VMS Software

VSI TCP/IP Services for OpenVMS ONC RPC Programming

Operating System and Version: VSI OpenVMS IA-64 Version 8.4-1H1 or higher
VSI OpenVMS Alpha Version 8.4-2L1 or higher

Software Version: VSI TCP/IP Services Version 5.7

VMS Software, Inc. (VSI)
Boston, Massachusetts, USA

VSI TCP/IP Services for OpenVMS ONC RPC Programming

Legal Notice

Table of Contents

Preface ix
1. About VSI ix
2. Intended Audience ix
3. Document Structure ix
4. Related Documents x
5. OpenVMS Documentation xi
6. VSI Encourages Your Comments xi
7. Conventions xi

Chapter 1. Introduction to Remote Procedure Calls 1
1.1. Overview 1
1.2. The RPC Model 1
1.3. RPC Procedure Versions 2
1.4. Using Portmapper to Determine the Destination Port Number of RPC Packets 2
1.4.1. Portmapper Notes for TCP/IP Services 3
1.4.2. Displaying Registered RPC Servers 3
1.5. RPC Independence from Transport Protocol 3
1.6. External Data Representation (XDR) 4
1.7. Assigning Program Numbers 5

Chapter 2. Writing RPC Applications with the RPCGEN Protocol Compiler 7
2.1. The RPCGEN Protocol Compiler 7
2.2. Simple Example: Using RPCGEN to Generate Client and Server RPC Code 7
2.2.1. RPC Protocol Specification File Describing Remote Procedure 9
2.2.2. Implementing the Procedure Declared in the Protocol Specification 10
2.2.3. The Client Program That Calls the Remote Procedure 11
2.2.4. Running RPCGEN 13
2.2.5. Compiling the Client and Server Programs 14
2.2.6. Copying the Server to a Remote System and Running It 14
2.3. Advanced Example: Using RPCGEN to Generate XDR Routines 14
2.3.1. The RPC Protocol Specification 15
2.3.2. Implementing the Procedure Declared in the Protocol Specification 16
2.3.3. The Client Program that Calls the Remote Procedure 18
2.3.4. Running RPCGEN 19
2.3.5. Compiling the File of XDR Routines 20
2.3.6. Compiling the Client and Server Programs 20
2.3.7. Copying the Server to a Remote System and Running It 20
2.4. Debugging Applications 21
2.5. The C Preprocessor 21
2.6. RPCGEN Programming 22
2.6.1. Network Types 22
2.6.2. User-Provided Define Statements 22
2.6.3. INETd Support 23
2.6.4. Dispatch Tables 23
2.7. Client Programming 24
2.7.1. Timeout Changes 24
2.7.2. Client Authentication 25
2.8. Server Programming 25
2.8.1. Handling Broadcasts 25
2.8.2. Passing Data to Server Procedures 26
2.9. RPC and XDR Languages 26

2.9.1. Definitions 26
2.9.2. Enumerations 27
2.9.3. Typedefs 27
2.9.4. Constants 28
2.9.5. Declarations 28
2.9.6. Structures 29
2.9.7. Unions 30
2.9.8. Programs 30
2.9.9. Special Cases 31
2.10. Command Reference 32
Chapter 3. RPC Application Programming Interface 37
3.1. RPC Layers 37
3.2. Middle Layer of RPC 38
3.2.1. Using callrpc 38
3.2.2. Using registerrpc and svc_run 39
3.2.3. Using XDR Routines to Pass Arbitrary Data Types 41
3.2.4. User-Defined XDR Routines 41
3.2.5. XDR Serializing Defaults 43
3.3. Lowest Layer of RPC 43
3.3.1. The Server Side and the Lowest RPC Layer 43
3.3.2. The Client Side and the Lowest RPC Layer 46
3.3.3. Memory Allocation with XDR 48
3.4. Raw RPC 49
3.5. Miscellaneous RPC Features 51
3.5.1. Using Select on the Server Side 51
3.5.2. Broadcast RPC 52
3.5.3. Batching 53
3.6. Authentication of RPC Calls 57
3.6.1. The Client Side 57
3.6.2. The Server Side 58
3.7. Using the Internet Service Daemon (INETd) 60
3.8. Additional Examples 61
3.8.1. Program Versions on the Server Side 61
3.8.2. Program Versions on the Client Side 62
3.8.3. Using the TCP Transport 64
3.8.4. Callback Procedures 68
Chapter 4. External Data Representation 73
4.1. Usefulness of XDR 73
4.1.1. A Canonical Standard 76
4.1.2. The XDR Library 76
4.2. XDR Library Primitives 78
4.2.1. Number and Single-Character Filters 78
4.2.2. Floating-Point Filters 79
4.2.3. Enumeration Filters 80
4.2.4. Possibility of No Data 80
4.2.5. Constructed Data Type Filters 80
4.2.5.1. Strings 80
4.2.5.2. Variable-Length Byte Arrays 81
4.2.5.3. Variable-Length Arrays of Arbitrary Data Elements 81
4.2.5.4. Fixed-Length Arrays of Arbitrary Data Elements 84
4.2.5.5. Opaque Data 84

4.2.5.6. Discriminated Unions 85
4.2.5.7. Pointers 86
4.2.6. Non-filter Primitives 87
4.3. XDR Operation Directions 88
4.4. XDR Stream Access 88
4.4.1. Standard I/O Streams 88
4.4.2. Memory Streams 88
4.4.3. Record (TCP/IP) Streams 89
4.4.4. XDR Stream Implementation 90
4.5. Advanced Topics 91

Chapter 5. ONC RPC Client Routines 95

auth_destroy 96
authnone_create 96
authunix_create 97
authunix_create_default 98
callrpc 99
clnt_broadcast 100
clnt_call 101
clnt_control 102
clnt_create 103
clnt_create_vers 105
clnt_destroy 106
clnt_freeres 107
clnt_geterr 107
clnt_pcreateerror 108
clnt_perrno 108
clnt_perror 109
clnt_spcreateerror 109
clnt_sperrno 110
clnt_sperror 111
clntraw_create 112
clnttcp_create 113
clntudp_bufcreate 114
clntudp_create 115
get_myaddress 117
get_myaddr_dest 117

Chapter 6. ONC RPC Portmapper Routines 119

pmap_getmaps 119
pmap_getmaps_vms 120
pmap_getport 120
pmap_rmtcall 121
pmap_set 123
pmap_unset 123

Chapter 7. ONC RPC Server Routines 125

registerrpc 126
seterr_reply 127
svc_destroy 128
svc_freeargs 128
svc_getargs 129
svc_getcaller 129
svc_getreqset 130

svc_register 131
svc_run 132
svc_sendreply 133
svc_unregister 133
svcerr_auth 134
svcerr_decode 135
svcerr_noproc 135
svcerr_noprog 136
svcerr_progvers 136
svcerr_systemerr 137
svcerr_weakauth 137
svcraw_create 138
svcfd_create 138
svctcp_create 139
svcudp_bufcreate 140
svcudp_create 141
xprt_register 141
xprt_unregister 142
_authenticate 142

Chapter 8. XDR Routine Reference 145

xdr_accepted_reply 146
xdr_array 147
xdr_authunix_parms 148
xdr_bool 149
xdr_bytes 149
xdr_callhdr 150
xdr_callmsg 151
xdr_char 151
xdr_double 152
xdr_enum 153
xdr_float 153
xdr_free 154
xdr_hyper 155
xdr_int 155
xdr_long 156
xdr_opaque 156
xdr_opaque_auth 157
xdr_pmap 158
xdr_pmap_vms 158
xdr_pmaplist 159
xdr_pmaplist_vms 159
xdr_pointer 160
xdr_reference 161
xdr_rejected_reply 162
xdr_replymsg 162
xdr_short 163
xdr_string 163
xdr_u_char 164
xdr_u_hyper 165
xdr_u_int 165
xdr_u_long 166
xdr_u_short 167

xdr_union 167
xdr_vector 168
xdr_void 169
xdr_wrapstring 169
xdrmem_create 170
xdrrec_create 171
xdrrec_endofrecord 172
xdrrec_eof 173
xdrrec_skiprecord 173
xdrstdio_create 174

Preface

The TCP/IP Services product is the VSI implementation of the TCP/IP networking protocol suite and Internet services for OpenVMS I64, Alpha, and VAX systems.

TCP/IP Services provides a comprehensive suite of functions and applications that support industry-standard protocols for heterogeneous network communications and resource sharing.

This *VSI TCP/IP Services for OpenVMS ONC RPC Programming* manual presents an overview of high-level programming using open network computing remote procedure calls (ONC RPCs). This manual also describes the RPC programming interface and how to use the RPCGEN protocol compiler to create applications.

See the *VSI TCP/IP Services for OpenVMS Installation and Configuration* manual for information about installing, configuring, and starting this product.

1. About VSI

VMS Software, Inc. (VSI) is an independent software company licensed by Hewlett Packard Enterprise to develop and support the OpenVMS operating system.

2. Intended Audience

This manual assumes a knowledge of network theory and is for experienced programmers who want to write network applications using ONC RPC without needing to know about the underlying network.

3. Document Structure

This manual contains eight chapters:

Chapter 1, "Introduction to Remote Procedure Calls"	Provides an overview of high-level programming through remote procedure calls (RPC), and discusses the RPC mode land versions, external data representation, and RPC independence from network transport protocol. This chapter is for anyone interested in ONC RPC.
Chapter 2, "Writing RPC Applications with the RPCGEN Protocol Compiler "	Describes how to write RPC client and server applications with the RPCGEN protocol compiler. It also provides some information on RPCGEN, client and server programming, debugging applications, the C preprocessor, and RPC language syntax. This chapter also describes how to create routines for external data representation (XDR). This chapter is for programmers who want to use RPCGEN to write RPC-based network applications.
Chapter 3, "RPC Application Programming Interface"	Describes the RPC programming interface layers, XDR serialization defaults, raw RPC, and miscellaneous RPC features.

	This chapter is for programmers who need to understand RPC mechanisms to write customized network applications.
Chapter 4, "External Data Representation"	Contains information about the XDR library. This chapter is for programmers who want to implement RPC and XDR on new systems.
Chapter 5, "ONC RPC Client Routines"	Contains descriptions of each of the RPC subroutine calls commonly used by client programs.
Chapter 6, "ONC RPC Portmapper Routines"	Contains descriptions of each of the RPC subroutine calls used by both client and server programs to access the Portmapper service.
Chapter 7, "ONC RPC Server Routines"	Contains descriptions of each of the RPC subroutine calls commonly used by client programs.
Chapter 8, "XDR Routine Reference"	Contains descriptions of each of the XDR subroutine calls.

4. Related Documents

The table below lists the documents available with this version of TCP/IP Services.

Table 1. TCP/IP Services Documentation

Manual	**Contents**
VSI TCP/IP Services for OpenVMS Concepts and Planning	This manual provides conceptual information about TCP/IP networking on OpenVMS systems, including general planning issues to consider before configuring your system to use the TCP/IP Services software. This manual also describes the manuals in the TCP/IP Services documentation set and provides a glossary of terms and acronyms for the TCP/IP Services software product.
VSI TCP/IP Services for OpenVMS Installation and Configuration	This manual explains how to install and configure the TCP/IP Services product.
VSI TCP/IP Services for OpenVMS User's Guide	This manual describes how to use the applications available with TCP/IP Services such as remote file operations, email, TELNET, TN3270, and network printing.
VSI TCP/IP Services for OpenVMS Management	This manual describes how to configure and manage the TCP/IP Services product.
VSI TCP/IP Services for OpenVMS Management Command Reference	This manual describes the TCP/IP Services management commands.
VSI TCP/IP Services for OpenVMS ONC RPC Programming	This manual presents an overview of high-level programming using open network computing remote procedure calls (ONC RPCs). This manual

Manual	Contents
	also describes the RPC programming interface and how to use the RPCGEN protocol compiler to create applications.
VSI TCP/IP Services for OpenVMS Sockets API and System Services Programming	This manual describes how to use the Sockets API and OpenVMS system services to develop network applications.
VSI TCP/IP Services for OpenVMS SNMP Programming and Reference	This manual describes the Simple Network Management Protocol (SNMP) and the SNMP application programming interface (eSNMP). It describes the subagents provided with TCP/IP Services, utilities provided for managing subagents, and how to build your own subagents.
VSI TCP/IP Services for OpenVMS Guide to IPv6	This manual describes the IPv6 environment, the roles of systems in this environment, the types and function of the different IPv6 addresses, and how to configure TCP/IP Services to access the IPv6 network.

For a comprehensive overview of the TCP/IP protocol suite, refer to the book *Internetworking with TCP/IP:Principles, Protocols, and Architecture*, by Douglas Comer.

5. OpenVMS Documentation

The full VSI OpenVMS documentation set can be found on the VMS Software Documentation webpage at https://docs.vmssoftware.com.

6. VSI Encourages Your Comments

You may send comments or suggestions regarding this manual or any VSI document by sending electronic mail to the following Internet address: <docinfo@vmssoftware.com>. Users who have VSI OpenVMS support contracts through VSI can contact <support@vmssoftware.com> for help with this product.

7. Conventions

The following conventions may be used in this manual:

Convention	Meaning
Ctrl/ *x*	A sequence such as **Ctrl/** *x* indicates that you must hold down the key labeled Ctrl while you press another key or a pointing device button.
PF1 *x*	A sequence such as PF1 *x* indicates that you must first press and release the key labeled PF1 and then press and release another key or a pointing device button.
Return	In examples, a key name enclosed in a box indicates that you press a key on the keyboard. (In text, a key name is not enclosed in a box.)
. . .	A horizontal ellipsis in examples indicates one of the following possibilities: ● Additional optional arguments in a statement have been omitted.

Convention	Meaning
	• The preceding item or items can be repeated one or more times. • Additional parameters, values, or other information can be entered.
. . .	A vertical ellipsis indicates the omission of items from a code example or command format; the items are omitted because they are not important to the topic being discussed.
( )	In command format descriptions, parentheses indicate that you must enclose the options in parentheses if you choose more than one.
[ ]	In command format descriptions, brackets indicate optional choices. You can choose one or more items or no items. Do not type the brackets on the command line. However, you must include the brackets in the syntax for OpenVMS directory specifications and for a substring specification in an assignment statement.
[|]	In command format descriptions, vertical bars separate choices within brackets or braces. Within brackets, the choices are options; within braces, at least one choice is required. Do not type the vertical bars on the command line.
{ }	In command format descriptions, braces indicate required choices; you must choose at least one of the items listed. Do not type the braces on the command line.
bold text	This typeface represents the introduction of a new term. It also represents the name of an argument, an attribute, or a reason.
italic text	Italic text indicates important information, complete titles of manuals, or variables. Variables include information that varies in system output (Internal error *number*), in command lines (/PRODUCER= *name*), and in command parameters in text (where *dd* represents the predefined code for the device type).
UPPERCASE TEXT	Uppercase text indicates a command, the name of a routine, the name of a file, or the abbreviation for a system privilege.
`Monospace type`	Monospace type indicates code examples and interactive screen displays. In the C programming language, monospace type in text identifies the following elements: keywords, the names of independently compiled external functions and files, syntax summaries, and references to variables or identifiers introduced in an example.
-	A hyphen at the end of a command format description, command line, or code line indicates that the command or statement continues on the following line.
numbers	All numbers in text are assumed to be decimal unless otherwise noted. Nondecimal radixes—binary, octal, or hexadecimal—are explicitly indicated.

Other conventions are:

- All numbers are decimal unless otherwise noted.
- All Ethernet addresses are hexadecimal.

Chapter 1. Introduction to Remote Procedure Calls

1.1. Overview

High-level programming through open network computing remote procedure calls (ONC RPC) provides logical client-to-server communication for network application development – without the need to program most of the interface to the underlying network. With RPC, the client makes a remote procedure call that sends requests to the server, which calls a dispatch routine, performs the requested service, and sends back a reply before the call returns to the client.

RPC does not require the client to be knowledgeable about the underlying network. For example, a program can simply call a local C routine that returns the number of users on a remote system much like making a system call. You can make remote procedure calls between different processes on the same system.

1.2. The RPC Model

The remote procedure call model is similar to that of the local model, which works as follows:

1. The caller places arguments to a procedure in a specific location (such as an argument variable).
2. The caller temporarily transfers control to the procedure.
3. When the caller gains control again, it obtains the results of the procedure from the specified location.
4. The caller then continues program execution.

As *Figure 1.1, "Basic Network Communication with Remote Procedure Call"* shows, the remote procedure call is similar to the local model, in that one thread of control logically winds through two processes – that of the client (caller) and that of the server:

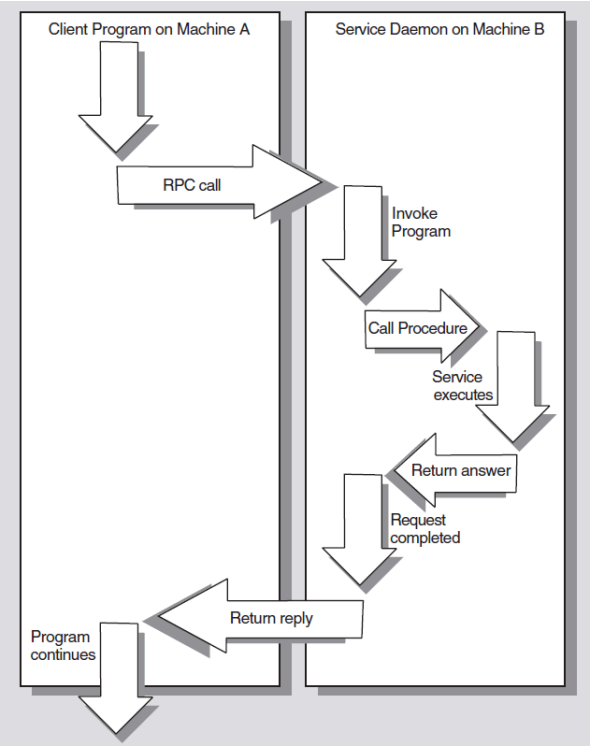
1. The client process sends a call message to the server process and blocks (that is, waits) for a reply message. The call message contains the parameters of the procedure and the reply message contains the procedure results.
2. When the client receives the reply message, it gets the results of the procedure.
3. The client process then continues executing.

On the server side, a process is dormant – awaiting the arrival of a call message. When one arrives, the server process computes a reply that it then sends back to the requesting client. After this, the server process becomes dormant again.

Figure 1.1, "Basic Network Communication with Remote Procedure Call" shows a synchronous RPC call, in which only one of the two processes is active at a given time. The remote procedure call hides the details of the network transport. However, the RPC protocol does not restrict the concurrency model. For example, RPC calls may be asynchronous so the client can do another task while waiting for the reply from the server. Another possibility is that the server could create a task to process a certain type

of request automatically, freeing it to service other requests. Although RPC provides a way to avoid programming the underlying network transport, it still allows this where necessary.

Figure 1.1. Basic Network Communication with Remote Procedure Call



1.3. RPC Procedure Versions

Each RPC procedure is defined uniquely by program and procedure numbers. The program number specifies 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 (adding a new procedure, for example), a new program number does not have to be assigned. When you want to call a procedure to find the number of remote users, you must know the appropriate program, version, and procedure numbers to use to contact the service. You can find this information in several places. On UNIX systems, the `/etc/rpc` file lists some RPC programs and the RPCINFO command lists the registered RPC programs and corresponding version numbers running on a particular system. On OpenVMS systems, the SHOW PORTMAPPER management command serves the same purpose as the RPCINFO command.

Typically, a service provides a protocol description so you can write client applications that call the service. The RPC Administrator at Sun Microsystems, Inc. has a list of programs that have been registered with Sun (that is, have received port numbers from them), but you can write your own local RPC programs. Knowing the program and procedure numbers is useful only if the program is running on a system to which you have access.

1.4. Using Portmapper to Determine the Destination Port Number of RPC Packets

The TCP/IP Services software starts the Portmapper network service when it receives the first network request for the Portmapper port. Interaction between RPC programs and the Portmapper occurs as follows:

1. After the system manager starts the Portmapper, it listens for UDP and TCP requests on port 111 of the host system.

2. When an RPC server program activates on a system, it registers itself with its local Portmapper. The Portmapper software keeps a table of all registered services.

3. To access the services available on a system, RPC client programs send RPC call messages to a system's Portmapper specifying the program and version number with which they wish to communicate.

4. The Portmapper program examines its local cache of registered RPC servers. If the server is registered, then the Portmapper uses an RPC reply message to return the port number that the RPC client program should use to communicate with the RPC server.

5. The RPC client program then uses the provided port number in all subsequent RPC calls.

Refer to the *VSI TCP/IP Services for OpenVMS Management* manual for more information about the Portmapper service.

1.4.1. Portmapper Notes for TCP/IP Services

The Portmapper service on TCP/IP Services differs from Portmapper software on other hosts in the following ways:

- When an RPC server that is registered with the Portmapper exits, the Portmapper purges any registrations for that server program.

- An RPC process can only register or unregister its own Portmapper entries. Any attempt to remove a registration for another RPC server will fail.

- The Portmapper includes its own mappings (on the UDP and TCP port 111). These mappings are available using the `pmap_getmaps` routine.

- All data structures used for the RPC `pmap_`*`xxxx`* routines are identical to other RPC implementations with the exception of the two additional structures `pmap_vms` and `pmaplist_vms`. These structures include the field `pm_pid` which is the OpenVMS process ID.

1.4.2. Displaying Registered RPC Servers

You can display current RPC registration information known to the Portmapper program. On UNIX systems use the `rpcinfo` command. On OpenVMS systems use the SHOW PORTMAPPER management command. The `rpcinfo` or SHOW PORTMAPPER commands can also find the RPC services registered on a specific host and report their port numbers and the transports for which the services are registered. For more information, see the *VSI TCP/IP Services for OpenVMS Management Command Reference* manual.

1.5. RPC Independence from Transport Protocol

The RPC protocol is concerned only with the specification and interpretation of messages; it is independent of transport protocols because it needs no information on how a message is passed among processes.

Also, RPC does not implement any kind of reliability; the application itself must be aware of the transport protocol type underlying RPC. With a reliable transport, such as TCP/IP, the application need not do much else. However, an application must use its own retransmission and timeout policy if it is running on top of an unreliable transport, such as UDP/IP.

Because of transport independence, the RPC protocol does not actively interpret anything about remote procedures or their execution. Instead, the application infers required information from the underlying protocol (where such information should be specified explicitly). For example, if RPC is running on top of an unreliable transport (such as UDP/IP) and the application retransmits RPC messages after short timeouts, and if the application receives no reply, then it can infer only that a certain procedure was executed zero or more times. If it receives a reply, then the application infers that the procedure was executed at least once.

With a reliable transport, such as TCP/IP, the application can infer from a reply message that the procedure was executed exactly once, but if it receives no reply message, it cannot assume the remote procedure was not executed.

Note

Even with a connection-oriented protocol such as TCP, an application still needs timeouts and reconnection procedures to handle server crashes.

ONC RPC is currently supported on both UDP/IP and TCP/IP transports. The selection of the transport depends on the application requirements. The UDP transport, which is connectionless, is a good choice if the application has the following characteristics:

- The procedures are idempotent; that is, the same procedure can be executed more than once without any side effects. For example, reading a block of data is idempotent; creating a file is not.
- The size of both the arguments and results is smaller than the UDP packet size of 8K bytes.
- The server is required to handle as many as several hundred clients. The UDP server can do so because it does not retain any information about the client state. By contrast, the TCP server holds state information for each open client connection and this limits its available resources.

TCP (connection-oriented) is a good transport choice if the application has any of the following characteristics:

- The application needs a reliable underlying transport.
- The procedures are non-idempotent.
- The size of either the arguments or the results exceeds 8K bytes.

1.6. External Data Representation (XDR)

RPC can handle arbitrary data structures, regardless of the byte order or structure layout convention on a particular system. It does this by converting them to a network standard called external data representation (XDR) before sending them over the network. XDR is a system-independent description and encoding of data that can communicate between diverse systems, such as a VAX, Sun workstation, IBM PC, or CRAY.

Converting from a particular system representation to XDR format is called serializing; the reverse process is deserializing.

1.7. Assigning Program Numbers

Program numbers are assigned in groups of 0x20000000 according to the following chart:

`0x00000000` – `0x1fffffff`	Defined by Sun Microsystems
`0x20000000` – `0x3fffffff`	Defined by user
`0x40000000` – `0x5fffffff`	Transient
`0x60000000` – `0x7fffffff`	Reserved
`0x80000000` – `0x9fffffff`	Reserved
`0xa0000000` – `0xbfffffff`	Reserved
`0xc0000000` – `0xdfffffff`	Reserved
`0xe0000000` – `0xffffffff`	Reserved

Sun Microsystems administers the first range of numbers, which should be identical for all ONC RPC users. An ONC RPC application for general use should have an assigned number in this first range. The second range of numbers is for specific, user-defined customer applications, and is primarily for debugging new programs. The third, called the Transient group, is reserved for applications that generate program numbers dynamically. The final groups are reserved for future use, and are not used.

To register a protocol specification, send a request by network mail to rpc@sun.com, or write to:

```
RPC Administrator
Sun Microsystems
2550 Garcia Ave.
Mountain View, CA 94043
```

Include a compilable RPCGEN .X file describing your protocol. You will then receive a unique program number. See *Chapter 2, "Writing RPC Applications with the RPCGEN Protocol Compiler "* for more information about RPCGEN .X files.

Chapter 2. Writing RPC Applications with the RPCGEN Protocol Compiler

2.1. The RPCGEN Protocol Compiler

The RPCGEN protocol compiler accepts a remote program interface definition written in RPC language, which is similar to C. It then produces C language output consisting of: client skeleton routines, server skeleton routines, XDR filter routines for both arguments and results, a header file that contains common definitions, and optionally, dispatch tables that the server uses to invoke routines that are based on authorization checks.

The client skeleton interface to the RPC library hides the network from the client program, and the server skeleton hides the network from the server procedures invoked by remote clients. You compile and link output files from RPCGEN as usual. The server code generated by RPCGEN supports INETd. You can start the server using INETd or at the command line.

You can write server procedures in any language that has system calling conventions. To get an executable server program, link the server procedure with the server skeleton from RPCGEN. To create an executable client program, write an ordinary main program that makes local procedure calls to the client skeletons, and link the program with the client skeleton from RPCGEN. If necessary, the RPCGEN options enable you to suppress skeleton generation and specify the transport to be used by the server skeleton.

The RPCGEN protocol compiler helps to reduce development time in the following ways:

- It greatly reduces network interface programming.
- It can mix low-level code with high-level code.
- For speed-critical applications, you can link customized high-level code with the RPCGEN output.
- You can use RPCGEN output as a starting point, and rewrite as necessary.

Refer to the RPCGEN command description at the end of this chapter for more information about programming applications that use remote procedure calls or for writing XDR routines that convert procedure arguments and results into their network format (or vice versa). For a discussion of RPC programming without RPCGEN, see *Chapter 3, "RPC Application Programming Interface"*.

2.2. Simple Example: Using RPCGEN to Generate Client and Server RPC Code

This section shows how to convert a simple routine – one that prints messages to the system console on a single system (OPCOM on OpenVMS) – to an ONC RPC application that runs remotely over the network. To do this, the RPCGEN protocol compiler is used to generate client and server RPC code. *Example 2.1, "Printing a Remote Message Without ONC RPC"* (see file SYS$COMMON:[SYSHLP.EXAMPLES.TCPIP.RPC]PRINTMSG.C) shows the routine before conversion.

Compile and run the program shown in the example (you will need OPER privileges):

```
$ CC/DECC PRINTMSG
$ LINK PRINTMSG
$ MCR SYS$DISK:[]PRINTMSG "Red rubber ball"
%%%%%%%%%%  OPCOM  27-SEP-1995 14:39:22.59  %%%%%%%%%%
Message from user GEORGE on BOSTON
Red rubber ball

Message Delivered!
$
```

If the `printmessage` procedure at the bottom of the `printmsg.c` program of *Example 2.1, "Printing a Remote Message Without ONC RPC"* were converted into a remote procedure, you could call it from anywhere in the network, instead of only from the program where it is embedded. Before doing this, it is necessary to write a protocol specification in RPC language that describes the remote procedure, as shown in the next section.

Example 2.1. Printing a Remote Message Without ONC RPC

```
/*
** printmsg.c: OpenVMS print a message on the console
*/
#include <descrip.h>
#include <opcdef.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

extern int SYS$SNDOPR(struct dsc$descriptor_s *, unsigned short);

static int printmessage(char *);

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

    if (argc != 2) {
        fprintf(stderr, "usage: %s <message>\n", argv[0]);
        exit (1);
    }
    message = argv[1];

    if (!printmessage(message)) {
        fprintf(stderr,"%s: couldn't print your message\n", argv[0]);
        exit (1);
    }
    printf("Message Delivered!\n");
    exit (0);
}

/*
** Print a message to the console.  Return a Boolean indicating
** whether the message was actually printed.
*/
```

```
static int
printmessage(msg)
    char *msg;
{
    struct dsc$descriptor_s desc;
    union {
        char  buffer[256]; /* Preallocate space for text */
        struct opcdef opc;
        } message;
    int status;

    /*
     ** Build the message request block.
     */
    message.opc.opc$b_ms_type   = OPC$_RQ_RQST;
    message.opc.opc$b_ms_target = OPC$M_NM_CENTRL;
    message.opc.opc$w_ms_status = 0;
    message.opc.opc$l_ms_rqstid = 0;
    strcpy((char *) &message.opc.opc$l_ms_text, msg);
    desc.dsc$a_pointer = (char *) &message.opc;
    desc.dsc$w_length  = (char *) &message.opc.opc$l_ms_text -
                         (char *) &message +
                         strlen((char *) &message.opc.opc$l_ms_text);
    /*
     ** Send the message to the console.
     */
    status = SYS$SNDOPR(&desc,        /* MSGBUF */
                        0);           /* CHAN */
    if (status & 1)
        return 1;
    return 0;
}
```

2.2.1. RPC Protocol Specification File Describing Remote Procedure

To create the specification file, you must know all the input and output parameter types. In *Example 2.1, "Printing a Remote Message Without ONC RPC"*, the `printmessage` procedure takes a string as input, and returns an integer as output. *Example 2.2, "RPC Protocol Specification File Simple Example"* (see SYS$COMMON:[SYSHLP.EXAMPLES.TCPIP.RPC]MSG.X) is the RPC protocol specification file that describes the remote version of the `printmessage` procedure.

Remote procedures are part of remote programs, so *Example 2.2, "RPC Protocol Specification File Simple Example"* actually declares a remote program containing a single procedure, `PRINTMESSAGE`. By convention, all RPC services provide for a NULL procedure (procedure 0), normally used for pinging. The RPC protocol specification file in *Example 2.2, "RPC Protocol Specification File Simple Example"* declares the `PRINTMESSAGE` procedure to be in version 1 of the remote program. No NULL procedure (procedure 0) is necessary in the protocol definition because RPCGEN generates it automatically.

In RPC language, the convention (though not a requirement) is to make all declarations in uppercase characters. Notice that the argument type is `string`, not `char *`, because a `char *` in C is ambiguous. Programmers usually intend it to mean a null-terminated string of characters, but it could also be a pointer to a single character or to an array of characters. In RPC language, a null-terminated string is unambiguously of type `string`.

Example 2.2. RPC Protocol Specification File Simple Example

```
/*
 * msg.x: Remote message printing protocol
 */
program MESSAGEPROG {
     version MESSAGEVERS {
          int PRINTMESSAGE(string) = 1;
     } = 1;
} = 0x20000099;
```

2.2.2. Implementing the Procedure Declared in the Protocol Specification

Example 2.3, "Remote Procedure Definition" (see SYS$COMMON: [SYSHLP.EXAMPLES.TCPIP.RPC]MSG_SERVER.C) defines the remote procedure declared in the RPC protocol specification file of the previous example.

Example 2.3. Remote Procedure Definition

```
/*
** msg_server.c: OpenVMS implementation of the remote procedure
** "printmessage"
*/

#include <descrip.h>  /* OpenVMS descriptor definitions */
#include <opcdef.h>   /* OpenVMS $SNDOPR() definitions */
#include <rpc/rpc.h>  /* always needed */ ❶
#include "msg.h"            /* msg.h will be generated by RPCGEN */

extern int SYS$SNDOPR(struct dsc$descriptor_s *, unsigned short);

/*
** Remote version of "printmessage"
*/
int *
printmessage_1(msg) ❷
    char **msg; ❸
{
    struct dsc$descriptor_s desc;
    union {
        char  buffer[256]; /* Preallocate space for text */
        struct opcdef opc;
        } message;
    static int result;
    int status;

    /*
    ** Build the message request block.
    */
    message.opc.opc$b_ms_type   = OPC$_RQ_RQST;
    message.opc.opc$b_ms_target = OPC$M_NM_CENTRL;
    message.opc.opc$w_ms_status = 0;
    message.opc.opc$l_ms_rqstid = 0;
    strcpy((char *) &message.opc.opc$l_ms_text, *msg);
    desc.dsc$a_pointer = (char *) &message.opc;
```

```
    desc.dsc$w_length  = (char *) &message.opc.opc$l_ms_text -
                         (char *) &message +
                         strlen((char *) &message.opc.opc$l_ms_text);
    status = SYS$SNDOPR(&desc,        /* MSGBUF */
                        0);           /* CHAN */
    if (status & 1)
        result = 1;
    else
        result = 0;
    return &result; ❹
}
```

In this example, the declaration of the remote procedure, `printmessage_1`, differs from that of the local procedure `printmessage` in four ways:

❶ It includes the `<rpc/rpc.h>` file and the `"msg.h"` header files. The `rpc/rpc.h` file is located in the directory TCPIP$RPC:. To ensure portability in header files references, most of the examples in this manual assume you have defined the symbol RPC to be equal to TCPIP$RPC:

```
$ DEFINE RPC TCPIP$RPC:
```

before using the RPCGEN compiler and the DECC compiler.

❷ It has `_1` appended to its name. In general, all remote procedures called by RPCGEN skeleton routines are named by the following rule: The name in the procedure definition (here, `PRINTMESSAGE`) is converted to all lowercase letters, and an underscore (_) and version number (here, 1) is appended to it.

❸ 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; if there are no arguments, specify `void`.

❹ It returns a pointer to an integer instead of an integer itself. This is also characteristic of remote procedures – they return pointers to their results. Therefore, it is important to have the result declared as a `static`; if there are no arguments, specify `void`.

2.2.3. The Client Program That Calls the Remote Procedure

Example 2.4, "Client Program that Calls the Remote Procedure" declares the main client program, `rprintmsg.c`, that calls the remote procedure. (See SYS$COMMON: [SYSHLP.EXAMPLES.TCPIP.RPC]RPRINTMSG.C.)

Example 2.4. Client Program that Calls the Remote Procedure

```
/*
** rprintmsg.c: remote OpenVMS version of "printmsg.c"
*/

#include
<stdio.h>
#include
<rpc/rpc.h>     /* always needed  */
#include "msg.h"         /* msg.h will be generated by RPCGEN */
```

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

    if (argc != 3) {
        fprintf(stderr, "usage: %s host message\n", argv[0]);
        exit(1);
        }
    server = argv[1];
    message = argv[2];

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

    /*
    ** Call the remote procedure "printmessage" on the server
    */
    result = printmessage_1(&message, cl);
❷

    if (result == NULL) {
❸
        /*
        ** An error occurred while calling the server.
        ** Print error message and stop.
        */
        clnt_perror(cl, server);
        exit(1);
        }

    /*
    ** Okay, we successfully called the remote procedure.
    */
    if (*result == 0) {
❹
        /*
        ** Server was unable to print our message.
        ** Print error message and stop.
```

```
        */
        fprintf(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);
    exit(0);
    }
```

In this example, the following events occur:

❶ First, the RPC library routine `clnt_create` creates a client "handle." The last parameter to `clnt_create` is `"tcp"`, the transport on which you want to run your application. (Alternatively, you could have used `"udp"`.)

❷ Next, the program calls the remote procedure `printmessage_1` in exactly the same way as specified in `msg_server.c`, except for the inserted client handle as the second argument.

❸ The remote procedure call can fail in two ways: The RPC mechanism itself can fail or there can be an error in the execution of the remote procedure. In the former case, the remote procedure, `printmessage_1`, returns `NULL`.

❹ In the later case, error reporting is application-dependent. In this example, the remote procedure reports any error via `*result`.

2.2.4. Running RPCGEN

Use the RPCGEN protocol compiler on the RPC protocol specification file, MSG.X, (from *Example 2.2, "RPC Protocol Specification File Simple Example"*) to generate client and server RPC code automatically:

```
$ RPCGEN MSG.X
```

Using RPCGEN like this – without options – automatically creates the following files from the input file MSG.X:

- A header file called MSG.H that contains `#define` statements for `MESSAGEPROG`, `MESSAGEVERS`, and `PRINTMESSAGE` so you can use them in the other modules. You must include MSG.H in both the client and server modules.
- A file containing client skeleton routines. RPCGEN forms the client skeleton file name, MSG_CLNT.C, by appending _CLNT to the file name and substituting the file type suffix, .C. The MSG_CLNT.C file contains only one client skeleton routine, `printmessage_1`, referred to in the `rprintmsg` client program.
- A file containing server skeleton routines. RPCGEN forms the server skeleton file name, MSG_SVC.C, by appending _SVC to the file name and substituting the file type suffix, .C. The `msg_svc.c` program calls the `printmessage_1` routine in the `msg_server.c` program.

Note

The /TABLE option of RPCGEN creates an additional output file of index information for dispatching service routines. See *Section 2.6.4, "Dispatch Tables"* for more information about dispatch tables.

2.2.5. Compiling the Client and Server Programs

After the RPCGEN protocol compilation, use two `cc` compilation statements to create a client program and a server program:

- To create the client program called `rprintmsg`, compile the client program, `rprintmsg.c`, and the client skeleton program (`msg_clnt.c`) from the original RPCGEN compilation, then link the two object files together with the RPC object library:

```
$ CC/DECC RPRINTMSG.C
$ CC/DECC MSG_CLNT.C
$ LINK RPRINTMSG,MSG_CLNT,TCPIP$RPC:TCPIP$RPCXDR/LIBRARY
```

- To create a server program called `msg_server`, compile the server program `msg_server.c` and the server skeleton program (`msg_svc.c`) from the original RPCGEN compilation, then link the two object files together with the RPC object library:

```
$ CC/DECC MSG_SERVER.C
$ CC/DECC MSG_SVC.C
$ LINK MSG_SERVER,MSG_SVC,TCPIP$RPC:TCPIP$RPCXDR/LIBRARY
```

Note

If you want to use the shareable version of the RPC object library, reference the shareable version of the library, SYS$SHARE:TCPIP$RPCXDR_SHR/SHARE, in your LINK options file.

2.2.6. Copying the Server to a Remote System and Running It

Copy the server program `msg_server` to a remote system called `space` in this example. Then, run it as a detached process there:

```
$ RUN/DETACHED MSG_SERVER
```

Note

You can invoke servers generated by RPCGEN from the command line as well as with port monitors such as INETd, if you generate them with the /INET_SERVICE option.

From a local system (`earth`) you can now print a message on the console of the remote system `space`:

```
$ MCR SYS$DISK:[]RPRINTMSG "space" "Hello out there..."
```

The message `Hello out there...` appears on the console of the system `space`. You can print a message on any console (including your own) with this program if you copy the server to that system and run it.

2.3. Advanced Example: Using RPCGEN to Generate XDR Routines

Section 2.2, "Simple Example: Using RPCGEN to Generate Client and Server RPC Code" explained how to use RPCGEN to generate client and server RPC code automatically to convert a simple procedure to

one that runs remotely over the network. The RPCGEN protocol compiler can also generate the external data representation (XDR) routines that convert local data structures into network format (and vice versa).

The following sections present a more advanced example of a complete RPC service – a remote directory listing service that uses RPCGEN to generate both the client and server skeletons as well as XDR routines.

2.3.1. The RPC Protocol Specification

As with the simple example, you must first create an RPC protocol specification file. This file, DIR.X, is shown in *Example 2.5, "RPC Protocol Specification File – Advanced Example"* (see SYS$COMMON: [SYSHLP.EXAMPLES.TCPIP.RPC]DIR.X).

Note

You can define types (such as `readdir_res` in *Example 2.5, "RPC Protocol Specification File – Advanced Example"*) by using the `struct`, `union`, and `enum` keywords, but do not use these keywords in later variable declarations of those types. For example, if you define `union results`, you must declare it later by using `results`, not `union results`. The RPCGEN protocol compiler compiles RPC unions into C structures, so it is an error to declare them later by using the `union` keyword.

Running RPCGEN on DIR.X creates four output files:

- Header file (DIR.H)
- Client skeleton file (DIR_CLNT.C)
- Server skeleton file (DIR_SVC.C)
- File of XDR routines (DIR_XDR.C)

The first three files have already been described. The fourth file, DIR_XDR.C, contains the XDR routines that convert the declared data types into XDR format (and vice versa). For each data type present in the .X file, RPCGEN assumes that the RPC/XDR library contains a routine with the name of that data type prefixed by `xdr_`, for example, `xdr_int`. If the .X file defines the data type, then RPCGEN generates the required XDR routines (for example, DIR_XDR.C). If the .X file contains no such data types, then RPCGEN does not generate the file. If the program uses a data type but does not define it, then you must provide that XDR routine. This enables you to create your own customized XDR routines.

Example 2.5. RPC Protocol Specification File – Advanced Example

```
/*
 * dir.x: Remote directory listing protocol
 */

/* maximum length of a directory entry */
const MAXNAMELEN = 255;
/* a directory entry */
typedef string nametype

<MAXNAMELEN>;

/* a link in the listing */
```

```
typedef struct namenode *namelist;
/*
 * A node in the directory listing
 */
struct namenode {
     nametype name;          /* name of directory entry */
     namelist next;          /* next entry */
};

/*
 * The result of a READDIR operation.
 */
union readdir_res switch (int Errno) {
case 0:
     namelist list;  /* no error: return directory listing */

default:
     void;           /* error occurred: nothing else to return */
};
/*
 * The directory program definition
 */
program DIRPROG {
     version DIRVERS {
          readdir_res
          READDIR(nametype) = 1;
     } = 1;
} = 0x20000076;
```

2.3.2. Implementing the Procedure Declared in the Protocol Specification

Example 2.6, "Remote Procedure Implementation" (see SYS$COMMON: [SYSHLP.EXAMPLES.TCPIP.RPC]DIR_SERVER.C) consists of the `dir_server.c` program that implements the remote `READDIR` procedure from the previous RPC protocol specification file.

Example 2.6. Remote Procedure Implementation

```
/*
** dir_server.c: remote OpenVMS readdir implementation
*/
#include
<errno.h>
#include
<rms.h>
#include
<rpc/rpc.h>   /* Always needed */
#include "dir.h"       /* Created by RPCGEN */

extern int SYS$PARSE(struct FAB *);
extern int SYS$SEARCH(struct FAB *);

extern char *malloc();

readdir_res *
readdir_1(dirname)
```

```
    nametype *dirname;
{
    char   expanded_name[NAM$C_MAXRSS+1];
    struct FAB fab;
    struct NAM nam;
    namelist       nl;
    namelist      *nlp;
    static readdir_res res; /* must be static! */
    char   resultant_name[NAM$C_MAXRSS+1];
    int exit();

    /*
    ** Initialize the FAB.
    */
    fab = cc$rms_fab;
    fab.fab$l_fna = *dirname;
    fab.fab$b_fns = strlen(*dirname);
    fab.fab$l_dna = "SYS$DISK:[]*.*;*";
    fab.fab$b_dns = strlen(fab.fab$l_dna);

    /*
    ** Initialize the NAM.
    */
    nam = cc$rms_nam;
    nam.nam$l_esa = expanded_name;
    nam.nam$b_ess = NAM$C_MAXRSS;
    nam.nam$l_rsa = resultant_name;
    nam.nam$b_rss = NAM$C_MAXRSS;
    fab.fab$l_nam = &nam;

    /*
    ** Parse the specification and see if it works.
    */
    if (SYS$PARSE(&fab) & 1) {
 /*
 ** Free previous result
 */
 xdr_free(xdr_readdir_res, &res);

        /*
        ** Collect directory entries.
        ** Memory allocated here will be freed by xdr_free
        ** next time readdir_1 is called
        */
        nlp = &res.readdir_res_u.list;
        while (SYS$SEARCH(&fab) & 1) {
     resultant_name[nam.nam$b_rsl] = '\0';
            nl = (namenode *) malloc(sizeof(namenode));
     *nlp = nl;
            nl->name = (char *) malloc(nam.nam$b_name +
                                       nam.nam$b_type +
                                       nam.nam$b_ver + 1);
     strcpy(nl->name, nam.nam$l_name);
            nlp = &nl->next;
            }
        *nlp = NULL;

        /*
```

```
        ** Return the result
        */
        res.Errno = 0;
        } /* SYS$PARSE() */
    else
        res.Errno = fab.fab$l_sts;

    return &res;
    }
```

2.3.3. The Client Program that Calls the Remote Procedure

Example 2.7, "Client Program that Calls the Server" (see SYS$COMMON: [SYSHLP.EXAMPLES.TCPIP.RPC]RLS.C) shows the client program, `rls.c`, that calls the remote server procedure.

Example 2.7. Client Program that Calls the Server

```
/*
* rls.c: Remote directory listing client
*/
#include
<errno.h>
#include
<rms.h>
#include
<stdio.h>
#include
<rpc/rpc.h>    /* always need this */
#include "dir.h"

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

    if (argc != 3) {
        fprintf(stderr, "usage: %s host directory\n", argv[0]);
        exit(1);
        }

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

    /*
    ** Create client "handle" used for calling DIRPROG on
    ** the server designated on the command line.  Use
```

```
    ** the tcp protocol when contacting the server.
    */
    cl = clnt_create(server, DIRPROG, DIRVERS, "tcp");
    if (cl == NULL) {
        /*
        ** Couldn't establish connection with server.
        ** Print error message and stop.
        */
        clnt_pcreateerror(server);
        exit(1);
        }

    /*
    ** Call the remote procedure readdir on the server
    */
    result = readdir_1(&dir, cl);
    if (result == NULL) {
        /*
        ** An RPC error occurred while calling the server.
        ** Print error message and stop.
        */
        clnt_perror(cl, server);
        exit(1);
        }

    /*
    ** Okay, we successfully called the remote procedure.
    */
    if (result->Errno != 0) {
        /*
        ** A remote system error occurred.
        ** Print error message and stop.
        **/
        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);
    exit(0);
    }
```

2.3.4. Running RPCGEN

As with the simple example, you must run the RPCGEN protocol compiler on the RPC protocol specification file DIR.X:

```
$ RPCGEN DIR.X
```

RPCGEN creates a header file, DIR.H, an output file of client skeleton routines, DIR_CLNT.C, and an output file of server skeleton routines, DIR_SVC.C. For this advanced example, RPCGEN also generates the file of XDR routines, DIR_XDR.C.

2.3.5. Compiling the File of XDR Routines

The next step is to compile the file of XDR routines, DIR_XDR.C:

```
$ CC/DECC DIR_XDR
```

2.3.6. Compiling the Client and Server Programs

After the XDR compilation, use two CC and LINK sequences to create the client program and the server program:

- To create the client program called `rls`, compile the client program, RLS.C and the client skeleton program from the original RPCGEN compilation DIR_CLNT.C. Then link the two object files and the object file produced by the recent compilation of the file of XDR routines together with the RPC object library:

  ```
  $ CC/DECC RLS.C
  $ CC/DECC DIR_CLNT.C
  $ LINK RLS,DIR_CLNT,DIR_XDR,TCPIP$RPC:TCPIP$RPCXDR/LIBRARY
  ```

- To create the server program called `dir_server`, compile the remote `READDIR` implementation program, DIR_SERVER.C and the server skeleton program from the original RPCGEN compilation, DIR_SVC.C. Then link the two object files and the object file produced by the recent compilation of the file of XDR routines together with the RPC object library:

  ```
  $ CC/DECC DIR_SERVER.C
  $ CC/DECC DIR_SVC.C
  $ LINK DIR_SERVER,DIR_SVC,DIR_XDR,TCPIP$RPC:TCPIP$RPCXDR/LIBRARY
  ```

Note

If you want to use the shareable version of the RPC object library, reference the shareable version of the library, SYS$SHARE:TCPIP$RPCXDR_SHR, in your LINK options file.

2.3.7. Copying the Server to a Remote System and Running It

Copy the server program `dir_server` to a remote system called `space` in this example. Then, run it as a detached process:

```
$ RUN/DETACHED DIR_SERVER
```

From the local system `earth` invoke the RLS program to provide a directory listing on the system where `dir_server` is running in background mode. The following example shows the command and output (a directory listing of `/usr/pub` on system `space`):

```
$ MCR SYS$DISK:[]RLS "space" "/usr/pub"
.
..
ascii
eqnchar
kbd
```

```
marg8
tabclr
tabs
tabs4
```

Note

Client code generated by RPCGEN does not release the memory allocated for the results of the RPC call. You can call `xdr_free` to deallocate the memory when no longer needed. This is similar to calling `free`, except that you must also pass the XDR routine for the result. For example, after printing the directory listing in the preceding example, you could call `xdr_free` as follows:

```
xdr_free(xdr_readdir_res, result);
```

2.4. Debugging Applications

It is difficult to debug distributed applications that have separate client and server processes. To simplify this, you can test the client program and the server procedure as a single program by linking them with each other rather than with the client and server skeletons. To do this, you must first remove calls to client creation RPC library routines (for example, `clnt_create`). To create the single debuggable file RLS.EXE, compile each file and then link them together as follows:

```
$ CC/DECC RLS.C
$ CC/DECC DIR_CLNT.C
$ CC/DECC DIR_SERVER.C
$ CC/DECC DIR_XDR.C
% LINK RLS,DIR_CLNT,DIR_SERVER,DIR_XDR,TCPIP$RPC:TCPIP$RPCXDR/LIBRARY
```

The procedure calls are executed as ordinary local procedure calls and you can debug the program with a local debugger. When the program is working, link the client program to the client skeleton produced by RPCGEN and the server procedures to the server skeleton produced by RPCGEN.

There are two kinds of errors possible in an RPC call:

1. A problem with the remote procedure call mechanism.

   This occurs when a procedure is unavailable, the remote server does not respond, the remote server cannot decode the arguments, and so on. As in *Example 2.7, "Client Program that Calls the Server"*, an RPC error occurs if `result` is `NULL`.

   The program can print the reason for the failure by using `clnt_perror`, or it can return an error string through `clnt_sperror`.

2. A problem with the server itself.

   As in *Example 2.6, "Remote Procedure Implementation"*, an error occurs if `opendir` fails; that is why `readdir_res` is of type `union`. The handling of these types of errors is the responsibility of the programmer.

2.5. The C Preprocessor

The C preprocessor, CC/DECC/PREPROCESSOR, runs on all input files before they are compiled, so all the preprocessor directives are legal within an .X file. RPCGEN may define up to five macro identifiers, depending on which output file you are generating. The following table lists these macros:

Identifier	Usage
RPC_HDR	For header file output
RPC_XDR	For XDR routine output
RPC_SVC	For server skeleton output
RPC_CLNT	For client skeleton output
RPC_TBL	For index table output

Also, RPCGEN does some additional preprocessing of the input file. Any line that begins with a percent sign (%) passes directly into the output file, without any interpretation. *Example 2.8, "Using the Percent Sign to Bypass Interpretation of a Line"* demonstrates this processing feature.

Example 2.8. Using the Percent Sign to Bypass Interpretation of a Line

```
/*
 * time.x: Remote time protocol
 */
program TIMEPROG {
     version TIMEVERS {
          unsigned int TIMEGET(void) = 1;
     } = 1;
} = 44;

#ifdef RPC_SVC
%int *
%timeget_1()
%{
%    static int thetime;
%
%    thetime = time(0);
%    return (&thetime);
%}
#endif
```

Using the percent sign feature does not guarantee that RPCGEN will place the output where you intend. If you have problems of this type, do not use this feature.

2.6. RPCGEN Programming

The following sections contain additional RPCGEN programming information about network types, defining symbols, INETd support, and dispatch tables.

2.6.1. Network Types

By default, RPCGEN generates server code for both UDP and TCP transports. The /TRANSPORT option creates a server that responds to requests on the specified transport. The following command creates a UDP server from a file called PROTO.X:

```
$ RPCGEN /TRANSPORT=UDP PROTO.X
```

2.6.2. User-Provided Define Statements

The RPCGEN protocol compiler provides a way to define symbols and assign values to them. These defined symbols are passed on to the C preprocessor when it is invoked. This facility is useful when,

for example, invoking debugging code that is enabled only when you define the DEBUG symbol. For example, to enable the DEBUG symbol in the code generated from the PROTO.X file, use the following command:

```
$ RPCGEN /DEFINE=DEBUG PROTO.X
```

2.6.3. INETd Support

The RPCGEN protocol compiler can create RPC servers that INETd can invoke when it receives a request for that service. For example, to generate INETd support for the code generated for the PROTO.X file, use the following command:

```
$ RPCGEN /INET_SERVICE PROTO.X
```

The server code in `proto_svc.c` supports INETd. For more information on setting up entries for RPC services, see *Section 3.7, "Using the Internet Service Daemon (INETd)"*.

In many applications, it is useful for services to wait after responding to a request, on the chance that another will soon follow. However, if there is no call within a certain time (by default, 120 seconds), the server exits and the port monitor continues to monitor requests for its services. You can use the /TIMEOUT_SECONDS option to change the default waiting time. In the following example, the server waits only 20 seconds before exiting:

```
$ RPCGEN /INET_SERVICE /TIMEOUT_SECONDS=20 PROTO.X
```

If you want the server to exit immediately, use /TIMEOUT_SECONDS = 0; if you want the server to wait forever (a normal server situation), use /TIMEOUT_SECONDS = -1.

2.6.4. Dispatch Tables

Dispatch tables are often useful. For example, the server dispatch routine may need to check authorization and then invoke the service routine, or a client library may need to control all details of storage management and XDR data conversion. The following RPCGEN command generates RPC dispatch tables for each program defined in the protocol description file, PROTO.X, and places them in the file PROTO_TBL.I (the suffix .I indicates index):

```
$ RPCGEN /TABLE PROTO.X
```

Each entry in the table is a `struct rpcgen_table` defined in the header file, PROTO.H, as follows:

```
struct rpcgen_table {
    char        *(*proc)();
    xdrproc_t   inproc;
    unsigned    len_in;
    xdrproc_t   outproc;
    unsigned    len_out;
};
```

In this definition:

- `proc` is a pointer to the service routine.
- `inproc` is a pointer to the input (arguments) XDR routine.
- `len_in` is the length in bytes of the input argument.

- `outproc` is a pointer to the output (results) XDR routine.
- `len_out` is the length in bytes of the output result.

The table `dirprog_1_table` is indexed by procedure number. The variable `dirprog_1_nproc` contains the number of entries in the table. The `find_proc` routine in the following example shows how to locate a procedure in the dispatch tables.

```
struct rpcgen_table *
find_proc(proc)
     long    proc;
{
     if (proc >= dirprog_1_nproc)

          /* error */
     else
          return (&dirprog_1_table[proc]);
}
```

Each entry in the dispatch table (in the file *input_file*_TBL.I) contains a pointer to the corresponding service routine. However, the service routine is not defined in the client code. To avoid generating unresolved external references, and to require only one source file for the dispatch table, the actual service routine initializer is `RPCGEN_ACTION(proc_ver)`. The following example shows the dispatch table entry for the procedure `printmessage` with a procedure number of 1:

```
     .....
     (char *(*)())RPCGEN_ACTION(printmessage_1),
     xdr_wrapstring,       0,
     xdr_int,              0,
     .....
```

With this feature, you can include the same dispatch table in both the client and the server. Use the following `define` statement when compiling the client:

```
#define RPCGEN_ACTION(routine)  0
```

Use the following `define` statement when compiling the server:

```
#define RPCGEN_ACTION(routine)  routine
```

2.7. Client Programming

The following sections contain client programming information about default timeouts and client authentication.

2.7.1. Timeout Changes

A call to `clnt_create` sets a default timeout of 25 seconds for RPC calls. RPC waits for 25 seconds to get the results from the server. If it does not get any results, then this usually means that one of the following conditions exists:

- The server is not running.
- The remote system has failed.
- The network is unreachable.

In such cases, the function returns `NULL`; you can print the error with `clnt_perrno`.

Sometimes you may need to change the timeout value to accommodate the application or because the server is slow or far away. Change the timeout by using `clnt_control`. The code segment in the following example demonstrates the use of `clnt_control`.

```
struct timeval tv;
CLIENT *cl;

cl = clnt_create("somehost", SOMEPROG, SOMEVERS, "tcp");
if (cl == NULL) {
     exit(1);
}
tv.tv_sec = 60; /* change timeout to 1 minute */
tv.tv_usec = 0; /* this should always be set  */
clnt_control(cl, CLSET_TIMEOUT, &tv);
```

2.7.2. Client Authentication

By default, client creation routines do not handle client authentication. Sometimes, you may want the client to authenticate itself to the server. This is easy to do, as shown in the following code segment:

```
CLIENT *cl;

cl = clnt_create("somehost", SOMEPROG, SOMEVERS, "udp");
if (cl != NULL) {
     /* To set UNIX style authentication */
     cl->cl_auth = authunix_create_default();
}
```

For more information on authentication, see *Section 3.6, "Authentication of RPC Calls"*.

2.8. Server Programming

The following sections contain server programming information about system broadcasts and passing data to server procedures.

2.8.1. Handling Broadcasts

Sometimes, clients broadcast to determine whether a particular server exists on the network, or to determine all the servers for a particular program and version number. You make these calls with `clnt_broadcast` (for which there is no RPCGEN support). Refer to *Section 3.5.2, "Broadcast RPC"*.

When a procedure is known to be called with broadcast RPC, it is best for the server not to reply unless it can provide useful information to the client. Otherwise, servers could overload the network with useless replies. To prevent the server from replying, a remote procedure can return `NULL` as its result; the server code generated by RPCGEN can detect this and prevent a reply.

In the following example, the procedure replies only if it acts as an NFS server:

```
void *
reply_if_nfsserver()
```

```
{
     char notnull;   /* just here so we can use its address */
     if (access("/etc/exports", F_OK)
< 0) {
          return (NULL);  /* prevent RPC from replying */
     }
     /*
      * return non-null pointer so RPC will send out a reply
      */
     return ((void *)&notnull);
}
```

If a procedure returns type `void *`, it must return a nonnull pointer if it wants RPC to reply for it.

2.8.2. Passing Data to Server Procedures

Server procedures often need to know more about an RPC call than just its arguments. For example, getting authentication information is useful to procedures that want to implement some level of security. This information is supplied to the server procedure as a second argument. (For details, see the structure of `svc_req` in *Section 3.6.2, "The Server Side"*.) The following code segment shows the use of `svc_req`, where the first part of the previous `printmessage_1` procedure is modified to allow only root users to print a message to the console:

```
int *
printmessage_1(msg, rqstp)
     char **msg;
     struct svc_req  *rqstp;
{
     static int result;      /* Must be static */
     FILE *f;
     struct authunix_parms *aup;

     aup = (struct authunix_parms *)rqstp->rq_clntcred;
     if (aup->aup_uid != 0) {
          result = 0;
          return (&result);
     }
/* Same code as before */
```

2.9. RPC and XDR Languages

The RPC language is an extension of the XDR language through the addition of the `program` and `version` types. The XDR language is similar to C. For a complete description of the XDR language syntax, see *RFC 1014: XDR: External Data Representation Standard*. For a description of the RPC extensions to the XDR language, see *RFC 1057: RPC: Remote Procedure Calls Protocol Specification Version 2*.

The following sections describe the syntax of the RPC and XDR languages, with examples and descriptions of how RPCGEN compiles the various RPC and XDR type definitions into C type definitions in the output header file.

2.9.1. Definitions

An RPC language file consists of a series of definitions:

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

RPC recognizes the following definition types:

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

2.9.2. 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
```

The following example defines an `enum` type with three values:

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

This coding compiles into the following:

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

2.9.3. Typedefs

XDR `typedef`s have the same syntax as C `typedef`s:

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

The following example in XDR defines an `fname_type` that declares file name strings with a maximum length of 255 characters:

```
typedef string fname_type <255>;
```

The following example shows the corresponding C definition for this:

```
typedef char *fname_type;
```

2.9.4. Constants

XDR constants are used wherever an integer constant is used (for example, in array size specifications), as shown by the following syntax:

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

The following XDR example defines a constant `DOZEN` equal to 12:

```
const DOZEN = 12;
```

The following example shows the corresponding C definition for this:

```
#define DOZEN 12
```

2.9.5. Declarations

XDR provides only four kinds of declarations, shown by the following syntax:

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

The following lists the syntax for each, followed by examples:

- Simple declarations

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

  For example, `colortype color` in XDR, is the same in C: `colortype color`.

- Fixed-length array declarations

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

  For example, `colortype palette[8]` in XDR, is the same in C: `colortype palette[8]`.

- Variable-length array declarations

  These have no explicit syntax in C, so XDR creates its own by using angle brackets, as in the following syntax:

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

  Specify the maximum size between the angle brackets. You can omit the value, indicating that the array can be of any size, as shown in the following example:

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

Variable-length arrays have no explicit syntax in C, so RPCGEN compiles each of their declarations into a `struct`. For example, RPCGEN compiles the `heights` declaration into the following `struct`:

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

Here, the `_len` component stores the number of items in the array and the `_val` component stores the pointer to the array. The first part of each of these component names is the same as the name of the declared XDR variable.

- Pointer declarations

  These are the same in XDR as in C. You cannot send pointers over the network, but you can use XDR pointers to send recursive data types, such as lists and trees. In XDR language, this type is called `optional-data`, not `pointer`, as in the following syntax:

  ```
  optional-data:
       type-ident "*"variable-ident
  ```

  An example of this (the same in both XDR and C) follows:

  ```
  listitem *next;
  ```

2.9.6. Structures

XDR declares a `struct` almost exactly like its C counterpart. The XDR syntax is the following:

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

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

The following example shows an XDR structure for a two-dimensional coordinate, followed by the C structure into which RPCGEN compiles it in the output header file:

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

The following example shows the C structure that results from compiling the preceding XDR structure:

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

Here, the output is identical to the input, except for the added `typedef` at the end of the output. This enables the use of `coord` instead of `struct coord` in declarations.

2.9.7. Unions

XDR unions are discriminated unions and are different from C unions. They are more analogous to Pascal variant records than to C unions. The syntax is shown here:

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

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

The following is an example of a type that might be returned as the result of a read data. If there is no error, it returns a block of data; otherwise, it returns nothing:

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

RPCGEN compiles this coding into the following:

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

Notice that the union component of the output structure has the same name as the structure type name, except for the suffix, `_u`.

2.9.8. Programs

You declare RPC programs 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
```

The following example shows a program specification for a time protocol program:

```
/*
* 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 coding compiles into the following `#define` statements in the output header file:

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

2.9.9. Special Cases

The following are exceptions to the syntax rules described in the previous sections:

- Booleans

  C has no built-in boolean type. However, the RPC library has a boolean type called `bool_t` that is either TRUE or FALSE. RPCGEN compiles items declared as type `bool` in the XDR language into `bool_t` in the output header file. For example, RPCGEN compiles `bool married` into `bool_t married`.

- Strings

  C has no built-in string type, but instead uses the null-terminated `char *` convention. In the XDR language, you declare strings by using the `string` keyword. RPCGEN compiles each string into a `char *` in the output header file. The maximum size contained in the angle brackets specifies the maximum number of characters allowed in the strings (excluding the `NULL` character). For example, RPCGEN compiles `string name <32>` into `char *name`. You can omit a maximum size to indicate a string of arbitrary length. For example, RPCGEN compiles `string longname<>` into `char *longname`.

- Opaque data

  RPC and XDR use opaque data to describe untyped data, which consists simply of sequences of arbitrary bytes. You declare opaque data as an array of either fixed or variable length. An `opaque` declaration of a fixed-length array is `opaque diskblock[512]`, whose C counterpart is `char`

`diskblock[512]`. An opaque declaration of a variable-length array is `opaque filedata <1024>`, whose C counterpart could be the following:

```
struct {
     u_int filedata_len;
     char *filedata_val;
} filedata;
```

- **Voids**

  In a `void` declaration, the variable is not named. The declaration is just a `void`. Declarations of `void` occur only in union and program definitions (as the argument or result of a remote procedure).

2.10. Command Reference

RPCGEN

RPCGEN — A code-generating tool for creating programming skeletons that implement the RPC mechanism.

Note

RPCGEN runs the C preprocessor, CC/DECC/PREPROCESSOR, on all input files before actually interpreted the files. Therefore, all the preprocessor directives are legal within an RPCGEN input file. For each type of output file, RPCGEN defines a special preprocessor symbol for use by the RPCGEN programmer:

RPC_HDR	Defined when compiling into header files.
RPC_XDR	Defined when compiling into XDR routines.
RPC_SVC	Defined when compiling into server skeletons.
RPC_CLNT	Defined when compiling into client skeletons.
RPC_TBL	Defined when compiling into RPC dispatch table.

In addition, RPCGEN does a little preprocessing of its own. RPCGEN passes any line beginning with a percent sign (%) directly into the output file, without interpreting the line.

Syntax

```
RPCGEN infile[[/HEADER_FILE ]
       [/CLIENT_STUBS_FILE | /DISPATCH_TABLE | /XDR_FILE]
       [/SERVER_STUBS_FILE | /TRANSPORT [=(TCP,UDP)]]]
       [[/TABLE]
       [/DEFINE = (name=[value][,....]) | /OUTPUT = file]
       [/DEFINE = (name=[value][,....]) | /ERRLOG | /INET_SERVICE | /OUTPUT
 = file |
       /TIMEOUT_SECONDS=seconds]]]
```

Parameters

infile

The input file to RPCGEN. The input file contains ONC RPC programming language. This language is very similar to the C language. By default, RPCGEN uses the name of the input file to create the four default output files as follows:

- *infile*.H – the header file
- *infile*_CLNT.C – the client skeleton
- *infile*_SVC.C – the server skeleton with support for both UDP and TCP transports
- *infile*_XDR.C – the XDR routines

If you specify the /DISPATCH_TABLE qualifier, RPCGEN uses the default name *infile*_TBL.I for the dispatch table.

Qualifiers

/CLIENT_STUBS_FILE

Optional.

UNIX equivalent: -l

Default: Create a client skeleton file.

Creates the client skeleton file.

Mutually exclusive with the /DISPATCH_TABLE, /HEADER_FILE, /SERVER_STUBS_FILE, / TRANSPORT, and XDR_FILE qualifiers.

/DEFINE = (*name[=value][,....]*)

Optional.

UNIX equivalent: -D

Default: No definitions.

Defines one or more symbol names. Equivalent to one or more `#define` directives. Names are defined as they appear in the argument to the qualifier. For example, /DEFINE=TEST=1 creates the line `#define TEST=1` in the output files. If you omit the value, RPCGEN defines the name with the value 1.

/DISPATCH_TABLE

Optional.

UNIX equivalent: -t

Default: No dispatch file created.

Creates the server dispatch table file. An RPCGEN dispatch table contains:

- Pointers to the service routines corresponding to a procedure
- A pointer to the input and output arguments

- The size of these routines

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

Mutually exclusive with the /CLIENT_STUBS_FILE, /HEADER_FILE, /SERVER_STUBS_FILE, /TRANSPORT, and XDR_FILE qualifiers.

/ERRLOG

Optional.

UNIX equivalent: -L

Default: Logging to `stderr`.

Specifies that servers should log errors to the operator console instead of using `fprintf` with `stderr`. You must install servers with OPER privilege in order to use this feature.

/HEADER_FILE

Optional.

UNIX equivalent: -h

Default: Create a header file.

Creates the C data definitions header file. Use the /TABLE qualifier in conjunction with this qualifier to generate a header file that supports dispatch tables.

Mutually exclusive with the /CLIENT_STUBS_FILE, /DISPATCH_TABLE, /SERVER_STUBS_FILE, /TRANSPORT, and XDR_FILE qualifiers.

/INET_SERVICE

Optional.

UNIX equivalent: -I

Default: No INETd support.

Compiles support for INETd in the server side stubs. You can start servers yourself or you can have INETd start them. Servers started by INETd log all error messages to the operator console.

If there are no pending client requests, the INETd servers exit after 120 seconds (default). You can change this default with the /TIMEOUT_SECONDS qualifier.

When RPCGEN creates servers with INETd support, it defines two global variables: `_rpcpmstart` and `rpcfdtype`. The runtime value of `_rpcpmstart` is 1 or 0 depending on whether INDEd started the server program. The value of `rpcfdtype` should be `SOCK_STREAM` or `SOCK_DGRAM` depending on the type of the connection.

/OUTPUT = ***file***

Optional.

UNIX equivalent: -o

Default: Direct output to one of the standard default files.

Use this qualifier to direct the output of the /CLIENT_STUBS_FILE, /DISPATCH_TABLE, /HEADER_FILE, /SERVER_STUBS_FILE, /TRANSPORT, and /XDR_FILE qualifiers.

/SERVER_STUBS_FILE

Optional.

UNIX equivalent: -m

Default: Create a server skeleton file.

Creates a server skeleton file without the `main` routine. Use this qualifier to generate a server skeleton when you wish to create your own `main` routine. This option is useful for programs that have callback routines and for programs that have customized initialization requirements.

Mutually exclusive with the /CLIENT_STUBS_FILE, /DISPATCH_TABLE, /HEADER_FILE, /TRANSPORT, and XDR_FILE qualifiers.

/TABLE

Optional.

UNIX equivalent: -T

Default: No dispatch table code created.

Creates the code in the header file to support an RPCGEN dispatch table. You can use this qualifier only when you are generating all files (the default) or when you are using the /HEADER_FILE qualifier to generate the header file. This /TABLE qualifier includes a definition of the dispatch table structure in the header file; it does not modify the server routine to use the table.

/TRANSPORT [= (TCP, UDP)]

Optional.

UNIX equivalent: -s

Default: Create a server skeleton that supports both protocols.

Creates a server skeleton that includes a `main` routine that uses the given transport. The supported transports are UDP and TCP. To compile a server that supports multiple transports, specify both.

/TIMEOUT_SECONDS= *seconds*

Optional.

UNIX equivalent: -K

Default: 120 seconds.

If INETd starts the server, this option specifies the time (in seconds) after which the server should exit if there is no further activity. By default, if there are no pending client requests, INETd servers exit after 120 seconds. This option is useful for customization. If *seconds* is 0, the server exits after serving a request. If *seconds* is -1, the server never exits after being started by INETd.

/XDR_FILE

Optional.

UNIX equivalent: -c

Default: Create an XDR file.

You can customize some of your XDR routines by leaving those data types undefined. For every data type that is undefined, RPCGEN assumes that there exists a routine with the name `xdr_` prepended to the name of the undefined type.

Mutually exclusive with the /CLIENT_STUBS_FILE, /DISPATCH_TABLE, /HEADER_FILE, /TRANSPORT, and /SERVER_STUBS_FILE qualifiers.

Examples

1. `RPCGEN /ERRLOG /TABLE PROTO.X`

   This example generates all of the five possible files using the default file names: PROTO.H, PROTO_CLNT.C, PROTO_SVC.C, PROTO_XDR.C, and PROTO_TBL.I. The PROTO_SVC.C code supports the use of the dispatch table found in PROTO_TBL.I. The server error messages are logged to the operator console instead of being sent to the standard error.

2. `RPCGEN /INET_SERVICE /TIMEOUT_SECONDS=20 PROTO.X`

   This example generates four output files using the default file names: PROTO.H, PROTO_CLNT.C, PROTO_SVC.C, and PROTO_XDR.C. INETd starts the server and the server exits after 20 seconds of inactivity.

3. `RPCGEN /HEADER_FILE /TABLE PROTO.X`

   This example sends the header file (with support for dispatch tables) to the default output file PROTO.H.

4. `RPCGEN /TRANSPORT=TCP PROTO.X`

   This example sends the server skeleton file for the transport TCP to the default output file PROTO_SVC.C.

5. `RPCGEN /HEADER_FILE /TABLE /OUTPUT=PROTO_TABLE.H PROTO.X`

   This example sends the header file (with support for dispatch tables) to the output file PROTO_TABLE.H.

Chapter 3. RPC Application Programming Interface

For most applications, you do not need the information in this chapter; you can simply use the automatic features of the RPCGEN protocol compiler (described in *Chapter 2, "Writing RPC Applications with the RPCGEN Protocol Compiler "*). This chapter requires an understanding of network theory; it is for programmers who must write customized network applications using remote procedure calls, and who need to know about the RPC mechanisms hidden by RPCGEN.

3.1. RPC Layers

The ONC RPC interface consists of three layers: highest, middle, and lowest. For ONC RPC programming, only the middle and lowest layers are of interest. For a complete specification of the routines in the remote procedure call library, see *Chapter 5, "ONC RPC Client Routines"* through *Chapter 8, "XDR Routine Reference"*.

The middle layer routines are adequate for most applications. This layer is “RPC proper ” because you do not need to write additional programming code for network sockets, the operating system, or any other low-level implementation mechanisms. At this level, you simply make remote procedure calls to routines on other systems. For example, you can make simple ONC RPC calls by using the following RPC routines:

- `registerrpc`, which obtains a unique systemwide procedure-identification number
- `callrpc`, which executes a remote procedure call
- `svc_run`, which calls a remote procedure in response to an RPC request

The middle layer is not suitable for complex programming tasks because it sacrifices flexibility for simplicity. Although it is adequate for many tasks, the middle layer does not provide the following:

- Timeout specifications
- Choice of transport
- Operating system process control
- Processing flexibility after occurrence of error
- Multiple kinds of call authentication

The lowest layer is suitable for programming tasks that require greater efficiency or flexibility. The lowest layer routines include client creation routines such as:

- `clnt_create`, which creates a client handle
- `clnt_call`, which calls the server
- `svcudp_create`, which creates a UDP server handle

- `svc_register`, which registers the server

3.2. Middle Layer of RPC

The middle layer is the simplest RPC program interface; from this layer you make explicit RPC calls and use the functions `callrpc` and `registerrpc`.

3.2.1. Using callrpc

The simplest way to make remote procedure calls is through the RPC library routine `callrpc`. The programming code in *Example 3.1, "Using callrpc"*, which obtains the number of remote users, shows the usage of `callrpc`.

The `callrpc` routine has eight parameters. In *Example 3.1, "Using callrpc"*, the first parameter, `argv[1]`, is the name of the remote server system as specified in the command line which invoked the `rnusers` program. The next three, `RUSERSPROG`, `RUSERSVERS`, and `RUSERSPROC_NUM`, are the program, version, and procedure numbers that together identify the procedure to be called (these are defined in `rusers.h`). The fifth and sixth parameters are an XDR filter (`xdr_void`) and an argument (`0`) to be encoded and passed to the remote procedure. You provide an XDR filter procedure to encode or decode system-dependent data to or from the XDR format.

The final two parameters are an XDR filter, `xdr_u_long`, for decoding the results returned by the remote procedure and a pointer, `&nusers`, to the storage location of the procedure results. Multiple arguments and results are handled by embedding them in structures.

If `callrpc` completes successfully, it returns zero; otherwise it returns a non-zero value. The return codes are found in `<rpc/clnt.h>`. The `callrpc` routine needs the type of the RPC argument, as well as a pointer to the argument itself (and similarly for the result). For `RUSERSPROC_NUM`, the return value is an unsigned long. This is why `callrpc` has `xdr_u_long` as its first return parameter, which means that the result is of type `unsigned long`, and `&nusers` as its second return parameter, which is a pointer to the location that stores the long result. `RUSERSPROC_NUM` takes no argument, so the argument parameter of `callrpc` is `xdr_void`. In such cases, the argument must be `NULL`.

If `callrpc` gets no answer after trying several times to deliver a message, it returns with an error code. Methods for adjusting the number of retries or for using a different protocol require you to use the lowest layer of the RPC library, which is discussed in *Section 3.3, "Lowest Layer of RPC"*.

The remote server procedure corresponding to the `callrpc` usage example might look like the one in *Example 3.2, "Remote Server Procedure"*.

This procedure takes one argument – a pointer to the input of the remote procedure call (ignored in the example) – and returns a pointer to the result. In the current version of C, character pointers are the generic pointers, so the input argument and the return value can be cast to `char *`.

Example 3.1. Using callrpc

```
/*
 * rnusers.c - program to return the number of users on a remote host
 */
#include
<stdio.h>
#include
<rpc/rpc.h>
#include "rusers.h"
```

```
main(argc, argv)
     int argc;
     char **argv;
{
     unsigned long nusers;
     int stat;

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

Example 3.2. Remote Server Procedure

```
unsigned long *
nuser(indata)
     char *indata;
{
     static unsigned long nusers;

     /*
      * Add code here to compute the number of users
      * and place result in variable nusers.
      * For this example, nusers is set to 5.
      */
     nusers = 5;
     return(&nusers);
}
```

3.2.2. Using registerrpc and svc_run

Normally, a server registers all the RPC calls it plans to handle, and then goes into an infinite loop while waiting to service requests. Using RPCGEN for this also generates a server dispatch function. You can write a server yourself by using `registerrpc`. *Example 3.3, "Using registerrpc in the Main Body of a Server Program"* is a program showing how you would use `registerrpc` in the main body of a server program that registers a single procedure; the remote procedure returns a single `unsigned long` result.

The `registerrpc` routine establishes the correspondence between a procedure and a given RPC procedure number. The first three parameters (defined in `rusers.h`), `RUSERPROG`, `RUSERSVERS`, and `RUSERSPROC_NUM`, are the program, version, and procedure numbers of the remote procedure to be registered; `nuser` is the name of the local procedure that implements the remote procedure; and `xdr_void` and `xdr_u_long` are the XDR filters for the remote procedure's arguments and results, respectively. (Multiple arguments or multiple results are passed as structures.)

The underlying transport mechanism for `registerrpc` is UDP.

Note

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

After registering the local procedure, the main procedure of the server program calls the RPC dispatcher using the `svc_run` routine. The `svc_run` routine calls the remote procedures in response to RPC requests and decodes remote procedure arguments and encodes results. To do this, it uses the XDR filters specified when the remote procedure was registered with `registerrpc`.

The remote server procedure, `nuser`, was already shown in *Example 3.2, "Remote Server Procedure"* and is duplicated in this example. This procedure takes one argument – a pointer to the input of the remote procedure call (ignored in the example) – and returns a pointer to the result. In the current version of C, character pointers are the generic pointers, so the input argument and the return value can be cast to `char *`.

Example 3.3. Using registerrpc in the Main Body of a Server Program

```
/*
 * nusers_server.c - server to return the number of users on a host
 */
#include
<stdio.h>
#include
<rpc/rpc.h>            /* required */
#include "rusers.h"             /* for prog, vers definitions */

unsigned long *nuser();

main()
{
     int exit();

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


unsigned long *
nuser(indata)
     char *indata;
{
     static unsigned long nusers;

     /*
      * Add code here to compute the number of users
      * and place result in variable nusers.
      * For this example, nusers is set to 5.
      */
     nusers = 5;
     return(&nusers);
}
```

3.2.3. Using XDR Routines to Pass Arbitrary Data Types

RPC can handle arbitrary data structures – regardless of system conventions for byte order and structure layout – by converting them to their external data representation (XDR) before sending them over the network. The process of converting from a particular system representation to XDR format is called **serializing**, and the reverse process is called **deserializing**. The type field parameters of `callrpc` and `registerrpc` can be a built-in procedure like `xdr_u_long` (in the previous example), or one that you supply. XDR has the built-in routines shown in *Table 3.1, "XDR Routines"*.

You cannot use the `xdr_string` routine with either `callrpc` or `registerrpc`, each of which passes only two parameters to an XDR routine. Instead, use `xdr_wrapstring`, which takes only two parameters and calls `xdr_string`.

Table 3.1. XDR Routines

Built-In XDR Integer Routines	
xdr_short	xdr_u_short
xdr_int	xdr_u_int
xdr_long	xdr_u_long
xdr_hyper	xdr_u_hyper
Built-In XDR Floating-Point Routines	
xdr_float	xdr_double
Built-In XDR Character Routines	
xdr_char	xdr_u_char
Built-In XDR Enumeration Routines	
xdr_bool	xdr_u_enum
Built-In XDR Array Routines	
xdr_array	xdr_bytes
xdr_vector	xdr_string
xdr_wrapstring	xdr_opaque
Built-In XDR Pointer Routines	
xdr_reference	xdr_pointer

3.2.4. User-Defined XDR Routines

Suppose that you want to send the following structure:

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

To send it, you would use the following `callrpc` call:

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

With this call to `callrpc`, you could define the routine `xdr_simple` as in the following example:

```
#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 (evaluates to TRUE in C) if it completes successfully; otherwise, it returns zero. For a complete description of XDR, see *RFC 1014: XDR: External Data Representation Standard* and *Chapter 4, "External Data Representation"* of this manual.

Note

It is best to use RPCGEN to generate XDR routines. Use the /XDR_FILE option of RPCGEN to generate only the _XDR.C file.

As another example, if you want to send a variable array of integers, you might package them as a structure like this:

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

Then, you would make an RPC call such as this:

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

You could then define `xdr_varintarr` as shown:

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

The `xdr_array` routine takes 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, and an XDR routine for handling each array element.

If you know the size of the array in advance, you can use `xdr_vector`, which serializes fixed-length arrays, as shown in the following example:

```
int intarr[SIZE];

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

```
{
     return (xdr_vector(xdrsp, intarr, SIZE, sizeof(int),
       xdr_int));
}
```

3.2.5. XDR Serializing Defaults

XDR always converts quantities to 4-byte multiples when serializing. If the examples in *Section 3.2.4, "User-Defined XDR Routines"* had used characters instead of integers, each character would occupy 32 bits. This is why XDR has the built-in routine `xdr_bytes`, which is like `xdr_array` except that it packs characters. The `xdr_bytes` routine has four parameters, similar to the first four of `xdr_array`. For null-terminated strings, XDR provides the built-in routine `xdr_string`, which is the same as `xdr_bytes` but without the length parameter.

When serializing, XDR gets the string length from `strlen`, and on deserializing it creates a null-terminated string. The following example calls the user-defined routine `xdr_simple`, as well as the built-in functions `xdr_string` and `xdr_reference` (the latter locates 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);
}
```

Note that `xdr_simple` could be called here instead of `xdr_reference`.

3.3. Lowest Layer of RPC

Examples in previous sections show how RPC handles many details automatically through defaults. The following sections describe how to change the defaults by using the lowest-layer RPC routines.

The lowest layer of RPC allows you to do the following:

- Use TCP as the underlying transport instead of UDP. Using TCP allows you to exceed the 8K-byte data limitation imposed by UDP.
- Allocate and free memory explicitly while serializing or deserializing with XDR routines.
- Use authentication on either the client or server side, through credential verification.

3.3.1. The Server Side and the Lowest RPC Layer

The server for the `nusers` program in *Example 3.4, "Server Program Using Lowest Layer of RPC"* does the same work as the previous `nusers_server.c` program that used `registerrpc` (see *Example*

3.3, "Using registerrpc in the Main Body of a Server Program"). However, it uses the lowest layer of RPC.

Example 3.4. Server Program Using Lowest Layer of RPC

```
#include
<stdio.h>
#include
<rpc/rpc.h>
#include
<rpc/pmap_clnt.h>
#include "rusers.h"

main()
{
     SVCXPRT *transp;
     unsigned long nuser();
     int exit();

     transp = svcudp_create(RPC_ANYSOCK);
❶
     if (transp == NULL){
          fprintf(stderr, "can't create an RPC server\n");
          exit(1);
     }
     pmap_unset(RUSERSPROG, RUSERSVERS);
❷
     if (!svc_register(transp, RUSERSPROG, RUSERSVERS,
❸
       nuser, IPPROTO_UDP)) {
          fprintf(stderr, "can't register RUSER service\n");
          exit(1);
     }
     svc_run();  /* Never returns */
❹
     fprintf(stderr, "should never reach this point\n");
}
unsigned long
nuser(rqstp, transp)
❺
     struct svc_req *rqstp;
     SVCXPRT *transp;
{
     unsigned long nusers;

     switch (rqstp->rq_proc) {
     case NULLPROC:
          if (!svc_sendreply(transp, xdr_void, 0))
               fprintf(stderr, "can't reply to RPC call\n");
          return;
     case RUSERSPROC_NUM:
          /*
           * Code here to compute the number of users
           * and assign it to the variable nusers
           * For this example, nusers is set to 5.
           */
          nusers = 5;
          if (!svc_sendreply(transp, xdr_u_long, &nusers))
```

```
                fprintf(stderr, "can't reply to RPC call\n");
            return;
        default:
            svcerr_noproc(transp);
            return;
        }
}
```

In this example, the following events occur:

❶ The server calls `svcudp_create` to get a transport handle for receiving and replying to RPC messages. If the argument to `svcudp_create` is `RPC_ANYSOCK`, the RPC library creates a socket on which to receive and reply to RPC calls. Otherwise, `svcudp_create` expects its argument to be a valid socket number. If you specify your own socket, it can be bound or unbound. If it is bound to a port by the user, the port numbers of `svcudp_create` and `clntudp_create` (the low-level client routine) must match. The `registerrpc` routine uses `svcudp_create` to get a UDP handle. If you need a more reliable protocol, call `svctcp_create` instead.

❷ The next step is to call `pmap_unset` so if the `nuser` server crashed earlier, any previous trace of it is erased before restarting. More precisely, `pmap_unset` erases the entry for `RUSERSPROG` from the Portmapper tables.

❸ Use a call to `svc_register` to associate the program number `RUSERSPROG` and the version `RUSERSVERS` with the procedure `nuser`. Unlike `registerrpc`, there are no XDR routines in the registration process, and registration is at the program level rather than the procedure level.

A service can register its port number with the local Portmapper service by specifying a nonzero protocol number in the final argument of `svc_register`. A client determines the server's port number by consulting the Portmapper on its server system. Specifying a zero port number in `clntudp_create` or `clnttcp_create` does this automatically.

❹ Finally, use a call to the `svc_run` routine to put the program into a wait state until RPC requests arrive.

❺ The server routine `nuser` must call and dispatch the appropriate XDR routines based on the procedure number. The `nuser` routine explicitly handles two cases that are taken care of automatically by `registerrpc`:

- The procedure `NULLPROC` (currently zero) returns with no results. This can be used as a simple test for detecting whether a remote program is running.
- There is a check for invalid procedure numbers; if the program detects one, it calls `svcerr_noproc` to handle the error.

The `nuser` service routine serializes the results and returns them to the RPC client using `svc_sendreply`. Its first parameter is the server handle, the second is the XDR routine, and the third is a pointer to the data to be returned. It is not necessary to have `nusers` declared as static here because the program calls `svc_sendreply` within that function itself.

To show how a server handles an RPC program that receives data, you could add to the previous example, a procedure called `RUSERSPROC_BOOL`, which has an argument `nusers` and which returns TRUE or FALSE depending on whether the number of users logged on is equal to `nusers`. For example:

```
    case RUSERSPROC_BOOL: {
        int bool;
```

```
          unsigned nuserquery;

          if (!svc_getargs(transp, xdr_u_int, &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, "can't reply to RPC call\n");
          return;
     }
```

Here, the `svc_getargs` routine takes as arguments a server handle, the XDR routine, and a pointer to where the input is to be placed.

3.3.2. The Client Side and the Lowest RPC Layer

When you use `callrpc`, you cannot control either the RPC delivery mechanism or the socket that transports the data. The lowest layer of RPC enables you to modify these parameters, as shown in *Example 3.5, "Using Lowest RPC Layer to Control Data Transport and Delivery"*, which calls the `nuser` service.

Example 3.5. Using Lowest RPC Layer to Control Data Transport and Delivery

```
#include
<stdio.h>
#include
<rpc/rpc.h>
#include
<sys/time.h>
#include
<netdb.h>
#include "rusers.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;
     int exit();

     if (argc != 2) {
          fprintf(stderr, "usage: nusers hostname\n");
          exit(-1);
```

```
     }

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

     pertry_timeout.tv_sec = 3;
     pertry_timeout.tv_usec = 0;
     bcopy(hp->h_addr, (caddr_t)&server_addr.sin_addr,
       hp->h_length);
     server_addr.sin_family = AF_INET;
     server_addr.sin_port =  0;
     if ((client = clntudp_create(&server_addr, RUSERSPROG,
❶
       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,
❷
       0, xdr_u_long, &nusers, total_timeout);

     if (clnt_stat != RPC_SUCCESS) {
          clnt_perror(client, "rpc");
          exit(-1);
     }

     printf("%d users on %s\n", nusers, argv[1]);
     clnt_destroy(client);
❸
     exit(0);
}
```

❶ This example calls the `clntudp_create` routine to get a client handle for the UDP transport. To get a TCP client handle, you would use `clnttcp_create`. The parameters to `clntudp_create` are the server address, the program number, the version number, a timeout value, and a pointer to a socket. If the client does not hear from the server within the time specified in `pertry_timeout`, the request may be sent again to the server. When the `sin_port` is 0, RPC queries the remote Portmapper to find out the address of the remote service.

❷ The lowest-level version of `callrpc` is `clnt_call`, which takes a client handle rather than a host name. The parameters to `clnt_call` are a client handle, the procedure number, the XDR routine for serializing the argument, a pointer to the argument, the XDR routine for deserializing the results, a pointer to where the results will be placed, and the time in seconds to wait for a reply. The number of times that `clnt_call` attempts to contact the server is equal to the `total_timeout` value divided by the `pertry_timeout` value specified in the `clntudp_create` call.

❸ The `clnt_destroy` call always deallocates the space associated with the `CLIENT` handle. It closes the socket associated with the `CLIENT` handle only if the RPC library opened it. If the

socket was opened by the user, it remains open. This makes it possible, in cases where there are multiple client handles using the same socket, to destroy one handle without closing 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, &sock,
  inbufsize, outbufsize);
```

Here, there is no timeout argument; instead, the “receive ” and “send ” buffer sizes must be specified. When the program makes a call to `clnttcp_create`, RPC creates a TCP client handle and establishes a TCP connection. All RPC calls using the client handle use the same TCP connection. The server side of an RPC call using TCP has `svcudp_create` replaced by `svctcp_create`:

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

The last two arguments to `svctcp_create` are “send ” and “receive ” sizes, respectively. If, as in the preceding example, `0` is specified for either of these, the system chooses default values.

The simplest routine that creates a `CLIENT` handle is `clnt_create`:

```
clnt=clnt_create(server_host,prognum,versnum,transport);
```

The parameters here are the name of the host on which the service resides, the program and version number, and the transport to be used. The transport can be either `udp` for UDP or `tcp` for TCP. You can change the default timeouts by using `clnt_control`. For more information, refer to Section 2.7.

3.3.3. Memory Allocation with XDR

To enable memory allocation, the second parameter of `xdr_bytes` is a pointer to a pointer to an array of bytes, rather than the pointer to the array itself. If the pointer has the value `NULL`, then `xdr_bytes` allocates space for the array and returns a pointer to it, putting the size of the array in the third argument. For example, the following XDR routine `xdr_chararr1`, handles 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));
}
```

Here, if space has already been allocated in `chararr`, it can be called from a server like this:

```
     char array[SIZE];

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

If you want XDR to do the allocation, you must rewrite this routine in this way:

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

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

The RPC call might look like this:

```
     char *arrayptr;

     arrayptr = NULL;
     svc_getargs(transp, xdr_chararr2, &arrayptr);
     /*
      * Use the result here
      */
     svc_freeargs(transp, xdr_chararr2, &arrayptr);
```

After using the character array, you can free it with `svc_freeargs`; this will not free any memory if the variable indicating it has the value `NULL`. For example, in the earlier routine `xdr_finalexample` in *Section 3.2.5, "XDR Serializing Defaults"*, if `finalp->string` was `NULL`, it would not be freed. The same is true for `finalp->simplep`.

To summarize, each XDR routine is responsible for serializing, deserializing, and freeing memory as follows:

- When called from `callrpc`, the XDR routine uses its serializing part.
- When called from `svc_getargs`, the XDR routine uses its deserializing part.
- When called from `svc_freeargs`, the XDR routine uses its memory deallocator part.

When building simple examples as shown in this section, you can ignore the three modes. See *Chapter 4, "External Data Representation"* for examples of more sophisticated XDR routines that determine mode and any required modification.

3.4. Raw RPC

Raw RPC refers to the use of pseudo-RPC interface routines that do not use any real transport at all. These routines, `clntraw_create` and `svcraw_create`, help in debugging and testing the noncommunications aspects of an application before running it over a real network. *Example 3.6, "Debugging and Testing the Noncommunication Parts of an Application"* shows their use.

In this example:

- All the RPC calls occur within the same thread of control.
- `svc_run` is not called.
- It is necessary that the server handle be created before the client handle.
- `svcraw_create` takes no parameters.

- The last parameter to `svc_register` is 0, which means that it will not register with Portmapper.
- The server dispatch routine is the same as it is for normal RPC servers.

Example 3.6. Debugging and Testing the Noncommunication Parts of an Application

```
/*
* A simple program to increment the number by 1
*/
#include
<stdio.h>
#include
<rpc/rpc.h>
#include
<rpc/raw.h>    /* required for raw */

struct timeval TIMEOUT = {0, 0};
static void server();

main()
     int argc;
     char **argv;
{
     CLIENT *clnt;
     SVCXPRT *svc;
     int num = 0, ans;
     int exit();

     if (argc == 2)
          num = atoi(argv[1]);
     svc = svcraw_create();

     if (svc == NULL) {
          fprintf(stderr,"Could not create server handle\n");
          exit(1);
     }

     svc_register(svc, 200000, 1, server, 0);
     clnt = clntraw_create(200000, 1);

     if (clnt == NULL) {
          clnt_pcreateerror("raw");
          exit(1);
     }

     if (clnt_call(clnt, 1, xdr_int, &num, xdr_int, &ans,
       TIMEOUT) != RPC_SUCCESS) {
          clnt_perror(clnt, "raw");
          exit(1);
     }

     printf("Client: number returned %d\n", ans);
     exit(0) ;
}

static void
server(rqstp, transp)
```

```
     struct svc_req *rqstp; /* the request */
     SVCXPRT *transp; /* the handle created by svcraw_create */
{
     int num;
     int exit();

     switch(rqstp->rq_proc) {
     case 0:
          if (svc_sendreply(transp, xdr_void, 0) == FALSE) {
               fprintf(stderr, "error in null proc\n");
               exit(1);
          }
          return;
     case 1:
          break;
     default:
          svcerr_noproc(transp);
          return;
     }

     if (!svc_getargs(transp, xdr_int, &num)) {
          svcerr_decode(transp);
          return;
     }

     num++;
     if (svc_sendreply(transp, xdr_int, &num) == FALSE) {
          fprintf(stderr, "error in sending answer\n");
          exit(1);
     }

     return;
}
```

3.5. Miscellaneous RPC Features

The following sections describe other useful features for RPC programming.

3.5.1. Using Select on the Server Side

Suppose a process simultaneously responds to RPC requests and performs another activity. If the other activity periodically updates a data structure, the process can set an alarm signal before calling `svc_run`. However, if the other activity must wait on a file descriptor, the `svc_run` call does not work. The code for `svc_run` is as follows:

```
void
svc_run()
{
     fd_set readfds;
     int dtbsz = getdtablesize();

     for (;;) {
          readfds = svc_fdset;
          switch (select(dtbsz, &readfds, NULL, NULL, NULL)) {
```

```
          case -1:
               if (errno != EBADF)
                    continue;
               perror("select");
               return;
          case 0:
               continue;
          default:
               svc_getreqset(&readfds);
          }
     }
}
```

You can bypass `svc_run` and call `svc_getreqset` if you know the file descriptors of the sockets associated with the programs on which you are waiting. In this way, you can have your own `select` that waits on the RPC socket, and you can have your own descriptors. Note that `svc_fds` is a bit mask of all the file descriptors that RPC uses for services. It can change whenever the program calls any RPC library routine, because descriptors are constantly being opened and closed, for example, for TCP connections.

Note

If you are handling signals in your application, do not make any system call that accidentally sets `errno`. If this happens, reset `errno` to its previous value before returning from your signal handler.

3.5.2. Broadcast RPC

The Portmapper required by broadcast RPC is a daemon that converts RPC program numbers into TCP/IP protocol port numbers. The main differences between broadcast RPC and normal RPC are the following:

- Normal RPC expects one answer, whereas broadcast RPC expects many answers (one or more from each responding server).
- Broadcast RPC supports only packet-oriented (connectionless) transport protocols such as UDP/IP.
- Broadcast RPC filters out all unsuccessful responses; if a version mismatch exists between the broadcaster and a remote service, the user of broadcast RPC never knows.
- All broadcast messages are sent to the Portmapper port; thus, only services that register themselves with their Portmapper are accessible with broadcast RPC.
- Broadcast requests are limited in size to 1400 bytes. Replies can be up to 8800 bytes (the current maximum UDP packet size).

In the following example, the procedure `eachresult` is called each time the program obtains a response. It returns a boolean that indicates whether the user wants more responses. If the argument `eachresult` is `NULL`, `clnt_broadcast` returns without waiting for any replies:

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

```
enum clnt_stat  clnt_stat;
u_long    progNum;        /* program number */
u_long    versnum;        /* version number */
u_long    procnum;        /* procedure number */
xdrproc_t inproc;         /* xdr routine for args */
caddr_t   in;             /* pointer to args */
xdrproc_t outproc;        /* xdr routine for results */
caddr_t   out;            /* pointer to results */
bool_t    (*eachresult)();/* call with each result gotten */
.
.
.
clnt_stat = clnt_broadcast(prognum, versnum, procnum,
  inproc, in, outproc, out, eachresult)
```

In the following example, if `done` is TRUE, broadcasting stops and `clnt_broadcast` returns successfully. Otherwise, the routine waits for another response. The request is rebroadcast after a few seconds of waiting. If no responses come back in a default total timeout period, the routine returns with `RPC_TIMEDOUT`:

```
bool_t done;
caddr_t resultsp;
struct sockaddr_in *raddr; /* Addr of responding server */
.
.
.
done = eachresult(resultsp, raddr)
```

For more information, see *Section 2.8.1, "Handling Broadcasts"*.

3.5.3. Batching

In normal RPC, a client sends a call message and waits for the server to reply by indicating that the call succeeded. This implies that the client must wait idle while the server processes a call. This is inefficient if the client does not want or need an acknowledgment for every message sent.

Through a process called batching, a program can place RPC messages in a “pipeline ” of calls to a desired server. In order to use batching, the following conditions must be true:

- No RPC call in the pipeline should require a response from the server. The server does not send a response message until the client program flushes the pipeline.
- The pipeline of calls is transported on a reliable byte-stream transport, such as TCP/IP.

Because the server does not respond to every call, the client can generate new calls in parallel with the server executing previous calls. Also, the TCP/IP implementation holds several call messages in a buffer and sends 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 the total elapsed time of a series of calls. Because the batched calls are buffered, the client must eventually do a nonbatched call to flush the pipeline. When the program flushes the connection, RPC sends a normal request to the server. The server processes this request and sends back a reply.

In the following example of server batching, assume that a string-rendering service (in the example, a simple print to `stdout`) has two similar calls – one provides a string and returns void results, and the other provides a string and does nothing else. The service (using the TCP/IP transport) may look like *Example 3.7, "Server Batching"*.

Example 3.7. Server Batching

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

void renderdispatch();

main()
{
     SVCXPRT *transp;
     int exit();

     transp = svctcp_create(RPC_ANYSOCK, 0, 0);
     if (transp == NULL){
          fprintf(stderr, "can't create an RPC server\n");
          exit(1);
     }

     pmap_unset(RENDERPROG, RENDERVERS);

     if (!svc_register(transp, RENDERPROG, RENDERVERS,
       renderdispatch, IPPROTO_TCP)) {
          fprintf(stderr, "can't register RENDER service\n");
          exit(1);
     }

     svc_run();  /* Never returns */
     fprintf(stderr, "should never reach this point\n");
}

void
renderdispatch(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, "can't reply to RPC call\n");
          return;
     case RENDERSTRING:
          if (!svc_getargs(transp, xdr_wrapstring, &s)) {
               fprintf(stderr, "can't decode arguments\n");
               /*
                * Tell client he erred
                */
               svcerr_decode(transp);
               return;
          }
          /*
           * Code here to render the string "s"
           */
```

```
            printf("Render: %s\n"), s;
            if (!svc_sendreply(transp, xdr_void, NULL))
                 fprintf(stderr, "can't reply to RPC call\n");
            break;
       case RENDERSTRING_BATCHED:
            if (!svc_getargs(transp, xdr_wrapstring, &s)) {
                 fprintf(stderr, "can't decode arguments\n");
                 /*
                  * We are silent in the face of protocol errors
                  */
                 break;
            }
            /*
             * Code here to render string s, but send no reply!
             */
            printf("Render: %s\n"), s;
            break;
       default:
            svcerr_noproc(transp);
            return;
       }
       /*
        * Now free string allocated while decoding arguments
        */
       svc_freeargs(transp, xdr_wrapstring, &s);
}
```

In *Example 3.7, "Server Batching"*, the service could have one procedure that takes the string and a boolean to indicate whether the procedure will respond. For a client to use batching effectively, the client must perform RPC calls on a TCP-based transport, and the actual calls must have the following attributes:

- The XDR routine of the result must be zero (NULL).
- The timeout of the RPC call must be zero. (Do not rely on `clnt_control` to assist in batching.)

If a UDP transport is used instead, the client call becomes a message to the server and the RPC mechanism becomes simply a message-passing system, with no batching possible. In *Example 3.8, "Client Batching"*, a client uses batching to supply several strings; batching is flushed when the client gets a null string (EOF).

In this example, the server sends no message, making the clients unable to receive notice of any failures that may occur. Therefore, the clients must handle any errors.

Using a UNIX-to-UNIX RPC connection, an example similar to this one was completed to render all of the lines (approximately 2000) in the UNIX file `/etc/termcap`. The rendering service simply discarded the entire file. The example was run in four configurations, in different amounts of time:

- System to itself, regular RPC – 50 seconds
- System to itself, batched RPC – 16 seconds
- System to another, regular RPC – 52 seconds
- System to another, batched RPC – 10 seconds

In the test environment, running only `fscanf` on `/etc/termcap` required 6 seconds. These timings show the advantage of protocols that enable overlapped execution, although they are difficult to design.

Example 3.8. Client Batching

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

main(argc, argv)
     int argc;
     char **argv;
{
     struct timeval total_timeout;
     register CLIENT *client;
     enum clnt_stat clnt_stat;
     char buf[1000], *s = buf;
     int exit(), atoi();
     char *host, *fname;
     FILE *f;
     int renderop;

     host = argv[1];
     renderop = atoi(argv[2]);
     fname = argv[3];

    f = fopen(fname, "r");
     if (f == NULL){
          printf("Unable to open file\n");
          exit(0);
     }
     if ((client = clnt_create(argv[1],
       RENDERPROG, RENDERVERS, "tcp")) == NULL) {
          perror("clnttcp_create");
          exit(-1);
     }

     switch (renderop) {
     case RENDERSTRING:
          total_timeout.tv_sec = 5;
          total_timeout.tv_usec = 0;
          while (fscanf(f,"%s", s) != EOF) {
               clnt_stat = clnt_call(client, RENDERSTRING,
                 xdr_wrapstring, &s, xdr_void, NULL, total_timeout);
               if (clnt_stat != RPC_SUCCESS) {
                    clnt_perror(client, "batching rpc");
                    exit(-1);
               }
          }
          break;
     case RENDERSTRING_BATCHED:
          total_timeout.tv_sec = 0;       /* set timeout to zero */
          total_timeout.tv_usec = 0;
          while (fscanf(f,"%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, "batching 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, "batching rpc");
                exit(-1);
            }
            break;
        default:
            return;
        }


        clnt_destroy(client);
        fclose(f);
        exit(0);
}
```

3.6. Authentication of RPC Calls

In the examples presented so far, the client never identified itself to the server, nor did the server require it from the client. Every RPC call is authenticated by the RPC package on the server, 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 default authentication type is `none`. The authentication subsystem of the RPC package, with its ability to create and send authentication parameters, can support commercially available authentication software.

This manual describes only one type of authentication – authentication through the operating system. The following sections describe client and server authentication through the operating system.

3.6.1. The Client Side

Assume that a client creates the following new RPC client handle:

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

The client handle includes a field describing the associated authentication handle:

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

The RPC client can choose to use authentication that is native to the operating system by setting `clnt->cl_auth` after creating the RPC client handle:

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

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

```
/*
 * credentials native to the operating system
```

```
 */
struct authunix_parms {
     u_long  aup_time;       /* credentials creation time  */
     char    *aup_machname;  /* host name where client is  */
     int     aup_uid;        /* client's OpenVMS uid       */
     int     aup_gid;        /* client's current group id  */
     u_int   aup_len;        /* element length of aup_gids */
                             /* (set to 0 on OpenVMS)      */
     int     *aup_gids;      /* array of groups user is in */
                             /* (set to NULL on OpenVMS)   */
};
```

In this example, the fields are set by `authunix_create_default` by invoking the appropriate system calls. Because the program created this new style of authentication, the program is responsible for destroying it (to save memory) with the following:

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

3.6.2. The Server Side

It is difficult for service implementors to handle authentication because the RPC package passes to the service dispatch routine a request that has an arbitrary authentication style associated with it. 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 credentials from wire */
     caddr_t rq_clntcred;        /* credentials (read only) */
};
```

The `rq_cred` is mostly opaque except for one field, 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 to the service dispatch routine:

- The `rq_cred` field of the request is well formed; that is, the service implementor can use the `rq_cred.oa_flavor` field of the request to determine the authentication style used by the client. The service implementor can also inspect other fields of `rq_cred` if the style is not supported by the RPC package.
- The `rq_clntcred` field of the request is either `NULL` or points to a well formed structure that corresponds to a supported style of authentication credentials.

The `rq_clntcred` field also could be cast to a pointer to an `authunix_parms` structure. If `rq_clntcred` is `NULL`, the service implementor can inspect the other (opaque) fields of `rq_cred`

to determine whether the service knows about a new type of authentication that is unknown to the RPC package.

Example 3.9, "Authentication on Server Side" extends the previous remote user's service (see *Example 3.3, "Using registerrpc in the Main Body of a Server Program"*) so it computes results for all users except UID 16.

Example 3.9. Authentication on Server Side

```
nuser(rqstp, transp)
      struct svc_req *rqstp;
      SVCXPRT *transp;
{
      struct authunix_parms *unix_cred;
      int uid;
      unsigned long nusers;

      /*
       * we don't care about authentication for null proc
       */
      if (rqstp->rq_proc == NULLPROC) {
           if (!svc_sendreply(transp, xdr_void, 0))
                fprintf(stderr, "can't reply to RPC call\n");
           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:        /* return weak authentication error */
           svcerr_weakauth(transp);
           return;
      }
      switch (rqstp->rq_proc) {
      case RUSERSPROC_NUM:
           /*
            * make sure client is allowed to call this proc
            */
           if (uid == 16) {
                svcerr_systemerr(transp);
                return;
           }
           /*
            * Code here to compute the number of users
            * and assign it to the variable nusers
            */
           if (!svc_sendreply(transp, xdr_u_long, &nusers))
                fprintf(stderr, "can't reply to RPC call\n");
           return;

      default:
```

```
            svcerr_noproc(transp);
            return;
        }
}
```

As in this example, it is not customary to check the authentication parameters associated with `NULLPROC` (procedure 0). Also, if the authentication parameter type is not suitable for your service, have your program call `svcerr_weakauth`.

The service protocol itself returns status for access denied; in *Example 3.9, "Authentication on Server Side"*, the protocol does not do this. Instead, it makes a call to the service primitive, `svcerr_systemerr`. RPC deals only with authentication and not with the access control of an individual service. The services themselves must implement their own access control policies and must reflect these policies as return statuses in their protocols.

3.7. Using the Internet Service Daemon (INETd)

You can start an RPC server from INETd. The only difference from the usual code is that it is best to have the service creation routine called in the following form because INETd passes a socket as file descriptor `0`:

```
    transp = svcudp_create(0);      /* For UDP */
    transp = svctcp_create(0,0,0); /* For listener TCP sockets */
    transp = svcfd_create(0,0,0);  /* For connected TCP sockets */
```

Also, call `svc_register` as follows, with the last parameter flag set to 0, because the program is already registered with the Portmapper by INETd:

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

If you want to exit from the server process and return control to INETd, you must do so explicitly, because `svc_run` never returns.

To show all the RPC service entries in the services database, use the following command:

```
TCPIP> SHOW SERVICES/RPC/PERMANENT

                          RPC                    Protocol Versions
Service                Program Number              Lowest / Highest

MEL                        101010                    1         10
TORME                       20202                    1          2

.
.
.
TCPIP>
```

To show detailed information about a single RPC service entry in the services database, use the following command:

```
TCPIP> SHOW SERVICES/FULL/PERMANENT MEL

Service: MEL
```

```
Port:             1111     Protocol:  UDP             Address:  0.0.0.0
Inactivity:          5     User_name: GEORGE          Process:  MEL
Limit:               1

File:         NLA0:
Flags:        Listen

Socket Opts:  Rcheck Scheck
 Receive:            0     Send:               0

Log Opts:     None
 File:        not defined

RPC Opts
 Program number:      101010  Lowest:      1   Highest:     10

Security
 Reject msg:  not defined
 Accept host: 0.0.0.0
 Accept netw: 0.0.0.0
TCPIP>
```

3.8. Additional Examples

The following sections present additional examples for server and client sides, TCP, and callback procedures.

3.8.1. Program Versions on the Server Side

By convention, the first version of program PROG is designated as PROGVERS_ORIG and the most recent version is PROGVERS. Suppose there is a new version of the user program that returns an `unsigned short` result rather than a `long` result. If you name this version RUSERSVERS_SHORT, then a server that wants to support both versions would register both. It is not necessary to create another server handle for the new version, as shown in this segment of code:

```
     if (!svc_register(transp, RUSERSPROG, RUSERSVERS_ORIG,
       nuser, IPPROTO_TCP)) {
          fprintf(stderr, "can't register RUSER service\n");
          exit(1);
     }
     if (!svc_register(transp, RUSERSPROG, RUSERSVERS_SHORT,
       nuser, IPPROTO_TCP)) {
          fprintf(stderr, "can't register new service\n");
          exit(1);
     }
```

You can handle both versions with the same C procedure, as in *Example 3.10, "C Procedure That Returns Two Different Data Types"*.

Example 3.10. C Procedure That Returns Two Different Data Types

```
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, "can't reply to RPC call\n");
               return;
          }
          return;
     case RUSERSPROC_NUM:
          /*
           * Code here to compute the number of users
           * and assign it to the variable, nusers
           */
          nusers2 = nusers;
          switch (rqstp->rq_vers) {
          case RUSERSVERS_ORIG:
               if (!svc_sendreply(transp, xdr_u_long, &nusers)) {
                    fprintf(stderr,"can't reply to RPC call\n");
               }
               break;
          case RUSERSVERS_SHORT:
               if (!svc_sendreply(transp, xdr_u_short, &nusers2)) {
                    fprintf(stderr,"can't reply to RPC call\n");
               }
               break;
          }
     default:
          svcerr_noproc(transp);
          return;
     }
}
```

3.8.2. Program Versions on the Client Side

The network can have different versions of an RPC server. For example, one server might run RUSERSVERS_ORIG, and another might run RUSERSVERS_SHORT.

If the version of the server running does not match the version number in the client creation routines, then `clnt_call` fails with an RPC_PROGVERSMISMATCH error. You can determine the version numbers supported by the server and then create a client handle with an appropriate version number. To do this, use `clnt_create_vers` (refer to *Chapter 5, "ONC RPC Client Routines"* for more information) or the routine shown in *Example 3.11, "Determining Server-Supported Versions and Creating Associated Client Handles"*.

❶ The program begins by creating the client handle with the `clnt_create` routine.

❷ Next, the `clnt_call` routine attempts to call the remote program. Because of the previous `clnt_create` call, the program version requested is RUSERVERS_SHORT. If the `clnt_call` routine is successful, the version was correct.

❸ If the `clnt_call` attempt failed, then the program checks the failure reason. If it is RPC_PROGVERSMISMATCH, the program goes on to find the versions supported.

❹ In this step, the program parses the error status and retrieves the highest and lowest versions supported by the server. The program then checks whether the version RUSERSVERS_SHORT is in the supported range.

❺ If the RUSERSVERS_SHORT version is supported, the program destroys the old client handle using the `clnt_destroy` routine. It then creates a new handle using the RUSERSVERS_SHORT version.

❻ Finally, the program uses the new client handle to make a call to the server using the RUSERSVERS_SHORT version.

Example 3.11. Determining Server-Supported Versions and Creating Associated Client Handles

```
/*
 * A sample client to sense server versions
 */
#include
<rpc/rpc.h>
#include
<stdio.h>
#include "rusers.h"

main(argc,argv)
     int argc;
     char **argv;
{
     struct rpc_err rpcerr;
     struct timeval to;
     CLIENT *clnt;
     enum clnt_stat status;
     int maxvers, minvers;
     int exit();
     u_short num_s;
     u_int num_l;
     char *host;

     host = argv[1];

     clnt = clnt_create(host, RUSERSPROG, RUSERSVERS_SHORT, "udp");
❶

     if (clnt == NULL) {
          clnt_pcreateerror("clnt");
          exit(-1);
     }

     to.tv_sec = 10; /* set the time outs */
     to.tv_usec = 0;
     status = clnt_call(clnt, RUSERSPROC_NUM,
❷
       xdr_void, NULL, xdr_u_short, &num_s, to);

     if (status == RPC_SUCCESS) {
          /* We found the latest version number */
          clnt_destroy(clnt);
          printf("num = %d\n",num_s);
          exit(0);
     }

     if (status != RPC_PROGVERSMISMATCH) {
❸
```

```
          /* Some other error */
          clnt_perror(clnt, "rusers");
          exit(-1);
     }

     clnt_geterr(clnt, &rpcerr);
❹
     maxvers = rpcerr.re_vers.high; /*highest version supported */
     minvers = rpcerr.re_vers.low;  /*lowest version supported */

     if (RUSERSVERS_ORIG
< minvers ||
          RUSERS_ORIG > maxvers) {
          /* doesn't meet minimum standards */
          clnt_perror(clnt, "version mismatch");
          exit(-1);
     }

     /* This version not supported */
     clnt_destroy(clnt);     /* destroy the earlier handle */
❺
     clnt = clnt_create(host, RUSERSPROG,
       RUSERSVERS_ORIG, "udp"); /* try different version */

     if (clnt == NULL) {
          clnt_pcreateerror("clnt");
          exit(-1);
     }

     status = clnt_call(clnt, RUSERSPROCNUM,
❻
       xdr_void, NULL, xdr_u_long, &num_l, to);

     if (status == RPC_SUCCESS) {
          /* We found the latest version number */
          printf("num = %d\n", num_l);
     } else {
          clnt_perror(clnt, "rusers");
          exit(-1);
     }
}
```

3.8.3. Using the TCP Transport

Examples *Example 3.12, "RPC Example That Uses TCP Protocol – XDR Routine"*, *Example 3.13, "RPC Example That Uses TCP Protocol – Client"*, and *Example 3.14, "RPC Example That Uses TCP Protocol – Server"* work like the remote file copy command RCP. The initiator of the RPC call, `snd`, takes its standard input and sends it to the server `rcv`, which prints it on standard output. The RPC call uses TCP. The example also shows how an XDR procedure behaves differently on serialization than on deserialization.

Example 3.12. RPC Example That Uses TCP Protocol – XDR Routine

```
/*
 * The XDR routine:
 *              on decode, read from wire, write onto fp
 *              on encode, read from fp, write onto wire
```

```
 */

#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, "can't fread\n");
                    return (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, "can't fwrite\n");
                    return (1);
               }
          }
     }
}
```

Example 3.13. RPC Example That Uses TCP Protocol – Client

```
/*
 * snd.c - the sender routines
 */
#include
<stdio.h>
#include
<netdb.h>
#include
<rpc/rpc.h>
#include
<sys/socket.h>
#include "rcp.h"        /* for prog, vers definitions */

main(argc, argv)
     int argc;
     char **argv;
{
```

```
     int xdr_rcp();
     int err;
     int exit();
     int callrpctcp();

     if (argc
< 2) {
          fprintf(stderr, "usage: %s servername\n", argv[0]);
          exit(-1);
     }
     if ((err = callrpctcp(argv[1], RCPPROG, RCPPROC,
       RCPVERS, xdr_rcp, stdin, xdr_void, 0) > 0)) {
          clnt_perrno(err);
          fprintf(stderr, "can't make RPC call\n");
          exit(1);
     }
     exit(0);
}

int
callrpctcp(host, prognum, procnum, versnum,
  inproc, in, outproc, out)
     char *host, *in, *out;
     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;
     void bcopy();

     if ((hp = gethostbyname(host)) == NULL) {
          fprintf(stderr, "can't get addr for '%s'\n", host);
          return (-1);
     }
     bcopy(hp->h_addr, (caddr_t)&server_addr.sin_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");
          return (-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);
}
```

Example 3.14. RPC Example That Uses TCP Protocol – Server

```
/*
 * rcv.c - the receiving routines
```

```
 */
#include
<stdio.h>
#include
<rpc/rpc.h>
#include
<rpc/pmap_clnt.h>
#include "rcp.h"        /* for prog, vers definitions */

main()
{
     register SVCXPRT *transp;
     int rcp_service(), exit();

     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");
}

int
rcp_service(rqstp, transp)
     register struct svc_req *rqstp;
     register SVCXPRT *transp;
{
     int xdr_rcp();

     switch (rqstp->rq_proc) {
     case NULLPROC:
          if (svc_sendreply(transp, xdr_void, 0) == 0)
               fprintf(stderr, "err: rcp_service");
          return;
     case RCPPROC:
          if (!svc_getargs(transp, xdr_rcp, stdout)) {
               svcerr_decode(transp);
               return;
          }
          if (!svc_sendreply(transp, xdr_void, 0))
               fprintf(stderr, "can't reply\n");
          return;
     default:
          svcerr_noproc(transp);
          return;
     }
}
```

3.8.4. Callback Procedures

It is sometimes useful to have a server become a client and to make an RPC call back to the process that is its client. An example of this is remote debugging, where the client is a window-system program and the server is a debugger running on the remote system. Mostly, the user clicks a mouse button at the debugging window (converting this to a debugger command), and 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 are reversed, and the debugger wants to make an RPC call to the window program so it can tell the user that a breakpoint has been reached.

Callbacks are also useful when the client cannot block (that is, wait) to hear back from the server (possibly because of excessive processing in serving the request). In such cases, the server could acknowledge the request and use a callback to reply.

To do an RPC callback, you need a program number on which to make the RPC call. The program number is generated dynamically, so it must be in the transient range 0x40000000 to 0c5fffffff. The sample routine `gettransient` returns a valid program number in the transient range and registers it with the Portmapper. It only communicates with the Portmapper running on the same system as the `gettransient` routine itself.

The call to `pmap_set` is a test-and-set operation because it indivisibly tests whether a program number has been registered; if not, it is reserved. The following example shows the sample `gettransient` routine:

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

gettransient(proto, vers, portnum)
     int proto;
     u_long vers;
     u_short portnum;
{
     static u_long prognum = 0x40000000;

     while (!pmap_set(prognum++, vers, proto, portnum))
          continue;
     return (prognum - 1);
}
```

Note that the call to `ntohs` for `portnum` is unnecessary because it was already passed in host byte order (as `pmap_set` expects).

The following list describes how the client/server programs in *Example 3.15, "Client Usage of the gettransient Routine"* and *Example 3.16, "Server Usage of the gettransient Routine"* use the `gettransient` routine:

- The client makes an RPC call to the server, passing it a transient program number.
- The client waits to receive a call back 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.
- At some random time (on receiving an SIGALRM signal in this example), it sends a callback RPC call, using the program number it received earlier.

In *Example 3.15, "Client Usage of the gettransient Routine"* and *Example 3.16, "Server Usage of the gettransient Routine"*, both the client and the server are on the same system; otherwise, host name handling would be different.

Example 3.15. Client Usage of the gettransient Routine

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

int callback();

main()
{
     int tmp_prog;
     char hostname[256];
     SVCXPRT *xprt;
     int stat;
     int callback(), gettransient();
     int exit();

     gethostname(hostname, sizeof(hostname));
     if ((xprt = svcudp_create(RPC_ANYSOCK)) == NULL) {
          fprintf(stderr, "rpc_server: svcudp_create\n");
          exit(1);
     }
     if ((tmp_prog = gettransient(IPPROTO_UDP, 1,
          xprt->xp_port)) == 0) {
          fprintf(stderr,"Client: failed to get transient number\n");
          exit(1);
     }
     fprintf(stderr, "Client: got program number %08x\n", tmp_prog);

     /* protocol is 0 - gettransient does registering */

     (void)svc_register(xprt, tmp_prog, 1, callback, 0);
     stat = callrpc(hostname, EXAMPLEPROG, EXAMPLEVERS,
       EXAMPLEPROC_CALLBACK,xdr_int,&tmp_prog,xdr_void,0);
     if (stat != RPC_SUCCESS) {
          clnt_perrno(stat);
          exit(1);
     }
     svc_run();
     fprintf(stderr, "Error: svc_run shouldn't return\n");
}
int
callback(rqstp, transp)
     register struct svc_req *rqstp;
     register SVCXPRT *transp;
{
     int exit();

     switch (rqstp->rq_proc) {
```

```
     case 0:
          if (!svc_sendreply(transp, xdr_void, 0)) {
               fprintf(stderr, "err: exampleprog\n");
               return (1);
          }
          return (0);

     case 1:
          fprintf(stderr, "Client: got callback\n");
          if (!svc_sendreply(transp, xdr_void, 0)) {
               fprintf(stderr, "Client: error replyingto exampleprog\n");
               return (1);
          }
          exit(0);
     }
     return (0);
}
```

Example 3.16. Server Usage of the gettransient Routine

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

char hostname[256];
void docallback(int);
int pnum = -1;          /* program number for callback routine */

main()
{
     char *getnewprog();

     gethostname(hostname, sizeof(hostname));
     registerrpc(EXAMPLEPROG, EXAMPLEVERS,
       EXAMPLEPROC_CALLBACK, getnewprog, xdr_int, xdr_void);
     signal(SIGALRM, docallback);
     alarm(10);
     svc_run();
     fprintf(stderr, "Server: error, svc_run shouldn't return\n");
}

char *
getnewprog(pnump)
     int *pnump;
{
     pnum = *(int *)pnump;
     return NULL;
}
void
docallback(int signum)
```

```
{
     int ans;

     if (pnum == -1) {
          fprintf(stderr, "Server: program number not received yet");
          signal(SIGALRM, docallback);
          alarm(10);
          return;
     }
     ans = callrpc(hostname, pnum, 1, 1, xdr_void, 0,
       xdr_void, 0);
     if (ans != RPC_SUCCESS) {
          fprintf(stderr, "Server: %s\n",clnt_sperrno(ans));
          exit(1);
     }
     if (ans == RPC_SUCCESS)
          exit(0);

}
```

Chapter 4. External Data Representation

This chapter describes the external data representation (XDR) standard, a set of routines that enable C programmers to describe arbitrary data structures in a system-independent way. For a formal specification of the XDR standard, see *RFC 1014: XDR: External Data Representation Standard*.

XDR is the backbone of ONC RPC, because data for remote procedure calls is transmitted using the XDR standard. ONC RPC uses the XDR routines to transmit data that is read or written from several types of systems. For a complete specification of the XDR routines, see *Chapter 8, "XDR Routine Reference"*.

This chapter also contains a short tutorial overview of the XDR routines, a guide to accessing currently available XDR streams, and information on defining new streams and data types.

XDR was designed to work across different languages, operating systems, and computer architectures. Most users (particularly RPC users) only need the information on number filters (*Section 4.2.1, "Number and Single-Character Filters"*), floating-point filters (*Section 4.2.2, "Floating-Point Filters"*) and enumeration filters (*Section 4.2.3, "Enumeration Filters"*). Programmers who want to implement RPC and XDR on new systems should read the rest of the chapter.

Note

You can use RPCGEN to write XDR routines regardless of whether RPC calls are being made.

C programs that need XDR routines must include the file `<rpc/rpc.h>`, which contains all necessary interfaces to the XDR system. The object library contains all the XDR routines, so you can link as you usually would when using a library. If you wish to use a shareable version of the library, reference the library SYS$SHARE:TCPIP$RPCXDR_SHR in your LINK options file.

4.1. Usefulness of XDR

Consider the following two programs, `writer.c` and `reader.c`:

```
#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);
          }
     }
     exit(0);
}
```

```
#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");
     exit(0);
}
```

The two programs appear to be portable because:

- They pass `lint` checking.
- They work the same when executed on two different hardware architectures, Sun Microsystem's SPARC architecture and VSI's OpenVMS Alpha or I64 architecture.

Piping the output of the `writer.c` program to the `reader.c` program gives identical results on an Alpha computer and on a Sun computer, as shown:

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

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

With local area networks and Berkeley UNIX 4.2 BSD came the concept of network pipes, in which a process produces data on one system, and a second process on another system uses this data. You can construct a network pipe with `writer.c` and `reader.c`. Here, the first process (on a Sun computer) produces data used by a second process (on an VSI Alpha computer):

```
sun% writer
| rsh alpha reader
0 16777216 33554432 50331648 67108864 83886080 100663296
117440512
sun%
```

You get identical results by executing `writer.c` on the VSI Alpha computer and `reader.c` on the Sun computer. These results occur because the byte ordering of long integers differs between the Alpha computer and the Sun computer, although the word size is the same. Note that 16777216 is equal to 224. When 4 bytes are reversed, the 1 is in the 24th bit.

Whenever data is shared by two or more system types, there is a need for portable data. You can make programs data-portable by replacing the `read` and `write` calls with calls to an XDR library routine

`xdr_long`, which is a filter that recognizes the standard representation of a long integer in its external form. Here are the revised versions of `writer.c` and `reader.c`:

```
/*        Revised Version of writer.c       */


#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);
          }
     }
     exit(0);
}

/*        Revised Version of reader.c      */

#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");
     exit(0);
}
```

The new programs were executed on an Alpha computer, a Sun computer, and from a Sun computer to an Alpha computer; the results are as follows:

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

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

sun% writer
| rsh alpha reader
0 1 2 3 4 5 6 7
sun%
```

Note

Arbitrary data structures create portability problems, particularly with alignment and pointers:

- Alignment on word boundaries may cause the size of a structure to vary on different systems.
- A pointer has no meaning outside the system where it is defined.

4.1.1. A Canonical Standard

The XDR approach to standardizing data representations is canonical, because XDR defines a single byte order (big-endian), a single floating-point representation (IEEE), and so on. A program running on any system can use XDR to create portable data by translating its local representation to the XDR standard. Similarly, any such program can read portable data by translating the XDR standard representation to the local equivalent.

The single standard treats separately those programs that create or send portable data and those that use or receive the data. A new system or language has no effect on existing portable data creators and users. Any new system simply uses the canonical standards of XDR; the local representations of other system are irrelevant. To existing programs on other systems, the local representations of the new system are also irrelevant. There are strong precedents for the canonical approach of XDR. For example, TCP/IP, UDP/IP, XNS, Ethernet, and all protocols below layer 5 of the ISO model, are canonical protocols. The advantage of any canonical approach is simplicity; in the case of XDR, a single set of conversion routines is written once.

The canonical approach does have one disadvantage of little practical importance. Suppose two little-endian systems transfer integers according to the XDR standard. The sending system converts the integers from little-endian byte order to XDR (big-endian) byte order, and the receiving system does the reverse. Because both systems observe the same byte order, the conversions were really unnecessary. Fortunately, the time spent converting to and from a canonical representation is insignificant, especially in networking applications. Most of the time required to prepare a data structure for transfer is not spent in conversion but in traversing the elements of the data structure.

4.1.2. The XDR Library

The XDR library enables you to write and read arbitrary C constructs consistently. This makes it useful even when the data is not shared among systems on a network. The XDR library can do this because it has filter routines for strings (null-terminated arrays of bytes), structures, unions, and arrays. Using more primitive routines, you can write your own specific XDR routines to describe arbitrary data structures, including elements of arrays, arms of unions, or objects pointed at from other structures. The structures themselves may contain arrays of arbitrary elements, or pointers to other structures.

The previous `writer.c` and `reader.c` routines manipulate data by using standard I/O routines, so `xdrstdio_create` was used. The parameters to XDR stream creation routines vary according to their function. For example, `xdrstdio_create` takes the following parameters:

- A pointer to an XDR structure that it initializes
- A pointer to a `FILE` that the input or output acts upon
- The operation – either `XDR_ENCODE` for serializing in `writer.c` or `XDR_DECODE` for deserializing in `reader.c`

It is not necessary for RPC users to create XDR streams; the RPC system itself can create these streams and pass them to the users. There is a family of XDR stream creation routines in which each member treats the stream of bits differently.

The `xdr_long` primitive is characteristic of most XDR library primitives and all client XDR routines for two reasons:

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

```
xdr_
xxx(xdrs, xp)
     XDR *xdrs;

xxx *xp;
{
}
```

In this case, *xxx* is `long`, and the corresponding XDR routine is a primitive, `xdr_long`. The client could also define an arbitrary structure `xxx`; in this case, the client would also supply the routine `xdr_xxx`, describing each field by calling XDR routines of the appropriate type. In all cases, the first parameter, `xdrs`, is treated as an opaque handle and passed to the primitive routines.

XDR routines are direction independent; that is, the same routines are called to serialize or deserialize data. This feature is important for portable data. Calling the same routine for either operation practically guarantees that serialized data can also be deserialized. Thus, one routine is used by both the producer and the consumer of networked data.

You implement direction independence by passing a pointer to an object rather than the object itself (only with deserialization is the object modified). If needed, the user can obtain the direction of the XDR operation. See *Section 4.3, "XDR Operation Directions"* for details.

For a more complicated example, assume that a person's gross assets and liabilities are to be exchanged among processes, and each is a separate 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);
}
```

In the preceding example, the parameter `xdrs` is never inspected or modified; it is only passed to subcomponent routines. The program must inspect the return value of each XDR routine call and stop immediately and return FALSE upon subroutine failure.

The preceding example also shows that the type `bool_t` is declared as an integer whose only value is TRUE (1) or FALSE (0). The following definitions apply:

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

With these conventions, 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));
}
```

Either coding style can be used.

4.2. XDR Library Primitives

The following sections describe the XDR primitives – basic and constructed data types – and XDR utilities. The include file `<rpc/xdr.h>` (automatically included by `<rpc/rpc.h>`), defines the interface to these primitives and utilities.

4.2.1. Number and Single-Character Filters

The XDR library provides primitives that translate between numbers and single characters and their corresponding external representations. Primitives include the set of numbers in:

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

Specifically, the ten primitives are:

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

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

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

```
     short *sip;

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

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

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

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

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

bool_t xdr_hyper(xdrs, hp)
     XDR *xdrs;
     longlong_t *hp;

bool_t xdr_u_hyper(xdrs, uhp)
     XDR *xdrs;
     u_longlong_t *uhp;
```

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

For more information on number filters, see *Chapter 8, "XDR Routine Reference"*.

4.2.2. Floating-Point Filters

The XDR library also provides primitive routines for floating-point types in C:

```
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 a pointer to an XDR stream handle. The second parameter is a pointer to the floating-point number that provides data to the stream or receives data from it. Both routines return TRUE if they complete successfully and FALSE if they do not.

Note

Because the numbers are represented in IEEE floating-point format over the network, routines may fail when decoding a valid IEEE representation into a system-specific representation, or vice versa.

To control the local representation of floating point numbers, you can choose the floating-point type when you compile your RPC program or you can use different XDR routines to explicitly control the local representation. For more information about floating-point filters, see the `xdr_double` and `xdr_float` routines in *Chapter 8, "XDR Routine Reference"*.

4.2.3. Enumeration Filters

The XDR library provides a primitive for generic enumerations; it assumes that a C `enum` has the same representation inside the system as a C `integer`. The `bool_t` (boolean) type is an important instance of the `enum` type. The external representation of a `bool_t` type is always TRUE (1) or FALSE (0), as shown here:

```
#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 pointers to the enumerations or booleans that provide data to or receive data from the stream `xdrs`.

For more information about enumeration filters, see *Chapter 8, "XDR Routine Reference"*.

4.2.4. Possibility of No Data

Occasionally, an XDR routine must be supplied to the RPC system, even when no data is passed or required. The following routine does this:

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

4.2.5. Constructed Data Type Filters

Constructed or compound data type primitives require more parameters and perform more complicated functions than the primitives previously discussed. The following sections include primitives for strings, arrays, unions, and pointers to structures.

Constructed data type primitives may use memory management. In many cases, memory is allocated when deserializing data with `XDR_DECODE`. XDR enables memory deallocation through the `XDR_FREE` operation. The three XDR directional operations are `XDR_ENCODE`, `XDR_DECODE`, and `XDR_FREE`.

For more information about constructed data filters, see *Chapter 8, "XDR Routine Reference"*.

4.2.5.1. Strings

In C, a string is defined as a sequence of bytes terminated by a `NULL` byte, which is not considered when calculating string length. When a string is passed or manipulated, there must be a pointer to it. Therefore, the XDR library defines a string to be a `char *`, not a sequence of characters. The external

and internal representations of a string are different. Externally, strings are represented as sequences of ASCII characters; internally, with character pointers. The `xdr_string` routine converts between the two, as follows:

```
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, `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 may specify that a file name cannot be longer than 255 characters. Keep `maxlength` small because overflow conditions may occur if `xdr_string` has to call `malloc` for space. The routine returns FALSE if the number of characters exceeds `maxlength`; otherwise, it returns TRUE.

The behavior of `xdr_string` is similar to that of other routines in this section. For the direction `XDR_ENCODE`, the parameter `sp` points to a string of a certain length; if the string does not exceed `maxlength`, the bytes are serialized.

For the direction `XDR_DECODE`, 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`, then a string of the appropriate length is allocated and `*sp` is set to this string. If the original value of `*sp` is not `NULL`, then XDR assumes that a target area (which can hold strings no longer than `maxlength`) has been allocated. In either case, the string is decoded into the target area, and the routine appends a `NULL` character to it.

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.

4.2.5.2. Variable-Length Byte Arrays

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

1. The length of the array (the byte count) is located explicitly in an unsigned integer.
2. The byte sequence is not terminated by a `NULL` character.
3. The external and internal byte representation is the same.

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 same parameters of `xdr_string` (*Section 4.2.5.1, "Strings"*). The length of the byte area is obtained by dereferencing `lp` when serializing; `*lp` is set to the byte length when deserializing.

4.2.5.3. Variable-Length Arrays of Arbitrary Data Elements

The XDR library 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 known to be 1, and the

external description of each element is built in. The generic array primitive, `xdr_array`, requires parameters identical to those of `xdr_bytes` in addition to two more: the size of array elements and an XDR routine to handle each of the elements.

This routine encodes or decodes each array element:

```
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 a pointer to 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 allowable number of array elements; `elementsiz` is the byte size of each array element. (You can also use the C function `sizeof` to obtain this value.) The `xdr_element` routine is called to serialize, deserialize, or free each element of the array.

Examples *Example 4.1, "Structure and Associated XDR Routine"*, *Example 4.2, "Declaration and Associated XDR Routines"*, and *Example 4.3, "Declarations and XDR Routines"* show the recursiveness of the XDR library routines already discussed.

A user on a networked system can be identified in three ways:

- The system name, such as `krypton` (use the `gethostname` socket routine)
- The user's UID (use the `geteuid` run-time routine)
- On UNIX systems, the group numbers to which the user belongs (not implemented on OpenVMS systems)

Example 4.1, "Structure and Associated XDR Routine" shows how a structure with this information and its associated XDR routine could be coded:

Example 4.1. Structure and Associated XDR Routine

```
struct netuser {
     char    *nu_systemname;
     int     nu_uid;
     u_int   nu_glen;
     int     *nu_gids;
};
#define NLEN 255        /* system names
< 256 chars */
#define NGRPS 20        /* user can't be in > 20 groups */

bool_t
xdr_netuser(xdrs, nup)
     XDR *xdrs;
     struct netuser *nup;

{
     return(xdr_string(xdrs, &nup->nu_systemname, NLEN) &&
```

```
        xdr_int(xdrs, &nup->nu_uid) &&
        xdr_array(xdrs, &nup->nu_gids, &nup->nu_glen,
        NGRPS, sizeof (int), xdr_int));
}
```

A party of network users could be implemented as an array of `netuser` structure. *Example 4.2, "Declaration and Associated XDR Routines"* shows the declaration and its associated XDR routines.

Example 4.2. Declaration and Associated XDR Routines

```
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));
}
```

The parameters to `main` (`argc` and `argv`) can be combined into a structure, and an array of these structures can make up a history of commands. *Example 4.3, "Declarations and XDR Routines"* shows how the declarations and XDR routines might look.

Example 4.3. Declarations and XDR Routines

```
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_wrapstring(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_wrapstring));
}
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));
}
```

In *Example 4.3, "Declarations and XDR Routines"*, the routine `xdr_wrapstring` is needed to package the `xdr_string` routine, because the implementation of `xdr_array` passes only two parameters to the array element description routine; `xdr_wrapstring` supplies the third parameter to `xdr_string`.

4.2.5.4. Fixed-Length Arrays of Arbitrary Data Elements

The XDR library provides a primitive, `xdr_vector`, for fixed-length arrays:

```
#define NLEN 255        /* system names must be
< 256 chars */
#define NGRPS 20        /* user belongs to exactly 20 groups */

struct netuser {
     char *nu_systemname;
     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_systemname, NLEN))
          return(FALSE);
     if (!xdr_int(xdrs, &nup->nu_uid))
          return(FALSE);
     if (!xdr_vector(xdrs, nup->nu_gids, NGRPS, sizeof(int),
       xdr_int)) {
          return(FALSE);
     }
     return(TRUE);
}
```

4.2.5.5. Opaque Data

Some protocols pass handles from a server to a client. The client later passes back the handles, without first inspecting them; that is, handles are opaque. The `xdr_opaque` primitive describes fixed-size, opaque bytes:

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

The first parameter `xdrs` is the XDR stream handle. The second parameter `p` is the location of the bytes and the third parameter `len` is the number of bytes in the opaque object. By definition, the data within the opaque object is not system-portable.

4.2.5.6. Discriminated Unions

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

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

In this example, the routine translates the discriminant of the union at `*dscmp`. The discriminant is always an `enum_t`. Next, the union at `*unp` is translated. The parameter `arms` is a pointer to an array of `xdr_discrim` structures. Each structure contains an ordered pair of `[value,proc]`.

If the union's discriminant is equal to the associated value, then `proc` is called to translate the union. The end of the `xdr_discrim` structure array is denoted by a routine of value `NULL`. If the discriminant is not in the `arms` array, then the `defaultarm` procedure is called if it is non-null; otherwise, the routine returns FALSE.

Example 4.4, "Constructs and XDR Procedure" shows how to serialize or deserialize a discriminated union. In the example, suppose that the type of a union is an integer, character pointer (a string), or a `gnumbers` structure (described in *Section 4.1.2, "The XDR Library"*). Also, assume the union and its current type are declared in a structure, as follows:

```
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;
};
```

Example 4.4, "Constructs and XDR Procedure" shows the constructs and XDR procedure that serialize or deserialize the discriminated union:

Example 4.4. Constructs and XDR Procedure

```
struct xdr_discrim u_tag_arms[4] = {
     { INTEGER, xdr_int },
     { GNUMBERS, xdr_gnumbers }
     { STRING, xdr_wrapstring },
     { __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 discussed in *Section 4.1.2, "The XDR Library"* and `xdr_wrapstring` was presented in *Example 4.3, "Declarations and XDR Routines"*. The default arm parameter to `xdr_union` (the last parameter) is `NULL` in *Example 4.4, "Constructs and XDR Procedure"*. Therefore, the value of the union's discriminant can only be a value listed in the `u_tag_arms` array. *Example 4.4, "Constructs and XDR Procedure"* also shows that the elements of the arm's array do not need to be sorted.

The values of the discriminant may be sparse, though in *Example 4.4, "Constructs and XDR Procedure"* they are not. It is always good practice to explicitly assign integer values to each element of the discriminant's type. This will document the external representation of the discriminant and guarantee that different C compilers provide identical discriminant values.

4.2.5.7. Pointers

In C it is useful to put within a structure any pointers to another structure. The `xdr_reference` primitive makes it easy to serialize, deserialize, and free these referenced structures. A structure of structure pointers is shown here:

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

Parameter `xdrs` is the XDR stream handle, `pp` is a pointer to the pointer to the structure, `ssize` is the size in bytes of the structure (use the C function `sizeof` to obtain this value), and `proc` is the XDR routine that describes the structure. When decoding data, storage is allocated if `*pp` is `NULL`.

There is no need for a primitive `xdr_struct` to describe a structure within a structure, because pointers are always sufficient.

Note

The `xdr_reference` and `xdr_array` primitives are not interchangeable external representations of data.

The following example describes a structure (and its corresponding XDR routine) that contains an item of data and a pointer to a `gnumbers` structure that has more information about that item of data.

Suppose there is a structure containing a person's name and a pointer to a `gnumbers` structure containing the person's gross assets and liabilities. This structure has the following construct:

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

```
};
```

This structure has the following corresponding XDR routine:

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

In many applications, C programmers attach double meaning to the values of a pointer. Typically the value `NULL` means data is not necessary, but some application-specific interpretation applies. In essence, the C programmer is encoding a discriminated union efficiently by overloading the interpretation of the value of a pointer.

For example, in the previous structure, a `NULL` pointer value for `gnp` could indicate that the person's assets and liabilities are unknown; that is, the pointer value encodes two things: whether the data is known and, if it is known, where it is located in memory. Linked lists are an extreme example of the use of application-specific pointer interpretation.

During serialization, the primitive `xdr_reference` cannot attach any special meaning to a pointer with the value `NULL`. That is, passing a pointer to a pointer whose value is `NULL` to `xdr_reference` when serializing data will most likely cause a memory fault and a core dump.

The `xdr_pointer` correctly handles `NULL` pointers. For more information about its use, see *Section 4.5, "Advanced Topics"*.

4.2.6. Non-filter Primitives

The non-filter primitives that follow are for manipulating XDR streams:

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

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

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; the routine returns a -1 in this case (though -1 should be a legitimate value).

The routine `xdr_setpos` sets a stream position to `pos`. However, in some XDR streams, setting a position is impossible; in such cases, `xdr_setpos` returns FALSE.

This routine also fails if the requested position is explicitly out of bounds. The definition of bounds varies according to the stream.

The `xdr_destroy` primitive destroys the XDR stream. Usage of the stream after calling this routine is undefined.

4.3. XDR Operation Directions

Though not recommended, you may want to optimize XDR routines by using the direction of the operation: `XDR_ENCODE`, `XDR_DECODE`, or `XDR_FREE`. For example, the value `xdrs->x_op` contains the direction of the XDR operation. An example in *Section 4.5, "Advanced Topics"* shows the usefulness of the `xdrs->x_op` field.

4.4. XDR Stream Access

An XDR stream is obtained by calling the appropriate creation routine, which takes arguments for the specific properties of the stream. Streams currently exist for serialization or deserialization of data to or from standard I/O FILE streams, TCP/IP connections and files, and memory.

4.4.1. Standard I/O Streams

XDR streams can be interfaced to standard I/O using the `xdrstdio_create` routine as follows:

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

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

The routine `xdrstdio_create` initializes an XDR stream pointed to by `xdrs`. The XDR stream interfaces to the standard I/O library. Parameter `fp` is an open file, and `x_op` is an XDR direction.

4.4.2. Memory Streams

A memory stream enables 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 that is pointed to by parameter `addr`; parameter `len` 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 ONC RPC uses `xdrmem_create`. Complete call or result messages are built-in memory before calling the `sendto` system routine.

4.4.3. Record (TCP/IP) Streams

A record stream is an XDR stream built on top of a record-marking standard that is, in turn, built on top of a file or a Berkeley UNIX 4.2 BSD connection interface, as shown:

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

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 are meant to be data in XDR form. The stream's primary use is for interfacing RPC to TCP connections. However, it can be used to stream data into or out of ordinary files.

The parameter `xdrs` is similar to the corresponding parameter described in *Section 4.4.1, "Standard I/O Streams"*. The stream does 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, defaults are used. When a buffer needs to be filled or flushed, the routine `readproc` or `writeproc` is called, respectively.

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, len)
     char *iohandle;
     char *buf;
     int nbytes;
```

The usage of these routines is similar to the system calls `read` and `write`. However, the first parameter to each routine is the opaque parameter `iohandle`. The other two parameters (`buf` and `nbytes`) and the results (byte count) are identical to the system routines.

The XDR stream enables you to delimit records in the byte stream. This is discussed in *Section 4.5, "Advanced Topics"*. The following primitives are specific to record streams:

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

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

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

The routine `xdrrec_endofrecord` causes the current outgoing data to be marked as a record. If the parameter `flushnow` is TRUE, then the stream's `writeproc` will be called; otherwise, `writeproc` will be called when the output buffer has been filled.

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. If there is no more data in the stream's input buffer, then the routine `xdrrec_eof` returns TRUE. This does not mean that there is no more data in the underlying file descriptor.

4.4.4. XDR Stream Implementation

This section provides the abstract data types needed to implement new instances of XDR streams. 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 is not expected to affect the implementation of a stream. The fields `x_private`, `x_base`, and `x_handy` pertain to a particular stream implementation. The field `x_public` is for the XDR client and must not be used by the XDR stream implementations or the XDR primitives. The macros `x_getpostn`, `x_setpostn`, and `x_destroy` access operations. The operation `x_inline` takes two parameters: an `XDR *`, and an unsigned integer, which is a byte count. The routine returns a pointer to a piece of the stream's internal buffer. The program can then use the buffer segment for any purpose. To the stream, the bytes in the buffer segment have been 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 maximizing efficient use of processor cycles. The resulting buffer is not data portable, so using this feature is not recommended.)

The operations `x_getbytes` and `x_putbytes` get and put sequences of bytes from or to the underlying stream; they return TRUE if successful, and FALSE otherwise. The routines have identical parameters (replace *xxx* with either `x_get` or `x_put`):

```
bool_t
xxxbytes(xdrs, buf, bytecount)
```

```
     XDR *xdrs;
     char *buf;
     u_int bytecount;
```

The `x_getlong` and `x_putlong` routines receive and put long numbers to and from the data stream. These routines must translate the numbers between the system representation and the (standard) external representation. The operating system primitives `htonl` and `ntohl` help to do this. The higher-level XDR implementation assumes that signed and unsigned long integers contain the same number of bits, and that nonnegative integers have the same bit representations as unsigned integers. The routines return TRUE if they succeed and FALSE if they do not. They have identical parameters (replace *xxx* with either `x_get` or `x_put`):

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

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

4.5. Advanced Topics

This section describes advanced techniques for passing data structures, such as linked lists (of arbitrary length). The examples in this section are written using both the XDR C library routines and the XDR data description language.

The last example in *Section 4.1.2, "The XDR Library"* presents a C data structure and its associated XDR routines for an individual's gross assets and liabilities. The example is duplicated here:

```
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);
}
```

If you want to implement a linked list of such information, you could construct the following data structure:

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

typedef struct gnumbers_node *gnumbers_list;
```

You can think of the head of the linked list as the data object; that is, the head is not merely a convenient shorthand for a structure. Similarly the `gn_next` field indicates whether the object has terminated.

Unfortunately, if the object continues, the `gn_next` field is also the address of where it continues. The link addresses carry no useful information when the object is serialized.

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

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

Here, the boolean indicates whether there is more data following it. If the boolean is FALSE, then it is the last data field of the structure; if TRUE, then it is followed by a `gnumbers` structure and (recursively) by a `gnumbers_list`. Note that the C declaration has no boolean explicitly declared in it (though the `gn_next` field implicitly carries the information), while the XDR data description has no pointer explicitly declared in it. From the XDR description in the previous paragraph, you can determine how to write the XDR routines for a `gnumbers_list`. That is, the `xdr_pointer` primitive would implement the XDR union. Unfortunately, because of recursion, using XDR on a list with the following routines causes the C stack to grow linearly with respect to the number of nodes in the list:

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

bool_t
xdr_gnumbers_list(xdrs, gnp)
     XDR *xdrs;
     gnumbers_list *gnp;
{
     return(xdr_pointer(xdrs, gnp,
     sizeof(struct gnumbers_node),
     xdr_gnumbers_node));
}
```

The following routine combines these two mutually recursive routines into a single, nonrecursive one:

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

     for (;;) {
          more_data = (*gnp != NULL);
          if (!xdr_bool(xdrs, &more_data)) {
               return(FALSE);
          }
```

```
            if (! more_data) {
                 break;
            }
            if (xdrs->x_op == XDR_FREE) {
                 nextp = &(*gnp)->gn_next;
            }
            if (!xdr_reference(xdrs, gnp,
              sizeof(struct gnumbers_node), xdr_gnumbers)) {

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

The first task is to find out if there is more data, so the boolean information can be serialized. Notice that this is unnecessary in the `XDR_DECODE` case, because the value of `more_data` is not known until it is deserialized in the next statement, which uses XDR on the `more_data` field of the XDR union. If there is no more data, this last pointer is set to `NULL` to indicate the list end, and a TRUE is returned to indicate completion. Setting the pointer to `NULL` is only important in the `XDR_DECODE` case, since it is already `NULL` in the `XDR_ENCODE` and `XDR_FREE` cases.

Next, if the direction is `XDR_FREE`, the value of `nextp` is set to indicate the location of the next pointer in the list. This is for dereferencing `gnp` to find the location of the next item in the list; after the next statement, the storage pointed to by `gnp` is deallocated and is no longer valid. This cannot be done for all directions because, in the `XDR_DECODE` direction, the value of `gnp` is not set until the next statement.

Next, XDR operates on the data in the node through the primitive `xdr_reference`, which is like `xdr_pointer` (which was used before). However, `xdr_reference` does not send over the boolean indicating whether there is more data; it is used instead of `xdr_pointer` because XDR has already been used on this information. Notice that the XDR routine passed is not the same type as an element in the list. The routine passed is `xdr_gnumbers`, for using XDR on `gnumbers`; however, each element in the list is of type `gnumbers_node`. The `xdr_gnumbers_node` is not passed because it is recursive; instead, use `xdr_gnumbers`, which uses XDR on all of the nonrecursive parts. Note that this works only if the `gn_numbers` field is the first item in each element, so the addresses are identical when passed to `xdr_reference`.

Finally, `gnp` is updated to point to the next item in the list. If the direction is `XDR_FREE`, it is set to the previously saved value; otherwise, `gnp` is dereferenced to get the proper value. Although more difficult to understand than the recursive version, the nonrecursive routine is much less likely to overflow the C stack. It also runs more efficiently because a lot of procedure call overhead has been removed. However, most lists are small (in the hundreds of items or less), and the recursive version should be sufficient for them.

Chapter 5. ONC RPC Client Routines

This chapter describes the client routines that allow C programs to make procedure calls to server programs across the network.

Table 5.1, "ONC RPC Client Routines" describes the task that each client routine performs.

Table 5.1. ONC RPC Client Routines

Routine	Task Category
`auth_destroy`	Destroys authentication information associated with an authentication handle (macro).
`authnone_create`	Creates and returns a null authentication handle for the client process.
`authunix_create`	Creates and returns a UNIX-style authentication handle for the client process.
`authunix_create_default`	Creates and returns a UNIX-style authentication handle containing default authentication information for the client process.
`callrpc`	Calls the remote procedure identified by the routine's arguments.
`clnt_broadcast`	Broadcasts a remote procedure call to all locally connected networks using the broadcast address.
`clnt_call`	Calls a remote procedure (macro).
`clnt_control`	Changes or retrieves information about an RPC client process (macro).
`clnt_create`	Creates an RPC client handle for a remote server procedure.
`clnt_create_vers`	Creates an RPC client handle for a remote server procedure having the highest supported version number within a specified range.
`clnt_destroy`	Destroys a client handle (macro).
`clnt_freeres`	Frees the memory that RPC allocated when it decoded a remote procedure's results (macro).
`clnt_geterr`	Returns an error code indicating why an RPC call failed (macro).
`clnt_pcreateerror`	Prints an error message indicating why RPC could not create a client handle.
`clnt_perrno`	Prints an error message indicating why a `callrpc` or `clnt_broadcast` routine failed.
`clnt_perror`	Prints an error message indicating why a `clnt_call` routine failed.
`clnt_spcreateerror`	Returns a message string indicating why RPC could not create a client handle.

Routine	Task Category
clnt_sperrno	Returns a message string indicating why a callrpc or clnt_broadcast routine failed.
clnt_sperror	Returns a message string indicating why a clnt_call routine failed.
clntraw_create	Creates an RPC client handle for a server procedure included in the same program as the client.
clnttcp_create	Creates an RPC client handle for a remote server procedure using the TCP transport.
clntudp_bufcreate	Creates an RPC client handle for a remote server procedure using a buffered UDP transport.
clntudp_create	Creates an RPC client handle for a remote server procedure using the UDP transport.
get_myaddress	Returns the local host's Internet address.
get_myaddr_dest	Returns the local host's Internet address as seen by the remote host.

auth_destroy

auth_destroy — A macro that frees the memory associated with the authentication handle created by the authnone_create and authunix_create routines.

Syntax

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

void auth_destroy(AUTH *auth_handle)
```

Arguments

auth_handle

An RPC authentication handle created by the authnone_create, authunix_create, or authunix_create_default routine.

Description

Frees the memory associated with the AUTH data structure created by the authnone_create, authunix_create, or authunix_create_default routine. Be careful not to reference the data structure after calling this routine.

authnone_create

authnone_create — Creates an authentication handle for passing null credentials and verifiers to remote systems.

Syntax

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

AUTH *authnone_create ( )
```

Description

Creates and returns an authentication handle that passes null authentication information with each remote procedure call. Use this routine if the server process does not require authentication information. RPC uses this routine as the default authentication routine unless you create another authentication handle using either the `authunix_create` or `authunix_create_default` routine.

Return Values

AUTH *	Authentication handle containing the pertinent information.
NULL	Indicates allocation of AUTH handle failed.

authunix_create

authunix_create — Creates and returns an RPC authentication handle that contains UNIX-style authentication information.

Syntax

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

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

Arguments

host

Pointer to the name of the host on which the information was created. This is usually the name of the system running the client process.

uid

The user's user identification.

gid

The user's current group.

len

The number of elements in `aup_gids` array.

Note

This parameter is ignored by the product's RPC implementation.

aup_gids

A pointer to an array of groups to which the user belongs.

Note

This parameter is ignored by the product's RPC implementation.

Description

Implements UNIX-style authentication parameters. The client uses no encryption for its credentials and only sends null verifiers. The server sends back null verifiers or, optionally, a verifier that suggests a new shorthand for the credentials.

Return Values

AUTH *	Authentication handle containing the pertinent information.
NULL	Indicates allocation of AUTH handle failed.

authunix_create_default

authunix_create_default — Returns a default authentication handle.

Syntax

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

AUTH *authunix_create_default( )
```

Arguments

None.

Description

Calls the `authunix_create` routine with the local host name, effective process ID and group ID, and the process default groups.

Return Values

AUTH *	Authentication handle containing the pertinent information.
NULL	Indicates allocation of AUTH handle failed.

Examples

```
auth_destroy(client->cl_auth)

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

This example overrides the default `authnone_create` action. The client handle, *`client`*, is returned by the `clnt_create`, `clnt_create_vers`, `clnttcp_create`, or `clntudp_create` routine.

callrpc

callrpc — Executes a remote procedure call.

Syntax

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

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

Arguments

host

A pointer to the name of the host on which the remote procedure resides.

prognum

The program number associated with the remote procedure.

versnum

The version number associated with the remote procedure.

procnum

The procedure number associated with the remote procedure.

inproc

The XDR routine used to encode the remote procedure's arguments.

in

A pointer to the remote procedure's arguments.

outproc

The XDR routine used to decode the remote procedure's results.

out

A pointer to the remote procedure's results.

Description

Calls the remote procedure associated with `prognum`, `versnum`, and `procnum` on the host `host`. This routine performs the same functions as a set of calls to the `clnt_create`, `clnt_call`, and `clnt_destroy` routines. This routine returns `RPC_SUCCESS` if it succeeds, or the value of `enum clnt_stat` cast to an integer if it fails. The routine `clnt_perrno` is handy for translating a failure status into a message.

Note

Calling remote procedures with this routine uses UDP/IP as a transport; see `clntudp_create` for restrictions. You do not have control of timeouts or authentication using this routine. If you want to use the TCP transport, use the `clnt_create` or `clnttcp_create` routine.

Returned Values

RPC_SUCCESS	Indicates success.
clnt_stat	Returns a value of type `enum clnt_stat` cast to type `int` containing the status of the `callrpc` operation.

clnt_broadcast

clnt_broadcast — Executes a remote procedure call that is sent to all locally connected networks using the broadcast address.

Syntax

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

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

Arguments

prognum

The program number associated with the remote procedure.

versnum

The version number associated with the remote procedure.

procnum

The procedure number associated with the remote procedure.

inproc

The XDR routine used to encode the remote procedure's arguments.

in

A pointer to the remote procedure's arguments.

outproc

The XDR routine used to decode the remote procedure's results.

out

A pointer to the remote procedure's results.

eachresult

Called each time the routine receives a response. Specify the routine as follows:

```
int eachresult(char *resultsp, struct sockaddr_in *addr)
```

`resultsp` is the same as the parameter passed to `clnt_broadcast()`, except that the remote procedure's output is decoded there. `addr` is a pointer to a `sockaddr_in` structure containing the address of the host that sent the results.

If `eachresult` is `NULL`, the `clnt_broadcast` routine returns without waiting for any replies.

Description

Performs the same function as the `callrpc` routine, except that the call message is sent to all locally connected networks using the broadcast address. Each time it receives a response, this routine calls the `eachresult` routine. If `eachresult` returns zero, `clnt_broadcast` waits for more replies; otherwise it assumes success and returns `RPC_SUCCESS`.

Note

This routine uses the UDP protocol. Broadcast sockets are limited in size to the maximum transfer unit of the data link. For Ethernet, this value is 1400 bytes. For FDDI, this value is 4500 bytes.

Returned Values

RPC_SUCCESS	Indicates success.
clnt_stat	Returns the buffer of type `enum clnt_stat` containing the status of the `clnt_broadcast` operation.

clnt_call

clnt_call — A macro that calls a remote procedure.

Syntax

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

enum clnt_stat clnt_call(CLIENT *handle, u_long procnum, xdrproc_t inproc,
char *in, xdrproc_t outproc, char *out, struct timeval timeout);
```

Arguments

handle

A pointer to a client handle created by any of the client-handle creation routines.

procnum

The procedure number associated with the remote procedure.

inproc

The XDR routine used to encode the remote procedure's arguments.

in

A pointer to the remote procedure's arguments.

outproc

The XDR routine used to decode the remote procedure's results.

out

A pointer to the remote procedure's results.

timeout

A structure describing the time allowed for results to return to the client. If you have previously used the `clnt_control` macro with the `CLSET_TIMEOUT` code, this value is ignored.

Description

Use the `clnt_call` macro after using one of the client-handle creation routines. After you are finished with the handle, return it using the `clnt_destroy` macro. Use the `clnt_perror` to print any errors that occurred.

Returned Values

RPC_SUCCESS	Indicates success.
clnt_stat	Returns the buffer of type `enum clnt_stat` containing the status of the `clnt_call` operation.

clnt_control

clnt_control — A macro that changes or retrieves information about an RPC client process.

Syntax

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

bool_t clnt_control(CLIENT *handle, u_int code, char *info);
```

Arguments

handle

A pointer to a client handle created by any of the client-handle creation routines.

code

A code designating the type of information to be set or retrieved.

info

A pointer to a buffer containing the information for a SET operation or the results of a GET operation.

Description

For UDP and TCP transports specify any of the following for `code`:

CLSET_TIMEOUT	struct timeval	Set total timeout
CLGET_TIMEOUT	struct timeval	Get total timeout
CLGET_SERVER_ADDR	struct sockaddr_in	Get server address
CLGET_FD	int	Get associated socket
CL_FD_CLOSE	void	Close socket on `clnt_destroy`
CL_FD_NCLOSE	void	Leave socket open on `clnt_destroy`

If you set the timeout using `clnt_control`, ONC RPC ignores the *timeout* parameter in all future `clnt_call` calls. The default total timeout is 25 seconds.

For the UDP transport two additional options are available:

CLSET_RETRY_TIMEOUT	struct timeval	Set retry timeout
CLGET_RETRY_TIMEOUT	struct timeval	Get retry timeout

The timeout value in these two calls is the time that UDP waits for a response before retransmitting the message to the server. The default time is 5 seconds. The retry timeout controls when UDP retransmits the request; the total timeout controls the total time that the client should wait for a response. For example, with the default settings, UDP will retry the transmission four times at 5-second intervals.

Returned Values

TRUE	Success
FALSE	Failure

clnt_create

clnt_create — Creates a client handle and returns its address.

Syntax

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

CLIENT *clnt_create(char *host, u_long prognum, u_long versnum, char
 *protocol);
```

Arguments

host

A pointer to the name of the remote host.

prognum

The program number associated with the remote procedure.

versnum

The version number associated with the remote procedure.

protocol

A pointer to a string containing the name of the protocol for transmitting and receiving RPC messages. Specify either `tcp` or `udp`.

Description

The `clnt_create` routine creates an RPC client handle for `prognum`. An RPC client handle is a structure containing information about the RPC client. The client can use the UDP or TCP transport protocol.

This routine uses the Portmapper. You cannot control the local port.

The default sizes of the send and receive buffers are 8800 bytes for the UDP transport, and 4000 bytes for the TCP transport. The retry time for the UDP transport is five seconds.

Use the `clnt_create` routine instead of the `callrpc` or `clnt_broadcast` routines if you want to use one of the following:

- The TCP transport
- A non-null authentication
- More than one active client at the same time

You can also use the `clnttcp_create` routine to use the TCP protocol, or the `clntudp_create` routine to use the UDP protocol.

The `clnt_create` routine uses the global variable `rpc_createerr`. `rpc_createerr` is a structure that contains the most recent service creation error. Use `rpc_createerr`if you want the client program to handle the error. The value of `rpc_createerr` is set by any RPC client creation routine that does not succeed.

Note

If the requested program is available on the host but the program does not support the requested version number, this routine still succeeds. A subsequent call to the `clnt_call` routine will discover the version mismatch. Use the `clnt_create_vers` routine if you want to avoid this condition.

Returned Values

CLIENT *	Client handle containing the server information.

NULL	Error occurred while creating the client handle. Use the `clnt_pcreateerror` or `clnt_spcreateerror` routine to obtain diagnostic information.

clnt_create_vers

clnt_create_vers — Creates a client handle and returns its address. Seeks to use a server supporting the highest version number within a specified range.

Syntax

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

CLIENT *clnt_create_vers(char *host, u_long prognum, u_long *versnum,
u_long min_vers, u_long max_vers, char *protocol);
```

Arguments

host

A pointer to the name of the remote host.

prognum

The program number associated with the remote procedure.

versnum

The version number associated with the remote procedure. This value is returned by the routine. The value is the highest version number supported by the remote server that is in the range of version numbers specified by `min_vers` and `max_vers`. The argument may remain undefined; see additional information in the Description section.

min_vers

The minimum acceptable version number for the remote procedure.

max_vers

The maximum acceptable version number for the remote procedure.

protocol

A pointer to a string containing the name of the protocol for transmitting and receiving RPC messages. Specify either `tcp` or `udp`.

Description

The `clnt_create_vers` routine creates an RPC client handle for `prognum`. An RPC client handle is a structure containing information about the RPC client. The client can use the UDP or TCP transport protocol.

This routine uses the Portmapper. You cannot control the local port.

The default sizes of the send and receive buffers are 8800 bytes for the UDP transport, and 4000 bytes for the TCP transport. The retry time for the UDP transport is 5 seconds.

The `clnt_create_vers` routine differs from the standard `clnt_create` routine in that it seeks out the highest version number supported by the server. If the server does not support any version numbers within the requested range, the routine returns `NULL` and the `versnum` variable is undefined.

The `clnt_create_vers` routine uses the global variable `rpc_createerr`. `rpc_createerr` is a structure that contains the most recent service creation error. Use `rpc_createerr` if you want the client program to handle the error. The value of `rpc_createerr` is set by any RPC client creation routine that does not succeed.

Returned Values

CLIENT *	Clien-thandle containing the server information.
NULL	Error occurred while creating the client handle. Usually the error indicates that the server does not support any version numbers within the requested range. Use the `clnt_pcreateerror` or `clnt_spcreateerror` routine to obtain diagnostic information.

clnt_destroy

clnt_destroy — A macro that frees the memory associated with an RPC client handle.

Syntax

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

void clnt_destroy(CLIENT *handle);
```

Arguments

handle

A pointer to a client handle created by any of the client-handle creation routines.

Description

The `clnt_destroy` routine destroys the client's RPC handle by deallocating all memory related to the handle. The client is undefined after the `clnt_destroy` call.

If the `clnt_create` routine had previously opened the socket associated with the client handle or the program had used the `clnt_control` routine to set CL_FD_CLOSE, this routine closes the socket. If the `clnt_create` routine had not previously opened the socket associated with the client handle or the program had used the `clnt_control` routine to set CL_FD_NCLOSE, this routine leaves the socket open.

Returned Values

None.

clnt_freeres

clnt_freeres — A macro that frees the memory that was allocated when the remote procedure's results were decoded.

Syntax

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

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

Arguments

handle

A pointer to a client handle created by any of the client-handle creation routines.

outproc

The XDR routine used to decode the remote procedure's results.

out

A pointer to the remote procedure's results.

Description

The `clnt_freeres` routine calls the `xdr_free` routine to deallocate the memory where the remote procedure's results are stored.

Returned Values

TRUE	Success.
FALSE	Error occurred while freeing the memory.

clnt_geterr

clnt_geterr — A macro that returns error information indicating why an RPC call failed.

Syntax

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

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

Arguments

handle

A pointer to a client handle created by any of the client-handle creation routines.

errp

A pointer to an `rpc_err` structure containing information that indicates why an RPC call failed. This information is the same information as `clnt_stat` contains, plus one of the following: the C error number, the range of server versions supported, or authentication errors.

Description

This macro copies the error information from the client handle to the structure referenced by `errp`. The macro is mainly for diagnostic use.

Return Values

None.

clnt_pcreateerror

clnt_pcreateerror — Prints a message explaining why ONC RPC could not create a client handle.

Syntax

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

void clnt_pcreateerror(char *sp);
```

Arguments

sp

A pointer to a string to be used as the beginning of the error message.

Description

The `clnt_pcreateerror` routine prints a message to SYS$OUTPUT. The message consists of the `sp` parameter followed by an RPC-generated error message. Use this routine when the `clnt_create`, `clnttcp_create`, or `clntudp_create` routine fails.

Returned Values

None.

clnt_perrno

clnt_perrno — Prints a message indicating why the `callrpc` or `clnt_broadcast` routine failed.

Syntax

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

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

Arguments

stat

A buffer containing status information.

Description

Prints a message to standard error corresponding to the condition indicated by the `stat` argument.

The data type declaration for `clnt_stat` in `rpc/rpc.h` lists the standard errors.

Returned Values

None.

clnt_perror

clnt_perror — Prints a message explaining why an ONC RPC routine failed.

Syntax

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

void clnt_perror(CLIENT *handle, char *sp);
```

Arguments

handle

A pointer to the client handle used in the call that failed.

sp

A pointer to a string to be used as the beginning of the error message.

Description

Prints a message to standard error indicating why an ONC RPC call failed. The message is prepended with string `sp` and a colon.

Returned Values

None.

clnt_spcreateerror

clnt_spcreateerror — Returns a message indicating why RPC could not create a client handle.

Syntax

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

char *clnt_spcreateerror(char *sp);
```

Arguments

sp

A pointer to a string to be used as the beginning of the error message.

Description

The `clnt_spcreateerror` routine returns the address of a message string. The message consists of the `sp` parameter followed by an error message generated by calling the `clnt_sperrno` routine. Use the `clnt_spcreateerror` routine when the `clnt_create`, `clnttcp_create`, or `clntudp_create` routine fails.

Use this routine if:

- You want to save the string.
- You do not want to use `fprintf` to print the message.
- The message format is different from the one that `clnt_perrno` supports.

The address that `clnt_spcreateerror` returns is the address of its own internal string buffer. The `clnt_spcreateerror` routine overwrites this buffer with each call. Therefore, you must copy the string to your own buffer if you wish to save the string.

Returned Values

char *	A pointer to the message string terminated with a `NULL` character.
NULL	The routine was not able to allocate its internal buffer.

clnt_sperrno

clnt_sperrno — Returns a message indicating why the `callrpc` or `clnt_broadcast` routine failed to create a client handle.

Syntax

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

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

Arguments

stat

A buffer containing status information.

Description

The `clnt_sperrno` routine returns a pointer to a string.

Use this routine instead if:

- The server does not have a `stderr` file; many servers do not.
- You want to save the string.
- You do not want to use `fprintf` to print the message.
- The message format is different from the one that `clnt_perrno` supports.

The address that `clnt_sperrno` returns is a pointer to the error message string for the error. Therefore, you do not have to copy the string to your own buffer in order to save the string.

Returned Values

char *	A pointer to the message string terminated with a `NULL` character.

clnt_sperror

clnt_sperror — Returns a message indicating why an ONC RPC routine failed.

Syntax

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

char *clnt_sperror(CLIENT *handle, char *sp);
```

Arguments

handle

A pointer to the client handle used in the call that failed.

sp

A pointer to a string to be used as the beginning of the error message.

Description

The `clnt_sperror` routine returns a pointer to a message string. The message consists of the `sp` parameter followed by an error message generated by calling the `clnt_sperrno` routine. Use this routine when the `clnt_call` routine fails.

Use this routine if:

- You want to save the string.
- You do not want to use `fprintf` to print the message.
- The message format is different from the one that `clnt_perrno` supports.

The address that `clnt_sperror` returns is a pointer to its own internal string buffer. The `clnt_sperror` routine overwrites this buffer with each call. Therefore, you must copy the string to your own buffer if you wish to save the string.

Returned Values

char *	A pointer to the message string terminated with a `NULL` character.
NULL	The routine was not able to allocate its internal buffer.

clntraw_create

clntraw_create — Creates a client handle for memory-based ONC RPC for simple testing and timing.

Syntax

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

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

Arguments

prognum

The program number associated with the remote program.

versnum

The version number associated with the remote program.

Description

Creates an in-program ONC RPC client for the remote program `prognum`, version `versnum`. The transport used to pass messages to the service is actually a buffer within the process's address space, so the corresponding server should live in the same address space; see `svcraw_create`. This allows simulation of and acquisition of ONC RPC overheads, such as round-trip times, without any kernel interference.

Returned Values

CLIENT *	A pointer to a client handle.
NULL	Indicates failure.

clnttcp_create

clnttcp_create — Creates an ONC RPC client handle for a TCP/IP connection.

Syntax

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

CLIENT *clnttcp_create(struct sockaddr_in *addr, u_long prognum,
u_long versnum, int *sockp, u_int sendsize, u_int recvsize);
```

Arguments

addr

A pointer to a buffer containing the Internet address where the remote program is located.

prognum

The program number associated with the remote procedure.

versnum

The version number associated with the remote procedure.

sockp

A pointer to the socket number to be used for the remote procedure call. If *sockp* is `RPC_ANYSOCK`, then this routine opens a new socket and sets *sockp*.

sendsize

The size of the send buffer. If you specify zero, the routine chooses a suitable default.

recvsize

The size of the receive buffer. If you specify zero, the routine chooses a suitable default.

Description

Creates an ONC RPC client handle for the remote program `prognum`, version `versnum` at address `addr`. The client uses TCP/IP as a transport. The routine is similar to the `clnt_create` routine, except `clnttcp_create` allows you to specify a socket and the send and receive buffer sizes.

If you specify the port number as zero by using `addr->sin_port`, the Portmapper provides the number of the port on which the remote program is listening.

The `clnttcp_create` routine uses the global variable `rpc_createerr`. `rpc_createerr` is a structure that contains the most recent service creation error. Use `rpc_createerr` if you want the client program to handle the error. The value of `rpc_createerr` is set by any RPC client creation routine that does not succeed. The `rpc_createerr` variable is defined in the CLNT.H file.

The socket referenced by *sockp* is copied into a private area for RPC to use. It is the client's responsibility to close the socket referenced by *sockp*.

The authentication scheme for the client, `client->cl_auth`, gets set to null authentication. The calling program can set this to something different if necessary.

Note

If the requested program is available on the host but the program does not support the requested version number, this routine still succeeds. A subsequent call to the `clnt_call` routine will discover the version mismatch. Use the `clnt_create_vers` routine if you want to avoid this condition.

Returned Values

CLIENT *	A pointer to the client handle.
NULL	Indicates failure.

clntudp_bufcreate

clntudp_bufcreate — Creates an ONC RPC client handle for a buffered I/O UDP connection.

Syntax

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

CLIENT *clntudp_bufcreate(struct sockaddr_in *addr, u_long prognum,
u_long versnum, struct timeval wait, register int *sockp, u_int sendsize,
u_int recvsize);
```

Arguments

addr

A pointer to a buffer containing the Internet address where the remote program is located.

prognum

The program number associated with the remote procedure.

versnum

The version number associated with the remote procedure.

wait

The amount of time used between call retransmission if no response is received. Retransmission occurs until the ONC RPC calls time out.

sockp

A pointer to the socket number to be used for the remote procedure call. If *sockp* is `RPC_ANYSOCK`, then this routine opens a new socket and sets *sockp*.

sendsize

The size of the send buffer. If you specify zero, the routine chooses a suitable default.

recvsize

The size of the receive buffer. If you specify zero, the routine chooses a suitable default.

Description

Creates an ONC RPC client handle for the remote program `prognum`, version `versnum` at address `addr`. The client uses UDP as the transport. The routine is similar to the `clnt_create` routine, except `clntudp_bufcreate` allows you to specify a socket, the UDP retransmission time, and the send and receive buffer sizes.

If you specify the port number as zero by using `addr->sin_port`, the Portmapper provides the number of the port on which the remote program is listening.

The `clntudp_bufcreate` routine uses the global variable `rpc_createerr`. `rpc_createerr` is a structure that contains the most recent service creation error. Use `rpc_createerr` if you want the client program to handle the error. The value of `rpc_createerr` is set by any RPC client creation routine that does not succeed. The `rpc_createerr` variable is defined in the CLNT.H file.

The socket referenced by *sockp* is copied into a private area for RPC to use. It is the client's responsibility to close the socket referenced by *sockp*.

The authentication scheme for the client, `client->cl_auth`, gets set to null authentication. The calling program can set this to something different if necessary.

Note

If `addr->sin_port` is 0 and the requested program is available on the host but the program does not support the requested version number, this routine still succeeds. A subsequent call to the `clnt_call` routine will discover the version mismatch. Use the `clnt_create_vers` routine if you want to avoid this condition.

Returned Values

CLIENT *	A pointer to the client handle.
NULL	Indicates failure.

clntudp_create

clntudp_create — Creates an ONC RPC client handle for a nonbuffered I/O UDP connection.

Syntax

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

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

Arguments

addr

A pointer to a buffer containing the Internet address where the remote program is located.

prognum

The program number associated with the remote procedure.

versnum

The version number associated with the remote procedure.

wait

The amount of time used between call retransmission if no response is received. Retransmission occurs until the ONC RPC calls time out.

sockp

A pointer to the socket number to be used for the remote procedure call. If *sockp* is `RPC_ANYSOCK`, then this routine opens a new socket and sets *sockp*.

Description

Creates an ONC RPC client handle for the remote program `prognum`, version `versnum` at address `addr`. The client uses UDP as the transport. The routine is similar to the `clnt_create` routine, except `clntudp_create` allows you to specify a socket and the UDP retransmission time.

If you specify the port number as zero by using `addr->sin_port`, the Portmapper provides the number of the port on which the remote program is listening.

The `clntudp_create` routine uses the global variable `rpc_createerr`. `rpc_createerr` is a structure that contains the most recent service creation error. Use `rpc_createerr` if you want the client program to handle the error. The value of `rpc_createerr` is set by any RPC client creation routine that does not succeed. The `rpc_createerr` variable is defined in the CLNT.H file.

The socket referenced by *sockp* is copied into a private area for RPC to use. It is the client's responsibility to close the socket referenced by *sockp*.

The authentication scheme for the client, `client->cl_auth`, gets set to null authentication. The calling program can set this to something different if necessary.

Notes

Since UDP/IP messages can only hold up to 8 KB of encoded data, this transport cannot be used for procedures that take large arguments or return huge results.

If `addr->sin_port` is 0 and the requested program is available on the host but the program does not support the requested version number, this routine still succeeds. A subsequent call to the `clnt_call` routine will discover the version mismatch. Use the `clnt_create_vers` routine if you want to avoid this condition.

Returned Values

CLIENT *	A pointer to the client handle.
NULL	Indicates failure.

get_myaddress

get_myaddress — Returns the local host's Internet address.

Syntax

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

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

Arguments

addr

A pointer to a `sockaddr_in` structure that the routine will load with the Internet address of the host where the local procedure resides.

Description

Puts the local host's Internet address into `addr` without doing any name translation. The port number is always set to `htons (PMAPPORT)`.

Returned Values

None.

get_myaddr_dest

get_myaddr_dest — Returns the local host's Internet address according to a destination address.

Syntax

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

void get_myaddr_dest(struct sockaddr_in *addr, struct sockaddr_in *dest);
```

Arguments

addr

A pointer to a `sockaddr_in` structure that the routine will load with the local Internet address that would provide a connection to the remote address specified in `dest`.

dest

A pointer to a `sockaddr_in` structure containing an Internet address of a remote host.

Description

Since the local host may have multiple network addresses (each on its own interface), this routine is used to select the local address that would provide a connection to the remote address specified in `dest`.

This is an alternative to `gethostbyname`, which invokes yellow pages. It takes a destination (where we are trying to get to) and finds an exact network match to go to.

Returned Values

None.

Chapter 6. ONC RPC Portmapper Routines

This chapter describes the routines that allow C programs to access the Portmapper network service.

Table 6.1, "ONC RPC Portmapper Routines" describes the task that each routine performs.

Table 6.1. ONC RPC Portmapper Routines

Routine	Task Category
`pmap_getmaps`	Returns a list of port mappings for the specified remote host.
`pmap_getmaps_vms`	Returns a list of port mappings (including OpenVMS process IDs) for the specified remote host.
`pmap_getport`	Returns the port number on which the specified service is waiting.
`pmap_rmtcall`	Requests the Portmapper on the specified remote host to call the specified procedure on that host.
`pmap_set`	Registers a remote server procedure with the host's Portmapper.
`pmap_unset`	Unregisters a remote server procedure with the host's Portmapper.

pmap_getmaps

pmap_getmaps — Returns a copy of the current port mappings on a remote host.

Syntax

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

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

Arguments

addr

A pointer to a `sockaddr_in` structure containing the Internet address of the host whose Portmapper you want to call.

Description

A client interface to the Portmapper, which returns a list of the current ONC RPC program-to-port mappings on the host located at the Internet address `addr`. The SHOW PORTMAPPER management command uses this routine.

Returned Values

struct pmaplist *	A pointer to the returned list of server-to-port mappings on host `addr`.
NULL	Indicates failure.

pmap_getmaps_vms

pmap_getmaps_vms — Returns a copy of the current port mappings on a remote host running TCP/IP Services software.

Syntax

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

struct pmaplist_vms *pmap_getmaps_vms(struct sockaddr_in *addr);
```

Arguments

addr

A pointer to a `sockaddr_in` structure containing the Internet address of the host whose Portmapper you wish to call.

Description

This routine is similar to the `pmap_getmaps` routine. However, `pmap_getmaps_vms` also returns the process identifiers (PIDs) that are required for mapping requests to TCP/IP Services hosts.

Returned Values

struct pmaplist *	A pointer to the returned list of server-to-port mappings on host `addr`.
NULL	Indicates failure.

pmap_getport

pmap_getport — Returns the port number on which the specified service is waiting.

Syntax

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

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

Arguments

addr

A pointer to a `sockaddr_in` structure containing the Internet address of the host where the remote Portmapper resides.

prognum

The program number associated with the remote procedure.

versnum

The version number associated with the remote procedure.

protocol

The transport protocol that the remote procedure uses. Specify either `IPPROTO_UDP` or `IPPROTO_TCP`.

Description

A client interface to the Portmapper. This routine returns the port number on which waits a server that supports program number `prognum`, version `versnum`, and speaks the transport protocol associated with `protocol` (`IPPROTO_UDP` or `IPPROTO_TCP`).

Notes

If the requested version is not available, but at least the requested program is registered, the routine returns a port number.

The `pmap_getport` routine returns the port number in host byte order not network byte order. For certain routines you may need to convert this value to network byte order using the `htons` routine. For example, the `sockaddr_in` structure requires that the port number be in network byte order.

Returned Values

x	The port number of the service on the remote system.
0	No mapping exists or RPC could not contact the remote Portmapper service. In the latter case, the global variable `rpc_createerr.cf_error` contains the ONC RPC status.

pmap_rmtcall

pmap_rmtcall — The client interface to the Portmapper service for a remote call and broadcast service. This routine allows a program to do a lookup and call in one step.

Syntax

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

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

```
u_long *port );
```

Arguments

addr

A pointer to a `sockaddr_in` structure containing the Internet address of the host where the remote Portmapper resides.

prognum

The program number associated with the remote procedure.

versnum

The version number associated with the remote procedure.

procnum

The procedure number associated with the remote procedure.

inproc

The XDR routine used to encode the remote procedure's arguments.

in

A pointer to the remote procedure's arguments.

outproc

The XDR routine used to decode the remote procedure's results.

out

A pointer to the remote procedure's results.

timeout

A `timeval` structure describing the time allowed for the results to return to the client.

port

A pointer to a location for the returned port number. Modified to the remote program's port number if the `pmap_rmtcall` routine succeeds.

Description

A client interface to the Portmapper, which instructs the Portmapper on the host at the Internet address `*addr` to make a call on your behalf to a procedure on that host. Use this procedure for a `ping` operation and nothing else. You can use the `clnt_perrno` routine to print any error message.

Note

If the requested procedure is not registered with the remote Portmapper, the remote Portmapper does not reply to the request. The call to `pmap_rmtcall` will eventually time out. The `pmap_rmtcall` does not perform authentication.

Returned Values

enum clnt_stat	Returns the buffer containing the status of the operation.

pmap_set

pmap_set — Called by the server procedure to have the Portmapper create a mapping of the procedure's program and version number.

Syntax

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

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

Arguments

prognum

The program number associated with the server procedure.

versnum

The version number associated with the server procedure.

protocol

The transport protocol that the server procedure uses. Specify either `IPPROTO_UDP` or `IPPROTO_TCP`.

port

The port number associated with the server program.

Description

A server interface to the Portmapper, which establishes a mapping between the triple `[prognum,versnum,protocol]` and `port` on the server's Portmapper service. The `svc_register` routine calls this routine to register the server with the local Portmapper.

Returned Values

TRUE	Indicates success.
FALSE	Indicates failure.

pmap_unset

pmap_unset — Called by the server procedure to have the Portmapper delete a mapping of the procedure's program and version number.

Syntax

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

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

Arguments

prognum

The program number associated with the server procedure.

versnum

The version number associated with the server procedure.

Description

A server interface to the Portmapper, which destroys all mapping between the triple `[prognum, versnum, *]` and ports on the local host's Portmapper.

Chapter 7. ONC RPC Server Routines

This chapter describes the server routines that allow C programs to receive procedure calls from client programs over the network.

Table 7.1, "ONC RPC Server Routines" describes the task that each routine performs.

Table 7.1. ONC RPC Server Routines

Routine	Task Category
`registerrpc`	Creates a server handle and registers the server program with the Portmapper.
`seterr_reply`	Fills in the error field in an RPC reply message with the specified error information.
`svc_destroy`	Destroys a server handle (macro).
`svc_freeargs`	Frees the memory allocated when RPC decoded the server procedure's arguments (macro).
`svc_getargs`	Decodes the server procedure's arguments (macro).
`svc_getcaller`	Returns the address of the client that called the server procedure (macro).
`svc_getreqset`	Reads data for each server connection.
`svc_register`	Registers the server program with the Portmapper.
`svc_run`	Waits for incoming RPC requests and dispatches to the appropriate service routine.
`svc_sendreply`	Sends the results of an RPC request to the client.
`svc_unregister`	Unregisters the server program with the Portmapper.
`svcerr_auth`	Sends an error message to the client indicating that the authentication information was not correctly formatted.
`svcerr_decode`	Sends an error message to the client indicating that the server could not decode the arguments.
`svcerr_noproc`	Sends an error message to the client indicating that the server does not implement the desired procedure.
`svcerr_noprog`	Sends an error message to the client indicating that the requested program is not available.
`svcerr_progvers`	Sends an error message to the client indicating that the requested version is not available.
`svcerr_systemerr`	Sends an error message to the client indicating that a system error occurred.

Routine	Task Category
svcerr_weakauth	Sends an error message to the client indicating that the authentication information was correctly formatted but was insufficient.
svcraw_create	Creates a server handle for a client that shares the same program space.
svcfd_create	Creates a server handle for a specified TCP socket.
svctcp_create	Creates a server handle using the TCP protocol.
svcudp_bufcreate	Creates a server handle using buffered UDP transport.
svcudp_create	Creates a server handle using the UDP transport.
xprt_register	Adds the UDP or TCP socket associated with the specified server handle to the list of registered sockets.
xprt_unregister	Removes the UDP or TCP socket associated with the specified server handle from the list of sockets.
_authenticate	Authenticates an RPC request message.

registerrpc

registerrpc — Obtains a unique systemwide procedure identification number.

Syntax

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

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

Arguments

prognum

The program number associated with the service procedure

versnum

The version number associated with the service procedure

procnum

The procedure number associated with the service procedure

progname

The address of the service procedure being registered with the ONC RPC service package

inproc

The XDR routine used to decode the service procedure's arguments

outproc

The XDR routine used to encode the service procedure's results

Description

The `registerrpc` routine performs the following tasks for a server:

- Creates a UDP server handle. See the `svcudp_create` routine for restrictions.
- Calls the `svc_register` routine to register the program with the Portmapper.
- Adds `prognum`, `versnum`, and `procnum` to an internal list of registered procedures. When the server receives a request, it uses this list to determine which routine to call.

A server should call `registerrpc` for every procedure it implements, except for the NULL procedure. If a request arrives for program `prognum`, version `versnum`, and procedure `procnum`, `progname` is called with a pointer to its parameters.

Returned Values

0	Indicates success.
1	Indicates failure.

seterr_reply

seterr_reply — Fills in the error text in a reply message.

Syntax

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

void seterr_reply(struct rpc_msg *msg, struct rpc_err *error);
```

Arguments

msg

A pointer to a reply message buffer

error

A pointer to an `rpc_err` structure containing the error associated with the reply message.

Description

Given a reply message, `seterr_reply` fills in the error field.

Returned Values

None.

svc_destroy

svc_destroy — A macro that frees the memory associated with an RPC server handle.

Syntax

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

void svc_destroy(SVCXPRT *xprt);
```

Arguments

xprt

A pointer to an RPC server handle created by any of the server-handle creation routines

Description

The `svc_destroy` routine returns all the private data structures associated with a server handle. If the server-handle creation routine received the value `RPC_ANYSOCK` as the socket, `svc_destroy` closes the socket. Otherwise, your program must close the socket.

Returned Values

None.

svc_freeargs

svc_freeargs — A macro that frees the memory allocated when the procedure's arguments were decoded.

Syntax

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

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

Arguments

xprt

A pointer to an RPC server handle created by any of the server-handle creation routines

inproc

The XDR routine used to decode the service procedure's arguments

in

A pointer to the service procedure's decoded arguments

Description

The `svc_destroy` routine returns the memory that the `svc_getargs` routine allocated to hold the service procedure's decoded arguments. This routine calls the `xdr_free` routine.

Returned Values

TRUE	Success; memory successfully deallocated.
FALSE	Failure; memory not deallocated.

svc_getargs

svc_getargs — A macro that decodes the service procedure's arguments.

Syntax

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

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

Arguments

xprt

A pointer to an RPC server handle created by any of the server-handle creation routines

inproc

The XDR routine used to decode the service procedure's arguments

in

A pointer to the service procedure's decoded arguments

Description

This routine calls the specified XDR routine to decode the arguments passed to the service procedure.

Returned Values

TRUE	Successfully decoded.
FALSE	Decoding unsuccessful.

svc_getcaller

svc_getcaller — A macro that returns the address of the client that called the service procedure.

Syntax

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

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

Arguments

xprt

A pointer to an RPC server handle created by any of the server-handle creation routines

Description

This routine returns a `sockaddr_in` structure containing the Internet address of the RPC client routine that called the service procedure.

Returned Values

struct sockaddr_in	A pointer to the socket descriptor.

svc_getreqset

svc_getreqset — Returns data for each server connection.

Syntax

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

void svc_getreqset(fd_set *rdfds);
```

Arguments

rdfds

A pointer to the read file descriptor bit mask modified by the `select` routine.

Description

The `svc_getreqset` routine is for servers that implement custom asynchronous event processing or that do not use the `svc_run` routine. You can only use `svc_fdset` when the server does not use `svc_run`.

You are unlikely to call this routine directly, because the `svc_run` routine calls it. However, there are times when you cannot call `svc_run`. For example, suppose a program services RPC requests and reads or writes to another socket at the same time. The program cannot call `svc_run`. It must call `select` and `svc_getreqset`.

The server calls `svc_getreqset` when a call to the `select` system call determines that the server has received one or more RPC requests. The `svc_getreqset` routine reads in data for each server connection, then calls the server program to handle the data.

The `svc_getreqset` routine does not return a value. It finishes executing after all sockets associated with the variable `rdfds` have been serviced.

You can use the global variable `svc_fdset` with `svc_getreqset`. The `svc_fdset` variable is the RPC server's read file descriptor bit mask.

To use `svc_fdset`:

1. Copy the global variable `svc_fdset` into a temporary variable.
2. Pass the temporary variable to the `select` routine. The `select` routine overwrites the variable and returns it.
3. Pass the temporary variable to the `svc_getreqset` routine.

Example

```
#define MAXSOCK 10

     int readfds[ MAXSOCK+1],    /* sockets to select from*/
         i, j;

     for(i = 0, j = 0; i
<
< MAXSOCK; i++)
          if((svc_fdset[i].sockname != 0) && (svc_fdset[i].sockname != -1))
               readfds[j++] = svc_fdset[i].sockname;
     readfds[j] = 0;                 /* list of sockets ends with a zero */
     switch(select(0, readfds, 0, 0, 0))
     {
       case -1:      /* an error happened */
       case 0:       /* time out */
            break;
       default:      /* 1 or more sockets ready for reading */
            errno = 0;
            svc_getreqset(readfds);
            if( errno == ENETDOWN || errno == ENOTCONN)
            sys$exit( SS$_THIRDPARTY);
     }
```

Returned Values

None.

svc_register

svc_register — Registers the server program with the Portmapper service.

Syntax

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

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

Arguments

xprt

A pointer to an RPC server handle created by any of the server-handle creation routines

prognum

The program number associated with the server procedure

versnum

The version number associated with the server procedure

dispatch

The address of the service dispatch procedure that the server procedure calls. The procedure `dispatch` has the following form:

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

The `svc_run` and `svc_getreqset` call the `dispatch` routine.

protocol

The protocol that the server procedure uses. Values for this parameter are zero, IPPROTO_UDP, or IPPROTO_TCP. If `protocol` is zero, the service is not registered with the Portmapper service.

Description

Associates `prognum` and `versnum` with the service dispatch procedure `dispatch`. If `protocol` is nonzero, then a mapping of the triple `[prognum, versnum, protocol]` to `xprt->xp_port` is also established with the local Portmapper service.

Returned Values

TRUE	Indicates success.
FALSE	Indicates failure.

svc_run

svc_run — Waits for incoming RPC requests and calls the `svc_getreqset` routine to dispatch to the appropriate RPC server program.

Syntax

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

void svc_run( );
```

Arguments

None.

Description

The `svc_run` routine calls the `select` routine to wait for RPC requests. When a request arrives, `svc_run` calls the `svc_getreqset` routine. Then `svc_run` calls the `select` routine again.

The `svc_run` routine never returns.

You may use the global variable `svc_fdset` with the `svc_run` routine. See the `svc_getreqset` routine for more information about `svc_fdset`.

Returned Values

Never returns.

svc_sendreply

svc_sendreply — Sends the results of a remote procedure call to an RPC client.

Syntax

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

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

Arguments

xprt

A pointer to an RPC server handle created by any of the server-handle creation routines

outproc

The XDR routine used to encode the server procedure's results

out

A pointer to the server procedure's results

Description

Called by an ONC RPC service's dispatch routine to send the results of a remote procedure call.

Returned Values

TRUE	Indicates success.
FALSE	Indicates failure.

svc_unregister

svc_unregister — Calls the Portmapper to unregister the specified program and version for all protocols. The program and version are removed from the list of active servers.

Syntax

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

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

Arguments

prognum

The program number associated with the server procedure

versnum

The version number associated with the server procedure

Description

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

Returned Values

None.

svcerr_auth

svcerr_auth — Sends an authentication error to the client.

Syntax

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

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

Arguments

xprt

A pointer to an RPC server handle created by any of the server-handle creation routines

why

The reason for the authentication error

Description

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

Returned Values

None.

svcerr_decode

svcerr_decode — Sends an error code to the client indicating that the server procedure cannot decode the client's arguments.

Syntax

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

void svcerr_decode(SVCXPRT *xprt);
```

Arguments

xprt

A pointer to an RPC server handle created by any of the server-handle creation routines

Description

Called by a service dispatch routine that cannot successfully decode its parameters. See also the `svc_getargs` routine.

Returned Values

None.

svcerr_noproc

svcerr_noproc — Sends an error code to the client indicating that the server program does not implement the requested procedure.

Syntax

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

void svcerr_noproc(SVCXPRT *xprt);
```

Arguments

xprt

A pointer to an RPC server handle created by any of the server-handle creation routines

Description

Called by a service dispatch routine that does not implement the procedure number that the client requested.

Returned Values

None.

svcerr_noprog

svcerr_noprog — Sends an error code to the client indicating that the server program is not registered with the Portmapper.

Syntax

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

void svcerr_noprog(SVCXPRT *xprt);
```

Arguments

xprt

A pointer to an RPC server handle created by any of the server-handle creation routines

Description

Called when the desired program is not registered with the ONC RPC package. Generally, the Portmapper informs the client when a server is not registered. Therefore, service implementors usually do not use this routine.

Returned Values

None.

svcerr_progvers

svcerr_progvers — Sends an error code to the client indicating that the requested program is registered with the Portmapper but the requested version of the program is not registered.

Syntax

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

void svcerr_progvers(SVCXPRT *xprt, u_long low_vers, u_long high_vers);
```

Arguments

xprt

A pointer to an RPC server handle created by any of the server-handle creation routines

low_vers

The lowest version of the requested program that the server supports

high_vers

The highest version of the requested program that the server supports

Description

Called when the desired version of a program is not registered with the ONC RPC package. Generally, the Portmapper informs the client when a requested program version is not registered. Therefore, service implementors usually do not use this routine.

Returned Values

None.

svcerr_systemerr

svcerr_systemerr — Sends an error code to the client indicating that an error occurred that is not handled by the protocol being used.

Syntax

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

void svcerr_systemerr(SVCXPRT *xprt);
```

Arguments

xprt

A pointer to an RPC server handle created by any of the server-handle creation routines

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.

Returned Values

None.

svcerr_weakauth

svcerr_weakauth — Sends an error code to the client indicating that an authentication error occurred. The authentication information was correct but was insufficient.

Syntax

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

void svcerr_weakauth(SVCXPRT *xprt);
```

Arguments

xprt

A pointer to an RPC server handle created by any of the server-handle creation routines

Description

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

Returned Values

None.

svcraw_create

svcraw_create — Creates a server handle for memory-based ONC RPC for simple testing and timing.

Syntax

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

SVCXPRT *svcraw_create( );
```

Arguments

None.

Description

Creates a in-program ONC RPC service transport, to which it returns a pointer. The transport is really a buffer within the process's address space, so the corresponding client should live in the same address space; see the `clntraw_create` routine. The `svcraw_create` and `clntraw_create` routines allow simulation and acquisition of ONC RPC overheads (such as round-trip times), without any kernel interference.

Returned Values

SVCXPRT *	A pointer to an RPC server handle for the in-memory transport.
NULL	Indicates failure.

svcfd_create

svcfd_create — Creates an RPC server handle using the specified open file descriptor.

Syntax

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

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

Arguments

fd

The number of an open file descriptor

sendsize

The size of the send buffer. If you specify zero, the routine chooses a suitable default

recvsize

The size of the receive buffer. If you specify zero, the routine chooses a suitable default

Description

Creates an RPC server handle using the specified TCP socket, to which it returns a pointer. The server should call the `svcfd_create` routine after it accepts an incoming TCP connection.

Returned Values

SVCXPRT *	A pointer to the server handle.
NULL	Indicates failure.

svctcp_create

svctcp_create — Creates an ONC RPC server handle for a TCP/IP connection.

Syntax

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

SVCXPRT *svctcp_create(int sock, u_int sendsize, u_int recvsize);
```

Arguments

sock

The socket with which the connection is associated. If *sock* is `RPC_ANYSOCK`, then this routine opens a new socket and sets *sock*. If the socket is not bound to a local TCP port, then this routine binds it to an arbitrary port.

sendsize

The size of the send buffer. If you specify zero, the routine chooses a suitable default.

recvsize

The size of the receive buffer. If you specify zero, the routine chooses a suitable default.

Description

Creates an RPC server handle using the TCP/IP transport, to which it returns a pointer. Upon completion, `xprt->xp_sock` is the transport's socket descriptor, and `xprt->xp_port` is the transport's port number. The service is automatically registered as a transporter (thereby including its socket in `svc_fds` such that its socket descriptor is included in all RPC `select` system calls).

Returned Values

SVCXPRT *	A pointer to the server handle.
NULL	Indicates failure.

svcudp_bufcreate

svcudp_bufcreate — Creates an ONC RPC server handle for a buffered I/O UDP connection.

Syntax

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

SVCXPRT *svcudp_bufcreate(int sock, u_int sendsize, u_int recvsize);
```

Arguments

sock

The socket with which the connection is associated. If *sock* is `RPC_ANYSOCK`, then this routine opens a new socket and sets *sock*.

sendsize

The size of the send buffer. If you specify zero, the routine chooses a suitable default.

recvsize

The size of the receive buffer. If you specify zero, the routine chooses a suitable default.

Description

Creates an RPC server handle using the UDP transport, to which it returns a pointer. Upon completion, `xprt->xp_sock` is the transport's socket descriptor, and `xprt->xp_port` is the transport's port number. The service is automatically registered as a transporter (thereby including its socket in `svc_fds` such that its socket descriptor is included in all RPC `select` system calls).

Returned Values

SVCXPRT *	A pointer to the server handle.
NULL	Indicates failure.

svcudp_create

svcudp_create — Creates an ONC RPC server handle for a nonbuffered I/O UDP connection.

Syntax

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

SVCXPRT *svcudp_create(int sock);
```

Arguments

sock

The socket with which the connection is associated. If *sock* is `RPC_ANYSOCK`, then this routine opens a new socket and sets *sock*.

Description

Creates an RPC server handle using the UDP transport, to which it returns a pointer. Upon completion, `xprt->xp_sock` is the transport's socket descriptor, and `xprt->xp_port` is the transport's port number. The service is automatically registered as a transporter (thereby including its socket in `svc_fds` such that its socket descriptor is included in all RPC `select` system calls).

Note

Since UDP/IP-based ONC RPC messages can only hold up to 8 KB of encoded data, this transport cannot be used for procedures that take large arguments or return huge results.

Returned Values

SVCXPRT *	A pointer to the server handle.
NULL	Indicates failure.

xprt_register

xprt_register — Adds a socket associated with an RPC server handle to the list of registered sockets.

Syntax

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

void xprt_register(SVCXPRT *xprt);
```

Arguments

xprt

A pointer to an RPC server handle created by any of the server-handle creation routines

Description

Activation of a transport handle involves setting the most appropriate bit for the socket associated with *xprt* in the `svc_fds` mask. When `svc_run`() is invoked, activity on the transport handle is eligible to be processed by the server.

The `svc_register` routine calls this routine; therefore, you are unlikely to use this routine directly.

Returned Values

None.

xprt_unregister

xprt_unregister — Removes a socket associated with an RPC server handle from the list of registered sockets.

Syntax

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

void xprt_unregister(SVCXPRT *xprt);
```

Arguments

xprt

A pointer to an RPC server handle created by any of the server-handle creation routines

Description

Removes the socket associated with the indicated handle from the list of registered sockets maintained in the `svc_fdset` variable. Activity on the socket associated with *xprt* will no longer be checked by the `svc_run` routine.

The `svc_unregister` routine calls this routine; therefore, you are unlikely to use this routine directly.

Returned Values

None.

_authenticate

_authenticate — Authenticates the request message.

Syntax

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

```
enum auth_stat _authenticate(struct svc_req *rqst, struct rpc_msg *msg);
```

Arguments

rqst

A pointer to an `svc_req` structure with the requested program number, procedure number, version number, and credentials passed by the client.

msg

A pointer to an `rpc_msg` structure with members that make up the RPC message.

Description

Returns `AUTH_OK` if the message is authenticated successfully. If it returns `AUTH_OK`, the routine also does the following:

- Sets `rqst->rq_xprt->verf` to the appropriate response verifier.
- Sets `rqst->rq_client_cred` to the “cooked ” form of the credentials.

The expression `rqst->rq_xprt->verf` must be preallocated and its length must be set appropriately.

The program still owns and is responsible for `msg->u.cmb.cred` and `msg->u.cmb.verf`. The authentication system retains ownership of `rqst->rq_client_cred`, the “cooked ” credentials.

Return Values

enum auth_stat	The return status code for the authentication checks:
	AUTH_OK=0—Authentication checks successful.
	AUTH_BADCRED=1—Invalid credentials(seal broken)
	AUTH_REJECTEDCRED=2—Client should begin new session
	AUTH_BADVERF=3—Invalid verifier (seal broken)
	AUTH_REJECTEDVERF=4—Verifier expired or was replayed
	AUTH_TOOWEAK=5—Rejected for security reasons
	AUTH_INVALIDRESP=6—Invalid response verifier
	AUTH_FAILED=7—Some unknown reason

Chapter 8. XDR Routine Reference

This chapter describes the routines that specify external data representation. They allow C programmers to describe arbitrary data structures in a system-independent fashion. These routines transmit data for remote procedure calls.

Table 8.1, "XDR Data Conversion Routines" indicates the type of task that each routine performs.

Table 8.1. XDR Data Conversion Routines

Routine	Encodes and Decodes...
`xdr_accepted_reply`	Accepted RPC messages
`xdr_array`	Variable-length arrays
`xdr_authunix_parms`	UNIX-style authentication information
`xdr_bool`	Boolean values
`xdr_bytes`	Single bytes
`xdr_callhdr`	Static part of RPC request message headers
`xdr_callmsg`	RPC request messages
`xdr_char`	Single characters
`xdr_double`	Double-precision floating-point numbers
`xdr_enum`	Enumerations
`xdr_float`	Single-precision floating-point numbers
`xdr_hyper`	Quad words (hyperintegers)
`xdr_int`	4-byte integers
`xdr_long`	Longwords
`xdr_opaque`	Fixed-length opaque data structures
`xdr_opaque_auth`	Opaque `opaque_auth` structures containing authentication information
`xdr_pmap`	Portmapper parameters
`xdr_pmap_vms`	Portmapper parameters (including OpenVMS process IDs)
`xdr_pmaplist`	Portmapper lists
`xdr_pmaplist_vms`	Portmapper lists (including OpenVMS process IDs)
`xdr_pointer`	Data structure pointers
`xdr_reference`	Data structure pointers
`xdr_rejected_reply`	Rejected RPC reply messages
`xdr_replymsg`	RPC reply messages
`xdr_short`	2-byte integers
`xdr_string`	Null-terminated strings

Routine	Encodes and Decodes...
xdr_u_char	Unsigned characters
xdr_u_hyper	Unsigned quadwords (hyperintegers)
xdr_u_int	Unsigned 4-byte integers
xdr_u_long	Unsigned long integers
xdr_u_short	Unsigned 2-byte integers
xdr_union	Unions
xdr_vector	Fixed-length arrays
xdr_void	(A dummy routine)
xdr_wrapstring	Null-terminated strings

This chapter also describes the XDR routines that manage XDR streams. They allow C programmers to handle XDR streams in a system-independent fashion.

Table 8.2, "XDR Stream Handling Routines" indicates the type of task that each routine performs.

Table 8.2. XDR Stream Handling Routines

Routine	Task
xdr_free	Deallocates an XDR data structure.
xdrmem_create	Creates an XDR stream handle describing a memory buffer.
xdrrec_create	Creates an XDR stream handle describing a record-oriented TCP-based connection.
xdrrec_endofrecord	Generates an end-of-record indication for an XDR record.
xdrrec_eof	Positions the data pointer to the end of the current XDR record and indicates whether any more records follow the current record.
xdrrec_skiprecord	Positions the data pointer at the end of the current XDR record.
xdrstdio_create	Creates an XDR stream handle describing a `stdio` stream.
xdr_accepted_reply	Accepts RPC messages.

xdr_accepted_reply

xdr_accepted_reply — Serializes and deserializes a message-accepted indication in an RPC reply message.

Syntax

```
#include <tcpip$rpcxdr.h>

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

Arguments

xdrs

A pointer to an XDR stream handle created by one of the XDR stream-handle creation routines.

arp

A pointer to a buffer to which the message-accepted indication is written.

Description

Used for encoding reply messages. This routine encodes the status of the RPC call and, in the case of success, the call results as well. This routine is useful for users who want to generate messages without using the ONC RPC package. It returns the message-accepted variant of a reply message union in the `arp` argument.

The `xdr_replymsg` routine calls this routine.

Return Values

TRUE	Indicates success.
FALSE	Indicates failure to encode the message.

xdr_array

xdr_array — Serializes and deserializes the elements of a variable-length array.

Syntax

```
#include <tcpip$rpcxdr.h>

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

Arguments

xdrs

A pointer to an XDR stream handle created by one of the XDR stream-handle creation routines.

arrp

A pointer to the pointer to the array.

sizep

A pointer to the number of elements in the array. This element count cannot exceed the `maxsize` parameter.

maxsize

The maximum size of the `sizep` parameter. This value is the maximum number of elements that the array can hold.

elsize

The size, in bytes, of each of the array's elements.

elproc

The XDR routine to call that handles each element of the array.

Description

A filter primitive that translates between variable-length arrays and their corresponding external representations.

Return Values

TRUE	Indicates success.
FALSE	Indicates failure.

xdr_authunix_parms

xdr_authunix_parms — Serializes and deserializes credentials in an authentication parameter structure.

Syntax

```
#include <tcpip$rpcxdr.h>

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

Arguments

xdrs

A pointer to an XDR stream handle created by one of the XDR stream-handle creation routines.

authp

A pointer to an `authunix_parms` structure.

Description

Used for externally describing standard UNIX credentials. On a TCP/IP Services host, this routine encodes the host name, the user ID, and the group ID. It sets the group ID list to `NULL`. This routine is useful for users who want to generate these credentials without using the ONC RPC authentication package.

Return Values

TRUE	Indicates success.

FALSE	Indicates failure.

xdr_bool

xdr_bool — Serializes and deserializes boolean data.

Syntax

```
#include <tcpip$rpcxdr.h>

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

Arguments

xdrs

A pointer to an XDR stream handle created by one of the XDR stream-handle creation routines.

bp

A pointer to the boolean data.

Description

A filter primitive that translates between booleans (integers) and their external representations. When encoding data, this filter produces values of either 1 or 0.

Return Values

TRUE	Indicates success.
FALSE	Indicates failure.

xdr_bytes

xdr_bytes — Serializes and deserializes a counted byte array.

Syntax

```
#include <tcpip$rpcxdr.h>

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

Arguments

xdrs

A pointer to an XDR stream handle created by one of the XDR stream-handle creation routines.

bpp

A pointer to a pointer to the byte array.

sizep

A pointer to the length of the byte array.

maxsize

The maximum size of the length of the byte array.

Description

A filter primitive that translates between a variable-length byte array and its external representation. The length of the array is located at `sizep`; the array cannot be longer than `maxsize`. If `*bpp` is `NULL`, `xdr_bytes` allocates `maxsize` bytes.

Return Values

TRUE	Indicates success.
FALSE	Indicates failure.

xdr_callhdr

xdr_callhdr — Serializes and deserializes the static part of a call message header.

Syntax

```
#include <tcpip$rpcxdr.h>

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

Arguments

xdrs

A pointer to an XDR stream handle created by one of the XDR stream-handle creation routines.

chdrp

A pointer to the call header data.

Description

Describes call header messages. This routine is useful for users who want to generate messages without using the ONC RPC package. The `xdr_callhdr` routine encodes the following fields: transaction ID, direction, RPC version, server program number, and server version.

Return Values

TRUE	Indicate success.
FALSE	Indicates failure.

xdr_callmsg

xdr_callmsg — Serializes and deserializes an ONC RPC call message.

Syntax

```
#include <tcpip$rpcxdr.h>

bool_t xdr_callmsg(XDR *xdrs, struct rpc_msg *cmsgp);
```

Arguments

xdrs

A pointer to an XDR stream handle created by one of the XDR stream-handle creation routines.

cmsgp

A pointer to an `rpc_msg` structure that describes the RPC call message.

Description

This routine is useful for users who want to generate messages without using the ONC RPC package. The `xdr_callmsg` routine encodes the following fields: transaction ID, direction, RPC version, server program number, server version number, server procedure number, and client authentication.

The `pmap_rmtcall` and `svc_sendreply` routines call `xdr_callmsg`.

Return Values

TRUE	Indicates success.
FALSE	Indicates failure.

xdr_char

xdr_char — Serializes and deserializes character data.

Syntax

```
#include <tcpip$rpcxdr.h>

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

Arguments

xdrs

A pointer to an XDR stream handle created by one of the XDR stream-handle creation routines.

cp

A pointer to a character.

Description

A filter primitive that translates between internal representations of characters and their XDR representations.

Return Values

TRUE	Indicates success.
FALSE	Indicates failure.

xdr_double

xdr_double — Serializes and deserializes VAX and IEEE double-precision floating-point numbers.

Syntax

```
#include <tcpip$rpcxdr.h>

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

Arguments

xdrs

A pointer to an XDR stream handle created by one of the XDR stream-handle creation routines.

dp

A pointer to the double-precision floating-point number.

Description

A filter primitive that translates between double-precision numbers and their external representations.

This routine is implemented by four XDR routines:

`xdr_double_D`	Converts VAX D-format floating-point numbers.
`xdr_double_G`	Converts VAX G-format floating-point numbers.
`xdr_double_T`	Converts IEEE T-format floating-point numbers.
`xdr_double_X`	Converts IEEE X-format floating-point numbers.

You can reference these routines explicitly or you can use compiler settings to control which routine is used when you reference the `xdr_double` routine.

Return Values

TRUE	Indicates success.

FALSE	Indicates failure.

xdr_enum

xdr_enum — Serializes and deserializes enumerations.

Syntax

```
#include <tcpip$rpcxdr.h>

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

Arguments

xdrs

A pointer to an XDR stream handle created by one of the XDR stream-handle creation routines.

ep

A pointer to the enumeration data.

Description

A filter primitive that translates between enumerations (actually integers) and their external representations.

Return Values

TRUE	Indicates success.
FALSE	Indicates failure.

xdr_float

xdr_float — Serializes and deserializes VAX and IEEE single-precision floating-point numbers.

Syntax

```
#include <tcpip$rpcxdr.h>

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

Arguments

xdrs

A pointer to an XDR stream handle created by one of the XDR stream-handle creation routines.

fp

A pointer to a single-precision floating-point number.

Description

A filter primitive that translates between single-precision floating-point numbers and their external representations.

This routine is implemented by two XDR routines:

`xdr_float_F`	Converts VAX F-format floating-point numbers.
`xdr_float_S`	Converts IEEE T-format floating-point numbers.

You can reference these routines explicitly or you can use compiler settings to control which routine is used when you reference the `xdr_float` routine.

Return Values

TRUE	Indicates success.
FALSE	Indicates failure.

xdr_free

xdr_free — Deallocates the memory associated with the indicated data structure.

Syntax

```
#include <tcpip$rpcxdr.h>

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

Arguments

proc

The XDR routine for the data structure being freed.

objp

A pointer to the data structure to be freed.

Description

Releases memory allocated for the data structure to which `objp` points. The pointer passed to this routine is not freed, but what it points to is freed (recursively). Use this routine to free decoded data that is no longer needed. Never use this routine for encoded data.

Return Values

TRUE	Indicates success.
FALSE	Indicates failure.

xdr_hyper

xdr_hyper — Serializes and deserializes VAX quadwords (known in XDR as hyperintegers).

Syntax

```
#include <tcpip$rpcxdr.h>

bool_t xdr_hyper(XDR *xdrs, quad *hp);
```

Arguments

xdrs

A pointer to an XDR stream handle created by one of the XDR stream-handle creation routines.

hp

A pointer to the hyperinteger data.

Description

A filter primitive that translates between hyperintegers and their external representations.

Return Values

TRUE	Indicates success.
FALSE	Indicates failure.

xdr_int

xdr_int — Serializes and deserializes integers.

Syntax

```
#include <tcpip$rpcxdr.h>

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

Arguments

xdrs

A pointer to an XDR stream handle created by one of the XDR stream-handle creation routines.

ip

A pointer to the integer data.

Description

A filter primitive that translates between integers and their external representations.

Return Values

TRUE	Indicates success.
FALSE	Indicates failure.

xdr_long

xdr_long — Serializes and deserializes long integers.

Syntax

```
#include <tcpip$rpcxdr.h>

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

Arguments

xdrs

A pointer to an XDR stream handle created by one of the XDR stream-handle creation routines.

lp

A pointer to a long integer.

Description

A filter primitive that translates between long integers and their external representations.

Return Values

TRUE	Indicates success.
FALSE	Indicates failure.

xdr_opaque

xdr_opaque — Serializes and deserializes opaque structures.

Syntax

```
#include <tcpip$rpcxdr.h>

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

Arguments

xdrs

A pointer to an XDR stream handle created by one of the XDR stream-handle creation routines.

op

A pointer to the opaque data.

cnt

The size of op in bytes.

Description

A filter primitive that translates between fixed-size opaque data and its external representation. This routine treats the data as a fixed length of bytes and does not attempt to convert the bytes.

Return Values

TRUE	Indicates success.
FALSE	Indicates failure.

xdr_opaque_auth

xdr_opaque_auth — Serializes and deserializes ONC RPC authentication information message.

Syntax

```
#include <tcpip$rpcxdr.h>

bool_t xdr_opaque_auth(XDR *xdrs, struct opaque_auth *authp);
```

Arguments

xdrs

A pointer to an XDR stream handle created by one of the XDR stream-handle creation routines.

authp

A pointer to an `opaque_auth` structure describing authentication information. The pointer should reference data created by the `authnone_create`, `authunix_create`, or `authunix_create_default` routine.

Description

Translates ONC RPC authentication information messages. This routine is useful for users who want to generate messages without using the ONC RPC package.

Return Values

TRUE	Indicates success.
FALSE	Indicates failure.

xdr_pmap

xdr_pmap — Serializes and deserializes Portmapper parameters.

Syntax

```
#include <tcpip$rpcxdr.h>

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

Arguments

xdrs

A pointer to an XDR stream handle created by one of the XDR stream-handle creation routines.

regs

A pointer to the `pmap` structure. This structure contains the program number, version number, protocol number, and port number.

Description

Describes parameters to various Portmapper procedures, externally. This routine is useful for users who want to generate these parameters without using the Portmapper interface.

Return Values

TRUE	Indicates success.
FALSE	Indicates failure.

xdr_pmap_vms

xdr_pmap_vms — Serializes and deserializes OpenVMS specific Portmapper parameters.

Syntax

```
#include <tcpip$rpcxdr.h>

bool_t xdr_pmap_vms(XDR *xdrs, struct pmap_vms *regs);
```

Arguments

xdrs

A pointer to an XDR stream handle created by one of the XDR stream-handle creation routines.

regs

A pointer to the `pmap_vms` structure. This structure contains the program number, version number, protocol number, port number and the OpenVMS specific process identification.

Description

This routine is similar to `xdr_pmap()`, except it also includes the process identification in the `pmap_vms` structure.

Return Values

TRUE	Indicates success.
FALSE	Indicates failure.

xdr_pmaplist

xdr_pmaplist — Serializes and deserializes a list of Portmapper port mappings.

Syntax

```
#include <tcpip$rpcxdr.h>

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

Arguments

xdrs

A pointer to an XDR stream handle created by one of the XDR stream-handle creation routines.

rpp

A pointer to a pointer to a `pmaplist` structure containing a list of Portmapper programs and their respective information. If the routine is used to decode a Portmapper listing, it sets `rpp` to the address of a newly allocated linked list of `pmaplist` structures.

Description

Describes a list of port mappings, externally. This routine is useful for users who want to generate these parameters without using the Portmapper interface.

Return Values

TRUE	Indicates success.
FALSE	Indicates failure.

xdr_pmaplist_vms

xdr_pmaplist_vms — Serializes and deserializes a list of Portmapper port mappings for OpenVMS systems.

Syntax

```
#include <tcpip$rpcxdr.h>
```

```
bool_t xdr_pmaplist_vms (XDR *xdrs, struct pmaplist_vms **rpp);
```

Arguments

xdrs

A pointer to an XDR stream handle created by one of the XDR stream-handle creation routines.

rpp

A pointer to a pointer to a `pmaplist_vms` structure containing a list of Portmapper programs and their respective information, including OpenVMS-specific information.

Description

This routine is similar to the `xdr_pmaplist` routine, except that it also includes the process identification in the `pmaplist_vms` structure.

Return Values

TRUE	Indicates success.
FALSE	Indicates failure.

xdr_pointer

xdr_pointer — Serializes and deserializes indirect pointers and the data being pointed to.

Syntax

```
#include <tcpip$rpcxdr.h>

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

Arguments

xdrs

A pointer to an XDR stream handle created by one of the XDR stream-handle creation routines.

objpp

A pointer to a pointer to the data being converted.

objsize

The size of the data structure in bytes.

objproc

The XDR procedure that filters the structure between its local form and its external representation.

Description

An XDR routine for translating data structures that contain pointers to other structures, such as a linked list. The `xdr_pointer` routine is similar to the `xdr_reference` routine. The differences are that the `xdr_pointer` routine handles pointers with the value `NULL` and that it translates the pointer values to a boolean. If the boolean is TRUE, the data follows the boolean.

Return Values

TRUE	Indicates success.
FALSE	Indicates failure.

xdr_reference

xdr_reference — Serializes and deserializes indirect pointers and the data being pointed to.

Syntax

```
#include <tcpip$rpcxdr.h>

bool_t xdr_reference(XDR *xdrs, char **objpp, u_int objsize, xdrproc_t
 objproc);
```

Arguments

xdrs

A pointer to an XDR stream handle created by one of the XDR stream-handle creation routines.

objpp

A pointer to a pointer to the structure containing the data being converted. If `objpp` is zero, the `xdr_reference` routine allocates the necessary storage when decoding. This argument must be nonzero during encoding.

objsize

The size of the structure in bytes.

objproc

The XDR procedure that filters the structure between its local form and its external representation.

Description

A primitive that provides pointer chasing within structures.

Return Values

TRUE	Indicates success.

FALSE	Indicates failure.

xdr_rejected_reply

xdr_rejected_reply — Serializes and deserializes the remainder of an RPC reply message after the header indicates that the reply is rejected.

Syntax

```
#include <tcpip$rpcxdr.h>

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

Arguments

xdrs

A pointer to an XDR stream handle created by one of the XDR stream-handle creation routines.

rrp

A pointer to the `rejected_reply` structure describing the rejected reply message.

Description

Describes ONC RPC reply messages. This routine is useful for users who want to generate messages without using the ONC RPC package.

Return Values

TRUE	Indicates success.
FALSE	Indicates failure.

xdr_replymsg

xdr_replymsg — Serializes and deserializes the RPC reply header and then calls the appropriate routine to interpret the rest of the message.

Syntax

```
#include <tcpip$rpcxdr.h>

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

Arguments

xdrs

A pointer to an XDR stream handle created by one of the XDR stream-handle creation routines.

rmsgp

A pointer to the `rpc_msg` structure describing the reply message.

Description

Describes ONC RPC reply messages. This routine is useful for users who want to generate messages without using the ONC RPC package. This routine interprets the message header and then calls either the `xdr_accepted_reply` or the `xdr_rejected_reply` routine to interpret the body of the RPC message.

Return Values

TRUE	Indicates success.
FALSE	Indicates failure.

xdr_short

xdr_short — Serializes and deserializes short integers.

Syntax

```
#include <tcpip$rpcxdr.h>

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

Arguments

xdrs

A pointer to an XDR stream handle created by one of the XDR stream-handle creation routines.

sp

A pointer to a short integer.

Description

A filter primitive that translates between short integers and their external representations.

Return Values

TRUE	Indicates success.
FALSE	Indicates failure.

xdr_string

xdr_string — Serializes and deserializes strings (arrays of bytes terminated by a `NULL` character).

Syntax

```
#include <tcpip$rpcxdr.h>

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

Arguments

xdrs

A pointer to an XDR stream handle created by one of the XDR stream-handle creation routines.

spp

A pointer to a pointer to a character string.

maxsize

The maximum size of the string.

Description

A filter primitive that translates between strings and their corresponding external representations. Strings cannot be longer than the value specified with the `maxsize` parameter.

While decoding, if `*spp` is `NULL`, this routine allocates the necessary storage to hold the `NULL`-terminated string and sets `*spp` to point to the allocated storage.

This routine is the same as the `xdr_wrapstring` routine, except that this routine allows you to specify `maxsize`.

Return Values

TRUE	Indicates success.
FALSE	Indicates failure.

xdr_u_char

xdr_u_char — Serializes and deserializes unsigned characters.

Syntax

```
#include <tcpip$rpcxdr.h>

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

Arguments

xdrs

A pointer to an XDR stream handle created by one of the XDR stream-handle creation routines.

ucp

A pointer to a character.

Description

A filter primitive that translates between internal representation of unsigned characters and their XDR representations.

Return Values

TRUE	Indicates success.
FALSE	Indicates failure.

xdr_u_hyper

xdr_u_hyper — Serializes and deserializes unsigned VAX quadwords (known in XDR as hyperintegers).

Syntax

```
#include <tcpip$rpcxdr.h>

bool_t xdr_u_hyper(XDR *xdrs, unsigned quad *uhp);
```

Arguments

xdrs

A pointer to an XDR stream handle created by one of the XDR stream-handle creation routines.

uhp

A pointer to the unsigned hyperinteger.

Description

A filter primitive that translates between unsigned hyperintegers and their external representations.

Return Values

TRUE	Indicates success.
FALSE	Indicates failure.

xdr_u_int

xdr_u_int — Serializes and deserializes unsigned integers.

Syntax

```
#include <tcpip$rpcxdr.h>
```

```
bool_t xdr_u_int(XDR *xdrs, unsigned *uip);
```

Arguments

xdrs

A pointer to an XDR stream handle created by one of the XDR stream-handle creation routines.

uip

A pointer to the unsigned integer.

Description

A filter primitive that translates between unsigned integers and their external representations.

Return Values

TRUE	Indicates success.
FALSE	Indicates failure.

xdr_u_long

xdr_u_long — Serializes and deserializes unsigned long integers.

Syntax

```
#include <tcpip$rpcxdr.h>

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

Arguments

xdrs

A pointer to an XDR stream handle created by one of the XDR stream-handle creation routines.

ulp

A pointer to the unsigned long integer.

Description

A filter primitive that translates between unsigned long integers and their external representations.

Return Values

TRUE	Indicates success.
FALSE	Indicates failure.

xdr_u_short

xdr_u_short — Serializes and deserializes unsigned short integers.

Syntax

```
#include <tcpip$rpcxdr.h>

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

Arguments

xdrs

A pointer to an XDR stream handle created by one of the XDR stream-handle creation routines.

usp

A pointer to the unsigned short integer.

Description

A filter primitive that translates between unsigned short integers and their external representations.

Return Values

TRUE	Indicates success.
FALSE	Indicates failure.

xdr_union

xdr_union — Serializes and deserializes discriminant unions.

Syntax

```
#include <tcpip$rpcxdr.h>

bool_t xdr_union(XDR *xdrs, enum *dscmp, char *unp, struct xdr_discrim
 *choices, xdrproc_t default);
```

Arguments

xdrs

A pointer to an XDR stream handle created by one of the XDR stream-handle creation routines.

dscmp

A pointer to the union's discriminant.

unp

A pointer to the union's data.

choices

A pointer to an array of `xdr_discrim` structures. Each structure contains an ordered pair of `[value,proc]`. The final structure in the array is denoted by a pointer with the value `NULL`.

default

The address of the default XDR routine to call if the `dscmp` argument is not found in the `choices` array.

Description

A filter primitive that translates between a discriminated union and its corresponding external representation. The `xdr_union` routine first translates the discriminant of the union located at `dscmp`. This discriminant is always of type `enum_t`.

Next, the routine translates the union data located at `unp`. To translate the union data the `xdr_union` routine first searches the structure pointed to by the `choices` argument for the union discriminant passed in the `dscmp` argument. If a match is found, the `xdr_union` routine calls `proc` to translate the union data.

The end of the `xdr_discrim` structure array must contain an entry with the value `NULL` for `proc`. If the `xdr_union` routine reaches this entry before finding a match, the routine calls the `default` procedure (if it is not `NULL`).

Return Values

TRUE	Indicates success.
FALSE	Indicates failure.

xdr_vector

xdr_vector — Serializes and deserializes the elements of a fixed-length array (known as a vector).

Syntax

```
#include <tcpip$rpcxdr.h>

bool_t xdr_vector(XDR *xdrs, char **vecpp, u_int elnum, u_int elsize,
 xdrproc_t elproc);
```

Arguments

xdrs

A pointer to an XDR stream handle created by one of the XDR stream-handle creation routines.

vecpp

A pointer to a pointer to the array.

elnum

The number of elements in the array.

elsize

The size, in bytes, of each element.

elproc

The XDR routine to handle each element.

Description

A routine that calls `elproc` to prepare the elements of an array for XDR messages.

Return Values

TRUE	Indicates success.
FALSE	Indicates failure.

xdr_void

xdr_void — When there is no data to convert, this routine is passed to ONC RPC routines that require an XDR procedure parameter.

Syntax

```
#include <tcpip$rpcxdr.h>

bool_t xdr_void( );
```

Description

This routine is used as a placeholder for a program that passes no data in a remote procedure call. Most client and server routines expect an XDR routine to be called, even when there is no data to pass.

Return Values

This routine always returns TRUE.

xdr_wrapstring

xdr_wrapstring — Serializes and deserializes `NULL`-terminated strings.

Syntax

```
#include <tcpip$rpcxdr.h>
```

```
bool_t xdr_wrapstring(XDR *xdrs, char **spp);
```

Arguments

xdrs

A pointer to an XDR stream handle created by one of the XDR stream-handle creation routines.

spp

A pointer to a pointer to a string.

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 ONC RPC client and server routines pass the XDR stream handle and a single pointer as parameters to any referenced XDR routines. The `xdr_string` routine, one of the most frequently used ONC RPC primitives, requires three parameters.

While decoding, if `*sp` is `NULL`, the necessary storage is allocated to hold the NULL-terminated string and `*sp` is set to point to it.

Return Values

TRUE	Indicates success.
FALSE	Indicates failure.

xdrmem_create

xdrmem_create — Initializes an XDR stream descriptor for a memory buffer.

Syntax

```
#include <tcpip$rpcxdr.h>

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

Arguments

xdrs

A pointer to the XDR stream handle being created. The routine `xdrmem_create` fills in `xdrs` with encoding and decoding information.

addr

A pointer to the memory buffer.

size

The length of the memory buffer.

op

An XDR operation, one of: `XDR_ENCODE`, `XDR_DECODE`, and `XDR_FREE`.

Description

The stream handle `xdrs` is initialized with the operation `op`, the buffer `addr` and `size`, and the operations context for an `xdrmem` stream.

Return Values

None.

xdrrec_create

xdrrec_create — Initializes a record-oriented XDR stream descriptor.

Syntax

```
#include <tcpip$rpcxdr.h>

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

Arguments

xdrs

A pointer to the XDR stream handle being created. The routine `xdrrec_create` fills in `xdrs` with encoding and decoding information.

sendsize

The send buffer size.

recvsize

The receive buffer size.

tcp_handle

A pointer to an opaque handle that is passed as the first parameter to the procedures (*readit)() and (*writeit)().

(* readit)()

Read procedure that takes the opaque handle `tcp_handle`. The routine must use the following format:

```
int readit(char *tcp_handle, char *buffer, u_long len)
```

where tcp_handle is the client or server handle, buffer is the buffer to fill, and len is the number of bytes to read. The `readit` routine should return either the number of bytes read or the value `-1` if an error occurs.

(* writeit)()

Write procedure that takes the opaque handle `tcp_handle`. The routine must use the following format:

```
int writeit(char *tcp_handle, char *buffer, u_long len)
```

where tcp_handle is the client or server handle, buffer is the buffer to write, and len is the number of bytes to write. The `readit` routine should return either the number of bytes written or the value `-1` if an error occurs.

Description

The stream descriptor for `xdrs` initializes the maximum allowable size for a request `recvsize` and reply `sendsize`, the addresses of the routine to perform the read `(readit)` and write `(writeit)`, and the TCP handle used for network I/O.

Return Values

None.

xdrrec_endofrecord

xdrrec_endofrecord — Generates an end-of-record for an XDR record.

Syntax

```
#include <tcpip$rpcxdr.h>

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

Arguments

xdrs

A pointer to an XDR stream handle created by one of the XDR stream-handle creation routines.

sendnow

Indicates whether the record should be sent. If *sendnow* is TRUE, `xdrrec_endofrecord` sends the record by calling the `writeit` routine specified in the call to `xdrrec_create`. If sendnow is FALSE, `xdrrec_endofrecord` marks the end of the record and calls `writeit` when the buffer is full.

Description

This routine lets an application support batch calls and pipelined procedure calls.

Return Values

TRUE	Indicates success.

FALSE	Indicates failure.

xdrrec_eof

xdrrec_eof — Moves the buffer pointer to the end of the current record and returns an indication if any more data exists in the buffer.

Syntax

```
#include <tcpip$rpcxdr.h>

bool_t xdrrec_eof (XDR *xdrs);
```

Arguments

xdrs

A pointer to an XDR stream handle created by one of the XDR stream-handle creation routines.

Description

Returns TRUE if there is no more input in the buffer after consuming the rest of the current record.

Return Values

TRUE	Indicates success.
FALSE	Indicates failure.

xdrrec_skiprecord

xdrrec_skiprecord — Guarantees proper record alignment during deserialization from an incoming stream.

Syntax

```
#include <tcpip$rpcxdr.h>

bool_t xdrrec_skiprecord (XDR *xdrs);
```

Arguments

xdrs

A pointer to an XDR stream handle created by one of the XDR stream-handle creation routines.

Description

This routine ensures that the stream is properly aligned in preparation for a subsequent read. It is recommended that, when a record stream is being used, this routine be called prior to any operations that would read from the stream.

This routine is similar to the `xdrrec_eof` routine, except that this routine does not verify whether there is more data in the buffer.

Return Values

TRUE	Indicates success.
FALSE	Indicates failure.

xdrstdio_create

xdrstdio_create — Initializes an `stdio` XDR stream.

Syntax

```
#include <tcpip$rpcxdr.h>

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

Arguments

xdrs

A pointer to the XDR stream handle being created. The routine `xdrstdio_create` fills in `xdrs` with encoding and decoding information.

file

A pointer to the `FILE` structure that is to be associated with the stream.

op

An XDR operation, one of: `XDR_ENCODE`, `XDR_DECODE`, and `XDR_FREE`.

Description

Initializes a `stdio` stream for the specified file.

Return Values

None.