeout.tv_usec = 0;
        clnt_stat = clnt_call(client, RUSERSPROC_NUM, xdr_void,
                0, xdr_u_long, &nusers, total_timeout);
        if (clnt_stat != RPC_SUCCESS) {
                clnt_perror(client, "rpc");
                exit(1);
        }
        clnt_destroy(client);
}
```

The low-level version of `callrpc()` is `clnt_call()`; it takes a `CLIENT` pointer rather than a host name. The parameters to `clnt_call()` are the following:

- The `CLIENT` pointer.
- The procedure number.
- The XDR routine for serializing the argument.
- A pointer to the argument.
- The XDR routine for deserializing the return value.
- A pointer to where the return value will be placed.
- The length of time to wait for a reply.

The `CLIENT` pointer is encoded with the transport mechanism. The `callrpc()` routine uses UDP and thus, calls `clntudp_create()` to obtain a `CLIENT` pointer. To use TCP, call `clnttcp_create()` instead.

The parameters to `clntudp_create()` are the following:

- The server address.
- The program number.
- The version number.
- A timeout value (between tries).
- A pointer to a file descriptor for a socket.

The final argument to `clnt_call()` is the total time to wait for a response. The number of tries is the `clnt_call()` timeout divided by the `clntudp_create()` timeout rounded down to the nearest integer.

The `clnt_destroy()` call deallocates any space associated with the `CLIENT` handle. It does not close the associated socket that was passed as an argument to `clntudp_create()`. The reason is that if there are multiple client handles using the same socket, then you can close one handle without destroying the socket that other handles are using.

To make a stream connection, replace the call to `clntudp_create()` with a call to `clnttcp_create()`.

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

No timeout argument exists; instead, you must specify the receive and send buffer sizes. When the `clnttcp_create()` call is made, a TCP connection is established. All RPC calls using that `CLIENT` handle will use this connection. The server side of an RPC call using TCP has `svcudp_create()` replaced by `svctcp_create()`.

Additional RPC Features

This section contains other RPC features you may occasionally find useful.

Select on the Server Side

Suppose a process is processing RPC requests while performing some other activity. If the other activity includes periodically updating a data structure, the process can set an alarm signal before calling `svc_run()`. However, if the other activity involves waiting on a file descriptor, the `svc_run()` call will not work. The code for `svc_run()` is as follows:

```
void
svc_run()
{
        fd_set readfds;

        for (;;) {
                readfds = svc_fds;
                switch (select(FD_SETSIZE, &readfds, NULL, NULL, NULL)) {

                case -1:
                        if (errno == EINTR)
                                continue;
                        perror("svc_run: select");
                        return;
                case 0:
                        break;
                default:
                        svc_getreqset(readfds);
                }
        }
}
```

You can bypass `svc_run()` and call `svc_getreqset()`. You only need to know the file descriptors of the socket associated with the programs on which you are waiting. Thus, you can have your own `select()` waiting on both the RPC socket and your own descriptors.

Broadcast RPC

The portmapper is a daemon that converts RPC program numbers into IP protocol port numbers. (See **portmap** in section 1M of the *HP-UX Reference.*) You cannot perform broadcast RPC without the portmapper in conjunction with standard RPC protocols. Refer to the following list of differences between broadcast RPC and normal RPC calls:

- Normal RPC expects one answer, whereas broadcast RPC expects many answers (one or more answers from each responding node).
- Only packet-oriented (connectionless) transport protocols (like UDP/IP) can support broadcast RPC.
- The broadcast RPC implementation ignores all unsuccessful responses. Thus, if a version mismatch occurs between the broadcaster and a remote service, the user of broadcast RPC never knows.
- Broadcast RPC sends all messages to the portmap port. Thus, only services that register with their portmapper are accessible via the broadcast RPC mechanism.

Broadcast RPC Synopsis

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

enum clnt_stat
clnt_broadcast(prog, vers, proc, xargs, argsp, xresults,
               resultsp, eachresult)
u_long  prog;                   /* program number */
u_long  vers;                   /* version number */
u_long  proc;                   /* procedure number */
xdrproc_t  xargs;               /* xdr routine for args */
caddr_t  argsp;                 /* pointer to args */
xdrproc_t  xresults;            /* xdr routine for results */
caddr_t  resultsp;              /* pointer to results */
bool_t  (*eachresult)();        /* call with each result gotten */
```

The `eachresult()` function is called each time a valid result is obtained. It returns a boolean indicating whether the client wants more responses.

```
bool_t
eachresult(resultsp, raddr)
caddr_t resultsp;               /* location of results */
struct sockaddr_in *raddr;      /* IP addr of responding machine */
```

If `eachresult()` returns `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`. To interpret `clnt_stat` errors, call `clnt_perrno()` with the error code.

Batching

In the RPC architecture, clients send a call message and wait for servers to reply that the call succeeded. This procedure implies that clients do not compute while servers are processing a call. It is inefficient if the client does not want or need an acknowledgement for every message sent. Using RPC batch facilities, clients can continue computing while waiting for a response.

Batching is the process of placing RPC messages in a pipeline of calls to a desired server. Batching assumes the following items:

- 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 (i.e., TCP/IP).

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

Note Since the batched calls are buffered, the client should eventually make a non-batched call to flush the pipeline.

EXAMPLE: Assume a string rendering service (like a window system) has two similar calls: one renders a string and returns void results, while the other renders a string and remains silent. The service using the TCP/IP transport may look like this example.

```
#include <stdio.h>
#include <rpc/rpc.h>
#include "windows.h"  /* contains the values of WINDOWPROG
                       * and WINDOWVERS
                       */

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");
                     exit(1);
             }
```

```
                return;
        case  RENDERSTRING:
                if (!svc_getargs(transp, xdr_wrapstring, &s)) {
                        fprintf(stderr, "cannot decode arguments\n");
                        /*
                         * tell caller that a problem exists
                         */
                        svcerr_decode(transp);
                        break;
                }
                /*
                 * call here to render the string s
                 */
                if (!svc_sendreply(transp, xdr_void, NULL)) {
                        fprintf(stderr, "cannot reply to RPC call\n");
                        exit(1);
                }
                break;
        case  RENDERSTRING_BATCHED:
                if (!svc_getargs(transp, xdr_wrapstring, &s)) {
                        fprintf(stderr, "cannot decode arguments\n");
                        /*
                         * the server cannot return errors to the client
                         * when using batched RPC
                         */
                        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 the procedure should respond. For a client to take advantage of batching, the client must perform RPC calls on a TCP-based transport and the actual calls must have the following attributes:

- The result's XDR routine must be zero.
- The RPC call's timeout must be zero.

EXAMPLE: This is an example of a client using batching to render strings. The batching is flushed when the client receives a null string.

```
#include <stdio.h>
#include <rpc/rpc.h>
#include <sys/socket.h>
#include <time.h>
#include <netdb.h>
#include "windows.h"

main(argc, argv)
     int argc;
     char *argv[ ];
{
     struct hostent *hp;
     struct timeval total_timeout;
     struct sockaddr_in server_addr;
     int sock = RPC_ANYSOCK;
     register CLIENT *client;
     enum clnt_stat clnt_stat;
     char buf[BUFSIZ], *s = buf;

     if (argc < 2) {
             fprintf(stderr, "usage: nusers hostname\n");
             exit(1);
     }

     if ((hp = gethostbyname(argv[1])) == NULL) {
             fprintf(stderr, "cannot get addr for %s\n",argv[1]);
             exit(1);
     }

     memcpy((caddr_t)&server->addr.sin_addr, hp->h_addr, hp->h_length);
     server_addr.sin_family = AF_INET;
     server_addr.sin_port =  0;
```

```
        if ((client = clnttcp_create(&server_addr,
                WINDOWPROG, WINDOWVERS, &sock, 0, 0)) == NULL) {
                        clnt_pcreateerror("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);
}
```

Since the server sends no message, the clients cannot be notified of any failures that may occur.

Authentication

In the previous examples the caller never identified itself to the server, and the server never required an ID from the caller. Some network services, such as a network file system, require stronger security than what has been presented thus far.

The RPC package on the server authenticates every RPC call, and similarly, the RPC client package generates and sends authentication parameters. Just as different transports (TCP/IP or UDP/IP) can be used when creating RPC clients and servers, different forms of authentication can be associated with RPC clients. The authentication type used as a default is type `AUTH_NULL`.

The authentication subsystem of the RPC package is open ended; numerous types of authentication are easy to support. However, this section deals only with UNIX type authentication which is the only supported type except `AUTH_NULL`.

RPC Client Side

When a caller creates a new RPC client handle as in

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

the appropriate transport instance defaults the associate authentication handle to be as follows.

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

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

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

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

```
/*
 * 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 effective UID */
     int aup_gid;           /* client's effective GID */
     u_int aup_len;         /* element length of aup_gids */
     int *aup_gids;         /* array of groups to which the user belongs */
}
```

These fields are set by **authunix_create_default()** by invoking the appropriate system calls. Since the RPC user created this new style of authentication, the user is responsible for destroying it to conserve memory.

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

RPC Server Side

Service implementors have a harder time handling authentication issues since the RPC package passes the service dispatch routine a request with an associated 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 vers num */
     u_long rq_proc;              /* desired procedure number */
     struct opaque_auth rq_cred;  /* raw credential from network */
     caddr_t rq_clntcred;         /* credentials (read only) */
};
```

The `rq_cred` is mostly opaque except for one field of interest: 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 the following two items to the service dispatch routine:

- The request's `rq_cred` is well-formed. Thus, 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 inspect the other fields of `rq_cred` if the style is not supported by the RPC package.
- The request's `rq_clntcred` field is either `NULL` or points to a well-formed structure corresponding to supported authentication credentials. Only UNIX `rq_clntcred` could be cast to a pointer to an `authunix_parms` structure. If `rq_clntcred` is `NULL`, the server may wish to inspect the other (opaque) fields of `rq_cred` if it knows about a new type of authentication about which the RPC package does not know.

Note The RPC protocol allows you to specify your own form of authentication, but to do so you must have access to the RPC authentication source files. Implementations based on NFS 3.2 (including HP-UX 8.0 on HP 9000 computers) do *not* allow you to define your own form of authentication.

EXAMPLE: This example extends the remote users service example 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
              * this disallows uid 16 to use this service
              */
             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 (procedure number zero).

If the authentication parameter's type is not suitable for your service, you should call **svcerr_weakauth()**.

The service protocol should return status for access denied. In the above example, the protocol does not have such a status, so the service primitive **svcerr_systemerr()** is called instead. This point underscores the relation between the RPC authentication package and the services; RPC deals only with authentication and not with individual services' access control. The services must implement their own access control policies and reflect these policies as return statuses in their protocols.

Using inetd

An RPC server can start from **inetd**. The only difference from the usual code is that **svcudp_create()** should be called as

```
transp = svcudp_create(0);
```

since **inetd(1M)** passes a socket as file descriptor zero (0). You should call **svc_register()** as

```
svc_register(transp, PROGNUM, VERSNUM, service, 0);
```

with the final parameter set to zero (0), since the program would already be registered by **inetd**. If you want to exit from the server process and return control to **inetd**, you must explicitly exit since **svc_run()** never returns.

To use TCP based RPC from the `inetd` daemon, call `svcfd_create()` instead of `svctcp_create()` since the socket (file descriptor zero (0)) is already an active socket.

The entry formats in /etc/inetd.conf for RPC services are as follows:

UDP:

```
rpc dgram udp wait user server program version name
```

TCP:

```
rpc stream tcp nowait user server program version name
```

/etc/inetd.conf Fields	Description
`user`	The user name that the process executes as
`server`	The server program
`program`	Program number of the service
`version`	Version number of the service
`name`	The server name and optional arguments

EXAMPLES:

```
rpc dgram udp wait root /usr/etc/rpc.mountd 100005 1 rpc.mountd
```

If the same program handles multiple versions, the version number can be a range as in the following line.

```
rpc dgram udp wait root /usr/etc/rpc.rstatd 100001 1-3 rpc.rstatd
```

Additional RPC Examples

Versions

By convention, the first version number of program PROG is PROGVERS_ORIG, and the most recent version is PROGVERS. Suppose there is a new version of the**user** program that returns an **unsigned short** rather than a long. If the name of this version is RUSERSVERS_SHORT, a server that wants to support both versions would perform a double register.

```
if (!svc_register(transp, RUSERSPROG, RUSERSVERS_ORIG,
            nuser, IPPROTO_UDP)) {
     fprintf(stderr, "cannot register RUSER service\n");
     exit(1);
}
if (!svc_register(transp, RUSERSPROG, RUSERSVERS_SHORT,
            nuser, IPPROTO_UDP)) {
     fprintf(stderr, "cannot register RUSER service\n");
     exit(1);
}
```

The same C procedure can handle both programs.

```
nuser(rqstp, transp)
     struct svc_req *rqstp;
     SVCXPRT *transp;
{
     unsigned long nusers;
     unsigned short nusers2;

     switch (rqstp->rq_proc) {
     case  NULLPROC:
             if (!svc_sendreply(transp, xdr_void, 0)) {
                     fprintf(stderr, "cannot reply to RPC call\n");
                     exit(1);
             }
             return;
     case  RUSERSPROC_NUM:
             /*
              * code here to compute the number of users
              * and put in variable nusers and in nusers2
              */
             if (rqstp->rq-vers == RUSERSVERS_ORIG) {
                     if (!svc_sendreply(transp,xdr_u_long, &nusers)) {
                             fprintf(stderr, "cannot reply to RPC call \n");
                             exit(1);
                     }
             } else if (!svc_sendreply (transp,xdr_u_short,&nusers2)) {
                     fprintf (stderr, "cannot reply to RPC call \n");
                     exit(1);
             }
             return;
     default:
             svcerr_noproc(transp);
             return;
     }
}
```

TCP

The following example is a routine to perform a remote copy. The initiator of the RPC call takes its standard input and sends it to the server to print it on standard output. The RPC call uses TCP. This example also illustrates an XDR procedure that behaves differently on serialization than on deserialization.

EXAMPLE:

```
/*
 * The xdr routine:
 * on decode, read from network, write onto fp
 * on encode, read from fp, write onto network
 */
#include <stdio.h>
#include <rpc/rpc.h>

xdr_rcp(xdrs, fp)
     XDR *xdrs;
     FILE *fp;
{
     unsigned long size;
     char buf[BUFSIZ], *p;

     if (xdrs->x_op == XDR_FREE)      /* nothing to free */
             return 1;
     while (1) {
             if (xdrs->x_op == XDR_ENCODE) {
                     if ((size = fread(buf, sizeof(char), BUFSIZ,
                                       fp)) == 0 & ferror(fp)) {
                     fprintf(stderr, "cannot fread\n");
                     exit(1);
             }
     }
     p = buf;
     if (!xdr_bytes(xdrs, &p, &size, BUFSIZ))
             return (0)
     if (size == 0)
             return (1)
     if (xdrs->x_op == XDR_DECODE) {
             if (fwrite(buf, sizeof(char), size,
                     fp) != size) {
                             fprintf(stderr, "cannot fwrite\n");
                             exit(1);
                     }
```

```
            }
        }
}

/*
 * The sender routines (client)
 */
#include <stdio.h>
#include <netdb.h>
#include <rpc/rpc.h>
#include <sys/socket.h>
#include <time.h>

int xdr_rcp(), callrpctcp();

main(argc, argv)
        int argc;
        char *argv[];
{
        int err;

        if (argc < 2) {
                fprintf(stderr, "usage: %s servername\n", argv[0]);
                exit(1);
        }
        if ((err = callrpctcp(argv[1], RCPPROG, RCPPROC_FP,
                        RCPVERS, xdr_rcp, stdin, xdr_void, 0) != 0)) {
                clnt_perrno(err);
                fprintf(stderr, "cannot make RPC call\n");
                exit(1);
        }
}
callrpctcp(host, prognum, procnum, versnum, inproc, in, outproc, out)
        char *host, *in, *out;
        int prognum, procnum, versnum;
        xdrproc_t inproc, outproc;
{
        struct sockaddr_in server_addr;
        int socket = RPC_ANYSOCK;
        enum clnt_stat clnt_stat;
        struct hostent *hp;
        register CLIENT *client;
        struct timeval total_timeout;

        if ((hp = gethostbyname(host)) == NULL) {
```

```
            fprintf(stderr, "cannot get addr for '%s'\n", host);
            exit(1);
        }

        memcpy((caddr_t)&server->addr.sin_addr, hp->h_addr, hp->h_length);
        server_addr.sin_family = AF_INET;
        server_addr.sin_port =  0;
        if ((client = clnttcp_create(&server_addr, prognum,
                        versnum, &socket, BUFSIZ, BUFSIZ)) == NULL) {
            clnt_pcreateerror("rpctcp_create");
            exit(1);
        }
        total_timeout.tv_sec = 20;
        total_timeout.tv_usec = 0;
        clnt_stat = clnt_call(client, procnum,
                        inproc, in, outproc, out, total_timeout);
        clnt_destroy(client);
        return (int)clnt_stat;
}

/*
 * The receiving routines (server)
 */
#include <stdio.h>
#include <rpc/rpc.h>

main()
{
        register SVCXPRT *transp;

        if ((transp = svctcp_create(RPC_ANYSOCK, BUFSIZ, BUFSIZ)) == NULL)
        {
            fprintf(stderr, "svctcp_create: error\n");
            exit(1);
        }
        pmap_unset(RCPPROG, RCPVERS);
        if (!svc_register(transp,
            RCPPROG, RCPVERS, rcp_service, IPPROTO_TCP)) {
                    fprintf(stderr, "svc_register: error\n");
                    exit(1);
        }
        svc_run();      /* never returns */
        fprintf(stderr, "svc_run should never return\n");
}
```

```
rcp_service(rqstp, transp)
     register struct svc_req *rqstp;
     register SVCXPRT *transp;
{
     switch (rqstp->rq_proc) {
     case  NULLPROC:
             if (svc_sendreply(transp, xdr_void, 0) == 0) {
                     fprintf(stderr, "err: rcp_service\n");
                     exit(1);
             }
             return;
     case  RCPPROC_FP:
             if (!svc_getargs(transp, xdr_rcp, stdout)) {
                     svcerr_decode(transp);
                     return;
             }
             if (!svc_sendreply(transp, xdr_void, 0)) {
                     fprintf(stderr, "cannot send reply\n");
                     return;
             }
             exit(0);
     default:
             svcerr_noproc(transp);
             return;
     }
}
```

Callback Procedures

You may want a server to become a client and make an RPC call back to the process which is its client. One example is remote debugging where the client is a window system program and the server is a debugger running on the remote node. Usually the user clicks a mouse button at the debugging window to select a debugger command. The application then makes an RPC call to the server (where the debugger is actually running), telling it to execute that command. However, when the debugger reaches a breakpoint, the roles reverse and the debugger makes an RPC call to the window program to inform the user that a breakpoint was reached.

To perform an RPC callback, you need a program number on which to make the RPC call. Since this program number is dynamically generated, it should be in the transient range, `0x40000000 - 0x5fffffff`. The routine `gettransient()` returns a valid program number in the transient range and registers it with the portmapper. It only talks to the portmapper running on the same node as the `gettransient()` routine. The call to `pmap_set()` is a test and set operation in that it indivisibly tests whether a program number was already registered. If it was not, then the `pmap_set` call reserves it. On return, the `sockp` argument contains a socket that can be used as the argument to an `svcudp_create()` or `svctcp_create()` call.

EXAMPLE:

```
#include <stdio.h>
#include <rpc/rpc.h>
#include <sys/socket.h>

u_long
gettransient(proto, vers, sockp)
     int proto;
     u_long vers;
     int *sockp;
{
     static u_long 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
      */
     (void) bind(s, &addr, len);
     if (getsockname(s, &addr, &len)< 0) {
```

```
                perror("getsockname");
                return (0);
        }
        while (!pmap_set(prognum++, vers, proto, addr.sin_port))
                continue;
        return (prognum-1);
}
```

The following pair of programs illustrate how to use the `gettransient()` routine:

- The client makes an RPC call to the server, passing it a transient program number.
- The client then waits to receive a callback from the server at that program number.
- The server registers the program **EXAMPLEPROG** so it can receive the RPC call informing it of the callback program number.
- After receiving a **SIGALRM** signal, the server sends a callback RPC call using the program number it received earlier.

EXAMPLE:

```
/*
 * client
 */
#include <stdio.h>
#include <rpc/rpc.h>
#include "example.h"

int callback();
u_long gettransient(), x;
char hostname[256];

main(argc, argv)
     int argc;
     char *argv[ ];
{
     int ans, s;
     SVCXPRT *xprt;

     gethostname(hostname, sizeof(hostname));
     s = RPC_ANYSOCK;
     x = gettransient(IPPROTO_UDP, 1, &s);
     fprintf(stderr, "client gets prognum %ld\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 (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 NULLPROC:
             if (!svc_sendreply(transp, xdr_void, 0)) {
                     fprintf(stderr, "err: callback\n");
                     exit(1);
             }
             pmap_unset(x,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: callbackd\n");
                     exit(1);
             }
     }
}

/*
 * server
 */
#include <stdio.h>
#include <rpc/rpc.h>
#include <sys/signal.h>
#include "example.h"

char *getnewprog();
char hostname[256];
int docallback();
u_long pnum=0;          /* program number for callback routine */

main(argc, argv)
     int argc;
     char *argv[];
{
     gethostname(hostname, sizeof(hostname));
     registerrpc(EXAMPLEPROG, EXAMPLEVERS,
             EXAMPLEPROC_CALLBACK, getnewprog, xdr_int, xdr_void);
     fprintf(stderr, "server going into svc_run\n");
     signal(SIGALRM, docallback);
     alarm(10);
```

```
        svc_run();
        fprintf(stderr, "Error: svc_run shouldn't return\n");
}

char *
getnewprog(pnump)
        u_long *pnump;
{
        pnum = *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");
        }
}
```

Synopsis of RPC Routines

Routine	`auth_destroy()`
Description	A macro that destroys the authentication information associated with `auth`. Destruction usually involves deallocation of private data structures. The use of `auth` is undefined after calling `auth_destroy()`.

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

Routine	`authnone_create()`
Description	Creates and returns an RPC authentication handle that passes no usable authentication information with each remote procedure call. This routine returns `NULL` if it fails.
Synopsis	`AUTH *` `authnone_create()`

Routine	`authunix_create()`
Description	Creates and returns an RPC authentication handle that contains authentication information. The parameter *host* is the node name on which the information was created. The parameter *uid* is the user's user ID. The parameter *gid* is the user's current group ID. The parameters *len* and *aup_gids* refer to a counted array of groups to which the user belongs. This routine returns `NULL` if it fails.

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

Routine	`authunix_create_default()`
Description	Calls `authunix_create()` with the appropriate parameters.
Synopsis	`AUTH *` `authunix_create_default()`

Routine	`callrpc()`
Description	Calls the remote procedure associated with **prognum**, **versum**, and **procnum** on the *host* node. The parameter *in* is the address of the procedure's argument(s), and *out* is the address of where to place the results. The parameter *inproc* encodes the procedure's parameters, and *outproc* decodes the procedure's results. The **clnt_perrno( )** routine is useful for translating **clnt_stat** return values into messages. This routine returns zero if it succeeds or the value of **enum clnt_stat** cast to an integer if it fails.

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

Note	Calling remote procedures with this routine uses UDP/IP as a transport. See **clntudp_create()** for restrictions.

Routine	`clnt_broadcast()`

Description	Works like `callrpc()` except the call message is broadcast to all locally connected broadcast networks.
	
	Each time this routine receives a response, it calls `eachresult( )`, whose form is as follows.
	
	`bool_t`
	`eachresult(out, addr)`
	`  char *out;`
	`  struct sockaddr_in *addr;`
	
	The parameter *out* is the same as `out` passed to `clnt_broadcast( )` except the remote procedure's output is decoded in `eachresult( )`.
	
	The parameter *addr* points to the host address that sent the results.
	
	If `eachresult( )` returns `FALSE`, `clnt_broadbast( )` waits for more replies. Otherwise, it returns the appropriate status.

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

Routine	`clnt_call()`
Description	A macro that calls the remote procedure **procnum** associated with the client handle **clnt**. The **clnt** handle is obtained with an RPC client creation routine such as **clntudp_create()**. The parameter *in* is the address of the procedure's arguments, and *out* is the address of where to place the results. The parameter *inproc* encodes the procedure's parameters, and *outproc* decodes the procedure's results. The parameter *tout* is the total time allowed for results to return.

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

Routine	`clnt_control( )`
Description	A macro that changes or retrieves information about an RPC client. The **req** parameter determines the type of operation and **info** is a pointer to the information. The information will be contained in various types of struct's depending on the value in req.

req	info	action
CLGET_TIMEOUT	struct timeval	Returns the value for the amount of time the client will wait on the server before returning a timeout error
CLSET_TIMEOUT	struct timeval	Sets the value for the amount of time the client will wait on the server before returning a timeout error
CLGET_SERVER_ADDR	struct sockaddr	Returns the address of the server
Note	`CLGET_TIMEOUT`, `CLSET_TIMEOUT`, and `CLGET_SERVER_ADDR` are valid ONLY for UDP based RPC.	
CLGET_RETRY_TIMEOUT	struct timeval	Returns the value for the amount of time the client will wait before resending a request
CLSET_RETRY_TIMEOUT	struct timeval	Sets the value for the amount of time the client will wait before resending a request

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

Note	If `CLSET_TIMEOUT` is used to set the timeout value, then the values that are sent in future calls to `clnt_call()` are ignored because the value set with `clnt_control` has overriding precedence.

Routine	`clnt_create()`
Description	A routine that will create an RPC client handle. *host* identifies the name of the remote host where the server is located. *prog* and *vers* are the program number and the version number of the server program. *proto* indicates which kind of transport protocol to use to link the server and client. Currently `udp` and `tcp` are the supported values for this parameter. Default timeout values are set, but can be modified using `clnt_control`.

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

Note	A UDP-based RPC message can hold up to 8K bytes of encoded data.

Routine	`clnt_destroy()`
Description	A macro that destroys the client's RPC handle. Destruction usually involves deallocation of private data structures, including `clnt`. You have the responsibility of closing sockets associated with `clnt`, and must do so before calling `clnt_destroy( )`. Use of `clnt` is undefined after calling `clnt_destroy( )`.

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

Routine	`clnt_freeres()`
Description	A macro that frees any data allocated by the RPC/XDR system when it decoded the results of an RPC call on `clnt`. The parameter *out* is the address of the results, and *outproc* is the XDR routine describing the results in simple primitives. This routine returns `TRUE` if the results were successfully freed or a `FALSE` if they were not.

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

Routine	`clnt_geterr()`
Description	A macro that copies the error structure out of the client handle to the structure at address *errp*.

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

Routine	`clnt_pcreateerror()`
Description	Prints a message to standard error indicating why a client RPC handle could not be created; prints the string *s* and a colon (:) before the message. Use `clnt_pcreateerror()` after a `clntraw_create()`

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

Routine	`clnt_perrno()`
Description	Prints a message to standard error corresponding to the condition indicated by *stat*. Use `clnt_perrno()` after `callrpc()`.

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

Routine	`clnt_perror()`
Description	Prints a message to standard error indicating why an RPC call failed; prints the string *s* and a colon (:) before the message. Use `clnt_perror()` after `clnt_call()`.

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

Routine	`clnt_spcreateerror()`
Description	Returns a string that contains a message telling why a client RPC handle could not be created. The message in the returned string will be preceded with the string *s* and a colon(:). The string will contain the same text as is printed when `clnt_pcreateerror()` is called.

Synopsis	`char *`
	`clnt_spcreateerror(s)`
	`  char *s;`

Note	`clnt_spcreateerror()` returns a pointer to static data so the contents of the string are overwritten on each call to the function.

Routine	`clnt_sperrno()`
Description	Returns a string that contains a message corresponding to the condition indicated by *stat*. The string will contain the same text as is printed when `clnt_perrno()` is called.

Synopsis	`char *`
	`clnt_sperror (stat)`
	`  enum clnt_stat stat;`

Routine	`clnt_sperror()`
Description	Returns a string that contains a message telling why an RPC call failed. The message in the returned string will be preceded with the string *s* and a colon(:). The string will contain the same text as is printed when `clnt_perror()` is called.

Synopsis	`char *`
	`clnt_sperror (s)`
	`char *s;`

Note	`clnt_sperror` returns a pointer to static data so the contents of the string are overwritten on each call to the function.

Routine	`clntraw_create()`
Description	This routine creates a simulated RPC client for the remote program *prognum*, version *versnum*. The transport used to pass messages to the service is actually a buffer within the process address space, so the corresponding RPC server must be in the same address space. (See `svcraw_create( )`). This pair of routines allow simulation of RPC and acquisition of RPC overheads (e.g., round trip times) without kernel interference. This routine returns `NULL` if it fails.

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

Routine	`clnttcp_create()`
Description	This routine creates an RPC client for the remote program *prognum,* version *versnum.* The client uses TCP/IP as a transport. The remote program is located at Internet address **addr.* If `addr->sin_port` is zero, it is set to the actual port on which the remote program is listening. (The `clnttcp_create()` function consults the remote `portmap` service for this information.) The parameter **sockp* is a socket file descriptor; if it is `RPC_ANYSOC`, then this routine opens a new one and sets **sockp.* Since TCP-based RPC uses buffered I/O, you can specify the size of the send and receive buffers with the parameters *sendsz* and *recvsz;* using values of zero causes `clnttcp_create( )` to choose reasonable defaults. This routine returns `NULL` if it fails.

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

Routine	`clntudp_create()`
Description	This routine creates an RPC client for the remote program *prognum,* version *versnum;* the client uses UDP/IP as a transport. The remote program is located at Internet address **addr.* If `addr->sin_port` is zero, then it is sent to the port on which the remote program is listening. (The `clntudp_create ( )` function consults the remote `portmap` service for this information.) The parameter **sockp* is a socket file descriptor; if it is `RPC_ANYSOCK`, this routine opens a new socket and sets **sockp.* The UDP transport resends the call message in intervals of *timeval wait* until a response is received or until the call times out. Use `clnt_call()` to specify the total timeout for the call. This routine returns `NULL` if it fails.

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

Note	UDP-based RPC messages can only hold up to 8K bytes of encoded data.

Routine	`get_myaddress()`
Description	Places the node's IP address into **addr* without consulting the library routines dealing with `/etc/hosts`. The port number is always set to *htons* (`PMAPPORT`).

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

Note	Use this routine to avoid using the NIS service.

Routine	`gettransient()`
Description	This function chooses a valid program number in the transient range (0x40000000 - 0x5fffffff) and registers it with the portmapper using the requested protocol *proto* and version *vers*. The value of *proto* is either `IPPROTO_TCP` or `IPPROTO_UDP`. If **sockp* is `RPC_ANYSOCK`, then `gettransient()` obtains a new socket and sets **sockp* to it. This routine returns the program number it registered or zero if it fails.

Synopsis	`u_long`
	`gettransient (proto, vers, sockp)`
	`  int proto;`
	`  u_long vers;`
	`  int *sockp;`

Routine	`pmap_getmaps()`
Description	A user interface to the **portmap** service; returns a list of the current RPC program-to-port mappings on the host located at IP address **addr.* The command **rpcinfo -p** uses this routine. This routine returns **NULL** if no mappings exist.

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

Routine	`pmap_getport()`
Description	A user interface to the **portmap** service; returns the port number associated with a service that supports program number *prognum* and version *versnum,* and speaks the transport protocol associated with *protocol.* A return value of zero means the mapping does not exist or the RPC system failed to contact the remote **portmap** service. In the latter case, the global variable *rpc_createerr* contains the RPC status.

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

Routine	`pmap_rmtcall()`
Description	A user interface to the **portmap** service; instructs *portmap* on the host at IP address **addr* to make an RPC call on your behalf to a procedure on that host. The parameter **portp* is modified to the program's port number if the procedure succeeds. Calls the remote procedure associated with *prognum*, *versnum*, and *procnum* on the host node. The parameter *in* is the address of the procedure's argument(s), and *out* is the address of where to place the results. The parameter *inproc* encodes the procedure's parameters, and *outproc* decodes the procedure's results. The parameter *tout* is the time allowed for results to return. Use this procedure for an "**are you there**" query and nothing else. (See `clnt_broadcast( )`.)

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

Routine	`pmap_set()`
Description	A user interface to the **portmap** service; establishes a mapping between the triple *[prognum,versnum,protocol]* and *port* on the node's **portmap** service. The value of protocol is either `IPPROTO_UDP` or `IPPROTO_TCP`. The `svc_register()` function automatically calls the `pmap_set()` function. This routine returns **TRUE** if it succeeds or **FALSE** if it does not.

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

Routine	`pmap_unset()`
Description	A user interface to the **portmap** service; destroys all mappings between the triple *[prognum,versnum,*]* and ports on the node's *portmap* service. This routine returns **TRUE** if it succeeds or **FALSE** if it does not.

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

Routine	`registerrpc()`
Description	Registers procedure *procname* with the RPC service package. If a request arrives for program *prognum*, version *versnum*, and procedure *procnum*, **procname** is called with a pointer to its parameter(s). The parameter *procname* should return a pointer to its static result(s). The parameter *inproc* decodes the parameters while *outproc* encodes the results. This routine returns a 0 (zero) if the registration succeeds or -1 if it does not.

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

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

Variable	`rpc_createerr`
Description	A global variable whose value is set by any RPC client creation routine that does not succeed. Use the `clnt_pcreateerror()` routine to print the reason why the creation routine did not succeed.
Synopsis	`struct rpc_createerr rpc_createerr;`

Routine	`svc_destroy()`
Description	A macro that destroys the RPC service transport handle *xprt*. Destruction usually involves deallocation of private data structures; including `xprt`. Use of `xprt` is undefined after calling this routine.

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

Routine	`svc_fds`
Description	A global variable reflecting the RPC service side's read file descriptor bit mask. This variable is of interest only if you do not call `svc_run()`, but rather implement asychronous event processing. This variable is read-only, yet it may change after calls to `svc_getreq()` or any creation routines.
Synopsis	`int svc_fds;`
Note	Do not use `svc_fds` by itself as an argument to `select()` since `select()` modifies its arguments. (Doing so will remove the RPC service side file descriptor mask.) You should copy the `svc_fds` value to a temporary variable for use.

Routine	`svc_fdset`
Description	A global variable reflecting the RPC service side's read file descriptor bit mask. This variable is of interest only if you do not call **`svc_run()`**, but rather implement asychronous event processing. This variable is read-only, yet it may change after calls to **`svc_getreqset()`**. It can handle up to **`FD_SETSIZE`** (as defined in **`/usr/include/sys/types.h`**) number of descriptors.
Synopsis	`fd_set svc_fdset;`
Note	Do not use **`svc_fdset`** by itself as an argument to **`select()`** since **`select()`** modifies its arguments (doing so will remove the RPC service side file descriptor mask). You should copy the **`svc_fdset`** value to a temporary data structure for use.

Routine	`svc_freeargs()`
Description	A macro that frees any data allocated by the RPC/XDR system when it decoded the arguments to a service procedure using **`svc_getargs()`**. This routine returns **`TRUE`** if the results were successfully freed or **`FALSE`** if they were not.

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

Routine	`svc_getargs()`
Description	A macro that decodes the arguments of an RPC request associated with the RPC service transport handle **xprt**. The parameter *in* is the address where the arguments will be placed. The parameter *inproc* is the XDR routine used to decode the arguments. This routine returns **TRUE** if decoding succeeds or **FALSE** if it does not.

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

Routine	`svc_getcaller()`
Description	The approved way in which the server with the RPC service transport handle **xprt** obtains the network address of the caller. This routine returns **NULL** if it fails.

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

Routine	`svc_getreq()`
Description	This routine is of interest only if you do not call **svc_run**, but rather implement custom asynchronous event processing. Use **svc_getreq()** when the **select()** system call determines that an RPC request arrived on an RPC socket. The parameter *rdfds* is the read file descriptor bit mask as modified by the **select()** call. The routine returns after all sockets associated with the value of *rdfds* are serviced.

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

Routine	`svc_getreqset()`
Description	This routine is of interest only if you do not call **svc_run**, but rather implement custom asynchronous event processing. Use **svc_getreqset()** when the **select()** system call determines that an RPC request arrived on an RPC socket. The parameter *rdfds* is the read file descriptor bit mask as modified by the **select()** call. The routine returns after all sockets associated with the value of *rdfds* are serviced. It can handle up to **FD_SETSIZE** (as defined in **/usr/include/sys/types.h**) number of descriptors.

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

Routine	`svc_register()`

Description	Associates *prognum* and *versnum* with the service dispatch procedure `dispatch()`.
	
	If *protocol* is zero, the service is not registered with the `portmap` service.
	
	If *protocol* is non-zero, a mapping of the triple *[prognum,versnum,protocol]* to `xprt->xp_port` is established with the local `portmap` service (generally *protocol* is zero, `IPPROTO_UDP`, or `IPPROTO_TCP`).
	
	The procedure `dispatch()` has the following form.
	
	`dispatch(request, xprt)`
	`  struct svc_req *request;`
	`  SVCXPRT *xprt;`
	
	The `svc_register()` routine returns `TRUE` if it succeeds or `FALSE` if it does not.
	
	The procedure `dispatch()` has the following form.

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

Routine	`svc_run()`
Description	This routine never returns. It waits for RPC requests to arrive and calls the appropriate service procedure using `svc_getreqset()` when one arrives. This procedure is usually waiting for a `select()` system call to return.
Synopsis	`void` `svc_run()`

Routine	`svc_sendreply()`
Description	Called by an RPC service's dispatch routine to send the results of a remote procedure call. The parameter *xprt* is the caller's associated transport handle. The parameter *outproc* is the XDR routine used to encode the results. The parameter *out* is the address of the results. This routine returns `TRUE` if it succeeds or `FALSE` if it does not.

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

Routine	`svc_unregister()`
Description	Removes all mappings of the double [*prognum,versnum*] to dispatch routines and of the triple [*prognum,versnum,**] to port number.

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

Routine	`svcerr_auth()`
Description	Called by a service dispatch routine that refuses to perform a remote procedure call because of an authentication error. See `<rpc/auth.h>` for valid `auth_stat` values.

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

Routine	`svcerr_decode()`
Description	Called by a service dispatch routine that cannot successfully decode its parameters. (See `svc_getargs()`.)

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

Routine	`svcerr_noproc()`
Description	Called by a service dispatch routine that does not implement the desired procedure number the caller requested.

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

Routine	`svcerr_noprog()`
Description	Called when the desired program is not registered with the RPC package.

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

Routine	`svcerr_progvers()`
Description	Called when the desired version of a program is not registered with the RPC package.

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

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

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

Routine	`svcerr_weakauth()`
Description	Called by a service dispatch routine that refuses to perform a remote procedure call because of insufficient, but possibly correct, authentication parameters.

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

Routine	`svcfd_create()`
Description	This routine creates a TCP/IP-based RPC service transport from an existing socket to which it returns a pointer. Use this routine when you receive a socket from the **inetd**. The *sock* parameter must be a valid file descriptor for an active socket (i.e., you already executed the **listen( )** and **accept( )** calls to obtain this socket). Since TCP-based RPC uses buffered I/O, you can specify the size of the **send()** and **recv()** buffers. Using values of zero causes **svcfd_create()** to choose reasonable defaults. Upon completion, the **xp_sock** field contains the transport's socket number and the **xp_port** field contains the transport's port number. See **clnttcp_create()**. This routine returns **NULL** if it fails.

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

Routine	`svcraw_create()`
Description	This routine creates a simulated RPC service transport to which it returns a pointer. The transport is a buffer within the process' address space, so the corresponding RPC client must exist in the same address space. (See **clntraw_create()**.) This routine allows simulation of RPC and acquisition of RPC overheads (e.g., round trip times) without kernel interference. This routine returns **NULL** if it fails.
Synopsis	`SVCXPRT *` `svcraw_create()`

Routine	`svctcp_create()`
Description	This routine creates a TCP/IP-based RPC service transport to which it returns a pointer. The transport is associated with the socket file descriptor *sock;* if the *sock* is **RPC_ANYSOCK**, a new socket is created. If the socket is not bound to a local TCP port, this routine binds it to an arbitrary port. Since TCP-based RPC uses buffered I/O, you can specify the size of the **send()** and **recv()** buffers; using values of zero causes **svctcp_create()** to choose reasonable defaults. Upon completion, the **xp_sock** field contains the transport's socket number and the **xp_port** field contains the transport's port number. See **clnttcp_create()**. This routine returns **NULL** if it fails.
Synopsis	**SVCXPRT *** **svctcp_create(sock, send_buf_size, recv_buf_size)** **int sock;** **u_int send_buf_size, recv_buf_size;**

Routine	`svcudp_create()`
Description	This routine creates a UDP/IP-based RPC service transport to which it returns a pointer. The transport is associated with the socket file descriptor *sock;* if *sock* is **RPC_ANYSOCK**, a new socket is created. If the socket is not bound to a local UDP port, this routine binds it to an arbitrary port. Upon completion, the **xp_sock** field contains the transport's socket number and the **xp_port** field contains the transport's port number. This routine returns **NULL** if it fails.

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

Note	UDP-based RPC messages only hold up to 8K bytes of encoded data.

Routine	`xdr_accepted_reply()`
Description	This routine is useful if you wish to generate RPC-style messages without using the RPC package. The **accepted_reply** structure is defined in **<rpc/rpc_msg.h>**. This routine returns **TRUE** if it succeeds or **FALSE** if it does not.

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

Routine	`xdr_authunix_parms()`
Description	This routine is useful if you wish to generate these credentials without using the RPC authentication package. The `authunix_parms` structure is defined in `<rpc/auth_unix.h>`

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

Routine	`xdr_callhdr()`
Description	This routine is useful if you wish to generate RPC-style messages without using the RPC package. The `rpc_msg` structure is defined in `<rpc/rpc_msg.h>`.

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

Routine	xdr_callmsg()
Description	This routine is useful if you wish to generate RPC-style messages without using the RPC package. The rpc_msg structure is defined in <rpc/rpc_msg.h>.
Synopsis	bool_t xdr_callmsg(xdrs, cmsg) XDR *xdrs; struct rpc_msg *cmsg;

Routine	xdr_opaque_auth()
Description	This routine is useful if you wish to generate RPC-style messages without using the RPC package. The opaque_auth() structure is defined in <rpc/auth.h>.
Synopsis	bool_t xdr_opaque_auth(xdrs, ap) XDR *xdrs; struct opaque_auth *ap;

Routine	xdr_pmap()
Description	This routine is useful if you wish to use XDR to encode or decode portmap structures without using the pmap interface. The pmap structure is defined in <rpc/pmap_prot.h>.
Synopsis	bool_t xdr_pmap(xdrs, regs) XDR *xdrs; struct pmap *regs;

Routine	`xdr_pmaplist()`
Description	This routine is useful if you wish to use XDR to encode or decode portmap structures without using the `pmap` interface. The `pmaplist` structure is defined in `<rpc/pmap_prot.h>`.

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

Routine	`xdr_rejected_reply()`
Description	This routine is useful if you wish to generate RPC-style messages without using the RPC package. The `rejected_reply` structure is defined in `<rpc/rpc_msg.h>`.

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

Routine	`xdr_replymsg()`
Description	This routine is useful if you wish to generate RPC-style messages without using the RPC package. The `rpc_msg` structure is defined in `<rpc/rpc_msg.h>`.

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

Routine	`xprt_register()`
Description	After RPC service transport handles are created, they should register with the RPC service package. This routine modifies the global variable **svc_fds**.

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

Routine	`xprt_unregister()`
Description	Before an RPC service transport handle is destroyed, it should unregister with the RPC service package. This routine modifies the global variable *svc_fds*.

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

4

RPCGEN Programming Guide

This chapter explains the use of the Remote Procedure Call Protocol Compiler (RPCGEN) to convert applications that run on a single computer to ones that run over a network.

The information in this chapter is based on the assumptions that you are familiar with HP-UX, the C programming language, Remote Procedure Calls (RPC), and networking. If you need a review of RPC programming without RPCGEN, see the "RPC Programming Guide" chapter.

Writing applications to use Remote Procedure Calls can be time-consuming and difficult. Perhaps the most difficult part is writing XDR routines necessary to convert arguments and results into their network form and vice versa. RPCGEN helps you write RPC applications simply and directly. It allows you to concentrate on debugging the main features of your application instead of the network interface code.

The Remote Procedure Call Protocol Compiler

RPCGEN is a compiler. It accepts remote program interface definitions written in RPC (Remote Procedure Call) language, which is similar to C. It produces C language output including the following:

- A header file.
- A client side subroutine file (client stub).
- A server side skeleton file (server side stub).
- An XDR routines file.

The client side subroutine file and the server side skeleton file are called "stubs." The client stubs interface with the RPC library, and effectively shield the user from the network. The server stub similarly shields the server procedures, invoked by remote clients, from the network.

RPCGEN's output files can be compiled and linked in the normal way with your C compiler. You write server procedures and link them with the server skeleton produced by RPCGEN to produce an executable server program. To use a remote program, you write an ordinary main program that makes local procedure calls to the client stubs produced by RPCGEN. Linking this program with RPCGEN's stubs creates an executable program.

Converting Local Procedures into Remote Procedures

The following section illustrates the conversion of a simple example application program running on a single computer to a version that runs over the network.

The first file is the user's application. The program prints a message on the console.

EXAMPLE:

```
/*
 * printmsg.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)) {
            fprint(stderr, "%s: couldn't print your message\n",
                    argv[0]);
            exit(1);
     }
     printf("Message delivered!\n");
     exit(0);
}
/*
 * Print a message to the console.
 * Return a boolean indicating whether the message was actually
 * printed.
 */
printmessage(msg)
     char *msg;
{
     FILE *f;

     f = fopen("/dev/console", "w");
     if (f == NULL) {
            return (0);
     }
     fprint(f, "%s\n", msg);
     fclose(f);
     return(1);
}
```

When you compile and run this simple application, this message is printed on your console:

```
%    cc printmsg.c -o printmsg
%    printmsg "Hello, there."
Message delivered!
%
```

If you were to convert your **printmessage** application into a remote procedure, it could be called from anywhere on the network. To convert a procedure into a remote procedure, you must work within the constraints of the C language, since it existed before RPC did. But even without language support, it is not very difficult to make a procedure remote.

In general, you must determine the types for all procedure inputs and outputs. In this case, you have a procedure **printmessage** which takes a string as input and returns an integer as output.

1. Writing the RPC Protocol Specification.

The first step in converting a program to a remote procedure is to write a protocol description file in RPC language that describes the remote version of your application program (`printmessage` in this case). The code for the `msg.x` description file is as follows:

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

Remote procedures are part of remote programs, so you actually declared an entire remote program here which contains the single procedure `PRINTMESSAGE`. 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.

The program, version, and procedure are declared using all capital letters. This is not required, but is a good convention to follow.

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

2. Writing the Remote Procedure.

The second step is to write the remote procedure itself. Following is the definition of a remote procedure (`msg_proc.c`) to implement the `PRINTMESSAGE` procedure you declared above:

EXAMPLE:

```
/*
  *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 version of " printmessage"
 */
int *
printmessage_1(msg)
     char **msg;
{
     static int result;  /* must be static! */
     FILE *f;

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

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

- It takes a pointer to a string instead of a string itself. This is true of all remote procedures: they always take pointers to their arguments rather than the arguments themselves.
- It returns a pointer to an integer instead of an integer itself. This is true of remote procedures: they always return a pointer to their results.
- It has a `_1` appended to its name. In general, all remote procedures called by RPCGEN are named by the following rule: convert the name in the program definition (here `PRINTMESSAGE`) to all lowercase letters, and append an underbar (_) and the version number (in this case, number 1).

3. Creating the Main Client Program.

The third step is to create the main client program (`rprintmsg.c`) that will call the remote procedure.

EXAMPLE:

```
/*
 * rprintmsg.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) {
            fprint(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. You 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 quit.
```

```
                */
                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 quit.
                 */
                clnt_perror(cl, server);
                exit(1);
        }

        /*
         * Okay, you successfully called the remote procedure.
         */
        if (*result == 0) {
                /*
                 * Server was unable to print our message.
                 * Print error message and quit.
                 */
                fprint(stderr, "%s: %s couldn't print your message\n",
                        argv[0], server);
                exit(1);
        }

        /*
         * The message got printed on the server's console.
         */
        printf("Message delivered to %s!\n", server);
}
```

A client **handle** is created using the RPC library routine `clnt_create` (a **handle** is a data structure that is used to specify a certain client when the rpc routines are called). This client handle will be passed to the stub routines which call the remote procedure.

The remote procedure `printmessage_1` is called the same way as it is declared in `msg_proc.c` except for the inserted client handle as the second argument.

4. Compiling the Files.

The next step is to execute RPCGEN on the `msg.x` file and then compile and link the files to form the client and server programs that comprise the example remote message printing application. The following example shows what to enter:

```
%   rpcgen msg.x
%   cc rprintmsg.c msg_clnt.c -o rprintmsg
%   cc msg_proc.c msg_svc.c -o msg_server
```

From the protocol description file (the input file `msg.x`), RPCGEN creates the following files:

- A header file named`msg.h` containing #define's for `MESSAGEPROG`, `MESSAGEVERS`, and `PRINTMESSAGE` for use in the other modules.
- 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 `printmsg` 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 `TEST.x`, the client stubs output file is called `TEST_clnt.c`.
- The server side skeleton file `msg_svc.c`. This server program calls `printmessage_1` in `msg_proc.c`. The rule for naming the server output file is similar to the previous one: for an input file called `TEST.x`, the server side skeleton file is named `TEST_svc.c`.

In addition, two programs are produced by the compiler:

- The client program `rprintmsg`.
- The server program `msg_server`.

5. Testing the Results.

Now you are ready to test the results.

1. Copy the server program to a remote computer and run it. In this example, the computer is named `node1`. Server processes are run in the background because they never exit.

```
node1% msg_server &
```

2. On your local computer (`node2`), print a message on `node1`'s console.

```
node2% rprintmsg node1 "Hello node1".
```

The message will be printed on `node1`'s console. You can print a message on anyone's console (including your own) with this program if you are able to copy the server to their computer and run it.

Generating XDR Routines

The example in the previous section only demonstrated the automatic generation of client and server RPC code. RPCGEN may also be used to generate XDR routines—the routines necessary to convert local data structures into network format and vice-versa.

You must provide three of the files required to convert a single-system application to run on a network. Four of the files are produced by the RPCGEN compiler.

Files you must produce

- **Protocol description file** (suffixed with `.x`).
- **Client side file** (suffixed with `.c`).
- Server side function file (suffixed with `_proc.c`).

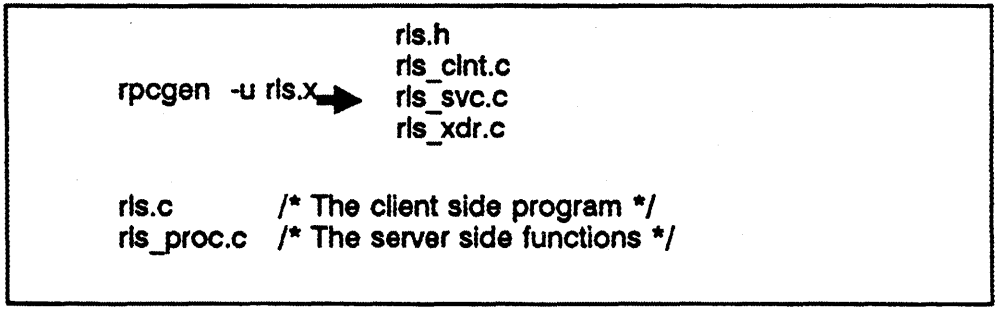
Files produced by RPCGEN

In addition to the file you create, RPCGEN produces four files from your `.x` file:

- **Header file** (suffixed with `.h`) containing the `const`'s, `typedef`'s, and `struct`'s used to communicate data structures among all of the portions of the application program.
- **Client side subroutine file** (suffixed with _clnt.c) which is a collection of the function stubs.
- **Server side skeleton file** (suffixed with _svc.c), the main C program for the server process.
- **XDR routine file** (suffixed with _xdr.c) used to translate the arguments and results between the client and server processes.

All of these files are prefixed with the main portion of the name of the `.x` file. For example, if you have a `.x` file named `remsh.x`, RPCGEN will produce the following files: `remsh.h`, `remsh_clnt.c`, `remsh_svc.c`, and `remsh_xdr.c`.

The following example files illustrate a complete RPC service—a remote directory listing service that uses RPCGEN not only to generate stub routines, but also to generate the XDR routines. The following illustration shows the files produced by RPCGEN acting on your `rls.x` file and the additional files that you must create.



Relationship of programmer supplied files to files created by RPCGEN

The Protocol Description File (The Input File)

The first file, produced by you, is the protocol description file (the input file). It is written in a C-like language and is stored in a file suffixed with `.x`. This file describes the necessary data structure involved in producing a remote directory listing.

EXAMPLE:

```
/*
 * rls.x: Remote directory listing protocol
 */
const MAXMANELEN = 255; /* maximum length of a directory entry */

     /* This definition is specific to RPCGEN. It is */
     /* different from C syntax. It defines a variable */
     /* length string.   */
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
 */

/* This definition is specific to RPCGEN. It is */
/* different from C syntax. It defines what a remote */
/* program consists of. */
program RLSPROG {
     version RLSVERS {
             readdir_res
             READDIR(nametype) = 1;
     } = 1;
} = 76;
```

The Header File

The next file is the header file (`rls.h` in this example). It is created by RPCGEN. This file ties all of the other files together. `rls.h` is a C language version of the `rls.x` file.

EXAMPLE:

```
#define MAXNAMELEN 255

typedef char *nametype;
bool_t xdr_nametype();

typedef struct namenode *namelist;
bool_t xdr_namelist();

struct namenode {
     nametype name;
     namelist next;
};
typedef struct namenode namenode;
bool_t xdr_namenode();
struct readdir_res {
     int errno;
     union {
             namelist list;
     } readdir_res_u;
};
typedef struct readdir_res readdir_res;
bool_t xdr_readdir_res();

#define RLSPROG ((u_long)76)
#define RLSVERS ((u_long)1)
#define READDIR ((u_long)1)
extern readdir_res *readdir_1();
```

The Client Side File

The client side file (`rls.c` in this example) is produced by you. It includes code to do the following:

- Create the user interface.
- Make the connection to the server computer.
- Make the call to the server and read a directory on the server.
- Decode and print the results.

EXAMPLE:

```
/*
 * rls.c Remote directory listing client
 */
#include <stdio.h>
#include <rpc/rpc.h>        /* always need this */
#include "rls.h"      /* need this too: generated by rpcgen */

extern int errno;

main(argc, argv)
     int argc;
     char *argv[ ];
{
     CLIENT *cl, *clnt_create();
     char *server;
     char *dir;
     readdir_res *result;
     namelist nl;

     if (argc !=3) {
             fprint(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. You tell the rpc
```

```
         * package to use the "tcp" protocol when contacting the server.
         */
        cl = clnt_create(server, RLSPROG, RLSVERS, "tcp");
        if (cl == NULL) {
                /*
                 * Couldn't establish connection with server.
                 */
                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, You 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 out.
         */
        for (nl = result->readdir_res_u.list; nl != NULL; nl = nl->next) {
                printf("%s\n", nl->name);
        }
}
```

The Client Side Subroutines File

The next file (`rls_clnt.c` in this example) is created by RPCGEN. The `rls_clnt.c` file contains the client side stubs that are called by `rls.c` to transmit the arguments and receive the results. The `rls_clnt.c` file defines only one routine, `readdir_1()`. This is because the program definition in the `rls.x` file contained only one procedure.

EXAMPLE:

```
#include <rpc/rpc.h>
#include <sys/time.h>
#include "rls.h"

#ifdef hpux

#ifndef      NULL
#define      NULL    0
#endif       NULL

#endif hpux

static struct timeval TIMEOUT = { 25, 0};

readdir_res *
readdir_1(argp, clnt)
     nametype *argp;
     CLIENT *clnt;
{
     static readdir_res res;

#ifdef hpux
     memset(&res, 0, sizeof(res));
#else hpux
     memset(&res, sizeof(res));
#endif hpux
     if (clnt_call(clnt, READDIR, xdr_nametype, argp,
             xdr_readdir_res, &res, TIMEOUT) !=RPC_SUCCESS ) {
                     return (NULL);
     }
     return (&res);
}
```

The Server Side Skeleton File

The next file (`rls_svc.c` in this example), created by RPCGEN, contains the main program for the server side. It registers the `rlsprog_1()` routine with the server computer and then waits for an incoming request by calling `svc_run()`. Note that by default, RPCGEN provides code to handle both TCP and UDP protocols. You can specify which protocol the server code will use by invoking the `-s` option when you execute RPCGEN. When `svc_run` receives a request, it calls `rlsprog_1()` which connects to the function supplied by you in the `rls_proc.c` file which does the actual work. The result of the call is then transmitted back to the requestor. The signal handling code is added when the "-u" option is used with RPCGEN.

EXAMPLE:

```
#include <stdio.h>
#include <rpc/rpc.h>
#include "rls.h"

void un_register_prog(signo)
int signo;
{
      pmap_unset(RLSPROG,RLSVERS);
      exit(1);
}

static void rlsprog_1();

main()
{
      SVCXPRT * transp;

       pmap_unset(RLSPROG, RLSVERS);

       (void)       signal(SIGHUP, un_register_prog);
       (void)       signal(SIGINT, un_register_prog);
       (void)       signal(SIGQUIT, un_register_prog):
       (void)       signal(SIGTERM, un_register_prog);

       transp = svcudp_create(RPC_ANYSOCK);
       if (transp == NULL) {
             fprintf(stderr, "cannot create udp service.\n");
             exit(1);
```

```
        }
        if (!svc_register(transp, RLSPROG, RLSVERS, rlsprog_1, IPPROTO_UDP)) {
                fprint(stderr,
                        "unable to register (RLSPROG, RLSVERS, udp).\n");
                exit(1);
        }
        transp = svctcp_create(RPC_ANYSOCK, 0, 0);
        if (transp == NULL) {
                fprint(stderr, "cannot create tcp service.\n");
                exit(1);
        }
        if (!svc_register(transp, RLSPROG, RLSVERS, rlsprog_1, IPPROTO_TCP)) {
                fprintf(stderr,
                          "unable to register (RLSPROG, RLSVERS, tcp).\n");
                exit(1);
         }
         svc_run();
         fprintf(stderr, "svc_run returned\n");
         exit(1);
}
static void
rlsprog_1(rqstp, transp)
        struct svc_req *rqstp;
        SVCXPRT *transp;
{
        union {
             nametype readdir_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 READDIR:
             xdr_argument = xdr_nametype;
             xdr_result = xdr_readdir_res;
             local = (char *(*) ()) readdir_1;
             return;

        default:
             svcerr_noproc(transp) ;
```

```
            return;
        }
#ifdef hpux
        memset(&argument, 0, sizeof(argument));
#else hpux
        memset(&argument, sizeof(argument));
#endif hpux
        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);
        }
}
```

The Server Side Function File

This file (`rls_proc.c` in this example) is written by you. It contains the code to produce the actual server portion of the application. In the following example, the code opens a directory, reads it and places the results in the result structure (`struct`) that was defined by the `rls.x` file.

EXAMPLE:

```
/*
 * rls_proc.c: remote readdir implementation
 */
#include <rpc/rpc.h>
#include <sys/dir.h>
#include <stdio.h>
#include "rls.h"

extern int errno;
extern char *malloc();
extern char *strcpy();

readdir_res*
```

```
readdir_1(dirname)
     nametype *dirname;
{
     DIR *dirp;
     stuct direct *d;
     namelist nl;
     namelist *nlp;
     static readdir_res res;     /* must be static! */

     /*
      * Free previous result
      */
     xdr_free(xdr_readdir_res, &res);

     /*
      * Open directory
      */
     dirp = opendir(*dirname);
     if (dirp == NULL) {
             res.errno = errno;
             return (&res);
     }

     /*
      * Collect directory entries
      */
     nlp = &res.readdir_res_u.list;
     while (d = readdir(dirp)) {
             nl=*nlp  =  (namenode  *)  malloc(sizeof(namenode));
             nl->name  =  malloc(strlen(d->d_name)+1);
             strcpy(nl->name, d->d_name);
             nlp = &nl->next;
     }
     *nlp = NULL;

     /*
      * Return the result
      */
     res.errno = 0;
     closedir(dirp);
     return (&res);
}
```

XDR Routine File

The `rls_xdr.c` file is created from the `rls.x` file by RPCGEN. This file manages the details of the XDR translation of requests and results. This file uses the definitions of the data structures in the `.x` file to produce functions which do the proper XDR translations. If there are data types in the `.x` file that you have not defined, the XDR routines for those data types will not be found in the `rls_xdr.c` file. RPCGEN will not object to having undefined data types. You must produce the translation functions for these data types.

EXAMPLE:

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

bool_t
xdr_nametype(xdrs, objp)
     XDR *xdrs;
     nametype *objp;
{
     if (!xdr_string(xdrs, objp, MAXNAMELEN)) {
             return (FALSE);
     }
     return (TRUE);
}

bool_t
xdr_namelist(xdrs, objp)
     XDR *xdrs;
     namelist *objp;
{
     if (!xdr_pointer(xdrs, (char **)objp, sizeof(struct namenode),
                     xdr_namenode)) {
             return (FALSE);
     }
     return (TRUE);
}

bool_t
xdr_namenode(xdrs, objp)
     XDR *xdrs;
     namenode *objp;
{
     if (!xdr_nametype(xdrs, &objp->name)) {
```

```
                return (FALSE);
        }
        if (!xdr_namelist(xdrs, &objp->next)) {
                return (FALSE);
        }
        return (TRUE);
}

bool_t
xdr_readdir_res(xdrs, objp)
        XDR *xdrs;
        readdir_res *objp;
{
        if (!xdr_int(xdrs, &objp->errno)) {
                return (FALSE);
        }
        switch (objp->errno) {
        case 0:
                if (!xdr_namelist(xdrs, &objp->readdir_res_u.list)) {
                        return(FALSE);
                }
                break;
        }
        return (TRUE);
}
```

Compiling the Files

The last step is to compile and link all of the files. The following example shows what to enter to compile and link everything, forming the client and server programs that comprise the example remote directory read application:

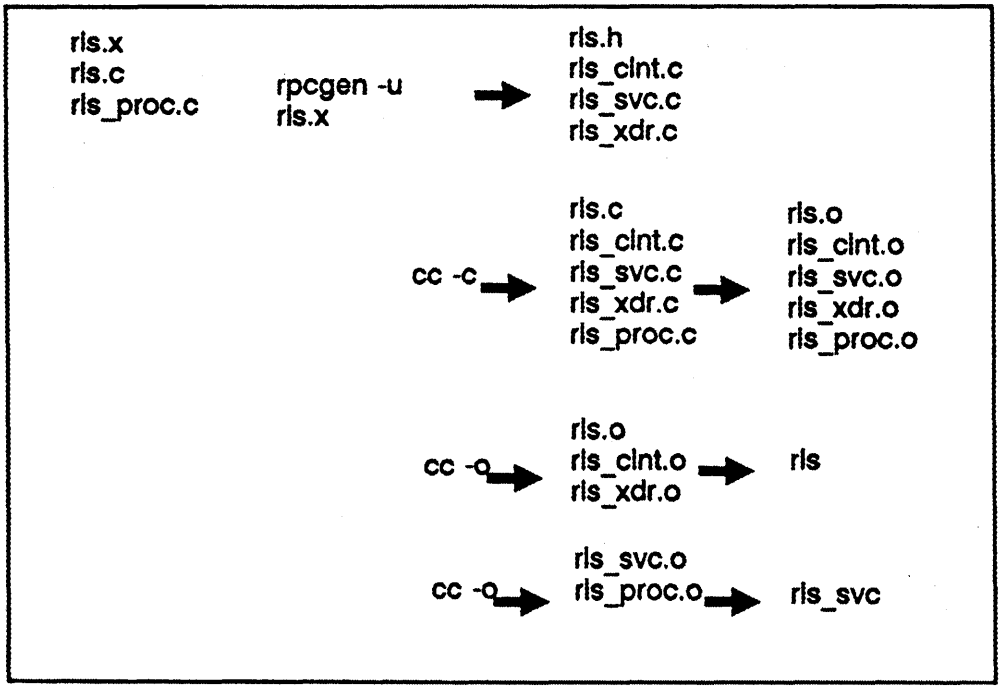
EXAMPLE:

```
node1%    rpcgen-urls.x
node1%    cc-crls_proc.c
node1%    cc-crls_svc.c
node1%    cc-crls_xdr.c
node1%    cc-crls.c
node1%    cc-crls_clnt.c
node1%    cc-orls_svcrls_proc.orls_svc.orls_xdr.o
node1%    cc-orlsrls.orls_clnt.orls_xdr.o
```

You can test the client program and the server procedure together as a single program by linking them with each other rather than with the client and server stubs. The procedure calls will be executed as ordinary local procedure calls and you can debug the program with a local debugger such as **xdb**. When the program is working, you can link the client program to the client stub produced by RPCGEN, and you can link the server procedures to the server stub produced by RPCGEN.

Note If you do this, you should comment out calls to the RPC library routines and have client routines call server routines directly.

The following illustration shows the entire RPCGEN process.



The RPCGEN process

RPCGEN Syntax

The syntax of the RPCGEN compiler is as follows:

```
rpcgen [-u] infile
rpcgen -c [-o outfile] [ infile]
rpcgen -h [-o outfile] [ infile]
rpcgen -l [-o outfile] [ infile]
rpcgen -m [-o outfile] [ infile]
rpcgen -s transport [-u] [-o outfile] [ infile]
```

Options.

-c	Compile into XDR routines.
-h	Compile into C data-definitions (a header file).
-l	Compile into client-side stubs.

`-s` *transport* Compile into server-side stubs, using the given transport. Supported transports are UDP and TCP. This option may be invoked more than once to compile a server using multiple transports.

Note If RPCGEN is called without the `-s` option, the server-side code that is generated will serve both UDP and TCP transports.

`-m` Compile into server-side stubs, but do not produce a `main()` routine. This option is useful if you want to supply your own `main()`.

`-u` Insert code into the server side `.c` stub file which traps signals sent to the server program.

This signal code will cause the RPC server program to unmap itself from the portmapper on the server computer. If this is not done, when the server receives a signal, it will stop execution and leave the portmapper thinking that it has that server program ready for incoming requests. This can cause a misleading error to be given on the client.

The signals `SIGHUP`, `SIGINT`, `SIGQUIT`, and `SIGTERM` are trapped by the signal handler. They are signals often sent to a program to cause it to terminate execution. The signal `SIGKILL` is not caught because it is not possible to trap it. The other available signals are not trapped because they are not associated with the concept of terminating a process.

Note The `-u` option can only be used when a server-side stub that contains a `main()` program is produced. It can be used with no other options given or with the `-s` option. It cannot be used when the `-h`, `-c`, `-l`, or `-m` options are present.

`-o` *outfile* Specify the name of the output file. If none is specified, standard output is used. This is usable only with the `-h`, `-c`, `-l`, or `-m` options.

Caution Nesting is not supported. As a work-around, structures can be declared at the top-level and their names used inside other structures in order to achieve the same effect. Name clashes can occur when using program definitions, since the apparent scoping does not really apply. Most of these can be avoided by using unique names for programs, versions, procedures, and types.

The C Preprocessor

The C preprocessor is run on the input file before it is compiled, so all the preprocessor directives are legal within `.x` files. Four symbols may be defined, depending upon which output file is being generated. The symbols are:

Symbol	Usage
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 does some preprocessing. Any line that begins with a percent sign is passed directly into the output file, without any interpretation of the line.

EXAMPLE The following example demonstrates the RPCGEN preprocessing features.

```
/*
 *time.x: Remote time protocol
 */
program TIMEPROG {
     version TIMEVERS {
             unsigned int TIMEGET(void) = 1;
     } = 1;
} = 44;
#ifdef RPC_SVC
%int *       /* This will only be added to */
%timeget_1() /* the _svc.c file */
%{
%    static int thetime;
%
%    thetime = time(0);
%    return (&thetime):
%}
#endif
```

Note The '%' feature is not generally recommended as there is no guarantee that the compiler will place the output where you intended.

RPC Language

The RPC language is similar to C. If you know the C language, you will understand RPC. This section describes the RPC language syntax, showing a few examples along the way. This section also describes how the various RPC and XDR type definitions are 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
```

Specifically, the six types of definitions are as follows:

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

The first five definitions are used to define data representations and are known as XDR definitions. The last definition is the RPC program definition.

Structures

An XDR structure (struct) in the RPC language is declared virtually the same as its C counterpart.

EXAMPLE: Following is an example of an XDR structure:

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

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

EXAMPLE: The following example of an XDR structure defines a two-dimensional coordinate and the C structure into which it is compiled in the output header file.

XDR structure	C structure

`struct coord {`
`  int x;`
`  int y;`
`};`
	`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 one 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 they are to C unions.

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

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

EXAMPLE: Following is an example of a type that might be returned as the result of a "read data" operation. If there is no error, a block of data is returned; otherwise, nothing is returned:

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

After it is compiled, the union component of output structure has the same name as the name type (except for the trailing _u):

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

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
```

EXAMPLE: The following example illustrates an XDR enumeration and the C enumeration that results after being compiled:

XDR enumeration	C enumeration

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

Typedef

XDR typedefs have the same syntax as C typedefs.

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

EXAMPLE: The following example defines a 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
```

EXAMPLE: The following example 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
```

EXAMPLE: In the following example, we take another look at 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 #defines in the output header file:

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

Declarations

In XDR there are only four types of declarations:

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

Simple Declarations

Simple XDR declarations are the same as simple C declarations.

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

EXAMPLE:

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

Fixed-Length Array Declarations

XDR fixed-length array declarations are the same as C array declarations:

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

EXAMPLE:

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

Variable-Length Array Declarations

Variable-length declarations have no explicit syntax in C, so XDR invents its own using angle-brackets.

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

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

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

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

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

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

Pointer Declarations

Pointer declarations are made the same in XDR as they are in C. You cannot use pointers over the network, but you can use XDR pointers for sending recursive data types such as lists and trees.

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

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 includes a boolean type called `bool_t` that is either `TRUE` or `FALSE`. Things declared as type bool in XDR language are compiled into `bool_t` in the output header file.

EXAMPLE:

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

Strings

C has no built-in string type, but instead uses the null-terminated "char*" convention. In XDR language, strings are declared using the "string" keyword, and compiled into "char *"s 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 characters). The maximum size may be omitted, indicating a string of arbitrary length.

EXAMPLES:

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

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

EXAMPLES:

```
opaque diskblock[512];        --> char diskblock[512];
opaque filedata <1024>;       --> struct {
                                      u_int filedata_len;
                                      char *filedata_val;
                                  } filedata;
```

Voids. In a void declaration, the variable is not named. The declaration is just `void` and nothing else. Void declarations can only occur in two places:

- Union definitions.
- Program definitions (as the argument or result of a remote procedure).

RPCGEN Error Messages

Command Line Error Messages

```
usage: rpcgen [-u] infile
rpcgen [-c | -h | -l | -m | -u] [-o outfile] [infile]
rpcgen [-s udp | tcp]* [-o outfile] [infile]
```

Cause: This message is given if the wrong number of arguments, the wrong arguments, or the wrong options are given when executing RPCGEN.

RPCGEN Execution Error Messages

```
RPCGEN: output would overwrite <input_file>
```

Cause: If the name of the input file and the name specified for the output file are the same, RPCGEN will print this message and quit. The name of the input file will be substituted for *<input_file>* in the message.

```
rpcgen: unable to open < output_file>: < perror message>
```

Cause: If RPCGEN is unable to open the output file, the message listed above appears. Possible causes are many, such as not having write permission to the parent directory. This is why the `perror` message is printed. It gives a text message for the `errno` that resulted during the attempt to open the file. The name of the output file will be substituted for *<output_file>* in the message.

```
rpcgen: No more processes
```

Cause: RPCGEN will try to execute the C preprocessor. If it cannot do this, it will print a `perror()` message stating what the problem was. The text message is based on the value in `errno`.

```
rpcgen: RPCGEN has too many files open
```

Cause: If RPCGEN opens too many files at once, this error message appears. Since RPCGEN only has a few files open at any one time, the message would appear if RPCGEN is executed from a process that had almost the maximum number of files already open.

Parsing Error Messages

The next group of error messages is produced because of an error detected in the contents of the `.x` file. They are similar to having compilation errors in a C program and as such are very context dependent. The general rule of thumb is that either RPCGEN could not recognize any of the input it is given, or it was able to start parsing a legal construction, but ran into a symbol that did not match what it expected. Because some of the messages are long, some have been placed on two lines in order to fit within the margin. In reality, they will be printed on one line. In addition to an error message, the line that contains the error is printed with the part of the line that caused the problem underscored with "^^^^" characters.

```
<beginning of the line><error> <rest_of_the_line>
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

<input_file>, line <line_number>: <error message>
```

EXAMPLE:

If the following line appeared in a `.x` file:

```
const ducks "mallard"
```

This is what the error message would look like:

```
const ducks "mallard"
^^^^^^^^^^^^^^^^^^^^
err.x, line 5: expected '='
```

Expecting a Keyword

```
<input_file>, line <line_number>: definition key word expected.
```

Cause: RPCGEN was expecting the start of a legal construction such as a struct declaration and it encountered a token from the input file that did not match one of the legal keywords (`struct`, `union`, `typedef`, `enum`, `program`, or `const`).

Array of Pointers

```
< input_file>, line < line_number>:
no array-of-pointer declarations -- use typedef
```

Cause: You tried to declare an array of pointers.

EXAMPLE: This example shows how an array of pointers can be declared. If you wish to refer to an array of pointers, use **typedef** to do so (as in the GOOD line shown in the following example).

```
typedef struct z *zptr;
struct z {
        int a;
        zptr t[2];              /* GOOD LINE */
        struct z *y[2];         /* BAD LINE #1 */
        struct z *y<2>;        /* BAD LINE #2 */
};
```

Bad Union

When declaring a union, do not use an array in the switching variable (as shown in the following example).

EXAMPLE:

```
union xxx switch (int the_array[2]) { /* File bad_union.x */
case 0:
     int a;
default:
     void;
}
```

If you do, the following message will be displayed:

```
bad_union.x, line 1: only simple declaration allowed in switch
```

Opaque Declarations

```
<input_file>, line <line_number>: array declaration expected
```

Cause: Data object incorrectly declared.

If you want to declare a data object to be opaque, declare it as an array.

EXAMPLE: The following example shows a correct and incorrect method of using the opaque declaration:

```
opaque group_of_bytes[777]; /*CORRECT*/
opaque bad_declaration;     /*INCORRECT*/
```

String Declaration Error

```
< input_file>, line < line_number>:
variable-length array declaration expected
```

A string must be declared using left and right angle braces ("<" and ">").

EXAMPLE: The following example shows a correct and incorrect method of using the string declaration:

```
rrectan
string first_name<50>;      /*CORRECT*/
string last_name 50;        /*INCORRECT*/
```

Void Declarations

```
< input_file>, line < line_number>:
voids allowed only inside union and program definitions
```

Cause: A void declaration used improperly.

The input language for RPCGEN has the concept of `void` declaration. This can be used only as a declaration for a variant in a union or as the argument or result of a remote procedure.

EXAMPLE: The following example shows a correct and incorrect method of using the void declarations:

```
void TIMESET(unassigned) = 2;    /* CORRECT */
voidbad_var;                     /* INCORRECT */
```

Unknown Types

```
<input_file>, line <line_number>: expected type specifier
```

Cause: An attempt was made to declare a variable to be something RPCGEN does not understand.

EXAMPLE: In the following example, the line with the comment of `OK` will not produce the "expected type specifier" message. This is because even though "flawid" is not a normally defined type specifier, it is simply a legal identifier and is the name of an unknown data type. RPCGEN assumes that the you will provide the appropriate definition and XDR routines for "flawid" data type in other files that will make up the client and server programs. The line with the comment of `NOT OK` will produce the "expected type specifier" message. This is because the "=" is not a legal value for a type specifier.

```
struct namenode {
     flawid a_var; /* OK */
     = wont_work;  /* NOT OK */
};
```

Illegal Characters

```
<input_file>, line < line_number>: illegal character in file:
```

Cause: An illegal character, such as "?", in the input file.

Missing Quotes

```
< input_file>, line < line_number>: unterminated string constant
```

Cause: A string constant is missing the terminating double quote.

General Syntax Errors

Other RPCGEN error messages that you may encounter are parsing errors that are context dependent. Because these messages depend on the type of construct being parsed, all possible messages and examples of their causes cannot be listed here.

5

XDR Protocol Specification

The RPC (Remote Procedure Call) package uses XDR (eXternal Data Representation) conventions for transmitting data. XDR works across different programming languages, operating systems, and node architectures.

This chapter explains library routines that allow you to describe arbitrary data structures in a machine-independent manner. It describes the following:

- XDR library routines.
- A guide to accessing currently available XDR streams.
- Information on defining new streams and data types.
- A formal definition of the XDR standard.

Note C programs using XDR routines must include the `<rpc/rpc.h>` file containing all the necessary interfaces to the XDR system. Since the C library `libc.a` contains all the XDR routines, compile programs as usual.

```
% cc program.c
```

Justification

The following two programs (**Writer** and **Reader**) appear to be portable because of the following:

- They pass `lint` checking.
- They exhibit the same behavior when executed locally on two different hardware architectures: an HP 9000 running HP-UX and a DEC VAX computer running the Berkeley Standard Distribution (BSD 4.3 or later) version of the UNIX operating system.

Writer Program

```
#include <stdio.h>

main()   /* writer.c */
{
        long i;

        for (i = 0; i < 8; i++) {
                if (fwrite((char *)&i, sizeof(i), 1, stdout) != 1) {
                        fprintf(stderr, "failed!\n");
                        exit(1);
                }
        }
}
```

Reader Program

```
#include <stdio.h>

main()   /* reader.c */
{
        long i, j;

        for (j = 0; j < 8; j++) {
                if (fread((char *)&i, sizeof (i), 1, stdin) != 1) {
                        fprintf(stderr, "failed!\n");
                        exit(1);
                }
                printf("%ld ", i);
        }
        printf("\n");
}
```

The concept of **network pipes** can be explained as a process producing data on one node and a second process consuming data on another node.

Piping the output of the **Writer** program to the **Reader** program gives identical results on an HP computer running the HP-UX operating system, or a DEC VAX computer running 4.3 BSD.

```
hp% writer | reader
0 1 2 3 4 5 6 7
hp%


vax% writer | reader
0 1 2 3 4 5 6 7
vax%
```

EXAMPLE: You can construct a network pipe with **Writer** and **Reader** programs. This example shows the results if the first process produces data on an HP computer and the second process consumes data on a DEC VAX computer.

```
hp% writer | remsh vax reader
0 16777216 33554432 50331648 67108864 83886080 100663296 117440512
hp%
```

You can obtain similar results by executing **Writer** on a DEC VAX computer running 4.3 BSD and **Reader** on an HP computer. These results occur because the byte ordering of long integers differs between the DEC VAX computer and the HP computer even though word size is the same. Note that 16777216 is 2^{24}. When 4 bytes are reversed, the 1 is in the 24th bit.

Whenever two or more machine types share data, the data format must be portable. You can make this program data-portable by replacing the **read()** and **write()** calls with calls to an XDR library routine **xdr_long()**. This filter knows the standard representation of a long integer in its external form.

EXAMPLE: Revised versions of **Writer** and **Reader** Programs

Writer Program

```
#include <stdio.h>
#include <rpc/rpc.h>   /* xdr is a sub-library of rpc */

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 Program

```
#include <stdio.h>
#include <rpc/rpc.h>   /* xdr is a sub-library of rpc */

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");
}
```

The new programs are executed on an HP computer, on a DEC VAX computer running 4.3 BSD, and from an HP to a DEC VAX computer running 4.3 BSD. The following sample shows the results.

```
hp% writer | reader
0 1 2 3 4 5 6 7
hp%


vax% writer | reader
0 1 2 3 4 5 6 7
vax%


hp% writer | remsh vax reader
0 1 2 3 4 5 6 7
hp%
```

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 system to system. Pointers are convenient to use, but have no meaning outside the process where they are defined.

XDR Library

The XDR library solves data portability problems. It allows you to write and read arbitrary C constructs in a consistent and specific manner. Thus, the XDR library is useful even if not sharing data among network nodes.

The XDR library has filter routines for strings (null-terminated arrays of bytes), structures, unions, and arrays. Using more primitive routines, you can write specific XDR routines to describe arbitrary data structures, including elements of arrays, arms (members) of unions, or objects pointed at from other structures.

These structures may contain arrays of arbitrary elements or pointers to other structures.

In a family of XDR stream creation routines each member treats the stream of bits differently. In this case, data is manipulated using standard I/O routines, so we use `xdrstdio_create()`. The parameters to XDR stream creation routines vary according to their function. For 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 operation may be `XDR_ENCODE` for serializing in the **Writer** program, or `XDR_DECODE` for deserializing in the **Reader** program.

Note If using standard RPC library routines, you will not need to create your own XDR streams since the RPC system creates them. The streams created by RPC are then passed to the programs.

The `xdr_long()` primitive is characteristic of most XDR library primitives and client XDR routines:

- The routine returns `TRUE` (1) if it succeeds and `FALSE` (0) if it fails.
- For each data type, `xxx`, there is an associated XDR routine of the following form.

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

In this case **xxx** is **long** so the corresponding XDR routine is the primitive **xdr_long**. The client could also define an arbitrary structure **xxx**. If it did so it would also supply the routine **xdr_xxx** describing each field by calling XDR routines of the appropriate type. You can treat the first parameter **xdrs** , as an **opaque handle** and pass it to the primitive routines. (An opaque handle is an object given to you from a lower level routine that you do not use directly, but pass it along elsewhere.)

XDR routines are direction independent; the same routines can serialize or deserialize data. This feature is critical to software engineering of portable data. You can call the same routine for either operation. (This process helps ensure serialized data can also be deserialized.) Both producer and consumer of networked data can use one routine. This is implemented by always passing the address of an object rather than the object. Only in the case of deserialization is the object modified. The value of this feature becomes obvious when nontrivial data structures are passed among nodes. If needed, you can obtain the direction of the XDR operation.

EXAMPLE:

Assume the following items.

- A person's gross assets and liabilities are to be exchanged among processes.
- 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 would be as follows.

```
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. You must inspect the return value of each XDR routine call. If the subroutine fails, quit immediately and return **FALSE** .

The above example also shows the type **bool_t** is an integer whose only values are **TRUE** (1) and **FALSE** (0). This document uses the following definitions.

```
#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()** as follows.

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

This document uses both coding styles.

XDR Library Primitives

This section gives a synopsis of each XDR primitive. It explains basic data types, constructed data types, and XDR utilities. The interface to these primitives and utilities is defined in the header file **<rpc/xdr.h>** that 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 following set of numbers.

[signed, unsigned] x [short, int, long]

Specifically, the six primitives are as follows.

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

bool_t
xdr_long(xdrs, lip)
     XDR *xdrs;
     long *lip;

bool_t
xdr_short(xdrs, sip)
     XDR *xdrs;
     short *sip;
```

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

bool_t
xdr_u_long(xdrs, lup)
     XDR *xdrs;
     u_long *lup;

bool_t
xdr_u_short(xdrs, sup)
     XDR *xdrs;
     u_short *sup;
```

The first parameter, **xdrs**, is an XDR stream handle. The second parameter is the address of the number that provides data to the stream or receives data from it. All routines return TRUE if they complete successfully or FALSE if they do not.

Floating Point Filters

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

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

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

The first parameter, **xdrs**, is an XDR stream handle. The second parameter is the address of the floating point number that provides data to the stream or receives data from it. All routines return **TRUE** if they complete successfully or **FALSE** if they do not.

Note The numbers are represented in ANSI-IEEE 754-1985 (ANSI-IEEE 754-1985 is a floating point standard that is accepted by the American National Standards Institute and the Institute of Electrical and Electronic Engineers.) floating point. Therefore, routines may fail when decoding a valid ANSI-IEEE 754-1985 representation into a machine-specific representation that is not ANSI-IEEE 754-1985, or vice versa.

Enumeration Filters

The XDR library provides a primitive for generic enumerations. This primitive assumes a C **enum** has the same representation inside the node as a C integer.

The boolean type is an important instance of the **enum**. The external representation of a boolean is always 1 (one) if **TRUE** or 0 (zero) if **FALSE**.

EXAMPLE

```
#define bool_t int
#define  FALSE  0
#define TRUE  1

#define  enum_t  int

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

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

The second parameters *ep* and *bp* are addresses of the associated type that provides data to the **xdrs** stream or receives data from it. The routines return **TRUE** if they complete successfully or **FALSE** if they do not.

No Data

Use the following function if an XDR routine must be supplied to an RPC routine even though no data is passed or required.

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

Constructed Data Type Filters

This section includes primitives for strings, arrays, unions, and pointers to structures. These constructed or compound data type primitives require more parameters and perform more complicated functions than the basic data type primitives previously discussed.

The three XDR directional operations are **XDR_ENCODE**, **XDR_DECODE**, and **XDR_FREE**. Constructed data type primitives can use memory management. In many cases, memory is allocated when deserializing data with **XDR_DECODE**. Therefore, the XDR package must provide a means to deallocate memory. The **XDR_FREE** operation performs this deallocation.

Strings

In C, a string is a sequence of bytes terminated by a null byte. However, when a string is passed or manipulated, a pointer to it is employed. Therefore, the XDR library defines a string to be a **char ***, not a sequence of characters.

The external representation of a string is very different from its internal representation. Externally, strings are sequences of ASCII characters; internally, they are character pointers. The routine **xdr_string()** converts the two representations.

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

The first parameter, *xdrs*, is the XDR stream handle. The second parameter, *sp*, is a pointer to a string (type **char ****). The third parameter, *maxlength*, specifies the maximum number of bytes allowed during encoding or decoding; its value is usually specified by a protocol. For example, a protocol specification may say a file name cannot be longer than 255 characters. The routine returns **FALSE** if the number of characters exceeds *maxlength* or if any other error occurs; it returns **TRUE** otherwise.

The behavior of **xdr_string()** is similar to the behavior of 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 it 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 value is **NULL**, a contiguous set of bytes 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 a target area was allocated that can hold strings no longer than *maxlength*.
- In either case, the string is decoded into the target area. The routine then 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 the *maxlength* parameter.

Byte Arrays

Often variable-length arrays of bytes are preferable to strings. Byte arrays differ from strings in the following three 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 their 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;
        char **bpp;
        u_int *lp;
        u_int maxlength;
```

The usage of the first, second, and fourth parameters are identical to the first, second, and third parameters of `xdr_string()`, respectively. 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. The **xdr_bytes()** routine treats a subset of generic arrays in which the size of array elements is one byte and the external description of each element is built-in. The generic array primitive, **xdr_array()**, requires parameters identical to those of **xdr_bytes()** plus two more: the size of array elements and an XDR routine to handle each of the elements. Call this routine to encode or decode arrays.

```
bool_t
xdr_array(xdrs, ap, lp, maxlength, elementsiz, xdr_element)
        XDR *xdrs;
        char **ap;
        u_int *lp;
        u_int maxlength;
        u_int elementsiz;
        bool_t (*xdr_element)();
```

The parameter *ap* is the address of the pointer to the array. 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 parameter *maxlength* is the maximum number of elements the array is allowed to have; *elementsiz* is the byte size of each element of the array. (You can use the C function **sizeof()** to obtain this value.) The **xdr_array()** function calls the **xdr_element()** routine to serialize, deserialize, or free each element of the array.

EXAMPLES:

Example A Identify a user on a networked node by the following:

- The host name, such as **krypton** (see **gethostname**).
- The user's UID (see **geteuid**).
- The group numbers to which the user belongs (see **getgroups**).

A structure with this information and its associated XDR routine could be coded as follows.

```
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));
}
```

Example B Identify a party of network users as an array of `netuser` structures. The declaration and its associated XDR routines are as follows.

```
struct party {
        u_int p_len;
        struct netuser *p_nusers;
};
#define   PLEN 500 /* max 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));
}
```

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 the following code.

```
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 **xdr_array()** function can only pass two arguments to the array element description routine, but the **xdr_string()** routine requires three arguments. The **xdr_wrap_string()** function requires only two arguments and provides the third argument to **xdr_string()**.

Opaque Data

In some protocols the server passes a handle to the client, and the client later passes the handle back to the server. Handles are opaque and never inspected by clients; they are obtained and submitted. Use the primitive **xdr_opaque()** for describing fixed sized, opaque bytes.

```
bool_t
xdr_opaque(xdrs, p, len)
        XDR *xdrs;
        char *p;
        u_int len;
```

The parameter *p* is the location of the bytes; *len* is the number of bytes in the opaque object. The actual data contained in the opaque object are system dependent.

Fixed Sized Arrays

The XDR library does not provide a primitive for fixed-length arrays. (The primitive `xdr_array()` is for varying-length arrays.)

EXAMPLE: You could rewrite the previous Example A to use fixed-sized arrays in the following manner.

```
#define NLEN 255    /* machine names must be < 256 chars */
#define NGRPS 20    /* user cannot belong to > 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);
        for (i = 0; i < NGRPS; i++) {
                if (!xdr_int(xdrs, &nup->nu_gids[i]))
                        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 a member 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 order 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 not null. Otherwise, the routine returns **FALSE**.

EXAMPLE: Assume the type of a union may be integer, character pointer (a string), or a **gnumbers** structure. Also, assume the union and its current type are declared in a structure.

```
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 structure and XDR procedure serialize or deserialize 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));
}
```

The routine **xdr_gnumbers()** was presented earlier; **xdr_wrap_string()** was presented in the previous Example C. The default **arm** parameter to **xdr_union()** (the last parameter) is **NULL** in this example. Therefore, the value of the union's discriminant may legally take on the values listed in the **u_tag_arms** array. This example also demonstrates that the elements of the arm's array do not need to be sorted.

The values of the discriminant may be sparse, though in the above example they are not. It is always good practice to assign explicitly integer values to each element of the discriminant's type. This practice documents the external representation of the discriminant and guarantees that different C compilers emit identical discriminant values.

Pointers

In C it is often convenient to put pointers to another structure within a structure. The primitive **xdr_reference()** makes it easy to serialize, deserialize, and free these referenced structures.

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

Parameter *pp* is the address of the pointer to the structure; parameter *ssize* is the size in bytes of the structure. (Use the C function **sizeof()** to obtain this value.) The XDR routine **proc** describes the structure. When decoding data, storage is allocated if **pp* is **NULL**.

The primitive **xdr_struct()** does not need to describe structures within structures since pointers are always sufficient.

Note The `xdr_reference()` and `xdr_array()` are *not* interchangeable external representations of data.

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

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

The corresponding XDR routine for this structure is as follows.

```
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);
}
```

Pointer Semantics and XDR

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. The C programmer is encoding a discriminated union efficiently by overloading the interpretation of the value of a pointer. In the above example, a NULL pointer value for **gnp** could indicate that the person's assets and liabilities are unknown.

The pointer value encodes two things: whether or not the data is known and if it is known, where it is located in memory. Linked lists are an example of the use of application-specific pointer interpretation.

The primitive **xdr_reference()** cannot attach any special meaning to a null-value pointer during serialization. Passing an address of a pointer whose value is NULL to **xdr_reference()** when serialing data may cause a memory fault and, on UNIX operating systems, a core dump for debugging.

You must expand non-dereferenceable pointers into their specific semantics. This process usually involves describing data with a two-armed discriminated union. One arm is used when the pointer is valid; the other is used when the pointer is NULL.

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;

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

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

Note In some XDR streams the returned value of **xdr getpos()** is meaningless. In this case, the routine returns a **(u_int) -1**.

The routine **xdr_setpos()** sets a stream position to **pos**.

Note In some XDR streams, setting a position is impossible. In such cases **xdr_setpos()** returns **FALSE**.

This routine fails if the requested position is invalid (out of bounds). The definition of bounds varies from stream to stream.

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

XDR Operation Directions

You may wish 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. Though you generally will not need this information, the field may be needed in some circumstances.

XDR Stream Access

Obtain an XDR stream by calling the appropriate creation routine. These creation routines take arguments tailored to the specific properties of the stream.

Streams currently exist for serialization and deserialization of data to or from standard I/O `FILE` streams, TCP/IP connections, UNIX operating system files, and memory.

Standard I/O Streams

The routine `xdrstdio_create()` initializes an XDR stream, pointed to by `xdrs` using the standard I/O library routines. The *fp* parameter is an open file, and *x_op* is an XDR direction.

```
#include <stdio.h>
#include <rpc/rpc.h>   /* xdr streams part of rpc */

void
xdrstdio_create(xdrs, fp, x_op)
        XDR *xdrs;
        FILE *fp;
        enum xdr_op x_op;
```

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;
        char *addr;
        u_int len;
        enum xdr_op x_op;
```

The routine **xdrmem_create()** initializes an XDR stream in local memory. The **addr** parameter points to the memory; the **len** parameter is the length in bytes of the memory. The parameters **xdrs** and **x_op** are identical to the corresponding parameters of **xdrstdio_create**. Currently, the UDP/IP implementation of RPC uses **xdrmem_create**. Complete call or result messages are built in 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 UNIX operating system file or 4.3 BSD connection interface.

```
#include <rpc/rpc.h>   /* xdr streams part of rpc */

void
xdrrec_create(xdrs, sendsize, recvsize, iohandle, readproc, writeproc)
        XDR *xdrs;
        u_int sendsize, recvsize;
        char *iohandle;
        int (*readproc)(), (*writeproc)();
```

The routine **xdrrec_create()** provides an XDR stream interface that allows for a bidirectional, arbitrarily long sequence of records. The contents of the records should be data in XDR form. The stream's primary use is for interfacing RPC to TCP connections. However, you can use it to stream data into or out of normal UNIX operating system files.

The parameter *xdrs* is similar to the corresponding parameter of *xdrstdio_create()*. The stream performs its own data buffering similar to that of standard I/O. The parameters *sendsize* and *recvsize* determine the size in bytes of the output and input buffers, respectively. If their values are zero (0), then predetermined defaults are used. When a buffer needs to be filled or flushed, the routine **readproc()** or **writeproc()** is called, respectively. The usage and behavior of these routines are similar to the UNIX system calls **read()** and **write()**. However, the first parameter to each of these 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**, then it has the following form:

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

The XDR stream provides a means for delimiting records in the byte stream. Refer to the "Synopsis of XDR Routines" section for implementation details of delimiting records in a stream. The primitives specific to record streams are as follows.

Primitives Specific to Record Streams	Description

`bool_t`
`xdrrec_endofrecord(xdrs, flushnow)`
`XDR *xdrs;`
`bool_t flushnow;`
	The routine **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 is filled.

`bool_t`
`xdrrec_skiprecord(xdrs)`
`XDR *xdrs;`
	The routine **xdrrec_skiprecord()** causes an input stream's position to be moved past the current record boundary and onto the beginning of the next record in the stream.

`bool_t`
`xdrrec_eof(xdrs)`
`XDR *xdrs;`
	If there is no more data in the stream's input buffer, the routine **xdrrec_eof()** returns TRUE. Note, this condition does not imply there is no more data in the underlying file descriptor.

XDR Stream Implementation

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

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 param */
     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)();    /* ptr 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 info */
     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, but should not affect a stream's implementation. A stream's implementation should not depend on this value. The fields **x_private**, **x_base**, and **x_handy** are private to the particular stream's implementation. The field **x_public** is for the XDR client and should never be used by the XDR stream implementations or the XDR primitives.

The operation **x_inline()** takes two parameters: an **XDR *** and an unsigned integer that is a byte count. The routine returns a pointer to a piece of the stream's internal buffer. The caller can then use the buffer segment for any purpose. From the stream's point of view, the bytes in the buffer segment were consumed or put. The routine may return **NULL** if it cannot return a buffer segment of the requested size. (The **x_inline()** routine is for directly accessing the underlying buffer. Use of the resulting buffer is not data-portable; therefore, we recommend you do not use this feature.)

The operations **x_getbytes()** and **x_putbytes()** blindly obtain and put sequences of bytes from or to the underlying stream; they return **TRUE** if they are successful or **FALSE** if they are not. The routines have identical parameters.

EXAMPLE:

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

The operations **x_getlong()** and **x_putlong()** receive and put long numbers from and to the data stream. These routines translate the numbers between the node representation and the (standard) external representation. The UNIX operating system primitives **htonl()** and **ntohl()** can be helpful in accomplishing this translation. The higher-level XDR implementation assumes the following:

- Signed and unsigned long integers contain the same number of bits.
- Non-negative integers have the same bit representations as unsigned integers.

The routines return **TRUE** if they succeed or **FALSE** if they do not. They have identical parameters.

EXAMPLE:

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

XDR Standard

The XDR standard is independent of languages, operating systems, and hardware architectures. Once data is shared among nodes, it should not matter if the data was produced on an HP computer and consumed by another vendor's computer, or vice versa. Similarly, the choice of operating systems should have no influence on how the data is represented externally. For programming languages, data produced by a C program should be readable by a Fortran or Pascal program.

The XDR standard depends on the assumption that bytes (or octets) are portable. (A byte is eight bits of data.) Hardware that encodes bytes onto various media should preserve the bytes' meanings across hardware boundaries. Both HP and DEC VAX computer hardware implementations adhere to the standard.

The XDR standard also suggests a language used to describe data. The language is a "changed" C; it is a data description language, not a programming language.

Basic Block Size

The representation of all items requires a multiple of 4 bytes (or 32 bits) of data. The bytes are numbered `0` through `n-1`, where `(n mod 4) = 0`. The bytes are read, or written to, a byte stream such that byte `m` always precedes byte `m+1`.

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 and least significant bytes are 0 and 3, respectively. The data description of integers is `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. The data description of unsigned integers is **unsigned**.

Enumerations

Enumerations have the same representation as integers and are useful for describing subsets of the integers. The data description of enumerated data is as follows.

```
typedef enum { name = value, ... } type-name;
```

For example, you could describe the three colors red, yellow, and blue by an enumerated type.

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

Booleans

Since booleans are important and occur frequently, they warrant their own explicit type in the standard. The boolean type is an enumeration with the following form.

```
typedef enum { FALSE = 0, TRUE = 1 } boolean;
```

Floating Point and Double Precision

The standard defines the encoding for the floating point data types `float` (32 bits or 4 bytes) and `double` (64 bits or 8 bytes). The standard encodes the following three fields to describe the floating point number.

S	The sign of the number. Values 0 and 1 represent positive and negative, respectively.
E	The exponent of the number, base 2. Type `float` devotes 8 bits to this field; `double` devotes 11 bits. The exponents for `float` and `double` are biased by 127 and 1023, respectively.
F	The fractional part of the number's mantissa, base 2. Type `float` devotes 23 bits to this field; `double` devotes 52 bits.

Therefore, the floating point number is described as follows.

$$(-1)^{S} * 2^{(E-Bias)} * (1.f)$$

Just as the most and least significant bytes of a number are 0 and 3, the most and least significant bits of a single-precision floating point number are 0 and 31. The beginning bit (and most significant bit) offsets of S, E, and F are 0, 1, and 9, respectively.

Type `double` has the analogous extensions. The beginning bit (and most significant bit) offsets of S, E, and F are 0, 1, and 12, respectively.

Consult the ANSI-IEEE 754-1985 specification concerning the encoding for signed zero, signed infinity (overflow), and denormalized numbers (underflow). Under ANSI-IEEE 754-1985 specifications, the "NaN" (not a number) is a system dependent and should not be used.

Opaque Data

You may need to pass fixed-sized uninterpreted data among nodes. This data is called **opaque** and is described as follows.

```
typedef opaque type-name[n];
opaque name[n];
```

The **n** is the (static) number of bytes necessary to contain the opaque data. If **n** is not a multiple of four, then the **n** bytes are followed by enough (up to three) zero-valued bytes to make the total byte count of the opaque object a multiple of four.

Counted Byte Strings

The XDR standard defines a string of **n** (numbered **0** through **n-1**) bytes to be the number **n** encoded as **unsigned** and followed by the **n** bytes of the string. If **n** is not a multiple of four, the **n** bytes are followed by enough (up to three) zero-valued bytes to make the total byte count a multiple of four. The data description of strings is as follows:

```
typedef string type-name<N>;
typedef string type-name<>;
string name<N>;
string name<>;
```

Note that the data description language uses angle brackets (< and >) to denote anything that varies in length (instead of square brackets to denote fixed-length sequences of data).

The constant **N** denotes an upper bound of the number of bytes that a string can contain. The protocol using XDR specifies **N** which must be less than $2^{32} - 1$. For example, a filing protocol may state that a file name can be no longer than 255 bytes.

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

The XDR specification does not define what the individual bytes of a string represent. This important information is left to higher-level specifications. A reasonable default is to assume the bytes encode ASCII characters.

Fixed Arrays

The data description for fixed-size arrays of homogeneous elements is as follows.

```
typedef elementtype type-name[n];
elementtype name[n];
```

Fixed-size 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`.

Counted Arrays

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. Array elements start with element `0` and progress through element `n-1`.

The data description for counted arrays is similar to that of counted byte strings.

```
typedef elementtype type-name<N>;
typedef elementtype type-name<>;
elementtype name<N>;
elementtype name<>;
```

The constant `N` specifies the maximum acceptable element count of an array that must be less than $2^{32} - 1$.

Structures

The data description for structures is very similar to that of standard C.

```
typedef struct {
        component-type component-name;
 ...
} type-name;
```

An XDR routine generally encodes the structure components in the order of their declaration in the structure, but need not do so.

Discriminated Unions

A discriminated union is a type composed of a discriminant followed by a type selected from a set of pre-arranged types according to the value of the discriminant. The type of the discriminant is always an enumeration. The component types are called "arms" of the union. The discriminated union is encoded as its discriminant followed by the encoding of the implied arm. The data description for discriminated unions is as follows.

```
typedef union switch (discriminant-type) {
      discriminant-value: arm-type;
 ...
      default: default-arm-type;
} type-name;
```

The default arm is optional. If it is not specified, a valid encoding of the union cannot take on unspecified discriminant values. Most specifications do not need or use default arms.

Missing Specifications

The XDR standard lacks representations for bit fields and bitmaps since it is based on bytes. However, this lack of representations does not mean bit fields and bit maps cannot be represented.

Library Primitive / XDR Standard Cross Reference

The following table describes the association between the C library primitives and the standard data types.

C Primitive	XDR Type
xdr_int xdr_long xdr_short	Integer
xdr_u_int xdr_u_long xdr_u_short	Unsigned
xdr_float	Float
xdr_double	Double
xdr_enum	enum_t
xdr_bool	bool_t
xdr_string xdr_bytes	String
xdr_array	(Varying arrays)
xdr_vector	(Fixed arrays)
xdr_opaque	Opaque
xdr_union	Union
xdr_reference xdr_pointer	Pointers
xdr_char xdr_u_char	Char
User Provided	Struct

Advanced XDR Topics

This section describes techniques for passing data structures that are not covered in the preceding sections. Such structures include linked lists (of arbitrary lengths).

Unlike the simpler examples covered in the earlier sections, the following examples use both the XDR C library routines and the XDR data description language.

Linked Lists

The following C data structure example contains XDR routines for a person's gross assets and liabilities.

EXAMPLE:

```
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);
}
```

Now assume you wish to implement a linked list of such information. You could construct a data structure as follows.

```
typedef struct gnnode {
        struct gnumbers gn_numbers;
        struct gnnode *nxt;
};

typedef struct gnnode *gnumbers_list;
```

Think of the head of the linked list as representing the entire link list. The **nxt** field indicates whether or not the object has terminated. If the object continues, the **nxt** 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 type declaration of **gnumbers_list**.

```
struct gnumbers {
        unsigned g_assets;
        unsigned g_liabilities;
};
typedef union switch (boolean) {
        case TRUE: struct {
                struct gnumbers current_element;
                gnumbers_list rest_of_list;
        };
        case FALSE: struct {};
} gnumbers_list;
```

In this description, the boolean indicates whether there is more data following it. 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 rest of the object). Note that the C declaration has no boolean explicitly declared in it (though the **nxt** field implicitly carries the information). The XDR data description has no pointer explicitly declared in it.

Hints for writing a set of XDR routines to successfully serialize or deserialize a linked list of entries are in the XDR description of the pointer-less data. This set includes the mutually recursive routines **xdr_gnumbers_list**, **xdr_wrap_list**, and **xdr_gnnode**.

```
bool_t
xdr_gnnode(xdrs, gp)
        XDR *xdrs;
        struct gnnode *gp;
{
        return(xdr_gnumbers(xdrs, &(gp->gn_numbers)) &
            xdr_gnumbers_list(xdrs, &(gp->nxt)) );
}

bool_t
xdr_wrap_list(xdrs, glp)
        XDR *xdrs;
        gnumbers_list *glp;
{
        return(xdr_reference(xdrs, glp, sizeof(struct gnnode),
            xdr_gnnode));
}

struct xdr_discrim choices[2] = {
            /*
            * called if another node needs (de)serializing
            */
            { TRUE, xdr_wrap_list },
            /*
            * called when no more nodes need (de)serializing
            */
            { FALSE, xdr_void }
}
```

```
bool_t
xdr_gnumbers_list(xdrs, glp)
        XDR *xdrs;
        gnumbers_list *glp;
{
        bool_t more_data;

        more_data = (*glp != (gnumbers_list)NULL);
        return(xdr_union(xdrs, &more_data, glp, choices, NULL));
}
```

The entry routine is **xdr_gnumbers_list()**; it translates between the boolean value **more_data** and the list pointer values. If there is no more data, the **xdr_union()** primitive calls **xdr_void()** and the recursion terminates. Otherwise, **xdr_union()** calls **xdr_wrap_list()** to dereference the list pointers. The **xdr_gnnode()** routine actually serializes or deserializes data of the current node of the linked list and recursively calls **xdr_gnumbers_list()** to handle the remainder of the list.

These routines function correctly in all three directions (**XDR_ENCODE**, **XDR_DECODE**, and **XDR_FREE**) for linked lists of any length (including zero). Note, the boolean **more_data** is always initialized, but in the **XDR_DECODE** case it is overwritten by an externally generated value. Also note the value of the **bool_t** is lost in the stack. The value is reflected in the list's pointers.

If serializing or deserializing a list with these routines, the C stack grows linearly with respect to the number of nodes in the list. This linear growth is due to the recursion. The routines are also hard to code and understand due to the number and nature of primitives involved (e.g., **xdr_reference**, **xdr_union**, and **xdr_void**).

EXAMPLE: This example routine collapses the recursive routines. It also has other optimizations as discussed afterwards.

```
bool_t
xdr_gnumbers_list(xdrs, glp)
        XDR *xdrs;
        gnumbers_list *glp;
{
        bool_t more_data;

        while (TRUE) {
              more_data = (*glp != (gnumbers_list)NULL);
              if (!xdr_bool(xdrs, &more_data))
                    return(FALSE);
              if (!more_data)
                    return(TRUE);  /* we are done */
              if (!xdr_reference(xdrs, glp, sizeof(struct gnnode),
                 xdr_gnumbers))
                    return(FALSE);
              glp = &((*glp)->nxt);
              }
}
```

This routine is easier to code and understand than the above three recursive routines, but still has difficulties. The parameter **glp** is treated as the address of the pointer to the head of the remainder of the list to be serialized or deserialized. Thus, **glp** is set to the address of the current node's **nxt** field at the end of the while loop. The discriminated union is implemented in-line; the variable **more_data** has the same use in this routine as in the above routines. Its value is recomputed and re-serialized or re-deserialized each iteration of the loop. Since ***glp** is a pointer to a node, the pointer is dereferenced using **xdr_reference**. Note that the third parameter is truly the size of a node (data values plus **nxt** pointer), while **xdr_gnumbers()** only serializes or deserializes the data values. This optimization works only because the **nxt** data occurs after all legitimate external data.

The routine has difficulties in the **XDR_FREE** case. The **xdr_reference()** frees the node ***glp**. Upon return, the assignment **glp = &((*glp)->nxt)**cannot be guaranteed to work since ***glp** is no longer a legitimate node.

The following rewrite works in all cases. Avoid dereferencing a pointer that was not initialized or already freed.

```
bool_t
xdr_gnumbers_list(xdrs, glp)
        XDR *xdrs;
        gnumbers_list *glp;
{
        bool_t more_data;
        bool_t freeing;
        gnumbers_list *next;  /* the next value of glp */

        freeing = (xdrs->x_op == XDR_FREE);
        while (TRUE) {
              more_data = (*glp != (gnumbers_list)NULL);
              if (!xdr_bool(xdrs, &more_data))
                    return(FALSE);
              if (!more_data)
                    return(TRUE);  /* we are done */
              if (freeing)
                    next = &((*glp)->nxt);
              if (!xdr_reference(xdrs, glp, sizeof(struct gnnode),
                    xdr_gnumbers))
                          return(FALSE);
              glp = (freeing) ? next : &((*glp)->nxt);
        }
}
```

Note that the previous example inspects the direction of the operation `xdrs->x_op`. The correct iterative implementation is still easier to understand or code than the recursive implementation. It is certainly more efficient with respect to C stack usage.

Record Marking Standard

Record marking (RM) is the process of delimiting one message from another when RPC messages pass on top of a byte stream protocol (like TCP/IP). RM helps detect and possibly recover from user protocol errors. This RM/TCP/IP transport passes RPC messages on TCP streams. One RPC message fits into one RM record. A record contains one or more record fragments. A record fragment is a 4-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 two values:

- A boolean indicating whether the fragment is the last fragment of the record (bit value 1 implies the fragment is the last fragment).
- A 31-bit unsigned binary value that 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. (Note that this record specification is not in XDR standard form.)

Synopsis of XDR Routines

Routine	xdr_array()
Description	A filter primitive that translates between arrays and their corresponding external representations. The parameter **arrp** is the address of the pointer to the array. The parameter **sizep** is the address of the element count of the array; this element count cannot exceed **maxsize**. The parameter **elsize** is the **sizeof()** each of the array's elements. The parameter **elproc** is an XDR filter that translates between the array elements' C form and their external representations. This routine returns **TRUE** if it succeeds or **FALSE** if it does not.
Synopsis	**bool_t** **xdr_array(xdrs, arrp, sizep, maxsize, elsize, elproc)** **XDR*xdrs;** **char **arrp;** **u_int *sizep, maxsize, elsize;** **xdrproc_t elrpoc;**

Routine	`xdr_bool()`
Description	A filter primitive that translates between booleans (C integers) and their external representations. When encoding data, this filter produces values of either **TRUE** or **FALSE**. This routine returns **TRUE** if it succeeds or **FALSE** if it does not.

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

Routine	`xdr_bytes()`
Description	A filter primitive that translates between counted byte strings and their external representations. The parameter **sp** is the address of the byte string pointer. The length of the byte string is located at address **sizep**; byte strings cannot be longer than **maxsize**. This routine returns **TRUE** if it succeeds or **FALSE** if it does not.

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

Routine	xdr_char()
Description	A filter primitive that translates between C characters and their external representations. This routine returns **TRUE** if it succeeds or **FALSE** if it does not.

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

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

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

Routine	`xdr_double()`
Description	A filter primitive that translates between C **double** precision numbers and their external representations. This routine returns **TRUE** if it succeeds or **FALSE** if it does not.

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

Routine	`xdr_enum()`
Description	A filter primitive that translates between the C **enum** (an integer) and its external representation. This routine returns **TRUE** if it succeeds or **FALSE** if it does not.

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

Routine	`xdr_float( )`
Description	A filter primitive that translates between the C `float` and its external representation. This routine returns `TRUE` if it succeeds or `FALSE` if it does not.

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

Routine	`xdr_getpos()`
Description	A macro that invokes the get-position routine associated with the XDR stream `xdrs`. The routine returns an unsigned integer to indicate the XDR byte stream position. A desirable feature of XDR streams is that simple arithmetic works with this number, although the XDR stream instances need not guarantee this. If this routine fails, it returns `(u_int) -1`.

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

Routine	xdr_free
Description	This routine frees the memory that an XDR data structure occupies. It can be used on arbitrary structures. The first parameter, **proc**, is a pointer to the XDR routine for the object being freed. The second parameter, **objp**, points to the object to be freed.

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

Note	The pointer passed to this routine is NOT freed, but what it points to is freed.

Routine	xdr_inline()
Description	A macro that invokes the in-line routine associated with the XDR stream **xdrs**. The routine returns a pointer to a contiguous piece of the stream's buffer; **len** is the byte length of the desired buffer. The pointer is cast to **long ***.

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

Note	The **xdr_inline( )** function may return **NULL** if it cannot allocate a contiguous piece of a buffer; therefore, the behavior may vary among stream instances. The **xdr_inline( )** routine exists for the sake of efficiency, though HP recommends that you do not use it.

Routine	xdr_int()
Description	A filter primitive that translates between C integers and their external representations. This routine returns **TRUE** if it succeeds or **FALSE** if it does not.

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

Routine	xdr_long()
Description	A filter primitive that translates between C **long** integers and their external representations. This routine returns **TRUE** if it succeeds or **FALSE** if it does not.

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

Routine	xdr_opaque()
Description	A filter primitive that translates between fixed size opaque data and its external representation. The parameter cp is the address of the opaque object, and cnt is its size in bytes. This routine returns TRUE if it succeeds or FALSE if it does not.

Synopsis	`bool_t`
	`xdr_opaque(xdrs, cp, cnt)`
	`XDR *xdrs;`
	`chap *cp;`
	`u_int cnt;`

Routine	xdr_pointer()
Description	A routine that is similar to **xdr_reference()** in that it provides pointer dereferencing within structures. It differs from **xdr_reference()** in its ability to handle NULL pointers. Therefore **xdr_pointer()** can create recursive data structures, such as binary trees or linked lists, correctly, whereas **xdr_reference()** will fail. The parameter **xproc** is an XDR procedure that filters the structure between its C form and its external representation. This routine returns **TRUE** if it succeeds or **FALSE** if it does not.

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

Routine	xdr_reference( )
Description	A primitive that provides pointer dereferencing within structures. The parameter pp is the address of the pointer. The parameter size is the sizeof( ) the structure to which *pp points. The parameter proc is an XDR procedure that filters the structure between its C form and its external representation. This routine returns TRUE if it succeeds or FALSE if it does not.
Synopsis	bool_t xdr_reference(xdrs, pp, size, proc) XDR *xdrs; char **pp; u_int size; xdrproc_t proc;

Routine	xdr_setpos( )
Description	A macro that invokes the set position routine associated with the XDR stream xdrs. The parameter pos is a position value obtained from xdr_getpos. This routine returns TRUE if the XDR stream could be repositioned or FALSE if it could not.
Synopsis	bool_t xdr_setpos(xdrs, pos) XDR *xdrs; u_int pos;
Note	Since it is difficult to reposition some types of XDR streams, this routine may fail with one type of stream and succeed with another.

Routine	xdr_short( )
Description	A filter primitive that translates between C **short** integers and their external representations. This routine returns **TRUE** if it succeeds or **FALSE** if it does not.

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

Routine	xdr_string( )
Description	A filter primitive that translates between null-terminated strings and their corresponding external representations. Strings cannot be longer than **maxsize**. The parameter **sp** is the address of the string's pointer. This routine returns **TRUE** if it succeeds or **FALSE** if it does not.

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

Routine	xdr_u_char()
Description	A filter primitive that translates between C unsigned characters and their external representations. This routine returns **TRUE** if it succeeds or **FALSE** if it does not.

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

Routine	xdr_union( )
Description	A filter primitive that translates between a discriminated C **union** and its corresponding external representation. The parameter **dscmp** is the address of the union's discriminant. The parameter **unp** in the address of the union. This routine returns **TRUE** if it succeeds or **FALSE** if it does not.

Synopsis	`bool_t`
	`xdr_union(xdrs, dscmp, ump, choises, dfault)`
	`  XDR *xdrs;`
	`  int *dscmp;`
	`  char *unp;`
	`  struct xdr_discrim *choises;`
	`  xdrproc_t dfault;`

Routine	xdr_vector()
Description	A filter primitive that translates between fixed-length arrays and their corresponding external representations. The parameter **arrp** is the address of the beginning of the array. The parameter **elsize** is the sizeof of each of the array's elements, and **elproc** is an XDR filter that translates between the array elements' C form and their external representation. This routine returns **TRUE** if it succeeds and **FALSE** if does not.

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

Routine	xdr_void( )
Description	This routine takes no arguments and always returns **TRUE**.
Synopsis	`bool_t` `xdr_void( )`
Note	Use this routine when an XDR routine is required

Routine	xdr_wrapstring( )
Description	A primitive that calls **xdr_string(xdrs,sp,MAXUNSIGNED)** where **MAXUNSIGNED** is the maximum value of an **unsigned** integer. This routine is useful because the RPC package passes only two parameter XDR routines; **xdr_string( )**, one of the most frequently used primitives, requires three parameters. This routine returns **TRUE** if it succeeds or **FALSE** if it does not.
Synopsis	**bool_t** **xdr_wrapstring(xdrs, sp)** **XDR *xdrs;** **char **sp;**

Routine	xdr_u_int( )
Description	A filter primitive that translates between C **unsigned** integers and their external representations. This routine returns **TRUE** if it succeeds or **FALSE** if it does not.
Synopsis	**bool_t** **xdr_u_int(xdrs, up)** **XDR *xdrs;** **unsigned *up;**

Routine	`xdr_u_long( )`
Description	A filter primitive that translates between C **`unsigned long`** integers and their external representations. This routine returns **`TRUE`** if it succeeds or **`FALSE`** if it does not.

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

Routine	`xdr_u_short( )`
Description	A filter primitive that translates between C **`unsigned short`** integers and their external representations. This routine returns **`TRUE`** if it succeeds or **`FALSE`** if it does not.

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

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

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

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

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

Note	Additional bytes in the stream are used to provide record boundary information.

Routine	xdrrec_endofrecord( )
Description	Invoke this routine only on streams created by **xdrrec_create**. The data in the output buffer is marked as a completed record. The output buffer is optionally written out if **sendnow** is nonzero. This routine returns **TRUE** if it succeeds or **FALSE** if it does not.

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

Routine	xdrrec_eof( )
Description	Invoke this routine only on streams created by **xdrrec_create**. After consuming the remainder of the current record in the stream, this routine returns **TRUE** if the stream has no more input or **FALSE** if it does.

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

Routine	xdrrec_skiprecord( )
Description	Invoke this routine only on streams created by **xdrrec_create**. This routine tells the XDR implementation that the rest of the current record in the stream's input buffer should be discarded. This routine returns **TRUE** if it succeeds or **FALSE** if it does not.

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

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

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

Note	The destroy routine associated with such XDR streams calls **fflush( )** on the **file** stream.

6

RPC Protocol Specification

This chapter explains the message protocol that is used to implement the RPC (Remote Procedure Call) package. The protocol is specified with the XDR (eXternal Data Representation) language.

You should be familiar with both RPC and XDR before reading this chapter.

RPC Model

The RPC model is similar to the local procedure call model. In the local case, the caller places arguments to a procedure in a specific location (e.g., a result register). It then transfers control to the procedure and eventually gains back control. The results of the procedure are extracted from the specified location, and the caller continues execution.

The remote procedure call is similar, except that one thread of control winds through two processes: one is a caller's process, the other is a server's process.

The caller process sends a call message to the server process and waits (blocks) for a reply message. The call message contains the procedure's parameters, and the reply message contains the procedure's results. After receiving the reply message, the caller process extracts the procedure results and resumes execution.

On the server side, a process is dormant while awaiting the arrival of a call message. When one arrives, the server process does the following:

- Extracts the procedure's parameters.
- Computes the results.
- Sends a reply message.
- Waits for the next call message.

Note that only one of the two processes is active at any given time. The RPC protocol does not explicitly support simultaneous execution of caller and server processes.

Transports and Semantics

Since the RPC protocol is independent of transport protocols, it does not care how a message passes from one process to another. It determines the specification interpretation of messages, but does not determine the specific semantics.

An RPC message-passing protocol using UDP/IP is unreliable. Thus, if the caller retransmits call messages after short time-outs, the only thing it can determine from no reply message is that the remote procedure was executed zero or more times. The only thing is can determine from a reply message is that the remote procedure was executed one or more times.

An RPC message-passing protocol using TCP/IP is reliable. No reply message means the remote procedure was executed at most once. A reply message means the remote procedure was executed exactly once.

Note RPC is implemented on top of the TCP/IP and UDP/IP transports.

Message Authentication

The RPC protocol provides the fields necessary for a client to identify itself to a service and vice versa. You can build security access-control mechanisms on top of the message authentication.

RPC Protocol Requirements

The RPC protocol must provide for the following:

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

The features that detect the following are required because of protocol roll-over errors, implementation defects, user error, and network administration:

- RPC protocol mismatches.
- Remote program protocol version mismatches.
- Protocol errors (e.g., mis-specification of a procedure's parameters).
- Reasons why remote authentication failed.
- Any other reasons why the desired procedure was not called.

Remote Programs and Procedures

The RPC call message has three unsigned fields:

- Remote program number.
- Remote program version number.
- Remote procedure number.

These fields uniquely identify the procedure being called. A central authority administers the program numbers. Once you have a program number, you can implement a remote program; the first implementation would most likely have the version number of 1. Since most new protocols evolve into more stable and mature protocols, a version field of the call message identifies which protocol version the caller is using. Version numbers enable you to speak old and new protocols through the same server process.

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

Just as remote program protocols may change over several versions, the actual RPC message protocol can also change. Therefore, the call message also has the RPC version number in it. This documentation describes version 2 of the RPC protocol.

The reply message to a request message has ample information to distinguish the following error conditions.

- The remote implementation of RPC does not speak protocol version 2.
- 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 invalid from the server's point of view. (This error results from a disagreement about the protocol between caller and service.)

Authentication

The call message has two authentication fields: the credentials and verifier. The reply message has one authentication field: the response verifier. The RPC protocol specification defines all three fields as the following opaque type.

```
enum auth_flavor {
     AUTH_NULL= 0,
     AUTH_UNIX= 1,
     AUTH_SHORT= 2
     /* and more to be defined */
};
struct opaque_auth  {
     union  switch (enum auth_flavor) {
            default: string  auth_body<400>;
     };
};
```

Any **opaque_auth** structure is an **auth_flavor** enumeration followed by a counted string whose bytes are opaque to the RPC protocol implementation.

Independent authentication protocol specifications describe the interpretation and semantics of the data contained within the authentication fields.

If the server rejects the RPC call due to authentication parameters, the response message states why they were rejected.

Refer to the "Portmapper Program Protocol" section for the definition of the three authentication protocols.

Program Numbers

Program numbers are assigned in groups of 0x20000000 as follows.

```
         0 -    1fffffff defined by Sun
  20000000 -    3fffffff defined by user
  40000000 -    5fffffff  transient
  60000000 -    7fffffff  reserved
  80000000 -    9fffffff  reserved
  a0000000 -    bfffffff  reserved
  c0000000 -    dfffffff  reserved
  e0000000 -    ffffffff  reserved
```

0 - 1fffffff defined by Sun

Sun Microsystems, Inc. administers the first group of numbers which should be identical for all systems. If you develop an application of general interest, that application should receive an assigned number in the first range.

20000000 - 3fffffff defined by user

The second group of numbers is reserved for specific customer applications. This range is primarily for debugging new programs.

40000000 - 5fffffff transient

The third group is reserved for applications that generate program numbers dynamically.

60000000 - 7fffffff reserved
80000000 - 9fffffff reserved
a0000000 - bfffffff reserved
c0000000 - dfffffff reserved
e0000000 - ffffffff reserved

The final groups are reserved for future use and should not be used.

To register a protocol specification, send a request to the following address. Please include a complete protocol specification similar to those in this manual. In return, you will receive a unique program number.

Network Administration Office, Dept. NET
Information Networks Division
Hewlett-Packard Company
19420 Homestead Road
Cupertino, California 95014
408-447-3444

Additional RPC Protocol Uses

This protocol is for calling remote procedures. Each call message generates a matching response message.

The protocol is also a message passing protocol with which you can implement other non-RPC protocols. RPC message protocols are used for the following two non-RPC protocols: batching (or pipelining) and broadcast RPC.

Batching

Batching allows a client to send an arbitrarily large sequence of call messages to a server; it uses reliable byte stream protocols (like TCP/IP) for their transport.

The client never waits for a reply from the server, and the server does not send replies to batch requests. A non-batched RPC call usually terminates a sequence of batch calls to flush the batched requests by waiting for positive acknowledgement.

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 (like UDP/IP) as their transports. Servers that support broadcast protocols only respond when the request is successfully processed and are silent when errors occur.

RPC Message Protocol

This section defines the RPC message protocol in the XDR data description language.

Note The following code is an XDR specification, *not* C code.

```
enum  msg_type  {
     CALL =  0,
     REPLY =  1
};

/*
 * A reply to a call message can take on two forms:
 * the message was either 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 */
};
```

```
    /*
     * 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; the service side cannot
     * treat this id as any type of sequence number.
     */
struct rpc_msg {
     unsigned  xid;
     union  switch (enum  msg_type) {
             CALL: struct call_body;
             REPLY: struct reply_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 rpcvers; /* must be equal to two (2) */
            unsigned prog;
            unsigned vers;
            unsigned proc;
            struct opaque_auth cred;
            struct opaque_auth verf;
            /* procedure specific parameters start here */
    };

    /*
     * Body of a reply to an RPC request.
     * The call message was either accepted or rejected.
     */
struct reply_body {
     union switch (enum reply_stat) {
             MSG_ACCEPTED:struct   accepted_reply;
             MSG_DENIED:struct   rejected_reply;
     };
};
```

```
/*
 * Reply to an RPC request that was accepted by the server.
 * Note: there could be an error even though the request
 * was accepted.  The first field is an authentication
 * verifier that the server generates in order 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_ARGS arms of the union are
 * void.  The PROG_MISMATCH arm specifies the lowest and
 * highest version numbers of the remote program that are
 * supported by the server.
 */
struct accepted_reply {
        struct    op aque_authverf;
        union switch (enum accept_stat) {
                SUCCESS: struct {
                        /*
                         * procedure-specific results start here
                         */
                };
                PROG_MISMATCH: struct {
                        unsigned low;
                        unsigned  high;
                };
                default: struct {
                        /*
                         * void. Cases include PROG_UNAVAIL,
                         * PROC_UNAVAIL, and GARBAGE_ARGS.
                         */
                };
        };
};
```

```
/*
 * Reply to an RPC request that was rejected by the server.
 * The request can be rejected because of 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 the case
 * of refused authentication, failure status is returned.
 */
struct rejected_reply {
        union switch (enum reject_stat) {
                RPC_MISMATCH: struct {
                        unsigned low;
                        unsigned high;
                };
                AUTH_ERROR: enum auth_stat;
        };
};
```

Authentication Parameter Specification

The RPC protocol does not define how to use authentication parameters. Instead, it passes them, unmodified, between client and server. The client and server applications are responsible for interpreting the authentication parameters.

Note The RPC protocol allows you to specify your own form of authentication, but to do so you must have access to the RPC authentication source files. Implementations based on NFS 3.2 (including HP-UX 8.0 for HP 9000 computers) do *not* allow you to define your own form of authentication.

NULL Authentication

The caller may not know who it is, or the server may not care who the caller is. In this case, the `auth_flavor` value (the discriminant of the `opaque_auth's` union) of the RPC message's credentials, verifier, and response verifier is `AUTH_NULL` (0). The bytes of the `auth_body` string are undefined. We recommend the string length be zero.

UNIX Authentication

The caller of a remote procedure may wish to identify himself as he is identified on a UNIX system.

- The value of the `credential`'s discriminant of an RPC call message is `AUTH_UNIX`, which has a value of one (1).
- The bytes of the `credential`'s string encode the following XDR structure.

```
struct auth_unix
{
        unsigned stamp;
        string machinename<255>;
        unsigned uid;
        unsigned gid;
        unsigned gids <8>;
};
```

Field	Description
stamp	An arbitrary ID the caller node may generate
machinename	The caller's host name
uid	The caller's effective user ID
gid	The caller's effective group ID
gids	A counted array of group IDs containing the caller as a member.

The verifier accompanying the credentials should be **AUTH_UNIX**.

The discriminate value of the response verifier received in the server's reply message may be **AUTH_NULL** or **AUTH_SHORT**.

For **AUTH_SHORT**, the bytes of the response verifier's string encode an **auth_opaque** structure. This new **auth_opaque** structure may now be passed to the server instead of the original **AUTH_UNIX** flavor credentials. The server keeps a cache that maps shorthand **auth_opaque** structures (passed back in a **AUTH_SHORT** style response verifier) to the caller's original credentials. The caller can save network bandwidth and server CPU cycles by using the new credentials.

The server may flush the shorthand **auth_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. The caller may wish to try the original **AUTH_UNIX** style of credentials.

Record Marking Standard

Record marking (RM) is the process of delimiting one message from another when RPC messages pass on top of a byte stream protocol (like TCP/IP). RM helps detect and possibly recover from user protocol errors. This RM/TCP/IP transport passes RPC messages on TCP streams. One RPC message fits into one RM record.

A record contains one or more record fragments. A record fragment is a 4-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 two values:

- A boolean indicating whether the fragment is the last fragment of the record (bit value 1 implies the fragment is the last fragment).
- A 31-bit unsigned binary value that 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. (Note that this record specification is not in XDR standard form.)

Portmapper Program Protocol

The portmapper program maps RPC program and version numbers to UDP/IP or TCP/IP port numbers. This program makes dynamic binding of remote programs possible.

This binding is desirable because the range of reserved port numbers is very small and the number of potential remote programs is very large. By running only the portmapper on a reserved port, the program can ascertain the port numbers of other remote programs by querying the portmapper.

RPC Protocol

The XDR description language specifies the portmapper RPC protocol.

```
Port Mapper RPC Program Number: 100000
     Version Number: 2
     Supported Transports:
            UDP/IP on port 111
            RM/TCP/IP on port 111
```

RPC Procedures

The following subsections describe the RPC procedures of the portmapper.

Function Procedure Version	Remote Procedure
Do Nothing Procedure 0 Version 2	`0. PMAPPROC_NULL () returns ()` This procedure performs *no* work. By convention, procedure zero of any protocol takes no parameters and returns no results.

Set a Mapping
 	
Procedure 1
Version 2
	`1. PMAPPROC_SET (prog,vers,prot,port) returns (resp)`
	`unsigned prog;`
	`unsigned vers;`
	`unsigned prot;`
	`unsigned port;`
	`boolean resp;`
	
	■ When a program is first available on a node, it registers with the portmapper program on the same node.
	
	■ 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 **resp**, whose value is **TRUE** if the procedure successfully established the mapping or **FALSE** if it did not.
	
	■ The procedure refuses to establish a mapping if one already exists for the tuple *[prog,vers,prot]*.

Unset a Mapping
 	
Procedure 2
Version 2
	`2. PMAPPROC_UNSET (prog,vers,dummy1,dummy2) returns (resp)`
	`unsigned prog;`
	`unsigned vers;`
	`unsigned dummy1; /* value always ignored */`
	`unsigned dummy2; /* value always ignored */`
	`boolean resp;`
	
	When a program becomes unavailable, it should unregister with the portmapper program on the same node. The parameters and results have meanings identical to those of **PMAPPROC_SET**.

Function Procedure Version	Remote Procedure
Look Up Mapping Procedure 3 Version 2	3. PMAPPROC_GETPORT (prog,vers,prot,dummy) returns (port) unsigned prog; unsigned vers; unsigned prot; unsigned dummy;/* this value always ignored */ unsigned port;/* zero means program not registered */ 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. A port value of zero means the program is not registered.

Dumping Mappings
 	
Procedure 4 Version 2
	4. PMAPPROC_DUMP () returns (maplist)
	struct maplist {
	union switch (boolean) {
	FALSE: struct { /* void, end of list */ };
	TRUE: struct {
	unsigned prog;
	unsigned vers;
	unsigned prot;
	unsigned port;
	struct maplist the_rest;
	};
	};
	} maplist;
	
	This procedure enumerates all entries in the portmapper's database. It takes no parameters and returns a list of program, version, protocol, and port values.

Function Procedure Version	Remote Procedure
Indirect Call Routine Procedure 5 Version 2	5. PMAPPROC_CALLIT (prog,vers,proc,args) returns (port,res) unsigned prog; unsigned vers; unsigned proc; string args<8K>; unsigned port; string res<8K>; This procedure allows a caller to call another remote procedure on the same node without knowing the remote procedure's port number. Its supports broadcasts to arbitrary remote programs via the well-known portmapper's port. Note: This procedure only sends a response if the procedure was successfully executed and is silent (no response) otherwise.

7

NIS Protocol Specification

The NIS (Network Information Service) distributed lookup service is a network service providing read access to replicated databases. The client interface uses the RPC (Remote Procedure Call) mechanism to access the NIS database servers.

The NIS operates on an arbitrary number of map databases. **Map** names provide the lower of two levels of a naming hierarchy. Maps are grouped into named sets called **NIS domains.** NIS domain names provide the second, higher level of naming. Map names must be unique within a domain, but may be duplicated in different NIS domains. The NIS client interface requires both a map name and an NIS domain name to access the NIS information.

The NIS achieves high availability by replication. Global consistency among the replicated database copies should be addressed, though it is not covered by the protocol. Every implementation should yield the same result at steady state when a request is made of any NIS database server. Update and update-propagation mechanisms must be implemented to supply the required degree of consistency.

Map Operations

Translating or mapping a name to its **value** is a very common operation performed in computer systems. Common examples include translating the following:

- A variable name to a virtual memory address.
- A user name to a system ID or list of capabilities.
- A network host name to an internet address.

You can perform two fundamental read-only operations on a map: match and enumerate. **Match** means to look up a name (a **key**) and return its current value. **Enumerate** means to return each key-value pair, one at a time.

The NIS supplies match and enumerate operations in a network environment. It provides availability and reliability by replicating both databases and database servers on multiple nodes within a single network. The database is replicated, but not distributed; all changes are made at a single server and eventually propagate to the remaining servers without locking.

Remote Procedure Call (RPC)

The RPC (Remote Procedure Call) mechanism defines a paradigm for interprocess communication modeled on function calls. Clients call functions that optionally return values. All inputs and outputs to the functions are in the client's address space. A server program executes the function.

Using RPC, clients address servers by a program number (to identify the application level protocol that the server speaks) and a version number. Additionally, each server procedure has a procedure number assigned to it.

In an internet environment, clients must also know the server's host internet address and the server's port number. The server listens for service requests at ports associated with a particular transport protocol: TCP/IP or UDP/IP.

The header files (included when the client interface functions are compiled) typically define the format of the data structures used as inputs to and outputs from the remotely executed procedures. Levels above the client interface package need not know specifics about the RPC interface to the server.

External Data Representation (XDR)

The XDR (eXternal Data Representation) specification establishes standard representations for basic data types (e.g., strings, signed and unsigned integers, structures, and unions) in a way that allows them to be transferred among nodes with varying architectures. XDR provides primitives to encode and decode basic data types. Constructor primitives allow arbitrarily complex data types to be made from the basic types.

The NIS uses XDR's data description language to describe RPC input and output data structures. Generally, the data description language looks like the C language with a few extra constructs. One such extra construct is the discriminated union. This construct is like a C language union in that it can hold various objects; it differs in that it indicates which object it currently holds. The discriminant is the first item across the network.

EXAMPLE:

```
union switch (long int) {
     1: string exmpl_name<16>
     0: unsigned int exmpl_error_code
     default: struct {}
}
```

The first object (the discriminant) encoded or decoded is a long integer. If it has the value one, the next object is a string. If the discriminant has the value zero, the next object is an unsigned integer. If the discriminant takes any other value, do not encode or decode any more data. The **string** data type in the XDR data definition language adds the ability to specify the maximum number of elements in a byte array or string of potentially variable size. For example

```
string domain<YPMAXDOMAIN>;
```

states that the byte sequence **domain** can be less than or equal to **YPMAXDOMAIN** bytes long.

An additional primitive data type is a **boolean** that takes the value one to mean **TRUE** and zero to mean **FALSE**.

NIS Database Servers

Maps and Map Operations

Map Structure

Maps are named sets of key-value pairs. Keys and their values are counted binary objects and may be ASCII information. The client applications that retrieve data from a map interpret the data comprising the map. The NIS has neither syntactic nor semantic knowledge of the map contents. Neither does the NIS determine or know any map's name. The NIS clients manage the map names. An administrator outside the NIS system should resolve conflicts in the map name space.

NIS maps are typically implemented as files or databases in a database management system. The design of the NIS map database is an implementation detail that the protocol does not specify.

Match Operation

The NIS supports an exact match operation in the `YPPROC_MATCH` procedure. If a match string and a key in the map are exactly the same, the value of the key is returned. The NIS does not support pattern matching, case conversion, or wild carding.

Map Entry Enumeration

You can obtain the first key-value pair in a map with `YPPROC_FIRST` and the next key-value pair with `YPPROC_NEXT`. To retrieve each entry once, call `YPPROC_FIRST` once and `YPPROC_NEXT` repeatedly until the return value indicates there are no more entries in the map. Making the same calls on the same map at the same NIS database server enumerates all entries in the same order. The actual order, however, is unspecified. Enumerating a map at a different NIS database server does not necessarily return entries in the same order.

Entire Map Retrieval

The YPPROC_ALL operation retrieves all key-value pairs in a map with a single RPC request. This operation is faster than map entry enumeration and it is more reliable since it uses TCP. Ordering is the same as when enumeration is applied.

Map Update

Updating the contents of an NIS map is an implementation detail that is outside the NIS service specification.

Master and Slave NIS Database Servers

Each map has one NIS database server called the map's **master.** Map updates occur only on the NIS master server. An updated map should transfer from the master to the rest of the NIS database servers (**slave** servers).

Each map may have a different NIS database server as its master, all maps may have the same master, or any other combination may exist. Implementation and administrative policy determine how to configure the map masters.

Map Propagation and Consistency

Map propagation is the process of copying map updates from the master to the slaves. The protocol does not specify technology or algorithms for map propagation. Map propagation may be entirely manual. For example, you can copy the maps from the master to the slaves at a regular interval or when a change is made on the master.

To escape from the idiosyncrasies of any particular implementation, all maps should be uniformly timestamped.

Functions to Aid in Map Propagation

The NIS protocol does not specify the way a map transfers from one server to another. One possibility is to transfer them manually. Another is for the NIS database server to activate another process to perform the map transfer. A third alternative is for a server to enumerate a recent version of the map using the normal client map enumeration functions.

The `YPPROC_XFR` procedure requests the NIS server to update a map and permits the actual transfer agent (a server process) to call back the requestor with a summary status.

NIS Domains

NIS domains provide a second level for naming within the NIS subsystem. Since they are names for sets of maps, you should create separate map name spaces. NIS domains provide an opportunity to divide large organizations into administrable portions and the ability to create parallel, non-interfering test and production environments.

Ideally, the NIS domain of interest to a client is associated with the invoking user. However, it is useful for client nodes to be in a default NIS domain. Implementations of the NIS client interface should supply some mechanism for telling processes the NIS domain name they should use. This mechanism is necessary because of the following:

- The NIS domain concept is not essential to most applications.
- It allows you to write programs that are insensitive to both location and the invoking user.

NIS Non-features

The following capabilities are not included in the current NIS protocols.

Map Update within the NIS

Direct modification to a NIS map is outside the NIS subsystem.

Version Commitment Across Multiple Requests

The NIS protocol keeps the NIS database server stateless with regard to its clients. Therefore, you do not have a facility for requesting a server to pre-allocate any resource beyond that required to service any single request. You do not have a way to commit a server to use a single version of a map while trying to enumerate that map's entries. Using `YPPROC_ALL` should help you avoid problems.

Guaranteed Global Consistency

No facility exists for locking maps during the update or propagation phases; therefore, map databases will probably be globally inconsistent during these phases. The set of client applications for which the NIS is an appropriate lookup service must be tolerant of transient inconsistencies.

Access Control

The NIS database servers do not attempt to restrict access to the map data. They will service all syntactically correct requests.

NIS Database Server Protocol Definition

This section describes the protocol version 2.

RPC Constants

All numbers are in decimal.

NIS RPC Constant	Description
YPPROG 100004	NIS database server protocol program number
YPVERS 2	Current NIS protocol version

Other Manifest Constants

All numbers are in decimal.

NIS Constants	Description
YPMAXRECORD 1024	The total maximum size of key and value for any pair The absolute sizes of the key and value may divide this maximum arbitrarily
YPMAXDOMAIN 64	The maximum number of characters in a NIS domain name
YPMAXMAP 64	The maximum number of characters in a map name
YPMAXPEER 64	The maximum number of characters in a NIS host name

Remote Procedure Return Values

This section presents the return status values returned by several of the NIS remote procedures. All numbers are in decimal.

Remote Procedures	Return Status Values
ypstat	typedef enum { YP_TRUE = 1, /* General purpose success code. */ YP_NOMORE = 2, /* No more entries in map. */ YP_FALSE = 0, /* General purpose failure code.*/ YP_NOMAP = -1, /* No such map in domain. */ YP_NODOM = -2, /* Domain not supported. */ YP_NOKEY = -3, /* No such key in map. */ YP_BADOP = -4, /* Invalid operation. */ YP_BADDB = -5, /* Server database is bad. */ YP_YPERR = -6, /* YP server error. */ YP_BADARGS = -7, /* Request arguments bad. */ YP_VERS = -8 /* YP server version mismatch.*/ } ypstat

Remote Procedures	Return Status Values
ypxfrstat	typedef enum { YPXFR_SUCC = 1, /* Success */ YPXFR_AGE = 2, /* Master's version not newer */ YPXFR_NOMAP = -1, /* Cannot find server for map */ YPXFR_NODOM = -2, /* Domain not supported */ YPXFR_RSRC = -3, /* Local resource alloc failure */ YPXFR_RPC = -4, /* RPC failure talking to server */ YPXFR_MADDR = -5, /* Cannot get master address */ YPXFR_YPERR = -6, /* YP server/map db error */ YPXFR_BADARGS= -7, /* Request arguments bad */ YPXFR_DBM = -8, /* Local database failure */ YPXFR_FILE = -9, /* Local file I/O failure */ YPXFR_SKEW = -10, /* Map version skew in transfer */ YPXFR_CLEAR = -11, /* Cannot clear local ypserv */ YPXFR_FORCE = -12, /* Must override defaults */ YPXFR_XFRERR = -13, /* ypxfr error */ YPXFR_REFUSED= -14 /* ypserv refused transfer */ } ypxfrstat

Basic Data Structures

This section defines the data structures used as inputs to and outputs from the NIS remote procedures.

Data Structure	Definition
domainname	typedef string domainname<YPMAXDOMAIN>
keydat	typedef string keydat<YPMAXRECORD>
mapname	typedef string mapname<YPMAXMAP>
peername	typedef string peername<YPMAXPEER>
valdat	typedef string valdat<YPMAXRECORD>

ypmaplist	`typedef struct {`
	`  mapname`
	`  ypmaplist *`
	`} ypmaplist`

ypmap_parms	`typedef struct {`
	`  domainname`
	`  mapname`
	`  unsigned long ordernum`
	`  peername`
	`} ypmap_parms`
	This structure contains parameters giving information about map *mapname* within NIS domain *domainname.*
	The *peername* element is the name of the map's NIS master database server.
	If any of the three strings is null, the information is unknown or unavailable.
	The *ordernum* element contains a binary value representing the map's creations time (order number); if unavailable, this number is zero.

Data Structure	Definition
ypreq_key	typedef struct { domainname mapname keydat } ypreq_key
ypreq_nokey	typedef struct { domainname mapname } ypreq_nokey
ypreq_xfr	typedef struct { struct ypmap_parms map_parms unsigned long transid unsigned long prog unsigned short port } ypreq_xfr
ypresp_all	typedef union switch (boolean more) { TRUE: ypresp_key_val FALSE: struct { } } ypresp_all
ypresp_key_val	typedef struct { ypstat keydat valdat } ypresp_key_val
ypresp_maplist	typedef struct { ypstat ypmaplist * } ypresp_maplist
ypresp_master	typedef struct { ypstat peername } ypresp_master

Data Structure	Definition
ypresp_order	typedef struct { ypstat unsigned long ordernum } ypresp_order
ypresp_val	typedef struct { ypstat valdat } ypresp_val
ypresp_xfr	typedef struct { unsigned long transid ypxfrstat xfrstat } ypresp_xfr

NIS Database Server Remote Procedures

This section contains a specification for each function you can call as a remote procedure. The XDR data definition language describes the input and output parameters.

Function Procedure Version	Remote Procedure
Do Nothing Procedure 0 Version 2	`0. YPPROC_NULL ( ) returns ( )` This procedure takes no arguments, does no work, and returns nothing. It is made available in all RPC services to allow server response testing and timing.

Do You Serve This Domain?
 	
Procedure 1
Version 2
	`1. YPPROC_DOMAIN (domain) returns (serves)`
	`domainname domain;`
	`boolean serves;`
	
	This procedure returns *TRUE* if the server serves *domain* or *FALSE* if it does not.
	
	This procedure allows a potential client to determine if a given server supports a certain NIS domain.

Answer Only If You Serve This Domain
 	
Procedure 2
Version 2
	`2. YPPROC_DOMAIN_NONACK (domain) returns (serves)`
	`domainname domain;`
	`boolean serves;`
	
	This procedure returns *TRUE* if the server serves *domain;* otherwise, it does not return.
	
	This function is useful in a broadcast environment when you want to restrict the number of useless messages.
	
	If you call this function, the client interface implementation must regain control in the negative case (e.g., by means of a timeout on the response).
	
	Note: The current implementation returns in the *FALSE* case by forcing an RPC decode error.

Function Procedure Version	Remote Procedure
Return Value of a Key Procedure 3 Version 2	3. YPPROC_MATCH (req) returns (resp) ypreq_key req; ypresp_val resp; This procedure returns the value associated with the datum *keydat* in *req.* If *resp.stat* has the value *YP_TRUE,* the value data are returned in the datum *valdat.*
Get First Key-Value Pair in Map Procedure 4 Version 2	4. YPPROC_FIRST (req) returns (resp) ypreq_nokey req; ypresp_key_val resp; If *resp.stat* has the value *YP_TRUE,* this procedure returns the first key-value pair from the map named in *req* to the *keydat* and *valdat* elements within *resp.* When status contains the value *YP_NOMORE,* the map is empty.
Get Next Key-Value Pair in Map Procedure 5 Version 2	5. YPPROC_NEXT (req) returns (resp) ypreq_key req; ypresp_key_val resp; If *resp.stat* has the value *YP_TRUE* , this procedure returns the key-value pair following the key-value named in *req* to the *keydat* and *valdat* elements within *resp.* If the passed key is the last key in the map, the value of *resp.stat* is *YP_NOMORE.*

Function Procedure Version	Remote Procedure

Transfer Map
 	
Procedure 6
Version 2
	`6. YPPROC_XFR (req) returns (resp)`
	`ypreq_xfr req;`
	`ypresp_xfr resp;`
	
	The NIS protocol specification does not declare what action is taken in response to this request. The action is implementation dependent. Use this procedure for the following:
	
	■ To indicate to the server that a map should be updated
	■ To allow the actual transfer agent (whether it be the NIS server process, or some other process)to call back the requestor with a summary status.
	
	The transfer agent should call back the program running on the requesting host with program number *req.prog*, program version 1, and listening at port *req.port*.
	
	The procedure number is 1, and the callback data is of type *ypresp_xfr*.
	
	The *transid* field should turn around *req.transid*, and the *xfrstat* field should be set appropriately.

Re-initialize Internal State Procedure 7 Version 2	`7. YPPROC_CLEAR ( ) returns ( )` The NIS protocol specification does not declare what action is taken in response to this request. The action is implementation dependent. Different server implementations may have different amounts of internal state (e.g., open files or the current map). This request signals that all such state information should be erased.

Function Procedure Version	Remote Procedure
Get All Key-Value Pairs in Map Procedure 8 Version 2	8. YPPROC_ALL (req) returns (resp) ypreq_nokey req; ypresp_all resp; This procedure transfers all key-value pairs from a map with a single RPC request. When the union's discriminant is *FALSE,* no more key-value pairs are returned. The status field of the last *ypresp_key_val* structure should be examined to determine why the flow of returned key-value pairs stopped.
Get Map Master Name Procedure 9 Version 2	9. YPPROC_MASTER (req) returns (resp) ypreq_nokey req; ypresp_master resp; This procedure returns the NIS master server's name inside the *resp* structure.
Get Map Order Number Procedure 10 Version 2	10. YPPROC_ORDER (req) returns (resp) ypreq_nokey req; ypresp_order resp; This procedure returns a map's order number as an unsigned long integer to indicate when the map was built. This quantity represents the number of seconds since 00:00:00 January 1, 1970, GMT.
Get All Maps in Domain Procedure 11 Version 2	11. YPPROC_MAPLIST (req) returns (resp) domainname req; ypresp_maplist resp; This procedure returns a list of all the maps in a NIS domain.

NIS Binders

For any network service to work, potential clients must be able to find the servers. This section describes the NIS binder, an optional element in the NIS subsystem that supplies NIS database server addressing information to potential NIS clients.

To address an NIS server in the Internet environment, a client must know the following:

- The server's internet address.
- The port at which the server is listening for service requests.

This addressing information is sufficient to bind the client to the server. One way to provide the addressing information is to allocate one entity on each NIS client to keep track of the NIS servers and provide that information to potential NIS clients on request. An NIS binder is useful under these conditions:

- It is easier for a client to find the NIS binder than to find a NIS database server.
- The NIS binder can find a NIS database server.

Assume the following statements about NIS binders to be true:

- An NIS binder should be present at every network node. This makes addressing the NIS binder easier than addressing an NIS database server. The scheme for finding a local resource is implementation specific. However, given that a resource is guaranteed to be local, there may be an efficient way of finding it.
- The NIS binder should be able to find an NIS database server. The means of doing so, however, may be complicated and consume time and resources.

If either of these assumptions is incorrect, your implementation of NIS binders is probably not a good solution for an NIS binder.

If an NIS binder is implemented, it can provide added value beyond the binding. For example, it can verify that the binding is correct and that the NIS database server is working. The degree of certainty in a binding that the NIS binder gives to a client is a parameter that can be configured appropriately in the implementation.

NIS Binder Protocol Definition

This section describes version 2 of the protocol.

RPC Constants

All numbers are decimal.

RPC Constant	Description
YPBINDPROG 100007	NIS binder protocol program number
YPBINDVERS 2	Current NIS binder protocol version

Other Manifest Constants

All numbers are decimal.

RPC Constant	Description
YPMAXDOMAIN 64	The maximum number of characters in an NIS domain name This constant is identical to the constant defined above in the "NIS Database Server Protocol" section.

ypbind_resptype	`enum ypbind_resptype {`
	`  YPBIND_SUCC_VAL = 1,`
	`  YPBIND_FAIL_VAL = 2`
	`}`
	
	This constant discriminates between success responses and failure responses to a YPBINDPROC_DOMAIN request.

ypbinderr	`typedef enum {`
	`  YPBIND_ERR_ERR = 1,  /* Internal error */`
	`  YPBIND_ERR_NOSERV = 2, /* No bound server */`
	`  YPBIND_ERR_RESC = 3  /* Can't allocate resource */`
	`} ypbinderr`
	
	The error case of most interest to an NIS binder client is YPBIND_ERR_NOSERV. This error means the binding request cannot be satisfied because the NIS binder does not know how to address any NIS database server in the named NIS domain.

Basic Data Structures

This section defines the data structures used as inputs to and outputs from the NIS binder remote procedures.

Remote Procedures	Return Status Values
domainname	`typedef string domainname <YPMAXDOMAIN>` This structure is identical to the domainname string defined above in the "NIS Database Server Protocol" section.

ypbind_binding	`typedef struct {`
	`  unsigned long ypbind_binding_addr`
	`  unsigned short ypbind_binding_port`
	`} ypbind_binding`
	This structure contains the information necessary to bind a client to an NIS database server in the Internet environment.
	The element **ypbind_binding_addr** holds the host IP address (4 bytes), and **ypbind_binding_port** holds the port address (2 bytes).
	Both IP address and port address must be in ARPA network byte order (most significant byte first), regardless of the host node's native architecture.

ypbind_resp	`typedef struct {`
	`  union switch (enum ypbind_resptype status) {`
	`  YPBIND_SUCC_VAL:`
	`  ypbind_binding`
	`  YPBIND_FAIL_VAL:`
	`  ypbinderr`
	`  default:`
	`  { }`
	`  }`
	`} ypbind_resp`
	This structure is the response to a YPBINDPROC_DOMAIN request.

Remote Procedures	Return Status Values

ypbind_setdom	`typedef struct {`
	`  domainname`
	`  ypbind_binding`
	`  version`
	`} ypbind_setdom`
	This structure is the input data structure for the *YPBINDPROC_SETDOM* procedure.

NIS Binder Remote Procedures

The XDR data definition language describes the NIS binder remote procedures.

Function Procedure Version	Binder Remote Procedure
Do Nothing Procedure 0 Version 2	`0. YPBINDPROC_NULL ( ) returns ( )` This procedure does no work. It is made available in all RPC services to allow server response testing and timing.

Get Current Binding for a Domain
Procedure 1 Version 2
	`1. YPBINDPROC_DOMAIN (domain) returns (resp)`
	`  domainname domain;`
	`  ypbind_resp resp;`
	This procedure returns the binding information necessary to address an NIS database server within the Internet environment.

Set Domain Binding
Procedure 2 Version 2
	`2. YPBINDPROC_SETDOM (setdom) returns ( )`
	`  ypbind_setdom setdom;`
	This procedure instructs an NIS binder to set its current binding using the passed information. This provides a way to override the process the NIS binder usually uses to bind to an NIS server.

Index

N

O

P

R

S

T

U

V

X

Y

SALES OFFICES
Arranged alphabetically by country

Please send directory corrections to:
Test & Measurement Catalog
Hewlett-Packard Company
3200 Hillview Avenue
Palo Alto, CA 94304
Tel: (415) 857-4706
Fax: (415) 857-3880

HEADQUARTERS OFFICES

If there is no sales office listed for your area, contact one of these headquarters offices.

ASIA

Hewlett-Packard Asia Ltd.
22/F Bond Centre, West Tower
89 Queensway, Central
HONG KONG
G.P.O. Box 863, Hong Kong
Tel: 5-8487777
Telex: 76793 HPA HX
Cable: HPASIAL TD

CANADA

Hewlett-Packard (Canada) Ltd.
6877 Goreway Drive
MISSISSAUGA, Ontario L4V 1M8
Tel: (416) 678-9430
Fax: (416) 678-9421

EASTERN EUROPE

Hewlett-Packard Ges.m.b.h.
Lieblgasse 1
P.O. Box 72
A-1222 **VIENNA,** Austria
Tel: (222) 2500
Telex: 1 3 4425 HEPA A

NORTHERN EUROPE

Hewlett-Packard S.A.
V. D. Hooplaan 241
P.O. Box 999
NL-118 LN 15 **AMSTELVEEN**
The Netherlands
Tel: 20 5479999
Telex: 18919 hpner

SOUTH EAST EUROPE

Hewlett-Packard S.A.
World Trade Center
110 Avenue Louis-Casai
1215 Cointrin, **GENEVA,** Switzerland
Tel: (022) 98 96 51
Telex: 27225 hpser
Mail Address:
P.O. Box
CH-1217 Meyrin 1
GENEVA
Switzerland

MIDDLE EAST AND CENTRAL AFRICA

Hewlett-Packard S.A.
International Sales Branch
Middle East/Africa
7, rue du Bois-du-Lan
P.O. Box 364
CH-1217 Meyrin 1
GENEVA
Switzerland
Tel: (41/22) 780 7111
Fax: 783 7535

European Operations
Hewlett-Packard S.A.
150, Route du Nant d'Avril
1217 Meyrin 2
GENEVA, Switzerland
Tel: (41/22) 780 .8111
Fax: (41/22) 780 .8542

UNITED KINGDOM

Hewlett-Packard Ltd.
Nine Mile Ride
WOKINGHAM
Berkshire, RG113LL
Tel: 0344 773100
Fax: (44/344) 763526

UNITED STATES OF AMERICA

Customer Information Center
(800) 752-0900
6:00 AM to 5:00 PM Pacific Time

EASTERN USA

Hewlett-Packard Co.
4 Choke Cherry Road
ROCKVILLE, MD 20850
Tel: (301) 670-4300

MIDWESTERN USA

Hewlett-Packard Co.
5201 Tollview Drive
ROLLING MEADOWS, IL 60008
Tel: (312) 255-9800

SOUTHERN USA

Hewlett-Packard Co.
2015 South Park Place
ATLANTA, GA 30339
Tel: (404) 955-1500

WESTERN USA

Hewlett-Packard Co.
5161 Lankershim Blvd.
NORTH HOLLYWOOD, CA 91601
Tel: (818) 505-5600

OTHER INTERNATIONAL AREAS

Hewlett-Packard Co.
Intercontinental Headquarters
3495 Deer Creek Road
PALO ALTO, CA 94304
Tel: (415) 857-5027
Telex: 034-8300
Cable: HEWPACK

Hewlett-Packard Trading S.A.
Bureau de Liaison/Bureau de Spport
Villa des Lions
9, Hai Galloul
DZ-BORDJ EL BAHRI
Tel: 76 02 07
Fax: 281 0387

ANGOLA

Telectra Angola LDA
Empresa Técnica de Equipamentos
16 rue Cons. Julio de Vilhema
LUANDA
Tel: 35515,35516
Telex: 3134

ARGENTINA

Hewlett-Packard Argentina S.A.
Montaneses 2140/50
1428 **BUENOS AIRES**
Tel: (54/1) 781 4059
(54/1) 781-4090

Laboratorio Rodriguez
Corswant S.R.L.
Misiones, 1156 - 1876
Bernal, Oeste
BUENOS AIRES
Tel: 252-3958, 252-4991

Argentina Esanco S.R.L.
A/ASCO 2328
1416 **BUENOS AIRES**
Tel: 541-58-1981, 541-59-2767
Telex: 22796 HEW PAC-AR

AUSTRALIA

Customer Information Centre
Tel: (008) 033821

Adelaide, South Australia Office

Hewlett-Packard Australia Ltd.
PARKSIDE, S.A. 5063
153 Greenhill Road
ADELAIDE (Parkside) Sales
Tel: (61-8-) 272-5911
Fax: (61/8) 373-1398

Brisbane, Queensland Office

Hewlett-Packard Australia Ltd.
10 Payne Road
THE GAP, Queensland 4061
Tel: 61-7-300-4133
Telex: 42133
Cable: HEWPARD Brisbane

Canberra, Australia Capital Territory Office

Hewlett-Packard Australia Ltd.
Thynne Street, Fern Hill Park
BRUCE, A.C.T. 2617
P.O. Box 257,
JAMISON, A.C.T. 2614
Tel: 61-62-51-6999
Telex: 62650
Cable: HEWPARD Canberra

Melbourne, Victoria Office

Hewlett-Packard Australia Ltd.
31-41 Joseph Street
P.O. Box 221
BLACKBURN, Victoria 3130
Tel: (61/3) 895-2895
Fax: (61/3) 898-7831
Cable: HEWPARD Melbourne

Perth, Western Australia Office

Hewlett-Packard Australia Ltd.
Herdsman Business Park
Cnr. Hasler & Gould Strs.
Osborne Park
CLAREMONT, W.A. 6017
Tel: (61/9) 242 1414
Fax: (61/9) 242-1682
Cable: HEWPARD Perth

Sydney, New South Wales Office

Hewlett-Packard Australia Ltd.
17-23 Talavera Road
P.O. Box 308
NORTH RYDE, N.S.W. 2113
Sydney, Australia
Tel: (61/2) 888-4444
Fax: (61/2) 888-9072
Cable: HEWPARD Sydney

AUSTRIA

Hewlett-Packard GmbH
Verkaufsbuero Graz
Grottenhofstrasse 94
A-8052 **GRAZ**
Tel: 43-316-291-5660
Telex: 312375

Hewlett-Packard GmbH
Lieblgasse 1
P.O. Box 72
A-1222 **VIENNA**
Tel: (43/222) 2500
Fax: (48/222) 2500 Ext 444

BAHRAIN

Modern Electronic Establishment
Hewlett-Packard Division
P.O. Box 22015
RIYADH 11495
SAUDI ARABIA
Tel: (966/1) 4763030
Telex: 595 (0495) 202049

Wael Pharmacy
P.O. Box 648
MANAMA
Tel: 256123
Telex: 8550 WAEL BN

BELGIUM

Hewlett-Packard Belgium S.A./N.V.
Blvd de la Woluwe, 100
Woluwedal
1200 **BRUSSELS**
Tel: (32/2) 761.31.11
Fax: (32/2) 763.06.13

BENIN

S.I.T.E.L.
Immeuble le General
Av. General de Gaulle
P.O. Box 161
ABIDJAN 01
Ivory Coast
Tel: 32 12 27
Telex: 22149

BERMUDA

Applied Computer Technologies
Atlantic House Building
P.O. Box HM 2091
Par-La-Ville Road
HAMILTON 5
Tel: 295-1616
Telex: 380 3589/ACT BA

BOLIVIA

Arrellano Ltda
Av. 20 de Octubre #2125
Casilla 1383
LA PAZ
Tel: 368541

Siser Ltda. (Sistemas de Importacion y Servicios Ltda.)
Gabriel Gozalvez 221
Casilla 4084
LA PAZ
Tel: (591/2) 340962/
363365/343245
Fax: 35-9268

BRAZIL

Hewlett-Packard do Brasil S.A.
Praia de Botafago 228-A-614
6. AND.-CONJ. 601
Edificio Argentina - Ala A
22250 **RIO DE JANEIRO, RJ**
Tel: (55/21) 552-6422
Telex: 21905 HPBR BR
Cable: HEWPACK Rio de Janeiro

BRUNEI

Komputer Wisman Sdn Bhd
G6, Chandrawaseh Cmplx,
Jalan Tutong
P.O. Box 1297,
BANDAR SERI BEGAWAN
NEGARA BRUNI DARUSSALAM
Tel: 673-2-2000-70/26711

BURKINA FASSO

S.I.T.E.L.
Immeuble le General
Avenue General de Gaulle
P.O. Box 161
ABIDJAN 01
Ivory Coast
Tel: 32 12 27
Telex: 22149

CAMEROON

R.T.I.
175 Rue Blomet
75015 **PARIS**
France
Tel: (1) 45 31 0906
Telex: 203376
Fax: (1) 45 31 09 18

CANADA

Alberta

Hewlett-Packard (Canada) Ltd.
3030 3rd Avenue N.E.
CALGARY, Alberta T2A 6T7
Tel: (1/403) 235-3100
Fax: (1/403) 272-2299

Hewlett-Packard (Canada) Ltd.
11120-178th Street
EDMONTON, Alberta T5S 1P2
Tel: (1/403) 486-6666
Fax: (1/403) 489-8764

SALES OFFICES
Arranged alphabetically by country (cont'd)

CANADA (Cont'd)

British Columbia
Hewlett-Packard (Canada) Ltd.
10691 Shellbridge Way
RICHMOND,
British Columbia V6X 2W8
Tel: (1/604) 270-2277
Fax: (1/604) 270-0859

Manitoba
Hewlett-Packard (Canada) Ltd.
1825 Inkster Blvd.
WINNIPEG, Manitoba R2X 1R3
Tel: (204) 694-2777

New Brunswick
Hewlett-Packard (Canada) Ltd.
814 Main Street
MONCTON, New Brunswick E1C 1E6
Tel: (506) 855-2841

Nova Scotia
Hewlett-Packard (Canada) Ltd.
201 Brownlow Avenue
DARTMOUTH, Nova Scotia B3B 1W2
Tel: (1/902) 469-7820
Fax: (1/902) 468-2817

Hewlett-Packard
(Canada) Ltd.
475 Hood Rd., Unit #2
MARKHAM, L3R 8H1
Tel: (416) 479-1770

Ontario
Hewlett-Packard (Canada) Ltd.
552 Newbold Street
LONDON, Ontario N6E 2S5
Tel: (1/519) 686-9181
Fax: (1/519) 686-9145

Hewlett-Packard (Canada) Ltd.
6877 Goreway Drive
MISSISSAUGA, Ontario L4V 1M8
Tel: (1/416) 678-9430
Fax: (1/416) 673-7253

Hewlett-Packard (Canada) Ltd.
2670 Queensview Dr.
OTTAWA, Ontario K2B 8K1
Tel: (1/613) 820-6483
Fax: (1/613) 820-0377

Hewlett-Packard (Canada) Ltd.
3790 Victoria Park Ave.
WILLOWDALE, Ontario M2H 3H7
Tel: (1/416) 499-2550

Quebec
Hewlett-Packard (Canada) Ltd.
17500 Trans Canada Highway
South Service Road
KIRKLAND, Quebec H9J 2X8
Tel: (1/514) 697-4232
Fax: (1/514) 697-6941

Saskatchewan
Hewlett-Packard (Canada) Ltd.
#1, 2175 Airport Rd.
SASKATOON, Saskatchewan S7L 7E1
Tel: (306) 242-3702

CHILE

Ricardo Borzutsky
Avanzados Sistemas de
Conocimientos S. A.
Austria 2041
SANTIAGO
America del Sur
Tel: (562) 223-5946/6148

CHINA, People's Republic of

China Hewlett-Packard Co., Ltd.
22/F Bond Centre, West Tower
89 Queensway, Central
HONG KONG
Tel: (852/5) 8487777
Fax: (852/5) 868 4997

China Hewlett-Packard Co., Ltd.
P.O. Box 9610, Beijing
38 Bei San Huan X1 Road
Shuang Yu Shu, Hai Dian District
BEIJING
Tel: 256-6888
Fax: 256-3207

China Hewlett-Packard Co., Ltd.
23/F Shanghai Union Building
100 Yan An Road
SHANG-HAI
Tel: 203-240
Fax: 202-149

COLOMBIA

Instrumentación
H. A. Langebaek & Kier S.A.
Carrerra 4A No. 52A-26
Apartado Aereo 6287
BOGOTA 1, D.E.
Tel: 212-1466
Telex: 44400 INST CO
Cable: AARIS Bogota

CONGO

Sema-Metra
16-18 Rue Barbes
92126 Montrouge Cedex
FRANCE
Tel: (1) 657/300
Telex: 200601 Semetra
Fax: (1) 46 56 96 53

COSTA RICA

Cientifica Costarricense S.A.
Avenida 2, Calle 5
San Pedro de Montes de Oca
Apartado 10159
SAN JOSÉ
Tel: 9-011-506-243-820
Telex: 3032367 GALGUR CR

Continex S.A.
Avenida 10C
Apartado 746-1000
34-36 **SAN JOSE**
Tel: (506) 33-0933
Telex: 2310 Continex CR
Fax: 21-6905

O. Fischel R. Y. Cia. S.A.
Apartados 434-10174
SAN JOSE
Tel: 23-72-44
Telex: 2379
Cable: OFIR

CYPRUS

Hellamco - M. Cotoyannis
2, Sikelianou St. & Kifissias Av.
P.O. Box 65074
N. Psyhiko 15410
ATHENS
Greece
Tel: 647 79 426, 647 79 427
Telex: 224903

Telerexa Ltd.
P.O. Box 1152
Valentine House
8 Stassandrou St.
NICOSIA
Tel: 45 628, 62 698
Telex: 5845 tlrx cy

DEMSTAR Ltd.
P.O. Box 2260
NICOSIA
Cyprus
Tel: 44 10 64
Telex: 3085
Fax: 46 46 35

DENMARK

Hewlett-Packard A/S
Kongevejen 25
3460 **BIRKEROD**
Tel: (45/42) 816640
Fax: (45/42) 815810

Hewlett-Packard A/S
Rolighedsvej 32
DK-8240 **RISSKOV,** Aarhus
Tel: 45-06-17-6000
Telex: 37409 hpas dk

DOMINICAN REPUBLIC

Microprog S.A.
Juan Tomás Mejía y Cotes No. 60
Arroyo Hondo
SANTO DOMINGO
Tel: 565-6268
Telex: 4510 ARENTA DR (RCA)

ECUADOR

CYEDE Cia. Ltda.
Avenida Eloy Alfaro 1749
y Belgica
Casilla 6423 CCI
QUITO
Tel: 9-011-593-2-450975
Telex: 39322548 CYEDE ED

EGYPT

International Engineering Associates
6 El Gamea Street
Agouza
CAIRO
Tel: 71-21-681 348 0904
Telex: 93830 IEA UN
Cable: INTEGASSO

EL SALVADOR

IPESA de El Salvador S.A.
29 Avenida Norte 1223
SAN SALVADOR
Tel: 9-011-503-266-858
Telex: 301 20539 IPESA SAL

ETHIOPIA

R.T.I.
175 rue Blomet
75016 **PARIS**
France
Tel: (1) 45 31 09 06
Telex: 203376
Fax: (1) 45 31 09 18

FINLAND

Hewlett-Packard Finland
Field Oy
Niittylanpolku IO
00620 **HELSINKI**
Tel: (90) 757-1011
Telex: 122022 Field SF

Hewlett-Packard Oy
Piispankalliontie 17
02200 **ESPOO**
Tel: (358/0) 88721
Fax: (358/0) 887 2277

Hewlett-Packard Oy
Väinönkatu 9 C
40100 **JYVÄSKYLÄ**
Tel: (358/41) 21 85 11

Hewlett-Packard Oy
Valtatie 57
90500 **OULU**
Tel: (358/81) 340 144

FRANCE

Hewlett-Packard France
Z.I. Mercure B
Rue Berthelot
13763 Les Milles Cedex
AIX-EN-PROVENCE
Tel: 33-42-59-41-02
Telex: 410770F

Hewlett-Packard France
ZA Kergaradec
Rue Fernand Forest
29239 **GOUEESNOU** (Brest)
Tel: (98) 41-87-90

Hewlett-Packard France
Chemin des Mouilles
Boite Postale 162
69131 **ECULLY** Cedex (Lyon)
Tel: (33) 72-29-32-93
Telex: 310617F

Hewlett-Packard France
Z.I. Mercure B
Rue Berthelot
F-13763 **LES MILLES** Cédex
Aix-en-Provence
Tel: (33/42) 59-41-02
Telex: 410770
Fax: 594872

Hewlett-Packard France
Parc Club des Tanneries
Batiment B4
4, Rue de la Faisanderie
67381/**LINGOLSHEIM** (Strasbourg)
Tel: (88) 76-15-00
Telex: 890141F

Hewlett-Packard France
Parc d'activités Cadéra
Quartier Jean-Mermoz
Avenue du Président JF Kennedy
33700 **MÉRIGNAC** (Bordeaux)
Tel: (33) 56-34-00-84
Telex: 550105F

Hewlett-Packard France
Miniparc-ZIRST
Chemin du Vieux Chêne
38240 **MEYLAN** (Grenoble)
Tel: (76) 90-38-40

Hewlett-Packard France
Ru de l'Hôtellerie
Le Petit Bel Air
44470 **CARQUEFOU** (Nantes)
Tel: 40-30-38-38
Telex: 711085F

Hewlett-Packard France
Parc Tertiaire Héliopolis
Route de Micy
45380 **La CHAPELLE ST MESMIN**
(Orléans)
Tel: 38-43-93-56
Telex: 783 497F

Hewlett-Packard France
Zone Industrielle de Courtaboeuf
1, av. du Canada
91947 **LES ULIS** Cedex (Orsay)
Tel: 69-82-60-60
Telex: 600048F

R.T.I. (Realisations
Telematiques Internationales)
175, rue Blomet
F-75015 **PARIS**
Tel: (33/1) 45310906
Telex: 42/203376

Hewlett-Packard France
Parc d'activités de la Poterie
Rue Louis Kerautel-Botmel
35000 **RENNES**
Tel: 99-51-42-44
Telex: 740912F

Hewlett-Packard France
45, rue des 3 Soeurs
Centre d'Affaires Paris Nord II
F-93420 Villepinte
B.P. 60020
F-95971 **ROISSY**
CHARLES DE GAULLE Cédex
Tel: (33/48) 91-68-00
Telex: 232366
Fax: 632183

Hewlett-Packard France
P.A.T. Lavatine
3, rue Jacques Monod
BP 185
76136 **MONT-ST-AIGNAN** (Rouen)
Tel: 35-59-19-20
Telex: 770035F

Hewlett-Packard France
Innoparc
BP 167 - Voie n°7
31328 **LABEGE** Cedex (T[illegible]ouse)
Tel: 61-39-11-40
Telex: 531639F

Hewlett-Packard France
Les Cardoulines
Batiment B2
Route des Dolines
Parc d'activite de Valbonne
Sophia Antipolis
06560 **VALBONNE** (Nice)
Tel: 93-65-39-40

Hewlett-Packard France
Parc d'activité des Prés
1, Rue Papin Cedex
59658 **VILLENEUVE D'ASCQ**
Tel: 20-91-41-25
Telex: 160124F

FRENCH WEST INDIES (Antilles)

R.T.I.
175, Rue Blomet
75015 **PARIS**
FRANCE
Tel: (1) 45 31 09 06
Telex: 203376
Fax: (1) 45 31 09 18

GABON

R.T.I. Cameroon
Distribution/Services
B.P. 3899
DOULA, CAMEROON
(Please contact R.T.I. France.)
Tel: (237) 423291
Telex: 970/5385

R.T.I.
175, rue Blomet
75015 **PARIS**
France
Tel: (1) 45 31 09 06
Telex: 203376
Fax: (1) 45 31 09 18

GERMAN FEDERAL REPUBLIC

Hewlett-Packard GmbH
Vertriebzentrale Deutschland
Hewlett-Packard-Strasse
Postfach 1641
D-6380 **BAD HOMBURG v.d.H**
Tel: (06172) 400-0
Telex: 410 844 hpbhg

Hewlett-Packard GmbH
Geschäftsstelle
Keithstrasse 2-4
D-1000 **BERLIN** 30
Tel: (030) 21 99 04-0
Telex: 018 3405 hpbln d

Hewlett-Packard GmbH
Verbindungsstelle Bonn
Friedrich-Ebert-Allee 26
5300 **BONN**
Tel: (0228) 234001
Telex: 8869421

Hewlett-Packard GmbH
Vertriebszentrun Südwest
Schickardstrasse 2
D-7030 **BÖBLINGEN**
Postfach 1427
Tel: (49/7031) 645
Fax: (49/7031) 645-429

Hewlett-Packard GmbH
Zeneralbereich Mktg
Herrenberger Strasse 130
D-7030 **BÖBLINGEN**
Tel: (49/7031) 14-0
Fax: (49/7031) 14-2999

Hewlett-Packard GmbH
Geschäftsstelle
Schleefstr. 28a
D-4600 **DORTMUND**-41
Tel: (0231) 45001
Telex: 822858 hepdod

Hewlett-Packard gmbH
Reparaturzentrum Frankfurt
Berner Strasse 117
6000 **FRANKFURT/MAIN 60**
Tel: (069) 500001-0
Telex: 413249 hpffm

Hewlett-Packard GmbH
Vertriebszentrum Nord
Kapstadtring 5
D-2000 **HAMBURG** 60
Tel: 49-40-63-804-0
Telex: 021 63 032 hphh d

Hewlett-Packard GmbH
Geschäftsstelle
Heidering 37-39
D-3000 **HANNOVER** 61
Tel: (49/511) 5706-0
Fax: (49/511) 5706-126

Hewlett-Packard GmbH
Geschäftsstelle
Rosslauer Weg 2-4
D-6800 **MANNHEIM**
Tel: 49-0621-70-05-0
Telex: 0462105 hpmhm

Hewlett-Packard GmbH
Geschäftsstelle
Messerschmittstrasse 7
D-7910 **NEU ULM**
Tel: 49-0731-70-73-0
Telex: 0712816 HP ULM-D

Hewlett-Packard GmbH
Geschäftsstelle
Emmericher Strasse 13
D-8500 **NÜRNBERG** 10
Tel: (0911) 5205-0
Telex: 0623 860 hpnbg

Hewlett-Packard GmbH
Vertriebszentrum Ratingen
Berliner Strasse 111
D-4030 **RATINGEN**
Postfach 31 12
Tel: (02102) 494-0
Telex: 589 070 hprad

Hewlett-Packard GmbH
Vertriebszentrum Muchen
Eschenstrasse 5
D-8028 **TAUFKIRCHEN**
Tel: 49-89-61-2070
Telex: 0524985 hpmch

Hewlett-Packard GmbH
Geschäftsstelle
Ermlisallee
7517 **WALDBRONN** 2
Postfach 1251
Tel: (07243) 602-0
Telex: 782 838 hepk

GREAT BRITAIN

See United Kingdom

GREECE

Hewlett-Packard Hellas
32, Kiffissias Avenue
15125 Amaroussion
ATHENS
Greece
Tel: 6828811
Telex: 216588 hpat gr
Fax: 6832978

GUATEMALA

IPESA DE GUATEMALA
Avenida Reforma 3-48, Zona 9
GUATEMALA CITY
Tel: 316627, 317853,66471/5
9-011-502-2-316627
Telex: 3055765 IPESA GU

GUINEA

R.T.I.
175, Rue Blomet
75015 **PARIS**
France
Tel: (1) 45 31 09 06
Telex: 203376
Fax: (1) 45 31 09 18

HONG KONG

Hewlett-Packard Asia, Ltd.
22/F, West Tower Bond Centre
89 Queensway Central
HONG KONG
Tel: (852/5) 848-7777
Fax: (85215) 868-4997
Cable: HEWPACK HONG KONG

ICELAND

Hewlett-Packard Iceland
Hoefdabakka 9
112 **REYKJAVIK**
Tel: 354-1-67-1000
Telex: 37409
Fax: 354-1-673031

INDIA

Computer products are sold through Blue Star Ltd. All computer repairs and maintenance service is done through Computer Maintenance Corp.

Blue Star Ltd.
B. D. Patel House
Near Sardar Patel Colony
AHMEDABAD 380 014
Tel: 403531, 403532
Telex: 0121-234
Cable: BLUE FROST

Blue Star Ltd.
40/4 Lavelle Road
BANGALORE 560 001
Tel: 57881, 867780
Telex: 0845-430 BSLBIN
Cable: BLUESTAR

Blue Star Ltd.
Sahas
414/2 Vir Savarkar Marg
Prabhadevi
BOMBAY 400 025
Tel: 422-6155
Telex: 011-71193 BSSS IN
Cable: FROSTBLUE

Blue Star Ltd.
Kalyan, 19 Vishwas Colony
Alkapuri, **BARODA,** 390 005
Tel: 65235, 65236
Cable: BLUE STAR

Blue Star Ltd.
7 Hare Street
P.O. Box 506
CALCUTTA 700 001
Tel: 230131, 230132
Telex: 031-61120 BSNF IN
Cable: BLUESTAR

Blue Star Ltd.
13 Community Center
New Friends Colony
NEW DELHI 110 065
Tel: 682547
Telex: 031-2463
Cable: BLUEFROST

Blue Star Ltd.
2-2-47/1108 Bolarum Rd.
SECUNDERABAD 500 003
Tel: 72057, 72058
Telex: 0155-459
Cable: BLUEFROST

Blue Star Ltd.
T.C. 7/603 Poornima
Maruthunkuzhi
TRIVANDRUM 695 013
Tel: 65799, 65820
Telex: 0884-259
Cable: BLUESTAR

Hewlett-Packard India
Meridian Commercial Complex
6th Floor
8 Windsor Place
Janpath
NEW DELHI 110 001
Tel: 91-11384911
Telex: 31-4935 HPNDIN

INDONESIA

BERCA Indonesia P.T.
P.O.Box 496/Jkt.
Jl. Abdul Muis 62
JAKARTA
Tel: 21-373009
Telex: 46748 BERSAL IA
Cable: BERSAL JAKARTA

BERCA Indonesia P.T.
P.O.Box 2497/Jkt
Antara Bldg., 12th Floor
Jl. Medan Merdeka Selatan 17
JAKARTA-PUSAT
Tel: 21-340417
Telex: 46748 BERSAL IA

BERCA Indonesia P.T.
Jalan Kutai 24
SURABAYA
Tel: 67118
Telex: 31146 BERSAL SB
Cable: BERSAL-SURABAYA

IRAQ

Hewlett-Packard Trading S.A.
Service Operation
Al Mansoor City 609/10/7
BAGHDAD
Tel: 551-49-73
Telex: 212-455 HEPAIRAQ IK

IRELAND

Hewlett-Packard Ireland Ltd.
Temple House, Temple Road
Blackrock, Co. **DUBLIN**
Tel: (353/1) 883399
Telex: 30439

Hewlett-Packard Ltd.
75 Belfast Rd, Carrickfergus
Belfast BT38 8PH
NORTHERN IRELAND
Tel: 09603-67333
Telex: 747626

ISRAEL

Eldan Electronic Instrument Ltd.
P.O. Box 1270
JERUSALEM 91000
16, Ohaliav St.
JERUSALEM 94467
Tel: 533 221, 553 242
Telex: 25231 AB/PAKRD IL

Computation and Measurement
Systems (CMS) Ltd.
11 Masad Street
67060
TEL-AVIV
Tel: 388 388
Telex: 33569 Motil IL

ITALY

Hewlett-Packard Italiana Spa
Via G. di Vittorio 10
20094 **CORSICO (MI)**
Tel: 02/4408351
Fax: 02/4409964

Hewlett-Packard Italiana Spa
Via Nuova Rivoltana 95
20090 **LIMITO (MI)**
Tel: 02/75761
Fax: 02/7576230

Hewlett-Packard Italiana Spa
Via Emilia 51/C
40011 **ANZOLA**
DELL'EMILIA (BO)
Tel: 051/731061

Hewlett-Packard Italiana Spa
Via M. Ricci 17 - Palombina Nuova
60100 **ANCONA**
Tel: 071/883782

Hewlett-Packard Italiana Spa
Traversa 99C Giulio Petroni 19
70124 **BARI**
Tel: 080/410744
Fax: 080/417891

Hewlett-Packard Italiana Spa
Via Principe Nicola 43G/C
95126 **CATANIA**
Tel: 095/371087
Fax: 095/388569

Hewlett-Packard Italiana Spa
Via G. Di Vittorio 9
20063 **CERNUSCO S/N (MI)**
Tel: 02/923691
Fax: 02/9237746

Hewlett-Packard Italiana Spa
Via Sacco e Vanzetti 1/A
50145 **FIRENZE**
Tel: 055/318533
Fax: 055/373965

Hewlett-Packard Italiana Spa
Viale Brigata Bisagno 2
16129 **GENOVA**
Tel: 010/541141
Fax: 010/591733

Hewlett-Packard Italiana Spa
Via Orazio 16
80122 **NAPOLI**
Tel: 081/7611444
Fax: 081/680164

Hewlett-Packard Italiana Spa
Via Pellizzo 15
35128 **PADOVA**
Tel: 049/8070166
Fax: 049/773097

Hewlett-Packard Italiana Spa
Via Del Tintoretto 200
00144 **ROMA**
Tel: 06/54831
Fax: 06/5408710

SALES OFFICES
Arranged alphabetically by country (cont'd)

ITALY (Cont'd)

Hewlett-Packard Italiana Spa
Corso Svizzera 185
10149 **TORINO**
Tel: 011/744044
Fax: 011/7710815

IVORY COAST

Ste Ivoirienne des Techniques
de l'Informatique
Immeuble C.N.A. - 5e etage
Avenue General de Gaulle
P.O. Box 161
ABIDJAN 01
Tel: 32 12 27
Telex: 22149

Engineering Business Concept (E.B.C.)
Angle Avenue J. Anoma et Bd.
République
08 B.P. 323 **ABIDJAN** 08
Tel: 32 50 24, 41 48 70
Fax: 35 37 90

JAPAN

Yokogawa-Hewlett-Packard Ltd.
Nihon-Seimei Akita
Chuo-Dori Bldg.
4-2-7 Naka-dori
AKITA, 010
Tel: (81/188) 36-5021

Yokogawa-Hewlett-Packard Ltd.
152-1, Onna
ATSUGI, Kanagawa, 243
Tel: (81/462) 25-0031
Fax: (81/462) 25-0064

Yokogawa-Hewlett-Packard Ltd.
3-1 Motochiba-Cho
CHIBA, 280
Tel: (81/472) 25-7701
Fax: (81/472) 21-0382

Yokogawa-Hewlett-Packard Ltd.
Dai-3 Hakata-Kaisei Bldg.
1-3-6 Hakata-eki Minami
Hakata-Ku, **FUKUOKA** 812
Tel: (81/92) 472-8731

Yokogawa-Hewlett-Packard Ltd.
Nihon-Dantai-Seimei-
Koriyama Bldg.
21-10 Toramaru-Cho,
Koriyama **FUKUSHIMA,** 963
Tel: (81/249) 39-7111

Yokogawa-Hewlett-Packard Ltd.
Yasuda-Seimei Hiroshima Bldg.
6-11, Hon-dori, Naka-ku
HIROSHIMA, 730
Tel: (81/82) 241-0611
Fax: (81/82) 241-0619

Yokogawa-Hewlett-Packard Ltd.
Issei Bldg.
2-3-17 Takezono, Tsukuba
IBARAGI, 305

Yokogawa-Hewlett-Packard Ltd.
Mito Mitsui Bldg.
1-4-73, Sanno-maru
Mito, **IBARAKI** 310
Tel: (81/292) 25-7470

Yokogawa-Hewlett-Packard Ltd.
Towa Building 2-2-3
Kaigan-dori, Chuo-ku
KOBE, 650
Tel: (81/78) 392-4791
Fax: (81/78) 392-4839

Yokogawa-Hewlett-Packard Ltd.
Kumagaya Asahi 82 Bldg.
3-4 Tsukuba
KUMAGAYA, Saitama 360
Tel: (81/485) 24-6563
Fax: (81/485) 24-9050

Yokogawa-Hewlett-Packard Ltd.
Shin-Kyoto Center Bldg.
614 Higashi-Shiokoji-cho
Karasuma-Nishi-Iru, Shiokoji-Dori
Shimogyo-Ku, **KYOTO,** 600
Tel: (81/75) 343-0921
Fax: (81/75) 343-4356

Yokogawa-Hewlett-Packard Ltd.
Mito Mitsui Bldg.
1-4-73, Sanno-Maru
MITO, Ibaraki 310
Tel: (81/292) 25-7470
Fax: (81/292) 31-6589

Yokogawa-Hewlett-Packard Ltd.
Nagano-Tokyokaiiyo Bldg.
1081, Minamiagata-Machi
Nagano-Shi, **NAGANO,** 380
Tel: (81/262) 24-8012
Fax: (81/262) 24-8016

Yokogawa-Hewlett-Packard Ltd.
Nagoya Kokusai Center Building
1-47-1, Nagono, Nakamura-ku
NAGOYA, AICHI 450
Tel: (81/52) 571-5171
Fax: (81/52) 565-0896

Yokogawa-Hewlett-Packard Ltd.
Sai-Kyo-Ren Building
1-2 Dote-cho
OOMIYA-SHI SAITAMA 330
Tel: (0486) 45-8031

Yokogawa-Hewlett-Packard Ltd.
Chuo Bldg., 5-4-20 Nishi-Nakajima
4-20 Nishinakajima, 5 Chome,
Yodogawa-ku
OSAKA, 532
Tel: (81/6) 304-6021
Fax: (81/6) 304-0216

Yokogawa-Hewlett-Packard Ltd.
1-27-15, Yabe
SAGAMIHARA Kanagawa, 229
Tel: (81/427) 59-1311

Yokogawa-Hewlett-Packard Ltd.
Hamamtsu Motoshiro-Cho Daichi
Seimei Bldg 219-21, Motoshiro-Cho
Hamamatsu-shi
SHIZUOKA, 430
Tel: (81/534) 56-1771
Fax: (81/534) 552371

Yokogawa-Hewlett-Packard Ltd.
Shinjuku Daiichi Seimei Bldg.
Nishi Shinjuku 2-7-1,
Shinjuku-ku,**TOKYO** 163
Tel: (81/3) 348-4611
Fax: (81/3) 348-7969

Yokogawa-Hewlett-Packard Ltd.
9-1, Takakura-cho
Hachioji-shi, **TOKYO,** 192
Tel: 81-426-42-1231

Yokogawa-Hewlett-Packard Ltd.
Tokyo-Nissam-Minato Bldg.
1-6-34 Konan, Minato-Ku
TOKYO 108
Tel: (81/3) 458-5411

Yokogawa-Hewlett-Packard Ltd.
29-21 Takaido-Higashi, 3-chome
Suginami-ku **TOKYO** 168
Tel: (03) 331-6111
Telex: 232-2024 YHPTOK

Yokogawa Hokushin Electric Corp.
(YEW)
Shinjuku-NS Bldg. 10F
9-32 Nokacho 2 Chome
Shinjuku-ku
TOKYO, 163
Tel: (03) 349-1859
Telex: J27584

Yokogawa-Hewlett-Packard Ltd.
Toyoda-Tokyo-Kaijo Bldg.
1-179 Miyuki-Hon-Cho
TOYODA 473
Tel: (81/565) 27-5611

Yokogawa-Hewlett-Packard Ltd.
Chiyodaseimei-Utsunomiya Bldg.
2-3-1, Ohdori, **UTSUNOMIYA,** Tochigi-
Shi 320
Tel: (81/286) 33-1153
Fax: (81/286) 33-1175

Yokogawa-Hewlett-Packard Ltd.
No. 2 Yasuda Bldg.
2-32-12, Tsuruyo-cho
Kanagawa-ku, **YOKOHAMA** 221
Tel: (81/45) 312-1252
Fax: (81/45) 311-8328

KENYA

ADCOM Ltd., Inc., Kenya
P.O. Box 30070
NAIROBI
Tel: 331955
Telex: 22639

KOREA

Samsung Hewlett-Packard Co. Ltd.
Dongbang Yeoeuido Building
36-1 Yeoeui Do-Dong
Youngdeungpo-Ku
SEOUL, 150
Tel: (82/2) 784-4666, 784-2666
Fax: (82/2) 784-7084
Telex: 25166 SAMSAN K

KUWAIT

Al-Khaldiya Trading & Contracting
P.O. Box 830
SAFAT 13009
Tel: 242 49 10, 241 17 26
Telex: 22481 AREEG KT
Cable: VISCOUNT

LEBANON

Computer Information Systems S.A.L.
Chammas Building
P.O. Box 11-6274
DORA BEIRUT
Tel: 89 31 13, 58 18 35
Telex: 42309 chacis le
Fax: 58 18 34

LUXEMBOURG

Hewlett-Packard Belgium S.A./N.V.
Blvd de la Woluwe, 100
Woluwedal
B-1200 **BRUSSELS**
Tel: (32/2) 761-3111
Telex: 23-494 paloben bru

MADAGASCAR

R.T.I.
175, Rue Blomet
75015 Paris
FRANCE
Tel: (1) 45 31 09 06
Tlx: 203376
Fax: (1) 45 31 09 18

MALAWI

Systron (Private) Ltd.
Manhattan Court
61 Second Street
P.O. Box 3458
HARARE
Zimbabwe
Tel: 739881/739885
Telex: 4122
Fax: 70 20 08

MALAYSIA

Hewlett-Packard Sales (Malaysia)
Sdn. Bhd.
9th Floor
Chung Khiaw Bank Building
46, Jalan Raja Laut
50350 **KUALA LUMPUR, MALAYSIA**
Tel: (60/3) 298-6555
Fax: (60/3) 291-5495

Protel Engineering
P.O.Box 1917
Lot 6624, Section 64
23/4 Pending Road
Kuching, **SARAWAK**
Tel: 36299
Telex: 70904 PROMAL MA
Cable: PROTELENG

MALTA

R.T.I.
175, Rue Blomet
75015 **PARIS**
France
Tel: (1) 45 31 09 06
Telex: 203376
Fax: (1) 45 31 09 18

MAURITANIA

R.T.I.
175, Rue Blomet
75015 **PARIS**
France
Tel: (1) 45 31 09 06
Tlx: 203376
Fax: (1) 45 31 09 18

MAURITIUS

R.T.I.
175, Rue Blomet
75015 **PARIS**
France
Tel: (1) 45 31 09 06
Telex: 203376
Fax: (1) 45 31 09 18

MEXICO

Hewlett-Packard de Mexico,
S.A. de C.V.
Rio Nio No. 4049 Desp. 12
Fracc. Cordoba
JUAREZ, Mexico
Tel: 161-3-15-62

Hewlett-Packard de Mexico,
S.A. de C.V.
Condominio Kadereyta
Circuito del Mezon No. 186 Desp. 6
Col. Del Prado - 76030
QRO, Mexico
Tel: 463-6-02-71

Hewlett-Packard de Mexico,
S.A. de C.V.
Monti Morelos No. 299
Fraccionamiento Loma Bonita 45060
GUADALAJARA, Jalisco
Tel: (52/36) 31 46 00
Telex: 0684 186 ECOME

Hewlett-Packard de Mexico,
S.A. de C.V.
Monte Pelvoux No. 111
Lomas de Chapultepec
11000 **MEXICO, D.F.**
Tel: (52/5) 596 79 33
Fax: (52/5) 596 42 08 (Ext 3231)

Hewlett-Packard de Mexico,
S.A. de C.V.
Czda. del Valle
409 Ote. 4th Piso
Colonia del Valle
Municipio de Garcia
66220 **NUEVO LEON**
Tel: 83-78-42-40
Telex: 382410 HPMY

Hewlett-Packard Co.
Latin America Region
Customer Support Center
7208 N.W. 31st St.
MIAMI, FL 33122
United States
Tel: (305) 599-0465
Telex: 441603 HPMIAMI
Fax: 599-0277

Hewlett-Packard de Mexico,
S.A. de C.V.
Blvd. Independencia No. 2000 Ote.
Ote Zerpiso
Co 1 Estrella
27010 **TORREON, COA.**
Tel: (52/171) 8 22 01

MOROCCO

Sicotel
Complexe des Habous
Tour C, avenue des Far
CASABLANCA 01
Tel: 31 22 70
Telex: 27604

R.T.I.
175, Rue Blomet
75015 **PARIS**
France
Tel: (1) 45 31 09 06
Tlx: 203376
Fax: (1) 45 31 09 18

Socofren Maroc
164, Boulevard D' Anfa
CASABLANCA
Tel: 36 08 84, 36 01 77
Telex: 23940

NETHERLANDS

Hewlett-Packard Nederland
Startbaan 16
1187 XR **AMSTELVEEN**
Tel: (31/20) 5476911
Telex: 13 216 HEPA NL
Fax: (31/20) 471825

Hewlett-Packard Nederland B.V.
Bongerd 2
NL 2900AA **CAPELLE A/D IJSSEL**
Tel: 31-20-51-6444
Telex: 21261 HEPAC NL

Hewlett-Packard Nederland B.V.
Pastoor Petersstraat 134-136
P.O. Box 2342
NL 5600 CH **EINDHOVEN**
Tel: 31-40-32-6911
Telex: 51484 hepae nl
Fax: (31/40) 446546

NEW ZEALAND

Hewlett-Packard (N.Z.) Ltd.
5 Owens Road
P.O. Box 26-189
Epsom, **AUCKLAND**
Tel: (64/9) 605-651
Fax: (64/9) 600-507

Hewlett-Packard (N.Z.) Ltd.
186-190 Willis Street
P.O. Box 9443
WELLINGTON
Tel: (64/4) 820-400
Fax: (64/4) 843-380

NIGER

S.I.T.E.L.
Immeuble le General
Avenue General de Gaulle
PO Box 161
ABIDJAN 01
Ivory Coast
Tel: 32 12 27
Telex: 22149

NIGERIA

Management Information Systems Ltd.
3 Gerrard Road, Ikoyi
LAGOS
Tel: 68 08 87
Telex: 23582
Fax: 68 54 87

NORTHERN IRELAND

See United Kingdom

NORWAY

Hewlett-Packard Norway A/S
Oesterndalen 16-18
P.O. Box 34
N-1345 **OESTERAAS**
Tel: (47/2) 24-6090
Telex: 76621 HPNAS N

Hewlett-Packard Norway A/S
Boemergt. 42
Box 2470
N-5037 **SOLHEIMSVIK**
Tel: (5/29) 10 72

OMAN

Suhail & Saud Bahwan
P.O.Box 169
MUSCAT/SULTANATE OF OMAN
Tel: 79 37 41
Telex: 3585 mb
Fax: 79 61 58

Imtac LLC
P.O. Box 9196
MINA AL FAHAL/SULTANATE OF OMAN
Tel: 70-77-27, 70-77-23
Telex: 3865 Tawoos On

PAKISTAN

Mushko & Company Ltd.
House No. 16, Street No. 16
Sector F-6/3
ISLAMABAD
Tel: 824545
Telex: 54001 Muski Pk
Cable: FEMUS Islamabad

Mushko & Company Ltd.
Oosman Chambers
Abdullah Haroon Road
KARACHI 0302
Tel: 524131, 524132
Telex: 2894 MUSKO PK
Cable: COOPERATOR Karachi

PANAMA

Electronico Balboa, S.A.
Calle Samuel Lewis, Ed. Alfa
Apartado 4929
PANAMA CITY
Tel: 9-011-507-636613
Telex: 368 3483 ELECTRON PG

PERU

Cia Electro Médica S.A. (ERMED)
Los Flamencos 145, Ofc. 301/2
San Isidro
Casilla 1030
LIMA 1 Peru
Tel: 9-011-511-4-414325, 41-3705
Telex: 39425257 PE PB SIS

PHILIPPINES

The Online Advanced Systems Corp.
2nd Floor, Electra House
115-117 Esteban Street
P.O. Box 1510
Legaspi Village, Makati
Metro **MANILA**
Tel: 815-38-10 (up to 16)
Telex: 63274 ONLINE PN

PORTUGAL

Mundinter Intercambio
Mundial de Commercio
Avenida Antonio 2761
LISBON
Tel: 53 21 31, 53 21 37
Telex: 16691

CPC Instrumentacao
Torre de Santo Antonio
rue Gregorio Lopes, Lote
Restelo
1400 **LISBON**
Tel: 617343/44/45/46
Telex: 27432/26054
Fax: 617345

C.P.C.S.I.
Rua de Costa Cabral 575
4200 **PORTO**
Tel: 493122
Telex: 26054, 27432
Fax: 48 87 21

PUERTO RICO

Hewlett-Packard Puerto Rico
Box 4048
Aguadilla, PR 00605
Tel: (809) 891-5235

Hewlett-Packard Puerto Rico
101 Munoz Rivera Avenue
Esu. Calle Ochoa
HATO REY, 00918
Tel: (809) 754-7800

QATAR

Qatar Datamation Systems
P.O. Box 350
DOHA
Tel: 41 32 82
Tlx: 4833
Fax: 42 63 78

REUNION ISLAND

R.T.I.
175, Rue Blomet
75015 **PARIS**
France
Tel: (33/1) 45310906
Telex: 42/203376

RWANDA

R.T.I.
175, Rue Blomet
75015 **PARIS**
France
Tel: (33/1) 45310906
Telex: 42/203376
Fax: (1) 45 31 09 18

SAUDI ARABIA

Modern Electronics Establishment
P.O. Box 281
Thuobah
AL-KHOBAR 31952
Tel: 895-1760, 895-1764
Telex: 671 106 HPMEEK SJ
Cable: ELECTA AL-KHOBAR

Modern Electronics Establishment
P.O. Box 1228
Redec Plaza, 6th Floor
JEDDAH
Tel: 644 96 28
Telex: 4027 12 FARNAS SJ
Cable: ELECTA JEDDAH

Modern Electronics Establishment
P.O.Box 22015
RIYADH 11495
Tel: 4763030
Telex: 402040 MEERYD SJ

SCOTLAND

See United Kingdom

SENEGAL

2SC
10, rue Tolbiac
B.P. 3716
DAKAR R.P.
(Please contact R.T.I. France.)
Tel: (221) 222248
Telex: 906/671

R.T.I.
175, Rue Blomet
75015 **PARIS**
France
Tel: 45310906
Telex: 203376
Fax: (1) 45 31 09 18

SINGAPORE

Hewlett-Packard Singapore Ltd.
1150 Depot Road
SINGAPORE, 0410
Tel: (65) 273 7388
Fax: (65) 278 8990

SOUTH AFRICA

Hi Performance Systems (Pty.) Ltd.
P.O. Box 120, Howard Place
CAPE TOWN 7450
Tel: (27/21) 53-7954
Fax: (27/21) 53-5119

Hi Performance Systems (Pty.) Ltd.
Private Bag Wendywood
SANDTON 2144
Tel: (27/11) 802-5111
Fax: (27/11) 802-6332

SPAIN

Hewlett-Packard Española, S.A.
Avda. Diagonal, 605
08028-**BARCELONA**
Tel: (34/3) 401 91 00
Telex: 52603 hpbee

Bilbao (Vizcaya) Sales
Hewlett-Packard Español, S.A.
Avda. Zugazarte, 8
48930 - Las Arenas - **VIZCAYA**

Hewlett-Packard Española, S.A.
Crta. N-VI, Km. 16, 500
Las Rozas
E-MADRID
Tel: (34/1) 6370011
Telex: 23515 HPE

Hewlett-Packard Española, S.A.
Avda. S. Francisco Javier, S/N
Planta 10. Edificio Sevilla 2
E-SEVILLA 5, SPAIN
Tel: (34/54/64) 4454
Telex: 72933

Hewlett-Packard Española, S.A.
Isabel La Catolica, 8
46004 **VALENCIA**
Tel: (34/6) 351 59 44
Telex: 63435
Fax: (34/6) 351 59 44

Hewlett-Packard Española, S.A.
Avda. de Zugazarte, 8
48930 - Las Arenas
VIZCAYA
Tel: (34/4) 464 32 55
Telex: 33032

SUDAN

Mediterranean Engineering &
Trading Co., Ltd.
P.O. Box 1025
KHARTOUM
Tel: (249) 41184
Telex: 24052

R.T.I.
175, Rue Blomet
75015 **PARIS**
France
Tel: (1) 45 31 09 06
Tlx: 203376
Fax: (1) 45 31 09 18

SWEDEN

Hewlett-Packard Sverige AB
Östra Tullgatan 3
20011 **MALMÖ**
Box 6132
Tel: (46/40) 702 70
Telex: (854) 17886 (via Spånga office)
Fax: (46/40) 97 74 18

Hewlett-Packard Sverige AB
Elementvagen 16
S-7022 7 **ÖREBRO**
Tel: (49/19) 10 48 80
Telex: (854) 17886 (via Spånga office)

Hewlett-Packard Sverige AB
Skalholtsgatan 9, Kista
P.O. Box 19
S- 16493 **KISTA**
Tel: 46/8/750-200
Telex: (854) 17886
Telefax: (08) 7527781

Hewlett-Packard Sverige AB
Box 266
Topasgatan 1A
S-42123 **VÄSTRA-FRÖLUNDA**
(Gothenburg)
Tel: (031) 891000
Telex: (854) 17886 (via Spånga office)

SWITZERLAND

Hewlett-Packard (Schweiz) AG
Clarastrasse 12
CH-4058 **BASEL**
Tel: (41/61) 681 59 20
Fax: (41/61) 681 98 59

Hewlett-Packard (Schweiz) AG
7, rue du Bois-du-Lan
Case postale 365-1366
CH-1217 **MEYRIN** 1
Tel: (41/22) 7804111
Telex:27333 HPAG CH

Hewlett-Packard (Schweiz) AG
Allmend
CH-8967 **WIDEN**
Tel: (41/57) 321 111
Telex: 53933 HPAG CH
Fax: (41/57) 321 475

SYRIA

Middle East Electronics
P.O.Box 2308
Abu Rumaneh
DAMASCUS
Tel: 33 45 92
Telex: 411 771 Meesy

TAIWAN

Hewlett-Packard Taiwan Ltd.
Taipei Office
8th Floor, Hewlett-Packard Building
337 Fu Hsing North Road
TAIPEI
Tel: (02) 712-0404
Telex: 24439 HEWPACK
Cable: HEWPACK Taipei

Hewlett-Packard Taiwan Ltd.
THM Office
2, Huan Nan Road
CHUNG LI, Taoyuan
Tel: (034) 929-666

SALES OFFICES
Arranged alphabetically by country (cont'd)

TAIWAN (Cont'd)

Hewlett-Packard Taiwan Ltd.
Taichung Office
5FL, 67, Sec. 3,
Wen-Hsin Road,
TAICHUNG
Tel: (04) 254-1201

Hewlett-Packard Taiwan Ltd.
Kaohsiung Office
11/F, 456, Chung Hsiao 1st Road
KAOHSIUNG
Tel: (07) 2412318

TANZANIA

Adcom Ltd, Inc. Kenya
P.O. Box 30070
NAIROBI
Kenya
Tel: 33 19 55
Telex: 22639

THAILAND

Unimesa Co. Ltd.
2540 Sukumwit Avenue
Bangna
BANGKOK 10260
Tel: 662-398-6953
Telex: 84439 Simonco TH
Cable: UNIMESA Bangkok

TOGO

S.I.T.E.L.
Immeuble le General
Avenue General de Gaulle
P.O. Box 161
ABIDJAN 01
Ivory Coast
Tel: 32 12 27
Telex: 22149

Societe Africaine De Promotion
Immeuble Sageb
Rue d'Atakpame
P.O. Box 4150
LOME
Tel: 21-62-88
Telex: 5357

TRINIDAD & TOBAGO

Caribbean Telecoms Ltd.
Corner McAllister Street &
Eastern Main Road, Laventille
P.O. Box 732
PORT-OF-SPAIN
Tel: 624-4213
Telex: 22561 CARTEL WG
Cable: CARTEL, PORT OF SPAIN

Computer and Controls Ltd.
P.O. Box 51
1 Taylor Street
PORT-OF-SPAIN
Tel: (809) 622-7719/622-7985
Telex: 38722798 COMCON WG
LOOGO AGENCY 1264

TUNISIA

Precision Electronique
5, rue de Chypre
Mutuelleville
1002 **TUNIS BELVEDERE**
Tunisia
Tel: 78 50 37
Tlx: 13238

TURKEY

E.M.A.
Mediha Eldem Sokak No. 41/6
Yenisehir
ANKARA
Tel: 131 4695, 131 9175
Telex: 46912n emsetr
Cable: EMATRADE ANKARA

Hewlett-Packard Bilgisayar Ve Olcum
Sistemleri A.S. (Headquarter)
Mesrutiyet Mah. 19 Mayis Cad.
Nova-Baran Plaza Kat: 11-12
SISLI / ISTANBUL
Tel: 175 29 70
Telex: 39150
Fax: 175 29 92

Hewlett-Packard Bilgisayar Ve Olcum
Sistemleri A.S.
Paris Caddesi No 3
Diare 9
06670 **ANKARA**
Tel: 125 83 13
Telex: 46180
Fax: 125 47 45

UGANDA

Adcom Ltd, Inc. Kenya
P O Box 30070
NAIROBI
Kenya
Tel: 33 19 55
Telex: 22639

UNITED ARAB EMIRATES

Emitac Ltd.
P.O. Box 1641
SHARJAH
Tel: 591181
Telex: 48710 EMITAC EM
Cable: EMITAC SHARJAH

Emitac Ltd.
P.O. Box 2711
ABU DHABI
Tel: 820419-20
Cable: EMITACH ABUDHABI

Emitac Ltd.
P.O. Box 8391
DUBAI,
Tel: 377591

Emitac Ltd.
P.O. Box 473
RAS AL KHAIMAH
Tel: 28133, 21270

UNITED KINGDOM

ENGLAND

Hewlett-Packard Ltd.
Customer Information Centre
King St. Lane
Winnersh, Wokingham
GB-**BERKSHIRE** RG11 5AR
Tel: (44/734) 784774
Telex: 847178
Fax: 777285

Hewlett-Packard Ltd.
Miller House
The Ring, **BRACKNELL**
Berkshire RG12 1XN
Tel: (44/344) 424-898
Fax: (44/344) 860015, Ext 56023

Hewlett-Packard Ltd.
Customer Sales & Support
Building 1, Filton Road
Stoke Gifford
BRISTOL, BS12 6QZ
Tel: (44/272) 236000

Hewlett-Packard Ltd.
Oakfield House, Oakfield Grove
Clifton **BRISTOL,** Avon BS8 2BN
Tel: 44-272-736 806
Telex: 444302

Hewlett-Packard Ltd.
9 Bridewell Place
LONDON EC4V 6BS
Tel: (44/583) 6565
Fax: (44/583) 6565, Ext 4713

Hewlett-Packard Ltd.
Heathside Park Rd.
Cheadle Heath, Stockport
GB-**MANCHESTER** SK3 ORB
Tel: (44/61) 428-0828
Telex: 668068
Fax: 4955009

Hewlett-Packard Ltd.
Harman House
No. 1 George St.
Uxbridge,
GB-**MIDDLESEX** UB8 1YH
Tel: (44/895) 72020
Telex: 893135
Fax: 73684

Hewlett-Packard Ltd.
Pontefract Road
NORMANTON,
West Yorkshire WF6 1RN
Tel: (44/924) 895-566
Fax: (44/924) 896-691
Telex: 557355

Hewlett-Packard Ltd.
The Quadrangle
106-118 Station Road
REDHILL, Surrey RH1 1PS
Tel: 44-737-686-55
Telex: 947234

Hewlett-Packard Ltd.
Avon House
435 Stratford Road
Shirley, **SOLIHULL,** West Midlands
B90 4BL
Tel: 44-21-745-8800
Telex: 339105

Hewlett-Packard Ltd.
Heathside Park Road
Cheadle Heath, Stockport
Cheshire SK3 ORB
Tel: (44/61) 428-0828
Fax: (44/61) 495-5009
Telex: 668068

Hewlett-Packard L'td.
Harman House
No. 1 George Street
UXBRIDGE, Middlesex UB8 1YH
Tel: (44/895) 72020
Fax: (44/895) 73684

Hewlett-Packard Ltd.
King Street Lane
Winnersh, **WOKINGHAM**
Berkshire RG11 5AR
Tel: (44/734) 784774
Fax: (44/734) 777285 Ext 52285

NORTHERN IRELAND

Hewlett-Packard (Ireland) Ltd.
Carrickfergus Industrial Centre
75 Belfast Road, Carrickfergus
CO. ANTRIM BT38 8PM
Tel: 09603 67333

Unit 5
Bridgewood House
Newforge Lane
Malone Road
BELFAST BT95 NW
Tel: (353/232) 664-851
Fax: (353/232) 665-619

SCOTLAND

Hewlett-Packard Ltd.
1/3 Springburn Place
College Milton North
EAST KILBRIDE, G74 5NU
Tel: 035-52-49261
Fax: 03552-35929
Telex: 779615

Hewlett-Packard Ltd.
SOUTH QUEENSFERRY
West Lothian, EH30 9TG
Tel: 031-331-1188
Fax: 031-331-7412

UNITED STATES

Hewlett-Packard Co.
Customer Information Center
Tel: (800) 752-0900
Hours: 6:00 AM to 5:00 PM
Pacific Time

Alabama

Hewlett-Packard Co.
2100 Riverchase Center
Building 100 - Suite 118
BIRMINGHAM, AL 35244
Tel: (205) 988-0547
Fax: (205) 988-5308

Hewlett-Packard Co.
620 Discovery Dr.
HUNTSVILLE, AL 35806
Tel: (205) 830-2000
Fax: (205) 830-1427

Alaska

Hewlett-Packard Co.
4000 Old Seward Highway
Suite 101
ANCHORAGE, AK 99503
Tel: (907) 563-8855
Fax: (907) 561-7409

Arizona

Hewlett-Packard Co.
8080 Pointe Parkway West
PHOENIX, AZ 85044
Tel: (602) 273-8000
Fax: (602) 273-8080

Hewlett-Packard Co.
3400 East Britannia Dr.
Bldg. C, Suite 124
TUCSON, AZ 85706
Tel: (602) 573-7400
Fax: (602) 573-7429

Arkansas

Hewlett-Packard Co.
10816 Executive Center Dr
Conway Bldg. Suite 116
LITTLE ROCK, AR 72211
Tel: (501) 225-7178
Fax: (501) 221-3614

California

Hewlett-Packard Co.
26701 W. Argoura Rd.
CALABASAS, CA 91302
Tel: (818) 880-3400
Fax: (818) 880-3437

Hewlett-Packard Co.
353 Lakeside Dr
FOSTER CITY, CA 94404
Tel: (415) 378-8400
Fax: (415) 378-8405

Hewlett-Packard Co.
1907 North Gateway Blvd.
FRESNO, CA 93727
Tel: (209) 252-9652
Fax: (209) 456-9302

Hewlett-Packard Co.
1421 S. Manhattan Av.
FULLERTON, CA 92631
Tel: (714) 999-6700
Fax: (714) 778-3033

Hewlett-Packard Co.
7408 Hollister Ave. #A
GOLETA, CA 93117
Tel: (805) 685-6100
Fax: (805) 685-6163

Hewlett-Packard Co.
9800 Muirlando Ave.
IRVINE, CA 92718
Tel: (714) 472-3000
Fax: (714) 581-3607 (Direct Dial only)

Hewlett-Packard Co.
2525 Grand Avenue
LONG BEACH, CA 90815
Tel: (213) 498-1111
Fax: (213) 494-1986

Hewlett-Packard Co.
5651 West Manchester Ave.
LOS ANGELES, CA 90045
Tel: (213) 337-8000
Fax: (213) 337-8338

Hewlett-Packard Co.
321 E. Evelyn Ave.
Bldg. 330
MOUNTAIN VIEW, CA 94039
Tel: (415) 694-2000
Fax: (415) 694-0600

Hewlett-Packard Co.
5161 Lankershim Blvd.
NORTH HOLLYWOOD, CA 91601
Tel: (818) 505-5600
Fax: (818) 505-5875

Hewlett-Packard Co.
5725 W. Las Positas Blvd.
PLEASANTON, CA 94566
Tel: (415) 460-0282
Fax: (415) 460-0713

Hewlett-Packard Co.
4244 So. Market Court, Suite A
SACRAMENTO, CA 95834
Tel: (916) 929-7222
Fax: (916) 927-7152

Hewlett-Packard Co.
9606 Aero Drive
SAN DIEGO, CA 92123
Tel: (619) 279-3200
Fax: (619) 268-8487

Hewlett-Packard Co.
50 Fremont St. Suite 200
SAN FRANCISCO, CA 94105
Tel: (415) 882-6800
Fax: (415) 882-6805

Hewlett-Packard Co.
3003 Scott Boulevard
SANTA CLARA, CA 95054
Tel: (408) 988-7000
Fax: (408) 988-7103

Hewlett-Packard Co.
5280 Valentine Rd. Suite 205
VENTURA, CA 93003
Tel: (805) 658-6898
Fax: (805) 650-0721

Colorado

Hewlett-Packard Co.
2945 Center Green Court South
Suite A
BOULDER, CO 80301
Tel: (303) 938-3065
Fax: (303) 938-3025

Hewlett-Packard Co.
24 Inverness Place, East
ENGLEWOOD, CO 80112
Tel: (303) 649-5000
Fax: (303) 649-5787

Connecticut

Hewlett-Packard Co.
3 Parkland Dr.
DARIEN, CT 06820
Tel: (203) 656-0040
Fax: (203) 656-5563

Hewlett-Packard Co.
115 Glastonbury Blvd
GLASTONBURY, CT 06033
Tel: (203) 633-8100
Fax: (203) 659-6087

Florida

Hewlett-Packard Co.
5900 N. Andrews, Suite 100
FORT LAUDERDALE, FL 33309
Tel: (305) 938-9800
Fax: (305) 938-2293

Hewlett-Packard Co.
6800 South Point Parkway
Suite 301
JACKSONVILLE, FL 32216
Tel: (904) 636-9955
Fax: (904) 636-9955

Hewlett-Packard Co.
255 East Drive, Suite B
MELBOURNE, FL 32901
Tel: (407) 729-0704
Fax: (407) 723-4557

Hewlett-Packard Co.
6177 Lake Ellenor Drive
ORLANDO, FL 32809
Tel: (407) 859-2900
Fax: (407) 826-9309

Hewlett-Packard Co.
4700 Bayou Blvd.
Building 5
PENSACOLA, FL 32503
Tel: (904) 476-8422
Fax: (904) 476-4116

Hewlett-Packard Co.
5550 Idlewild, #150
TAMPA, FL 33634
Tel: (813) 884-3282
Fax: (813) 889-4445

Georgia

Hewlett-Packard Co.
2015 South Park Place
ATLANTA, GA 30339
Tel: (404) 955-1500
Fax: (404) 980-7669

Hewlett-Packard Co.
3607 Parkway Lane
Suite 300
NORCROSS, GA 30092
Tel: (404) 448-1894
Fax: (404) 246-5206

Hawaii

Hewlett-Packard Co.
Pacific Tower
1001 Bishop St.
Suite 2400
HONOLULU, HI 96813
Tel: (808) 526-1555
Fax: (808) 536-7873

Idaho

Hewlett-Packard Co.
11309 Chinden Blvd.
BOISE, ID 83714
Tel: (208) 323-2700
Fax: (208) 323-2528

Illinois

Hewlett-Packard Co.
2205 E. Empire St.
BLOOMINGTON, IL 61704
Tel: (309) 662-9411
Fax: (309) 662-0351

Hewlett-Packard Co.
525 W. Monroe St., Suite 1308
CHICAGO, IL 60606
Tel: (312) 930-0010
Fax: (312) 930-0986

Hewlett-Packard Co.
1200 East Diehl Road
NAPERVILLE, IL 60566
Tel: (312) 357-8800
Fax: (312) 357-9896

Hewlett-Packard Co.
5201 Tollview Drive
ROLLING MEADOWS, IL 60008
Tel: (312) 255-9800
Fax: (312) 259-5878

Indiana

Hewlett-Packard Co.
11911 N. Meridian St.
CARMEL, IN 46032
Tel: (317) 844-4100
Fax: (317) 843-1291

Hewlett-Packard Co.
111 E. Ludwig Road
Suite 108
FT. WAYNE, IN 46825
Tel: (219) 482-4283
Fax: (219) 482-9907

Iowa

Hewlett-Packard Co.
4050 River Center Court
CEDAR RAPIDS, IA 52402
Tel: (319) 393-0606
Fax: (319) 378-1024

Hewlett-Packard Co.
4201 Corporate Dr.
WEST DES MOINES, IA 50265
Tel: (515) 224-1435
Fax: (515) 224-1870

Kansas

Hewlett-Packard Co.
North Rock Business Park
3450 N. Rock Rd.
Suite 300
WICHITA, KS 67226
Tel: (316) 636-4040
Fax: (316) 636-4504

Kentucky

Hewlett-Packard Co.
305 N. Hurstbourne Lane,
Suite 100
LOUISVILLE, KY 40222
Tel: (502) 426-0100
Fax: (502) 426-0322

Louisiana

Hewlett-Packard Co.
160 James Drive East
ST. ROSE, LA 70087
Tel: (504) 467-4100
Fax: (504) 467-4100 x 291

Maryland

Hewlett-Packard Co.
3701 Koppers Street
BALTIMORE, MD 21227
Tel: (301) 644-5800
Fax: (301) 362-7650

Hewlett-Packard Co.
2 Choke Cherry Road
ROCKVILLE, MD 20850
Tel: (301) 948-6370
Fax: (301) 948-5986

Massachusetts

Hewlett-Packard Co.
1775 Minuteman Road
ANDOVER, MA 01810
Tel: (508) 682-1500
Fax: (508) 794-2619

Hewlett-Packard Co.
29 Burlington Mall Rd.
BURLINGTON, MA 01803-4514
Tel: (617) 270-7000
Fax: (617) 221-5240

Michigan

Hewlett-Packard Co.
3033 Orchard Vista S.E.
GRAND RAPIDS, MI 49546
Tel: (616) 957-1970
Fax: (616) 956-9022

Hewlett-Packard Co.
39550 Orchard Hill Place Drive
NOVI, MI 48050
Tel: (313) 349-9200
Fax: (313) 349-9240

Hewlett-Packard Co.
560 Kirts Rd.
Suite 101
TROY, MI 48084
Tel: (313) 362-5180
Fax: (313) 362-3028

Minnesota

Hewlett-Packard Co.
2025 W. Larpenteur Ave.
ST. PAUL, MN 55113
Tel: (612) 644-1100
Fax: (612) 641-9787

Mississippi

Hewlett-Packard Co.
800 Woodland Parkway, Suite 101
RIDGELAND, MS 39157
Tel: (601) 957-0730
Fax: (601) 957-2515

Missouri

Hewlett-Packard Co.
13001 Hollenberg Drive
BRIDGETON 63044
Tel: (314) 344-5100
Fax: (314) 344-5273

Hewlett-Packard Co.
6601 Winchester Ave.
KANSAS CITY, MO 64133
Tel: (816) 737-0071
Fax: (816) 737-4690

Montana

Hewlett-Packard Co.
13001 Hollenberg Drive
BRIDGETON, MT 63044
Tel: (314) 344-5100
Fax: (314) 344-5273

Nebraska

Hewlett-Packard
11626 Nicholas St.
OMAHA, NE 68154
Tel: (402) 493-0300
Fax: (402) 493-4334

New Jersey

Hewlett-Packard Co.
120 W. Century Road
PARAMUS, NJ 07653
Tel: (201) 599-5000
Fax: (201) 599-5382

Hewlett-Packard Co.
10 Sylvan Way
PARSIPPANY, NJ 07054
Tel: (201) 682-4000
Fax: (201) 682-4031

Hewlett-Packard Co.
20 New England Av.
PISCATAWAY, NJ 08854
Tel: (201) 562-6100
Fax: (201) 562-6246

New Mexico

Hewlett-Packard Co.
7801 Jefferson N.E.
ALBUQUERQUE, NM 87109
Tel: (505) 823-6100
Fax: (505) 823-1243

Hewlett-Packard Co.
1362-C Trinity Dr.
LOS ALAMOS, NM 87544
Tel: (505) 662-6700
Fax: (505) 662-4312

New York

Hewlett-Packard Co.
5 Computer Drive South
ALBANY, NY 12205
Tel: (518) 458-1550
Fax: (518) 458-1550 x 0393

Hewlett-Packard Co.
130 John Muir Dr.
AMHERST, NY 14228
Tel: (716) 689-3003
Fax: (716) 636-7034

Hewlett-Packard Co.
200 Cross Keys Office Park
FAIRPORT, NY 14450
Tel: (716) 223-9950
Fax: (716) 223-6331

Hewlett-Packard Co.
7641 Henry Clay Blvd.
LIVERPOOL, NY 13088
Tel: (315) 451-1820
Fax: (315) 451-1820 x 255

Hewlett-Packard Co.
No. 1 Pennsylvania Plaza
55th Floor
34th Street & 7th Avenue
MANHATTAN NY 10119
Tel: (212) 971-0800
Fax: (212) 330-6967

Hewlett-Packard Co.
2975 Westchester Ave
PURCHASE, NY 10577
Tel: (914) 935-6300
Fax: (914) 935-6497

Hewlett-Packard Co.
Executive Square Office Bldg.
66 Middlebush Rd.
WAPPINGERS FALLS, NY 12590
Tel: (914) 298-9125
Fax: (914) 298-9154

Hewlett-Packard Co.
3 Crossways Park West
WOODBURY, NY 11797
Tel: (516) 682-7800
Fax: (516) 682-7806 (2)

North Carolina

Hewlett-Packard Co.
305 Gregson Dr.
CARY, NC 27511
Tel: (919) 467-6600
Fax: (919) 460-2296
(919) 460-2297

Hewlett-Packard Co.
P.O. Box 240318
CHARLOTTE, NC 28224
Tel: (704) 527-8780
Fax: (704) 523-7857

Hewlett-Packard Co.
7029 Albert Pick Rd. #100
GREENSBORO, NC 27409
Tel: (919) 665-1800
Fax: (919) 668-1797
Mailing Address
PO Box 26500
Greensboro, NC 27426

Ohio

Hewlett-Packard Co.
2717 S. Arlington Rd.
AKRON 44312
Tel: (216) 644-2270
Fax: (216) 644-7415

Hewlett-Packard Co.
4501 Erskine Road
CINCINNATI, OH 45242
Tel: (513) 891-9870
Fax: (513) 891-0033

SALES OFFICES
Arranged alphabetically by country (cont'd)

UNITED STATES (Cont'd)

Hewlett-Packard Co.
Moutroffe West Ave.
COPLEY, OH 44321
Tel: (216) 666-7711
Fax: (216) 666-6054

Hewlett-Packard Co.
7887 Washington Village Dr.
DAYTON, OH 45459
Tel: (513) 433-2223
Fax: (513) 433-8633

Hewlett-Packard Co.
9080 Springboro Pike
MIAMISBURG 45342
Tel: (513) 433-2223
Fax: (513) 433-3633

Hewlett-Packard Co.
15885 Sprague Road
STRONGSVILLE, OH 44136
Tel: (216) 243-7300
Fax: (216) 234-7230

Hewlett-Packard Co.
One Maritime Plaza, 5th Floor
720 Water Street
TOLEDO, OH 43604
Tel: (419) 242-2200
Fax: (419) 241-7655

Hewlett-Packard Co.
675 Brooksedge Blvd.
WESTERVILLE, OH 43081
Tel: (614) 891-3344
Fax: (614) 891-1476

Oklahoma

Hewlett-Packard Co.
3525 N.W. 56th St.
Suite C-100
OKLAHOMA CITY, OK 73112
Tel: (405) 946-9499
Fax: (405) 942-2127

Hewlett-Packard Co.
6655 South Lewis,
Suite 105
TULSA, OK 74136
Tel: (918) 481-6700
Fax: (918) 481-2250

Oregon

Hewlett-Packard Co.
9255 S. W. Pioneer Court
WILSONVILLE, OR 97070
Tel: (503) 682-8000
Fax: (503) 682-8155

Pennsylvania

Hewlett-Packard Co.
Heatherwood Industrial Park
50 Dorchester Rd.
P.O. Box 6080
HARRISBURG, PA 17112
Tel: (717) 657-5900
Fax: (717) 657-5946

Hewlett-Packard Co.
111 Zeta Drive
PITTSBURGH, PA 15238
Tel: (412) 782-0400
Fax: (412) 963-1300

Hewlett-Packard Co.
2750 Monroe Boulevard
VALLEY FORGE, PA 19482
Tel: (215) 666-9000
Fax: (215) 666-2034

South Carolina

Hewlett-Packard Co.
Brookside Park, Suite 122
1 Harbison Way
COLUMBIA, SC 29212
Tel: (803) 732-0400
Fax: (803) 732-4567

Hewlett-Packard Co.
545 N. Pleasantburg Dr.
Suite 100
GREENVILLE, SC 29607
Tel: (803) 232-8002
Fax: (803) 232-8739

Tennessee

Hewlett-Packard Co.
One Energy Center Suite 200
Pellissippi Pkwy.
KNOXVILLE, TN 37932
Tel: (615) 966-4747
Fax: (615) 966-8147

Hewlett-Packard Co.
889 Ridge Lake Blvd.,
Suite 100
MEMPHIS, TN 38119
Tel: (901) 763-4747
Fax: (901) 762-9723

Hewlett-Packard Co.
44 Vantage Way,
Suite 160
NASHVILLE, TN 37228
Tel: (615) 255-1271
Fax: (615) 726-2310

Texas

Hewlett-Packard Co.
9050 Capital of Texas Highway, North
#290
AUSTIN, TX 78759
Tel: (512) 346-3855
Fax: (512) 338-7201
Mailing Address
PO Box 9431
Austin, TX 78766-9430

Hewlett-Packard Co.
5700 Cromo Dr
EL PASO, TX 79912
Tel: (915) 833-4400
Fax: (915) 581-8097

Hewlett-Packard Co.
10535 Harwin Drive
HOUSTON, TX 77036
Tel: (713) 776-6400
Fax: (713) 776-6495

Hewlett-Packard Co.
3301 West Royal Lane
IRVING, TX 75063
Tel: (214) 869-3377
Fax: (214) 830-8951

Hewlett-Packard Co.
109 E. Toronto, Suite 100
McALLEN, TX 78503
Tel: (512) 630-3030
Fax: (512) 630-1355

Hewlett-Packard Co.
930 E. Campbell Rd.
RICHARDSON, TX 75081
Tel: (214) 231-6101
Fax: (214) 699-4337

Hewlett-Packard Co.
14100 San Pedro Ave., Suite 100
SAN ANTONIO, TX 78232
Tel: (512) 494-9336
Fax: (512) 491-1299

Utah

Hewlett-Packard Co.
3530 W. 2100 South
SALT LAKE CITY, UT 84119
Tel: (801) 974-1700
Fax: (801) 974-1780

Virginia

Hewlett-Packard Co.
840 Greenbrier Circle
Suite 101
CHESAPEAKE, VA 23320
Tel: (804) 424-7105
Fax: (804) 424-1494

Hewlett-Packard Co.
4401 Water Front Dr.
GLEN ALLEN, VA 23060
Tel: (804) 747-7750
Fax: (804) 965-9297

Hewlett-Packard Co.
2800 Electric Road Suite 100
ROANOKE, VA 24018
Tel: (703) 774-3444
Fax: (703) 989-8049

Washington

Hewlett-Packard Co.
15815 S.E. 37th Street
BELLEVUE, WA 98006
Tel: (206) 643-4000
Fax: (206) 643-8748

Hewlett-Packard Co.
N. 1225 Argonne Rd
SPOKANE, WA 99212-2657
Tel: (509) 922-7000
Fax: (509) 927-4236

West Virginia

Hewlett-Packard Co.
501 56th Street
CHARLESTON, WV 25304
Tel: (304) 925-0492
Fax: (304) 925-1910

Wisconsin

Hewlett-Packard Co.
275 N. Corporate Dr.
BROOKFIELD, WI 53005
Tel: (414) 792-8800
Fax: (414) 792-0218

URUGUAY

Pablo Ferrando S.A.C. e I.
Avenida Italia 2877
Casilla de Correo 370
MONTEVIDEO
Tel: 59-82-802-586
Telex: 398802586

Olympia de Uruguay S.A.
Maquines de Oficina
Avda. del Libertador 1997
Casilla de Correos 6644
MONTEVIDEO
Tel: 91-1809, 98-3807
Telex: 6342 OROU UY

VENEZUELA

Analytical Supplies, CA
Quinta #103 Impermes
Av El Centro
Los Chorros
Apartado 75472
CARACAS
Tel: 364904, 2394047
Telex: 26274 CABIC

Hewlett-Packard de Venezuela C.A.
Residencias Tia Betty Local 1
Avenida 3 Y con Calle 75
MARACAIBO, Estado Zulia
Apartado 2646
Tel: 586175669
Telex: 62464 HPMAR

YUGOSLAVIA

Do Hermes
General Zdanova 4
YU-11000 **BEOGRAD**
Tel: (011) 342 641
Telex: 11433

Do Hermes
Celovska 73
YU-61000 **LJUBLJANA**
Tel: (061) 553 170
Telex: 31583

ZAIRE

C.I.E.
Computer & Industrial Engineering
25 Ave. de la Justice Gombe
Boite Postale 10976
KINSHASA
Tel: 32 063, 32 633, 28 251
Telex: 21552
Fax: 22 850

ZAMBIA

R.J. Tilbury (Zambia) Ltd.
P.O. Box 32792
LUSAKA
Tel: 21 55 80
Telex: 40128

ZIMBABWE

Field Consolidated (Private) Ltd.
Systron Division
Manhattan Court
61 Second Street
P.O. Box 3458
HARARE
Tel: 73 98 81
Telex: 26241
Fax: 70 20 08

Please send directory corrections to:
Test & Measurement Catalog
Hewlett-Packard Company
3200 Hillview Avenue
Palo Alto, CA 94304
Tel: (415) 857-4706
Fax: (415) 857-3880

September 1989

Reader Comment Card

HP 9000 Computers
Programming and Protocols for NFS Services
B1013-90010 E0291

We welcome your evaluation of this manual. Your comments and suggestions will help us improve our publications. Please tear this card out and mail it in. Use and attach additional pages if necessary.

Please circle the following Yes or No:

• Is this manual well organized?	Yes	No
• Is the information technically accurate?	Yes	No
• Are instructions complete?	Yes	No
• Are concepts and wording easy to understand?	Yes	No
• Are examples and pictures helpful?	Yes	No
• Are there enough examples and pictures?	Yes	No

Comments:_____

_____

_____

_____

_____

Name:_____

Title:_____

Company:_____

Address:_____

City & State: _____

Zip:_____

HEWLETT PACKARD

Printed in USA

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO.

POSTAGE WILL BE PAID BY ADDRESSEE

Hewlett-Packard Company
Information Networks Division
19420 Homestead Road
Cupertino, CA 95014

ATTN: Network Usability Department

Fold Here

Tape

Please do not staple

Tape

Printed in England 02/91

Customer Order No.
B1013-90010

Manufacturing No.
B1013-91010
Mfg. number is for HP internal use only

B1013-91010