

Distinct® ONC RPC/XDR

**Remote Procedure Calls
and
External Data Representation
Dynamic Link Library
for Microsoft® Windows™**

Version 4.0

**Copyright © 1988 - 2002 by Distinct Corporation
All rights reserved**

Distinct Corporation

3315 Almaden Expressway, Suite 10

San Jose, CA 95118 - USA

Tel: (408) 445 - 3270

Fax: (408) 445 - 3274

[Http://www.distinct.com](http://www.distinct.com)

Disclaimer

Distinct Corporation makes no warranties as to the contents of this documentation and specifically disclaims any implied warranties of merchantability or fitness for any particular purpose. Information in this manual is subject to change without notice and does not represent a commitment on the part of Distinct Corporation. The software described in this manual is furnished under a license agreement and may be used or copied only in accordance with the terms of that agreement.

Copyright Notice

Copyright © 1988 - 2002 by Distinct Corporation. All rights reserved.

No part of this publication may be reproduced, transmitted, transcribed, stored in any retrieval system, or translated into any language by any means without the express written permission of Distinct Corporation.

Trademarks

Distinct is a registered trademark and Distinct RPC is a trademark of Distinct Corporation. UNIX is a registered trademark in the United States and other countries, licensed exclusively through X/Open Company Limited. Microsoft Windows is a registered trademark of Microsoft Corporation. Solaris is a registered trademark of Sun Microsystems. Borland is a registered trademark and Delphi is a trademark of Borland Software Corporation. All other product names are trademarks or registered trademarks of their respective owners.

Version 4.0

Printed in the United States of America.

Table of Contents

Overview	5
Introduction	5
RPC/XDR-32 Toolkit Files	7
Using RPC/XDR-32 Library Functions	8
Registry Entries	9
RPC Components	11
Special Considerations	12
Authentication Issues	13
External Data Representation	14
RPCBIND and RPC Information	21
The RPC Protocol Compiler: RPCGEN	23
Samples	25
RPC Error Return Values	27
D32-RPC.H	28
D32-RPC.PAS	41
Getting Started with Distinct ONC RPC/XDR	63
Before you Start	63
Using the Toolkit in the Microsoft C++ Environment	63
Using the Toolkit in the Borland C++ Environment	64
Facts about the RPC Deployment Licenses	65
Reference	66
auth_destroy ()	66
authdes_create ()	67
authnone_create ()	68
authsys_create ()	69
authunix_create ()	70
callrpc ()	71
clnt_broadcast ()	72
clnt_call ()	73
clnt_control ()	74
clnt_create ()	75
clnt_destroy ()	76
clnt_freeres ()	77
clnt_geterr ()	78
clnt_geterror ()	79
clnt_perror ()	80
clnt_sperror ()	81
clnttcp_create ()	82
clntudp_bufcreate ()	83
clntudp_create ()	84
pmap_getmaps ()	85
pmap_getport ()	86
pmap_rmtcall ()	87
pmap_set ()	88
pmap_unset ()	89
rac_drop ()	90
rac_poll ()	91
rac_recv ()	92
rac_send ()	93
registerrpc ()	94
rpc_broadcast ()	95

rpc_call ()	96
svc_destroy ()	97
svc_exit ()	98
svc_fdset ()	99
svc_freeargs ()	100
svc_getargs ()	101
svc_getcaller ()	102
svc_getreqset ()	103
svc_register ()	104
svc_run ()	105
svc_run_ex ()	106
svc_sendreply ()	107
svc_set_timeout ()	108
svc_unregister ()	109
svc_update_threads ()	110
svcerr_auth ()	111
svcerr_decode ()	112
svcerr_noproc ()	113
svcerr_noprogram ()	114
svcerr_progvers ()	115
svcerr_systemerr ()	116
svcerr_weakauth ()	117
svctcp_create ()	118
svctcp_create_secure ()	119
svcudp_bufcreate ()	120
svcudp_create ()	121
svcudp_create_secure ()	122
Database Functions	123
Advanced Topics	125
Broadcast RPC	125
Batching	127
Authentication	128
Setting up DES Authentication for Secure RPC	129
RPC Callback and RPC Transient Program Numbers	131
Multithreaded RPC Programming	132
Asynchronous RPC Client Call	133

Overview

Introduction

With Distinct ONC RPC/XDR you are able to include Windows workstations and Windows-based servers in your distributed application across heterogeneous networks. The Distinct ONC RPC/XDR-Toolkit implements the Sun Microsystems RPC standard and contains all necessary tools for developing both Windows client and Windows server portions of a **distributed network application** using remote procedure calls (RPCs). Such an application will be able to communicate with RPC clients and servers running on different systems, as data is transferred using the external data representation (XDR) in a format independent of the processors and operating systems involved. The Distinct RPC/XDR-library is a Microsoft Windows dynamic link library (DLL), which can be used by any Windows application. It is an extension of the Microsoft Windows Software Development Kit (SDK) and allows a Windows application to issue remote procedure calls to a network server and to register its own remote procedure services for use by one or more network clients.

Remote Procedure Calls (RPCs) are similar to local procedure calls. In a local procedure call, the program places the parameters in a well-known place (usually the stack), transfers control to the procedure and, when the procedure completes, resumes execution. There is an inherent assumption here that the local system is capable of executing a procedure correctly and efficiently, which is often not correct. For example, a small computer running Windows may not have the resources to execute complicated floating point operations or have the disk space to store and query a large database. With remote procedure calls, any computer can make procedure calls to another remote machine.

To achieve this, the program makes a normal function call to a local procedure called the **client stub** with the necessary arguments. The client stub, after any necessary local operations, locates the server and sends the message to the **server stub**, which executes a local procedure call with the given parameters. When done, the server stub contacts the client stub with the results and the client stub returns this result to the program which can at this point resume execution.

The **client portion** of the Distinct ONC RPC/XDR library contains both a high level and a low level interface to RPC calls. The **high level interface** consists of `rpc_call` and `callrpc` (the two functions only differ in how the server address is specified). Both functions default to the UDP/IP transport protocol. To use the **low level interface**, the application must first call `clntudp_create` or `clnttcp_create` to set up an RPC client connection using either the UDP/IP or the TCP/IP transport protocol. Then one or more RPC calls can be issued by calling `clnt_call`. Finally, the application must free all resources allocated to the RPC client connection by calling `clnt_destroy`. Both interface levels include the same functionality. The caller must provide pointers to the argument ('out') and result ('in') buffers (usually containing basic data types or structures, sometimes arrays or even linked lists) along with pointers to the routines used to convert data to and from a host independent network format: the external data representation (XDR). The outgoing data is then encoded into network format using the XDR routine specified and the RPC call is sent to the server. When the RPC reply is received, the incoming data is decoded into host (local) format using the given XDR routine. All necessary byte swapping is performed automatically by the XDR routines. RPC calls do not return to the caller until an answer has been received or until a timeout has occurred. A detailed description of the above mentioned client RPC routines are given in the reference section of this manual.

The Distinct RPC/XDR-library includes an RPCBIND daemon, which is dynamically loaded when the first RPC server registers a TCP/IP or UDP/IP service. The RPC Information utility (RPCINF32.EXE) can be used to obtain a listing of all registered services and their ports either on the local machine or on other hosts on the network. Please refer to the section 'RPCBIND and RPC Information' for more details. Server applications can register a service using `pmap_set` and can unregister a service using `pmap_unset`. The `pmap_getport` function can be used to look up the port number of an RPC service registered on any host (including the local machine). With the `pmap_getmaps` function, an application can obtain a list of all the RPC services registered on the local or any remote machine. A detailed description of the four above mentioned port mapper routines is given in the reference section of this manual.

The **server portion** of the Distinct RPC/XDR-library allows a Windows application or DLL to register one or more services for the UDP/IP or TCP/IP transport protocol. Each such service has associated with it an XDR encode routine, an XDR decode routine and a service routine to be called by the port mapper daemon whenever a request for the service arrives. A server process associates its service number (e.g. 100002 for the rusers service) with an actual port on the local machine by first allocating the necessary resources with a call to **svctcp_create** or **svcudp_create** depending on the desired transport protocol. Many servers provide the same service for both UDP/IP and TCP/IP. In this case, the same service routine and the same XDR routines can be used for both transport protocols. Next, the port mapper is instructed to add the new service to the list of available services on the local machine by calling **svc_register**. Every time an RPC request for a particular service arrives on the local port, the port mapper calls the associated service routine. Upon completion of the request, the service routine must call **svc_sendreply** to transmit the reply buffer to the client (in case of an error, the **svcerr_** functions may be used to return an error message). Before the server terminates, it must call **svc_destroy** for all of the services it registered to free all resources allocated to the services. A detailed description of the above mentioned server RPC routines are given in the reference section of this manual.

External data representation (XDR) is used to encode data into a **host independent network data format** and decode network data back into the host specific data format. Data sent as an argument to an RPC call and the data making up the reply part of the call are transferred in network data format. Before this data can be used, the correct XDR routine must be called for decoding. Any byte swapping or differences in the length of data elements (for example, two versus four byte integers) are handled transparently in the XDR routines. The **XDR primitives** included in this library can be used to encode or decode call arguments and results directly or to build more complex **custom XDR routines**. Short descriptions and function prototypes of all XDR primitives are given in the section 'External Data Representation'. For information on how to generate C source code for custom XDR routines which handle arbitrary data structures, please refer to the section - RPC Protocol Compiler (RPCGEN)!

The Distinct ONC RPC/XDR library also allows encoding and decoding of data to and from XDR memory buffers ('streams') without actually making a remote procedure call. Applications can, for example, encode and decode data that is transmitted over sockets with a protocol other than RPC. For more information on the XDR stream functions **xdrmem_create**, **xdrstdio_create**, **xdrrec_create**, **xdr_destroy**, **xdr_putlong**, **xdr_getlong**, **xdr_putbytes**, **xdr_getbytes**, **xdr_setpos** and **xdr_getpos**, please refer to the section 'External Data Representation'.

To illustrate the use of the Distinct ONC RPC/XDR library, **sample RPC programs** with source code were copied to your hard disk during installation.

RPC/XDR-32 Toolkit Files

The Distinct ONC RPC/XDR toolkit is a complete 32-bit environment to enable development of remote procedure calls. With the Distinct ONC RPC/XDR library it is simple to write the three necessary portions of RPC (i.e. the client side, the server side and the XDR portion - the port mapper is built into the system). The Distinct ONC RPC/XDR toolkit provides the necessary include files, libraries, the DLL and other utilities, which make the development of RPC, based client/server routines very simple. The Distinct ONC RPC/XDR toolkit includes the following files.

D32-RPC.DLL

This file contains the actual dynamic link library, which resolves all the RPC client calls.

DRPCSRVR.DLL

This file contains the actual dynamic link library, which resolves all the RPC server calls.

D32-RPC.H

This C and C++ header file includes definitions of all constants and structures used in Distinct RPC/XDR-library functions in addition to function prototypes. It should be included in all source modules to allow for argument type checking.

D32-RPC.LIB

This library link file must be added to your project or must be specified on the linker command line to resolve references to Distinct RPC/XDR library functions. This is needed when writing either a client or a server application.

DRPCSRV.LIB

This library must be added to your project or must be specified on the linker command line to resolve references to Distinct RPC/XDR library functions. This is needed only when writing a server application.

NEWKEY.EXE

This tool generates a pair of Diffie-Hellman public key and secret key for Secure RPC applications, where the secret key is usually encrypted with a password. The key pair generated by this application needs to be stored in the *publickey* database.

RPCBIND.EXE

The RPCBIND daemon is started automatically when the first RPC service is registered. RPCBIND is designed to automatically close once the last service that is registered with it is closed. You may however opt to install RPCBIND as a service on NT, 2000 and XP systems.

RPCGEN.EXE

The RPC protocol compiler generates C source code and C header files for XDR routines for your custom data types from an XDR definition file you supply. Refer to section '[1.10 - RPC Protocol Compiler \(RPCGEN\)](#)'.

RPCINFO.EXE

This application obtains a list of all registered RPC services from the local machine or from any host on the network. Refer to section '[1.9 - RPCBIND and RPC Information](#)'.

In addition to the above files, sample source code for some RPC based procedures is included to illustrate the development process for RPC based applications.

Using RPC/XDR-32 Library Functions

The Distinct RPC/XDR library functions are called from your application. Because they are all defined as WINAPI (required for routines in DLLs) and to allow for type checking, they must be forward declared before they can be used. All Distinct ONC RPC/XDR library routines are declared in the include file D32-RPC.H. By including the statement

```
#include <d32-rpc.h>
```

at the beginning of each C or C++ source file, all RPC/XDR-library functions and their parameters will be automatically declared.

The Distinct ONC RPC/XDR library is provided as a DLL, which is loaded when the program starts executing. However, to resolve the function calls provided with the Distinct ONC RPC/XDR library, the library file D32-RPC.LIB must be linked to the CLIENT program when generating the client executable or DLL, D32-RPC.LIB & DRPCSRVR.LIB must be linked to the SERVER program when generating the server executable or DLL.

Other Microsoft Windows development environments (such as Borland® C++ Builder and, Borland Delphi™) also allow calls to third-party DLLs. The appropriate options or settings must be used to link to the D32-RPC.LIB file to produce the executable or DLL.

Using the Distinct ONC RPC/XDR library directly from Microsoft **Visual Basic** is not supported. To use remote procedure calls in a VB application, a dynamic link library (DLL) must be written in an environment that supports callback functions (such as C or Pascal). Then the VB application can use the DLL as an interface to the Distinct ONC RPC/XDR library.

Registry Entries

Various parameters used to control the behavior of the Distinct ONC RPC/XDR library are stored under the following registry path.

HKEY_LOCAL_MACHINE\SOFTWARE\Distinct\DLLS\RPC32

The following entries, listed with their correspondent types, default values, and allowable ranges, specifies the various settings.

BufferSize **REG_DWORD** **9216** **512 -**

Specifies the send and receive buffer size, in Bytes, to be used. Default is set to 9216 bytes. The maximum size of the packet sent over the UDP protocol is limited. Usually it is 64 kilobytes. Care must be taken not to exceed the maximum packet size of the underlying subnets, which can be obtained by using function **getsockopt** provided by **winsock**. Generally, the data to be transferred becomes four times after encoding and approximately 1 kilobyte of space is required by the rpc header, which is sent for each data transfer. Hence, make sure that amount of data being transferred does not exceed the specified buffer sizes. If the data to transfer is more than the default size, either change this registry value to the desired size or specify the desired size while creating the client and/or server handles.

DebugLevel **REG_DWORD** **0** **0 - 2**

This entry is only required is requested by Distinct Technical Support. It specifies the level debug information to be written into a log file. The debug information from D32-RPC.DLL will be written to D32-RPC.DBG in the same directory where D32-RPC.DLL resides. The debug information from DRPCSRVR.DLL will be written to DRPCSRVR.DBG in the same directory where DRPCSRVR.DLL resides.

MultiThread **REG_DWORD** **1** **0 - 1**

Specifies whether to create worker threads or not. 0 disables the creation and 1 enables it. By default it is enabled. This value is useful only if function `svc_run` is called to service the client requests. If you use `svc_run_ex` to service the client requests, this value is ignored.

RequestListSize **REG_DWORD** **128** **1 -**

Specifies the maximum number of unserviced requests the server can hold or queue. The default value is 128. This value is used only if the `svc_run_ex` function is used for servicing client requests and it has one or more worker threads.

Resend **REG_DWORD** **5** **0 -**

Specifies the number of seconds client waits before retransmitting the request in case it did not receive an answer from the server. This value applies only if the UDP protocol is used. The default value is 5 seconds.

SelectTimeout **REG_DWORD** **60** **0 -**

Specifies the number of seconds that the server waits for RPC requests before checking for the exit flag. The default value is 60 seconds.

Timeout **REG_DWORD** **20** **0 -**

Specifies the number of seconds the client waits for an answer from the server before returning a timeout error. The default value is 20 seconds.

Note that setting 'Resend' to a higher value than 'Timeout' will disable retransmission.

Various parameters used to configure RPCBIND are stored under the following registry path

HKEY_LOCAL_MACHINE\SOFTWARE\Distinct\DLLS\RPC32\RPCBind

The following entry applies to the RPCBIND daemon.

ShutDown **REG_DWORD** **10** **0 -**

Specifies the time in seconds after which RPCBIND will automatically shut down if no servers are registered with it. Shut down will occur only if last service registered with it unregisters successfully. Automatic shut down can be disabled by setting this value to 0. Note that is you install RPCBIND as a service this value is ignored.

RPC Components

The Distinct RPC system essentially consists of four components. The client stub, server stub, data conversion module and the port mapper.

Client Stub

The client stub is called by the client program as a normal local procedure call with all the necessary parameters. The client stub accepts all the parameters (dereferencing pointers if necessary) , encapsulates them into a standard format, sends them to the server and waits for the result. Once the result value or structure has been returned by the server, the client stub returns the result of the remote call to the client program.

Server Stub

The server stub waits for client requests on the server. Every time a client request is received, the parameters are decoded into the local data format, and the appropriate local procedure is executed. The server stub then returns the either an error code or the result in a standard format.

An error can occur both in the server procedure and in the RPC protocol. An RPC protocol error could be caused by invalid authentication, failure to decode client parameters or some network error. A server procedure error could be caused by invalid or incorrect client parameters. In case of a procedure error, the RPC call will still return success.

Data Representation

When client and server need to exchange data, the XDR standard is used to represent data on the network. XDR is a standard for encoding and decoding data which assumes that one byte is portable between different hardware architectures. Any program running on any machine can send portable data by translating its local data representation to the XDR standard representation and another program running on another type of machine can read portable data by translating the XDR standard representations to its local data representation. A detailed discussion of XDR is beyond the scope of this manual. Please refer to 'RFC1832 - XDR: External Data Representation standard' for more details. Section '1.8 - External Data Representation' contains an overview of the XDR primitives included in the Distinct ONC RPC/XDR toolkit.

RPCBIND

Distinct RPCBIND supports version 3 and 4 as well as the older port mapper features. RPCBIND is an RPC server which allows clients to locate the port on which a particular server is running. Whenever a server is ready to accept incoming calls, it registers the port on which it can receive client requests with the port mapper. The client must then contact the port mapper on the remote machine to retrieve the port number of the desired server before it can make the first remote procedure call. RPCBIND accepts client requests on the well-known reserved port 111. RPCBIND must be started before any service is started. The Distinct library will start it automatically but you may also install this as a service on NT, 2000 or XP servers.

Special Considerations

Although a procedure call is executed on a remote machine and is generally transparent to the application, the following important issues regarding remote procedure calls need to be considered.

Server Location

Unlike many well-known TCP services (such as ftp or telnet), RPC servers do not operate at a reserved port. Therefore the client stub must be able to locate the port number on which the server is ready to accept client requests. To accomplish this, the client stub must communicate with the remote RPCBIND or port mapper.

Parameter Passing

Normally parameters can be passed either by reference (as pointers) or by value (by putting a copy of the parameter on the stack). Since client and server are running on different machines, RPC calls can pass parameters only by value.

Data Representation

RPC client and server may have different hardware architectures and therefore may have different data representations (i.e., the method used to store data). For example, a big endian system stores a four byte unsigned long in a completely different format from a little endian system. For this reason a consistent format for data representation has to be chosen. Data (RPC call parameters) is then transmitted in a machine independent format (the external data representation or XDR) over the network and converted to the local data format on both the server and the client.

Security

Allowing remote clients to execute procedure calls on a server carries with it a certain risk. Without security, potentially damaging operations could be performed by any client on the network. To provide a means for checking the identity and permissions of a client, each RPC call includes an authentication structure. The authentication structure is opaque (i.e., left for the application to provide) but a few standard authentication flavors are also available. This allows an application to implement its own security checks or to use a standard mechanism. Distinct ONC RPC/XDR 4.0 supports authentication using DES AUTH.

Timing

Unlike local procedure calls, remote procedure calls may encounter problems with the network or the server. Because communication links or servers sometimes go down, remote procedure calls must use both a resend period (over the connectionless UDP transport protocol) and a timeout period. If the response is not received within the timeout period, then an error is returned to the client.

Authentication Issues

Allowing clients to execute remote procedure calls on a server often requires additional security. For this reason, each client request contains authentication information. This information can be validated by the server before allowing the client to execute the remote procedure. The authentication parameters are currently opaque (i.e., handled by the application) but the following standard flavors are supported.

NULL Security Flavor (AUTH_NONE)

No security or authentication information is passed to the server. This authentication flavor is used when either the client cannot supply the information or the server does not care about the information. Using AUTH_NONE allows any machine on the network to issue remote procedure calls.

UNIX Security Flavor (AUTH_UNIX)

This flavor passes UNIX type authentication to the server. This authentication flavor sends the user id, the users primary group id along with auxiliary group ids to the server. It also contains the time of day and the name of the host from which the server is being called.

DES Security Flavor (AUTH_DES)

DES authentication offers more security features than UNIX authentication. The users of DES authentication need Diffie-Hellman public keys assigned in database. Additionally, each user's secret key (private key) must be decrypted using password known only to that user only. The security of DES authentication is based on a sender's ability to encrypt the current time, which the receiver can then decrypt and check against its own clock. The timestamp is encrypted with DES.

It is up to the server to choose how to authenticate a client. The RPC protocol allows for empty, default and custom authentication but the protocol itself does not directly deal with authentication issues.

External Data Representation

The following XDR routines are used to create and modify an XDR stream in memory. Once a stream has been created with `xdrmem_create`, the XDR primitives listed below can be used to either **encode or decode data in a memory buffer**. This method is useful for a developer who would like to use XDR to translate data into a machine independent format without actually making any remote procedure calls. The encoded data could then, for example, be transmitted over a socket. After encoding or decoding the data, the stream must be freed using `xdr_destroy`. The data in the buffer is not affected by this call, because the buffer was provided by the caller in the `xdrmem_create` call.

```
XDR * WINAPI xdrmem_create (buf, size, op)
char *buf;
unsigned int size;
unsigned int op;
```

The `xdrmem_create` function is used to create an XDR stream in memory. The `op` parameter must be set to either `XDR_ENCODE`, `XDR_DECODE` or `XDR_FREE`. The size of the data buffer pointed to by `buf` is given in `size`. If `size` does not specify a multiple of four then the next smaller number which is a multiple of four is used. The function returns a pointer to the XDR stream structure.

```
XDR * WINAPI xdrstdio_create (file, op)
FILE *file;
unsigned int op;
```

The `xdrstdio_create` function is used to create an XDR stream in standard I/O stream. The `op` parameter must be set to either `XDR_ENCODE`, `XDR_DECODE` or `XDR_FREE`. The function returns a pointer to the XDR stream structure.

```
XDR * WINAPI xdrrec_create (sendsz, recvsz, tcphandle, readit, writeit)
unsigned int sendsz;
unsigned int recvsz;
char *tcphandle;
int (*readit)(void *readhandle, char *buf, int len);
int (*writeit)(void *writehandle, char *buf, int len);
```

The `xdrrec_create` function is used to create a record-oriented XDR stream. The stream's data are written to a buffer of size `sendsz`; a value of 0 indicates the system should use a suitable default. The stream's data are read from a buffer of size `recvsz`; it too can be set to a suitable default by passing a 0 value. When a stream's output buffer is full, `writeit` is called. Similarly, when a stream's input buffer is empty, `readit` is called. Note: XDR stream's `op` field must be set by the caller.

```
void WINAPI xdr_destroy (xdr)
XDR *xdr;
```

The `xdr_destroy` function frees the XDR stream structure pointed to by `xdr`. The buffer specified in the `xdrmem_create` call is not affected by this call.

```
void WINAPI xdr_free (proc, buf)
xdr_proc proc;
char *buf;
```

The `xdr_free` function frees the primitive or structure pointed to by `buf` by calling the XDR function `proc` for an XDR free operation. The `xdr_free` function should be used to free memory that was allocated by an XDR routine on behalf of the application.

The following XDR routines can be used for **direct access to an XDR stream** created with `xdrmem_create`, `xdrstdio_create`, and `xdrrec_create`. The routines are used by the XDR primitives internally to move data to and from the XDR structure. These routines should only be used if the desired effect cannot be accomplished with the XDR primitives listed below. The routines also allow a developer to completely replace the XDR primitives.

```
BOOL WINAPI xdr_putlong (xdr, val)
XDR *xdr;
unsigned long val;
```

The `xdr_putlong` function appends a four byte long integer to the XDR stream. The long integer is converted from host to network byte order. The current position in the XDR stream is updated. This function can only be called during an encode operation.

```
BOOL WINAPI xdr_getlong (xdr, val)
XDR *xdr;
unsigned long *val;
```

The `xdr_getlong` function retrieves a four byte long integer from the XDR stream. The long integer is converted from network to host byte order. The current position in the XDR stream is updated. This function can only be called during a decode operation.

```
BOOL WINAPI xdr_putbytes (xdr, buf, cnt)
XDR *xdr;
char *buf;
unsigned int cnt;
```

The `xdr_putbytes` function appends `cnt` bytes (which must be a multiple of four) to the XDR stream. No byte order conversion is performed. The current position in the XDR stream is updated. This function can only be called during an encode operation.

```
BOOL WINAPI xdr_getbytes (xdr, buf, cnt)
XDR *xdr;
char *buf;
unsigned int cnt;
```

The `xdr_getbytes` function retrieves `cnt` bytes (which must be a multiple of four) from the XDR stream. No byte order conversion is performed. The current position in the XDR stream is updated. This function can only be called during a decode operation.

```
BOOL WINAPI xdr_setpos (xdr, pos)
XDR *xdr;
unsigned int pos;
```

The `xdr_setpos` function sets the current position in the XDR stream. The `pos` parameter (which must be a multiple of four) specifies the offset in bytes from the beginning of the buffer. This function can be called during encode and decode operations.

```
BOOL WINAPI xdr_getpos (xdr, pos)
XDR *xdr;
unsigned int *pos;
```

The `xdr_getpos` function retrieves the current position in the XDR stream. The integer pointed to by the `pos` parameter will contain the current offset in bytes from the beginning of the buffer. This function can be called during encode and decode operations.

The following XDR **primitives** can either be used directly to encode arguments or decode results of RPC calls or can be used to write custom XDR routines which encode or decode structures.

```
BOOL WINAPI xdr_void (xdr, p)
XDR *xdr;
void *p;
```

The `xdr_void` function is used if no argument is needed. The pointer `p` should be set to NULL.

```
BOOL WINAPI xdr_enum (xdr, p)
XDR *xdr;
enum *p;
```

The `xdr_enum` function encodes or decodes a four-byte enumeration.

```
BOOL WINAPI xdr_bool (xdr, p)
XDR *xdr;
BOOL *p;
```

The `xdr_bool` function encodes or decodes a four byte Boolean.

```

BOOL WINAPI xdr_char (xdr, p)
XDR *xdr;
char *p;

```

The **xdr_char** function encodes or decodes a signed one byte character.

```

BOOL WINAPI xdr_u_char (xdr, p)
XDR *xdr;
unsigned char *p;

```

The **xdr_u_char** function encodes or decodes an unsigned one byte character.

```

BOOL WINAPI xdr_short (xdr, p)
XDR *xdr;
short *p;

```

The **xdr_short** function encodes or decodes a signed two byte integer.

```

BOOL WINAPI xdr_u_short (xdr, p)
XDR *xdr;
unsigned short *p;

```

The **xdr_u_short** function encodes or decodes an unsigned two byte integer.

```

BOOL WINAPI xdr_int (xdr, p)
XDR *xdr;
int *p;

```

The **xdr_int** function encodes or decodes a signed four byte integer.

```

BOOL WINAPI xdr_u_int (xdr, p)
XDR *xdr;
unsigned int *p;

```

The **xdr_u_int** function encodes or decodes an unsigned four byte integer.

```

BOOL WINAPI xdr_long (xdr, p)
XDR *xdr;
long *p;

```

The **xdr_long** function encodes or decodes a signed four byte long integer.

```

BOOL WINAPI xdr_u_long (xdr, p)
XDR *xdr;
unsigned long *p;

```

The **xdr_u_long** function encodes or decodes an unsigned four byte long integer.

```

BOOL WINAPI xdr_hyper (xdr, p)
BOOL WINAPI xdr_longlong_t (xdr, p)
XDR *xdr;
LONGLONG *p;

```

The **xdr_hyper** and **xdr_longlong_t** function encodes or decodes a signed eight byte long integer.

```

BOOL WINAPI xdr_u_hyper (xdr, p)
BOOL WINAPI xdr_u_longlong_t (xdr, p)
XDR *xdr;
ULONGLONG *p;

```

The **xdr_u_hyper** and **xdr_u_longlong_t** function encodes or decodes an unsigned eight byte long integer.

```

BOOL WINAPI xdr_float (xdr, p)
XDR *xdr;
float *p;

```

The **xdr_float** function encodes or decodes a four byte IEEE float.

```
BOOL WINAPI xdr_double (xdr, p)
XDR *xdr;
double *p;
```

The `xdr_double` function encodes or decodes an eight byte IEEE double.

```
BOOL WINAPI xdr_quadruple (xdr, p)
XDR *xdr;
long double *p;
```

The `xdr_quadruple` function encodes or decodes a sixteen byte IEEE quadruple.

```
BOOL WINAPI xdr_array (xdr, p, cnt, max, size, proc)
XDR *xdr;
char **p;
unsigned int *cnt;
unsigned int max;
unsigned int size;
xdr_proc proc;
```

The `xdr_array` function encodes or decodes the array of primitives or structures pointed to by `p`. The `cnt` parameter specifies the number of elements to encode or returns the number of elements decoded (not to exceed `max` elements). The size of each element is given by `size` and the XDR procedure pointed to by `proc` is called to encode or decode each element.

```
BOOL WINAPI xdr_vector (xdr, p, cnt, size, proc)
XDR *xdr;
char *p;
unsigned int cnt;
unsigned int size;
xdr_proc proc;
```

The `xdr_vector` function encodes or decodes the array of primitives or structures pointed to by `p`. The `cnt` parameter specifies the number of elements to encode or decode. The size of each element is given by `size` and the XDR procedure pointed to by `proc` is called to encode or decode each element.

```
BOOL WINAPI xdr_bytes (xdr, p, cnt, max)
XDR *xdr;
char *p;
unsigned int *cnt;
unsigned int max;
```

The `xdr_bytes` function encodes or decodes the array of bytes pointed to by `p`. The `cnt` parameter specifies the number of bytes to encode or returns the number of bytes decoded (not to exceed `max` bytes)

```
BOOL WINAPI xdr_opaque (xdr, p, len)
XDR *xdr;
void *p;
unsigned int len;
```

The `xdr_opaque` function directly copies `len` bytes without encoding or decoding.

```
BOOL WINAPI xdr_string (xdr, p, max)
XDR *xdr;
void *p;
unsigned int max;
```

The `xdr_string` function encodes or decodes a string of up to `max` characters.

```

BOOL WINAPI xdr_union (xdr, type, p, choice, default)
XDR *xdr;
int *type;
char *p;
xdr_discrim *choice;
xdr_proc default;

```

The **xdr_union** function encodes or decodes a discriminated union pointed to by *p*. The type given by *type* specifies which of the entries in the array *choice* to use.

```

BOOL WINAPI xdr_pointer (xdr, p, size, proc)
XDR *xdr;
char **p;
unsigned int size;
xdr_proc proc;

```

The **xdr_pointer** function encodes or decodes primitives or structures of the size specified by *size* by calling the XDR function *proc* if the flag preceding the primitive or structure is non-zero. During an XDR decode operation it will allocate memory for the object to be decoded if *p* points to a NULL pointer. Also, this function will free the global memory pointed to by *p* during an XDR free operation.

```

BOOL WINAPI xdr_reference (xdr, p, size, proc)
XDR *xdr;
char **p;
unsigned int size;
xdr_proc proc;

```

The **xdr_reference** function encodes or decodes the primitive or structure pointed to by the pointer at address *p* by calling the XDR function *proc*.

```

BOOL WINAPI xdr_wrapstring (xdr, p)
XDR *xdr;
void *p;

```

The **xdr_wrapstring** function encodes or decodes a string of unknown length up to 65535 characters.

The following **XDR routines** for RPCBind version 3 and version 4 data types are also included in the Distinct ONC RPC/XDR library. They are used internally by the port mapper and by the RPC functions. Refer to `d32-rpc.h` or RFC 1833 for their structure definitions. In general, use of these functions should not be necessary either.

```

BOOL WINAPI xdr_rpcb (xdr, p)
XDR *xdr;
rpcb *p;

BOOL WINAPI xdr_rp_list (xdr, p)
XDR *xdr;
rp_list *p;

BOOL WINAPI xdr_rpcblist_ptr (xdr, p)
XDR *xdr;
rpcblist_ptr *p;

BOOL WINAPI xdr_rpcb_rmtcallargs (xdr, p)
XDR *xdr;
rpcb_rmtcallargs *p;

BOOL WINAPI xdr_rpcb_rmtcallres (xdr, p)
XDR *xdr;
rpcb_rmtcallres *p;

```

```
BOOL WINAPI xdr_rpcb_entry (xdr, p)
XDR *xdr;
rpcb_entry *p;

BOOL WINAPI xdr_rpcb_entry_list (xdr, p)
XDR *xdr;
rpcb_entry_list *p;

BOOL WINAPI xdr_rpcb_entry_list_ptr (xdr, p)
XDR *xdr;
rpcb_entry_list_ptr *p;

BOOL WINAPI xdr_rpcbs_addrlist (xdr, p)
XDR *xdr;
rpcbs_addrlist *p;

BOOL WINAPI xdr_rpcbs_addrlist_ptr (xdr, p)
XDR *xdr;
rpcbs_addrlist_ptr *p;

BOOL WINAPI xdr_rpcbs_rmtcalllist (xdr, p)
XDR *xdr;
rpcbs_rmtcalllist *p;

BOOL WINAPI xdr_rpcbs_rmtcalllist_ptr (xdr, p)
XDR *xdr;
rpcbs_rmtcalllist_ptr *p;

BOOL WINAPI xdr_rpcbs_proc (xdr, p)
XDR *xdr;
rpcbs_proc p;

BOOL WINAPI xdr_rpcb_stat (xdr, p)
XDR *xdr;
rpcb_stat *p;

BOOL WINAPI xdr_rpcb_stat_byvers (xdr, p)
XDR *xdr;
rpcb_stat_byvers p;

BOOL WINAPI xdr_netbuf (xdr, p)
XDR *xdr;
netbuf *p;
```

The following additional **XDR routines** are included in the Distinct ONC RPC/XDR library. They are used internally by the port mapper and by the RPC client and server functions. In general, use of these functions should not be necessary.

```
BOOL WINAPI xdr_opaque_auth (xdr, p)
XDR *xdr;
struct opaque_auth *p;
```

The `xdr_opaque_auth` function encodes or decodes an RPC authentication structure.

```
BOOL WINAPI xdr_authunix_parms (xdr, p)
XDR *xdr;
struct authunix_parms *p;
```

The `xdr_authunix_parms` function encodes or decodes a UNIX style authentication structure.

```
BOOL WINAPI xdr_authdes_cred (xdr, p)
XDR *xdr;
struct authdes_cred *p;
```

The `xdr_authdes_cred` function encodes or decodes a DES style authentication credential structure.

```
BOOL WINAPI xdr_authdes_verf (xdr, p)
XDR *xdr;
struct authdes_verf *p;
```

The `xdr_authdes_verf` function encodes or decodes a DES style authentication verifier structure.

```
BOOL WINAPI xdr_pmap (xdr, p)
XDR *xdr;
pmap *p;
```

The `xdr_pmap` function encodes or decodes a structure containing a port mapper service.

```
BOOL WINAPI xdr_pmaplist (xdr, p)
XDR *xdr;
GLOBALHANDLE *p;
```

The `xdr_pmaplist` function encodes or decodes a list of structures containing port mapper services.

RPCBIND and RPC Information

RPCBIND - Port Mapper

The RPCBIND program supports version 2, 3 and 4 of the RPCBIND protocol. It is started automatically when the first RPC service is registered but can also be started manually or as an NT/2000.XP service. RPCBIND maps RPC program and version numbers to universal addresses. In so doing it makes dynamic binding of remote programs possible. RPCBIND listens on port 111 for both UDP and TCP requests. When an RPC service is started, it tells rpcbind the address at which it is listening, and the RPC program numbers it is prepared to serve. When a client wishes to make an RPC call to a given program number, it first contacts RPCBIND on the server machine to determine the address where RPC requests should be sent. So for example the RPC service number 100002 for the rusers service is translated to the actual TCP or UDP port associated with the service. This allows the definition of standard service id numbers which can reside on almost any port on a server.

When RPCBIND is started automatically by an RPC service, it will shut down automatically when no more RPC services are registered with it. The waiting time value is determined by the registry DWORD value:

```
\\HKEY_LOCAL_MACHINE\\Software\\Distinct\\DLLS\\RPC32\\RPCBind\\ShutDown
```

This value is ignored if RPCBIND was started manually or as an NT/2000/XP service.

Note that RPCBIND is backward compatible and supports the older Portmapper calls.

Installing RPCBIND as a Service

RPCBIND may be installed as a service on Windows NT, 2000 or XP servers. Please note that this can only be done with an actual licensed copy of the RPC Server run time or a licensed copy of the toolkit and cannot be done with a trial copy.

To install RPCBIND as a service, first make sure that you must have administrator privileges, then open a command prompt window and change the directory to the one where the rpcbind.exe and pmapsvc.exe files are installed, by default this will be C:\Program Files\Common Files\DistinctShared. Next type:

```
Pmapsvc -install
```

to install the service. To uninstall the service type

```
Pmapsvc -remove
```

RPCINFO - RPC Service Information

RPCINFO.EXE makes an RPC call to an RPC server and lists all the RPC services that are registered with the RPCBIND daemon on that host. The Distinct RPCINFO application supports all versions (2, 3, 4) of the rpcbind protocol.

The format of the information depends on the version of RPCBIND running on the RPC server. If the portmapper is the version 2 of the rpcbind protocol, the information will contain "program", "version", "protocol", "port" and "service". Otherwise, the version will be 3 or 4, and the information will contain "program", "version", "netid", "address", "service" and "owner".

- program - the program number of the RPC service
- version - the version number of the RPC service
- protocol - the transport protocol of the RPC service, version 2 only
- netid - the network transport protocol id of the RPC service, version 3 & 4
- port - the port number of the RPC service, version 2 only
- address - the universal address of the RPC service, version 3 & 4
- service - the name of the RPC service
- owner - the user id of the RPC service, version 3 & 4

This RPCInfo application provides more features which are listed in the "Action" menu and the pop up menu when the right mouse button is clicked.

- "Show Details" - When the version of RPCBIN on the server is 4, RPCINFO can display detailed information about the listed service.
- "Ping It" - RPCINFO can send an RPC call to procedure 0 of the selected RPC service to test whether the service is still available or not.
- "Unregister It" - When RPCINFO is running on the same host with RPC server, RPCINFO can request to unregister the selected service from portmapper.
- "RPCBind Details" - When the version of RPCBIN on the server is 4, RPCINFO can display detailed information about RPCBIND returned by an RPC call to procedure RPCBPROC_GETSTAT (12).

The RPC Protocol Compiler: RPCGEN

The RPCGEN protocol compiler is used to generate C code to implement an RPC protocol. The input to RPCGEN is a language similar to C known as the RPC Language (Remote Procedure Call Language). The syntax of the RPC Language is described in RFC 1831. RPCGEN is normally invoked with an input file to generate up to four output files. If, for example, the input file is named *q.x*, then RPCGEN generates by default the header file *q.h*, the XDR source file *q_xdr.c*, the server side stubs in a file called *q_svc.c*, and the client side stubs in *q_clnt.c*. Option settings allow the generation of only some of the above four files.

A small amount of preprocessing is done by RPCGEN. Any line beginning with '%' is passed directly into the output file, uninterpreted by RPCGEN. For every data type referred to in the input file, RPCGEN assumes that there exists a routine with the string 'xdr_' prepended to the data type. If this XDR primitive routine does not exist in the Distinct ONC RPC/XDR library, it must be provided by the developer. Providing an undefined data type allows customization of XDR routines.

Running RPCGEN

RPCGEN can be run from the MS-DOS command prompt by typing 'RPCGEN' followed by the appropriate parameters at the prompt. A brief description about RPCGEN usage will be given if no parameters are given or wrong parameters are used.

The basic syntax for the RPCGEN command when run from a DOS box in Windows is

```
RPCGEN infile
```

where *infile* is the input file containing the RPC protocol definition as described in the RPC language definition. RPCGEN generates four files which contain the C code required to implement the RPC protocol in a Windows application or DLL. For example, the command

```
RPCGEN TIME.X
```

will generate the files

TIME.H	Header file containing all declarations.
TIME_XDR.C	C code to implement the XDR routines for data structures defined in TIME.X.
TIME_SVC.C	C code to implement the server defined in TIME.X.
TIME_CLNT.C	C code for client portion of RPC protocol.

However the generic usage is

```
RPCGEN [-f] [-P | -B | -p command] [-D (name=value)] [-M] infile
RPCGEN [-c | -h | -l | -m] [-o outfile] [infile]
RPCGEN [-s udp|tcp]* [-o outfile] [infile]
```

where the flags have the following meaning:

-c	Generate only XDR primitives.
-h	Generate only the header file output.
-l	Generate only the client stub.
-m	Generate only the server stub.
-f	Generate the output files with 8.3 DOS format. If this parameter is not specified, it defaults to the long file naming convention.
-P	Disable C preprocessor command, it defaults to be enabled.
-B	Use Borland C/C++ or Borland C++ Builder preprocessor command (cpp32 -C -P- -oCON) and developing environment.
-p command	Specify C preprocessor command to be used. The default C preprocessor command is `cl /C /EP /nologo`.
-D (name=value)	Define macros for the C preprocessor, this only works when the C preprocessor is enabled.
-M	Generate the multi-thread safe code.
-s udp tcp	Generate codes for the specified network protocol.
-o outfile	Save the generated code to the specified output file. In the second and third RPCGEN command syntax, standard output will be used if no <i>outfile</i> is specified.
infile	The input RPC language file name, standard input will be the default for the second and third RPCGEN command syntax.

For example, the command

```
RPCGEN -l -o CLIENT.C TIME.X
```

will generate only the client stub and store it in CLIENT.C.

```
RPCGEN -s udp -o SERVER.C TIME.X
```

will create the file SERVER.C containing only the code for a UDP based server.

Multi-Arguments RPCL Support

This version of RPCGEN supports multi-arguments RPCL. So you can write the XDR file without constructing the input argument structure (*demo.x*): Using multiple arguments allows you to write more efficient code and reduce your network traffic.

```
program DEMO_PROG {
    version DEMO_VERS {
        int ADD(int, int) = 1;
    } = 1;
} = 0x20000023;
```

Then after running “RPCGEN *demo.x*”, the *add_1* prototype in *demo.h* will be:

```
int * add_1(int *, int *, void *);
```

Samples

The toolkit contains several sample applications for Microsoft Visual C++, Borland C Builder and Borland Delphi. These samples demonstrate how to use the RPC library to create Client/Server applications.

The following are the sample applications built with Microsoft Visual C++. Note that the auth.sol, auth.win, batch, broadcast, callback, multiarg, nomap, rac, rac_mt and rpcinfos are command line samples that need to be run from a Command prompt box.

TIME

This RPC sample demonstrates how to write a single threaded RPC Client/Server application using the WIN32 API. It does not use MFC.

RUSERS

This RPC sample demonstrates how to write a single threaded RPC Client application using the WIN32 API. It does not use MFC. The client talks to a UNIX host and gets the list of users logged in.

DIRLIST

This RPC sample demonstrates how to write a multi threaded RPC Client/Server application using MFC.

AUTH.SOL

This sample demonstrates how to do authentication with Solaris systems.

AUTH.WIN

This sample illustrates the use of the authentication feature on Windows machines.

BATCH

Batching is a new feature added to the latest release, where a client can send a request to the server, and does not expect the result immediately instead keeps sending the request and asks for the results when required. This sample demonstrates how to use it. It sends a few numbers in sequence to the server and asks for the sum when all the numbers have been sent.

BROADCAST

The ONC RPC system includes a high-level client-side call that allows you to probe the network for servers matching a certain description. This sample attempts to ask all reachable portmappers if they have a record of the service designated on the command line. The client uses the `clnt_broadcast()` to query the local network repeatedly. Each time a server replies, the reply procedure is called. Only UDP packets can be broadcast.

Broadcast packets are limited in size by the underlying data-link transfer unit. For Ethernet, this limit the caller's argument portion of the request packet to be around 1400 bytes.

CALLBACK

This sample illustrates the use of callbacks by using the `gettransient` routine. The client makes a remote procedure call to the server, passing it a transient program number. The client then waits around to receive a callback from the server at that program number. The server registers the `CALLBACKPROG` program so that it can receive the remote procedure call that will give it the callback program number. When receiving the client call with the callback program number, the server sends a callback remote procedure call, using the program number it received earlier.

RAC

This sample illustrates the use of an RPC asynchronous client call. Unlike `clnt_call()`, which returns only when requested server replies, `rac_send()` returns immediately after sending out the remote procedure call to the server. Later `rac_poll()` can be used to check whether the reply has arrived or not.

This sample calls `rac_xxxx()` functions in multi-thread mode for the same client object. The XDR procedure is the same as that of the sample for an asynchronous call.

RACMT

RACMT is an extension of the RAC sample with multithreaded calls.

NOMAP

This sample illustrates how an RPC client and server can be written to communicate without the use of RPCBIND or Portmapper.

RPCINFOS

This is a command line RPCINFO sample.

MULTARG

This sample illustrates how to use multiargument XDR.

RPC Error Return Values

The following gives a brief description of the possible RPC call return codes.

Value	Meaning
RPC_SUCCESS	Successful completion.
RPC_CANT_ENCODE_ARGS	XDR error encoding arguments.
RPC_CANT_DECODE_RES	XDR error decoding results.
RPC_CANT_SEND	Socket transmit error.
RPC_CANT_RECV	Socket receive error.
RPC_TIMED_OUT	No response from server.
RPC_VERS_MISMATCH	Server does not support RPC version.
RPC_AUTH_ERROR	Permission denied.
RPC_PROG_UNAVAIL	Program not registered on server.
RPC_PROG_VERS_MISMATCH	Incompatible program version on server.
RPC_PROC_UNAVAIL	Procedure not registered.
RPC_CANT_DECODE_ARGS	Server unable to decode arguments.
RPC_SYSTEM_ERROR	Local internal RPC error (e.g. no memory).
RPC_UNKNOWN_HOST	Specified host does not exist.
RPC_PMAP_FAILURE	Unable to obtain remote port from port mapper.
RPC_PROG_NOT_REGISTERED	Remote port for program not known.
RPC_UNABLE_TO_REGISTER	Unable to register service with port mapper
RPC_UNABLE_TO_REMOVE	Unable to remove service from port mapper
RPC_FAILED	General 'other' error.
RPC_CALL_INPROGRESS	Asynchronous RPC call is in progress
RPC_STALE_RACHANDLE	Rac handle is no longer valid

D32-RPC.H

```

/*****
        Distinct TCP/IP RPC/XDR-32 Library
        (C) Copyright 1988 - 2001 Distinct Corporation
*****/
#ifndef _RPC_H_INCLUDED_
#define _RPC_H_INCLUDED_
// Default maximum RPC send and receive buffer size
#define MAX_RPC_SEND    9216
#define MAX_RPC_RECV    9216
typedef PVOID          HASYNCRPC;
typedef unsigned char  u_char;
typedef unsigned short u_short;
typedef unsigned int   u_int;
typedef unsigned long  u_long;
typedef ULARGE_INTEGER u_LARGE_INTEGER;
typedef unsigned int   enum_t;
/*****
        XDR functions and structures
*****/
#ifndef _XDR_HEADER_DEF
#define _XDR_HEADER_DEF
#include <stdio.h>
// XDR operations
enum xdr_op {
    XDR_ENCODE = 0,
    XDR_DECODE = 1,
    XDR_FREE   = 2
};
#define XDR_TRUE    1
#define XDR_FALSE  0
#define MAX_NAME_LEN    255
#define BYTES_PER_XDR_UNIT (4)
#define RNDUP(x) \
    (((x)+BYTES_PER_XDR_UNIT-1)/BYTES_PER_XDR_UNIT)*BYTES_PER_XDR_UNIT
// XDR Stream
struct _XDR {
    enum xdr_op    x_op;          /* operation; fast additional param */
    struct xdr_ops {
        BOOL      (WINAPI *x_getlong)();
    };
};

```

```
    BOOL    (WINAPI *x_putlong)();
    BOOL    (WINAPI *x_getbytes)();
    BOOL    (WINAPI *x_putbytes)();
    BOOL    (WINAPI *x_getpostn)();
    BOOL    (WINAPI *x_setpostn)();
    long *  (WINAPI *x_inline)();
    void    (WINAPI *x_destroy)();
} *x_ops;
char *     x_public;    /* users' data */
char *     x_private;  /* pointer to private data */
char *     x_base;     /* private used for position info */
u_int     x_handy;    /* extra private word */
};
typedef struct _XDR XDR;
// XDR En/Decode function types
typedef BOOL (WINAPI *xdr_proc) (XDR *, VOID *);
#define NULL_XDR_PROC ((xdr_proc) 0)
// Support for unions
struct _xdr_discrim {
    int      value;
    xdr_proc proc;
};
typedef struct _xdr_discrim xdr_discrim;
typedef struct _xdr_discrim xdr32_discrim;
// XDR primitives
#define xdr_getlong(xdr, longp)      \
    (*(xdr)->x_ops->x_getlong)(xdr, longp)
#define xdr_putlong(xdr, longv)      \
    (*(xdr)->x_ops->x_putlong)(xdr, longv)
#define xdr_getbytes(xdr, buf, cnt)  \
    (*(xdr)->x_ops->x_getbytes)(xdr, buf, cnt)
#define xdr_putbytes(xdr, buf, cnt)  \
    (*(xdr)->x_ops->x_putbytes)(xdr, buf, cnt)
#define xdr_getpos(xdr, posp)        \
    (*(xdr)->x_ops->x_getpostn)(xdr, posp)
#define xdr_setpos(xdr, pos)         \
    (*(xdr)->x_ops->x_setpostn)(xdr, pos)
#define xdr_inline(xdr, cnt)         \
    (*(xdr)->x_ops->x_inline)(xdr, cnt)
#ifdef __cplusplus
```

```

extern "C" {
#ifdef
// version 4 backward compatibility functions
XDR *      WINAPI xdr_create (char *, u_int, enum xdr_op);
void       WINAPI xdr_destroy (XDR *);
BOOL       WINAPI xdr_put_long (XDR *, u_long);
BOOL       WINAPI xdr_get_long (XDR *, u_long *);
BOOL       WINAPI xdr_put_bytes (XDR *, char *, u_int);
BOOL       WINAPI xdr_get_bytes (XDR *, char *, u_int);
BOOL       WINAPI xdr_set_pos (XDR *, u_int);
BOOL       WINAPI xdr_get_pos (XDR *, u_int *);
// xdr_admin
XDR *      WINAPI xdrmem_create (char *, u_int, enum xdr_op);
XDR *      WINAPI xdrstdio_create (FILE *, enum xdr_op);
XDR *      WINAPI xdrrec_create (u_int, u_int, char *, int (*)(), int (*)());
BOOL       WINAPI xdrrec_skiprecord (XDR *);
BOOL       WINAPI xdrrec_eof (XDR *);
BOOL       WINAPI xdrrec_endofrecord (XDR *, BOOL);
int        WINAPI xdrrec_readbytes (XDR *, char *, u_int);
// xdr_simple
BOOL       WINAPI xdr_bool (XDR *, BOOL *);
BOOL       WINAPI xdr_char (XDR *, char *);
BOOL       WINAPI xdr_double (XDR *, double *);
BOOL       WINAPI xdr_enum (XDR *, int *);
BOOL       WINAPI xdr_float (XDR *, float *);
void       WINAPI xdr_free (xdr_proc, char *);
BOOL       WINAPI xdr_hyper (XDR *, LONGLONG *);
BOOL       WINAPI xdr_int (XDR *, int *);
BOOL       WINAPI xdr_LARGE_INTEGER (XDR *, LARGE_INTEGER *);
BOOL       WINAPI xdr_long (XDR *, long *);
BOOL       WINAPI xdr_longlong_t (XDR *, LONGLONG *);
BOOL       WINAPI xdr_quadruple (XDR *, long double *);
BOOL       WINAPI xdr_short (XDR *, short *);
BOOL       WINAPI xdr_u_char (XDR *, u_char *);
BOOL       WINAPI xdr_u_hyper (XDR *, ULONGLONG *);
BOOL       WINAPI xdr_u_int (XDR *, u_int *);
BOOL       WINAPI xdr_u_LARGE_INTEGER (XDR *, ULARGE_INTEGER *);
BOOL       WINAPI xdr_u_long (XDR *, u_long *);
BOOL       WINAPI xdr_u_longlong_t (XDR *, ULONGLONG *);
BOOL       WINAPI xdr_u_short (XDR *, u_short *);

```

```

BOOL          WINAPI  xdr_void (XDR *, void *);
// xdr_complex
BOOL          WINAPI  xdr_array (XDR *, char **, u_int *, u_int, u_int, xdr_proc);
BOOL          WINAPI  xdr_bytes (XDR *, char **, u_int *, u_int);
BOOL          WINAPI  xdr_opaque (XDR *, char *, u_int);
BOOL          WINAPI  xdr_pointer (XDR *, char **, u_int, xdr_proc);
BOOL          WINAPI  xdr_reference (XDR *, char **, u_int, xdr_proc);
BOOL          WINAPI  xdr_string (XDR *, char **, u_int);
BOOL          WINAPI  xdr_union (XDR *, int *, char *, xdr_discrim *, xdr_proc);
BOOL          WINAPI  xdr_vector (XDR *, char *, u_int, u_int, xdr_proc);
BOOL          WINAPI  xdr_wrapstring (XDR *, char **);
#ifdef __cplusplus
}
#endif
#endif // _XDR_HEADER_DEF
/*****
    Authentication functions and structures
*****/
#define MAX_AUTH_BYTES      400
#define MAX_GIDS            32
// Authentication type
#define AUTH_NONE           0
#define AUTH_UNIX           1
#define AUTH_SHORT          2
#define AUTH_DES             3
#define AUTH_OK              0
#define AUTH_BAD_CRED        1
#define AUTH_REJECTED_CRED  2
#define AUTH_BAD_VERF        3
#define AUTH_REJECTED_VERF   4
#define AUTH_TOO_WEAK        5
#define AUTH_INVALID_RESP    6
#define AUTH_FAILED          7
typedef u_char des_block[8];
struct opaque_auth {
    u_int      type;
    char *     base;
    u_int      len;
#define oa_flavor type
#define oa_base  base

```

```

#define oa_length len
};
typedef struct opaque_auth auth;
struct _AUTH {
    auth      ah_cred;
    auth      ah_verf;
    des_block ah_key;
    struct auth_ops {
        void (*ah_nextverf)(struct _AUTH *);
        BOOL (*ah_marshall)(struct _AUTH *, void *);
        BOOL (*ah_validate)(struct _AUTH *, auth *);
        BOOL (*ah_refresh) (struct _AUTH *);
        void (*ah_destry)  (struct _AUTH *);
    } *      ah_ops;
    void *    ah_private;
};
typedef struct _AUTH AUTH;
struct authunix_parms {
    u_long    time;
    char *    machine;
    long      uid;
    long      gid;
    u_int     len;
    long *    gids;
#define aup_time      time
#define aup_machname  machine
#define aup_uid       uid
#define aup_gid       gid
#define aup_len       len
#define aup_gids      gids
};
typedef struct authunix_parms auth_unix;
#define MAXNETNAMELEN 255
enum authdes_namekind {
    ADN_FULLNAME = 0,
    ADN_NICKNAME = 1
};
struct authdes_fullname {
    char *    name; // network name of client, up to MAXNETNAMELEN
    des_block key;  // PK_encrypted(conversation key)
};

```

```

    u_long      window; // time to live
};
struct authdes_cred {
    enum authdes_namekind      adc_namekind;
    struct authdes_fullname    adc_fullname;
    u_long                     adc_nickname;
};
struct authdes_verf {
    union {
        struct timeval         adv_ctime;      // clear time
        des_block              adv_xtime;     // crypt time
    }                          adv_time_u;
    u_long                     adv_int_u;
#define adv_timestamp         adv_time_u.adv_ctime
#define adv_xtimestamp        adv_time_u.adv_xtime // encrypted(current time)
#define adv_winverf           adv_int_u       // encrypted(windows + 1)
#define adv_timeverf         adv_time_u.adv_ctime
#define adv_xtimeverf        adv_time_u.adv_xtime // encrypted(client timestamp
                                                // + client window)
#define adv_nickname         adv_int_u       // nickname for client
};
#ifdef __cplusplus
extern "C" {
#endif
BOOL      WINAPI  xdr_des_block (XDR *, des_block);
BOOL      WINAPI  xdr_auth (XDR *, auth *);
BOOL      WINAPI  xdr_opaque_auth (XDR *, struct opaque_auth *);
BOOL      WINAPI  xdr_auth_unix (XDR *, auth_unix *);
BOOL      WINAPI  xdr_authunix_parms (XDR *, struct authunix_parms *);
AUTH *    WINAPI  authnone_create (void);
AUTH *    WINAPI  authunix_create (char *, ULONG, ULONG, ULONG, ULONG *);
AUTH *    WINAPI  authsys_create (char *, ULONG, ULONG, ULONG, ULONG *);
void      WINAPI  auth_destroy (AUTH *);
AUTH *    WINAPI  authdes_create (char *, char *, char *, u_int);
BOOL      WINAPI  xdr_authdes_cred (XDR *, struct authdes_cred *);
BOOL      WINAPI  xdr_authdes_verf (XDR *, struct authdes_verf *);
void      WINAPI  bin2hex(unsigned char *, char *, int);
void      WINAPI  hex2bin(char *, char *, int);
void      WINAPI  passwd2des(char *, des_block *);
BOOL      WINAPI  xencrypt(char *, char *, char *);

```

```

BOOL      WINAPI  xdecrypt(char *, char *, char *);
BOOL      WINAPI  getpublickey(char *, char *);
BOOL      WINAPI  getsecretkey(char *, char *, char *);
BOOL      WINAPI  getcommonkey(char *, char *, char *, des_block);
#ifdef __cplusplus
}
#endif
/*****
    RPC -> Client side functions and structures
*****/
// Default RPC socket
#define RPC_ANYSOCK          -1
// Client I/O Control Codes
#define CLSET_TIMEOUT        0x01      // Set Total Timeout
#define CLGET_TIMEOUT        0x02      // Get Total Timeout
#define CLGET_FD             0x03      // Get Associated file desc.
#define CLGET_SVC_ADDR       0x04      // Get Server's Address
#define CLSET_FD_CLOSE       0x05      // Close descriptor
#define CLSET_RETRY_TIMEOUT  0x07      // Resend Timeout for UDP
#define CLGET_RETRY_TIMEOUT  0x08      // Resend Timeout for UDP
// RPC call return codes
#define RPC_SUCCESS          0
// Local Errors
#define RPC_CANT_ENCODE_ARGS  1
#define RPC_CANT_DECODE_RES   2
#define RPC_CANT_SEND         3
#define RPC_CANT_RECV         4
#define RPC_TIMED_OUT         5
// Remote Errors
#define RPC_VERS_MISMATCH     6
#define RPC_AUTH_ERROR        7
#define RPC_PROG_UNAVAIL      8
#define RPC_PROG_VERS_MISMATCH 9
#define RPC_PROC_UNAVAIL     10
#define RPC_CANT_DECODE_ARGS  11
#define RPC_SYSTEM_ERROR      12
// Other Errors
#define RPC_UNKNOWN_HOST      13
#define RPC_PMAP_FAILURE      14
#define RPC_PROG_NOT_REGISTERED 15

```

```

// Unspecified Error
#define RPC_FAILED                16
// Portmapper Errors
#define RPC_UNABLE_TO_REGISTER    17
#define RPC_UNABLE_TO_REMOVE     18
#define RPC_UNKNOWN_PROTOCOL      19
// RAC Errors
#define RPC_CALL_INPROGRESS       20
#define RPC_STALE_RACHANDLE       21
// Error info.
typedef struct _rpc_err {
    ULONG    re_status;
    union {
        int    RE_errno;        // Related system error
        ULONG  RE_why;         // Why the auth error occurred
        struct {
            u_long  low;        // Lowest version supported
            u_long  high;       // Highest version supported
        } RE_vers;
        struct {                // Maybe meaningful if RPC_FAILED
            long    s1;
            long    s2;
        } RE_lb;                // Life boot & debugging only
    } ru;
#define re_errno    ru.RE_errno
#define re_why      ru.RE_why
#define re_vers     ru.RE_vers
#define re_lb       ru.RE_lb
} ClRpcError, *PclRpcError;
// Client Handle
typedef struct _HCLIENT {
    PVOID    Client;
    auth     *cl_auth;
} HCLIENT, *PHCLIENT;
typedef BOOL (*eachresult) (PVOID, SOCKADDR_IN *);
#ifdef __cplusplus
extern "C" {
#endif
// Blocking functions
PHCLIENT WINAPI clnttcp_create (SOCKADDR_IN *, ULONG, ULONG, INT *, UINT, UINT);

```

```

PHCLIENT WINAPI clntudp_create (SOCKADDR_IN *, ULONG, ULONG, TIMEVAL, INT *);
PHCLIENT WINAPI clntudp_bufcreate (SOCKADDR_IN *, ULONG, ULONG, TIMEVAL, INT *,
                                   UINT, UINT);

VOID WINAPI clnt_destroy (PHCLIENT);
int WINAPI clnt_call (PHCLIENT, ULONG, xdr_proc, PVOID, xdr_proc, PVOID,
                     TIMEVAL);

INT WINAPI callrpc (LPSTR, ULONG, ULONG, ULONG, xdr_proc, PVOID, xdr_proc,
                  PVOID);

INT WINAPI rpc_call (LPSTR, ULONG, ULONG, ULONG, xdr_proc, PVOID, xdr_proc,
                   PVOID, LPSTR);

void WINAPI clnt_seterror (PHCLIENT, UINT);
int WINAPI clnt_geterror (PHCLIENT);
VOID WINAPI clnt_getterr (GLOBALHANDLE, PClRpcError);
INT WINAPI clnt_broadcast (ULONG, ULONG, ULONG, xdr_proc, PVOID, xdr_proc,
                          PVOID, eachresult);

PHCLIENT WINAPI clnt_create (char *, ULONG, ULONG, char *);
INT WINAPI rpc_broadcast( ULONG, ULONG, ULONG, xdr_proc, PVOID, xdr_proc,
                        PVOID, eachresult);

void WINAPI clnt_perror (PHCLIENT, char *);
char * WINAPI clnt_sperror (PHCLIENT);
BOOL WINAPI clnt_control (PHCLIENT, UINT, PVOID);
BOOL WINAPI clnt_freeres (PHCLIENT, xdr_proc, PVOID);
// Asynchronous functions
HASYNCRPC WINAPI rac_send (PHCLIENT, ULONG, xdr_proc, PVOID, xdr_proc, PVOID,
                          TIMEVAL);

INT WINAPI rac_recv (PHCLIENT, HASYNCRPC);
INT WINAPI rac_poll (PHCLIENT, HASYNCRPC);
VOID WINAPI rac_drop (PHCLIENT, HASYNCRPC);
#ifdef __cplusplus
}
#endif
/*****
    RPC -> Server side functions and structures
*****/
#define NULLPROC    0        // Added for the benefit of Rpcgen32
// Server handle
typedef PVOID HSERVER;
typedef struct _svc_request {
    unsigned long    prog;
    unsigned long    vers;

```

```
    unsigned long   proc;
    auth            cred;
    char *          clntcred;
    GLOBALHANDLE    hSvc;
} svc_request;
// Dispatch routine
typedef VOID (*dispatch) (svc_request *, HSERVER);
typedef VOID * (*progrname) (VOID *);
typedef char *(*local_proc)(void *, void *);
typedef int (*local_proc_m)(void *, void *, void*);
#ifdef __cplusplus
extern "C" {
#endif
HSERVER    WINAPI svctcp_create (INT, SHORT, ULONG, ULONG);
HSERVER    WINAPI svcudp_create (INT, SHORT);
HSERVER    WINAPI svctcp_create_secure (INT, SHORT, ULONG, ULONG, PCHAR, PCHAR);
HSERVER    WINAPI svcudp_create_secure (INT, SHORT, PCHAR, PCHAR);
HSERVER    WINAPI svcudp_bufcreate (INT, SHORT, ULONG, ULONG);
BOOL       WINAPI svc_register (HSERVER, ULONG, ULONG, VOID (*dispatch)(),
                                ULONG);

BOOL       WINAPI svc_unregister (ULONG, ULONG);
VOID       WINAPI svc_destroy (HSERVER);
VOID       WINAPI svc_run (VOID);
VOID       WINAPI svc_run_ex (INT);
VOID       WINAPI svc_exit (VOID);
VOID       WINAPI svc_getreqset (FD_SET *);
VOID       WINAPI svc_fdset (FD_SET *);
INT        WINAPI svc_update_threads (INT);
VOID       WINAPI svc_set_timeout (TIMEVAL);
BOOL       WINAPI svc_getargs (HSERVER, xdr_proc, PVOID);
BOOL       WINAPI svc_freeargs (HSERVER, xdr_proc, PVOID);
SOCKADDR_IN WINAPI svc_getcaller (HSERVER);
BOOL       WINAPI svc_sendreply (HSERVER, xdr_proc, PVOID);
VOID       WINAPI svcerr_auth (HSERVER, UINT);
VOID       WINAPI svcerr_decode (HSERVER);
VOID       WINAPI svcerr_noprogram (HSERVER);
VOID       WINAPI svcerr_noproc (HSERVER);
VOID       WINAPI svcerr_progvers (HSERVER);
VOID       WINAPI svcerr_systemerr (HSERVER);
VOID       WINAPI svcerr_weakauth (HSERVER);
```

```

BOOL          WINAPI registrerrpc(ULONG , ULONG , ULONG , progname, xdr_proc,
                                   xdr_proc);

#ifdef __cplusplus
}
#endif

/*****
    PortMapper functions and structures
*****/

// Id of PortMapper
#define PMAP_PORT          111
#define PMAP_PROGRAM      ((unsigned long) 100000)
#define PMAP_VERSION      ((unsigned long) 2)
// PortMapper services
#define PMAP_NULL         ((unsigned long) 0)
#define PMAP_SET          ((unsigned long) 1)
#define PMAP_UNSET        ((unsigned long) 2)
#define PMAP_GETPORT      ((unsigned long) 3)
#define PMAP_DUMP         ((unsigned long) 4)
#define PMAP_CALLIT       ((unsigned long) 5)
typedef struct _pmap {
    unsigned long prog;
    unsigned long vers;
    unsigned long prot;
    unsigned long port;
} pmap;
typedef struct _pmaplist {
    pmap          map;
    GLOBALHANDLE next;
} pmaplist;
// Structure to pass the arguments to the CALLIT procedure in the port mapper
typedef struct _call_args {
    unsigned long prog;
    unsigned long vers;
    unsigned long proc;
    unsigned int  count;
    char          *buf;
} call_args;
// Structure for the results of the CALLIT procedure in the port mapper
typedef struct _call_results {
    unsigned long port;

```

```

        unsigned int    count;
        char            *buf;
    } call_results;
#ifdef __cplusplus
extern "C" {
#endif
BOOL            WINAPI xdr_pmap (XDR *, pmap * );
BOOL            WINAPI xdr_pmaplist (XDR *, GLOBALHANDLE *);
BOOL            WINAPI xdr_broadcast_args (XDR *, call_args *);
BOOL            WINAPI xdr_broadcast_results (XDR *, call_results *);
BOOL            WINAPI xdr_call_args ( XDR *, call_args *);
BOOL            WINAPI xdr_call_results ( XDR *, call_results *);
UINT            WINAPI pmap_getport (SOCKADDR_IN *, ULONG, ULONG, ULONG);
pmaplist *     WINAPI pmap_getmaps (SOCKADDR_IN *);
BOOL            WINAPI pmap_set (ULONG, ULONG, ULONG, ULONG);
BOOL            WINAPI pmap_unset (ULONG, ULONG);
INT             WINAPI pmap_rmtcall (SOCKADDR_IN *, ULONG, ULONG, ULONG, xdr_proc,
                                     PVOID, xdr_proc, PVOID, TIMEVAL, PULONG);

#ifdef __cplusplus
}
#endif
/*****
    Database definitions and functions
*****/
struct rpcent {
    char    *r_name;
    char    **r_aliases;
    int     r_number;
};
#ifdef __cplusplus
extern "C" {
#endif
extern struct rpcent *getrpcent ();
extern struct rpcent *getrpcbyname (char *name);
extern struct rpcent *getrpcbynumber (int number);
extern void endrpcent (void);
extern void setrpcent (int);
#ifdef __cplusplus
}
#endif

```

```
#endif      // _RPC_H_INCLUDED_  
/*****  
    End of D32-Rpc.h  
*****/
```

D32-RPC.PAS

The following header file is required when programming in Delphi.

```
{ Distinct ONC RPC/XDR for Windows }
{ (c) Copyright 1988 - 2002 Distinct Corporation }

unit D32RPC;

interface

uses idWinsock, WinTypes;

const
  { library name }
  Lib = 'D32-RPC.dll';
  LibSrv = 'DRPCSRVR.DLL';

  MAX_RPC_SEND = 9216;
  MAX_RPC_RECV = 9216;

  XDR_TRUE = 1;
  XDR_FALSE = 0;
  MAX_NAME_LEN = 255;

  { XDR operations }
  _XDR_ENCODE = 0;
  _XDR_DECODE = 1;
  _XDR_FREE = 2;

  { authentication limits }
  MAX_AUTH_BYTES = 400;
  MAX_GIDS = 10;

  { authentication types }
  _AUTH_NONE = 0;
  _AUTH_UNIX = 1;
  _AUTH_SHORT = 2;
  _AUTH_DES = 3;

  { authentication codes }
```

```
AUTH_OK = 0;
AUTH_BAD_CRED = 1;
AUTH_REJECTED_CRED = 2;
AUTH_BAD_VERF = 3;
AUTH_REJECTED_VERF = 4;
AUTH_TOO_WEAK = 5;
AUTH_INVALID_RESP = 6;
AUTH_FAILED = 7;

// fullname and nickname

ADN_FULLNAME = 0;
ADN_NICKNAME = 1;

{ RPC Server }
NULLPROC = 0;

{ port mapper port }
PMAP_PORT = 111;
_PMAP_PROGRAM = 100000;
_PMAP_VERSION = 2;

{ port mapper services }
_PMAP_NULL = 0;
_PMAP_SET = 1;
_PMAP_UNSET = 2;
_PMAP_GETPORT = 3;
_PMAP_DUMP = 4;
_PMAP_CALLIT = 5;

{ default RPC socket }
RPC_ANYSOCK = -1;

{ Client TO CONTROL CODES }
CLSET_TIMEOUT = $01; { Set Total Timeout }
CLGET_TIMEOUT = $02; { Get Total Timeout }
CLGET_FD = $03; { Get Associated file desc. }
CLGET_SVC_ADDR = $04; { Get Server's Address }
CLSET_FD_CLOSE = $05; { Close descriptor }
```

```
CLSET_RETRY_TIMEOUT = $06; { Resend Timeout for UDP }
CLGET_RETRY_TIMEOUT = $07; { Resend Timeout for UDP }

{ RPC call return codes }
RPC_SUCCESS = 0;

{ local errors }
RPC_CANT_ENCODE_ARGS = 1;
RPC_CANT_DECODE_RES = 2;
RPC_CANT_SEND = 3;
RPC_CANT_RECV = 4;
RPC_TIMED_OUT = 5;

{ remote errors }
RPC_VERS_MISMATCH = 6;
RPC_AUTH_ERROR = 7;
RPC_PROG_UNAVAIL = 8;
RPC_PROG_VERS_MISMATCH = 9;
RPC_PROC_UNAVAIL = 10;
RPC_CANT_DECODE_ARGS = 11;
RPC_SYSTEM_ERROR = 12;

{ other errors }
RPC_UNKNOWN_HOST = 13;
RPC_PMAP_FAILURE = 14;
RPC_PROG_NOT_REGISTERED = 15;

{ unspecified error }
RPC_FAILED = 16;

{ portmapper errors }
RPC_UNABLE_TO_REGISTER = 17;
RPC_UNABLE_TO_REMOVE = 18;
RPC_UNKNOWN_PROTOCOL = 19;

RPCBSTAT_HIGHPROC=13;
RPCBVERS_STAT=3;
RPCBVERS_4_STAT=2;
RPCBVERS_3_STAT=1;
```

```
RPCBVERS_2_STAT=0;
```

```
type
```

```
HASYNCRPC = Pointer;  
HSERVER = Pointer;
```

```
type
```

```
u_char = byte;  
u_short = Word;  
u_int = Longword;  
u_long = Longword;  
enum_t = Integer;
```

```
type
```

```
{ double reference pointer }  
PPtr = ^Pointer;
```

```
type
```

```
{ XDR function type }  
xdr_proc = TFarProc;
```

```
type
```

```
{ support for unions }  
xdr32_discrim = record  
    v : Integer;  
    proc : xdr_proc;  
end;
```

```
type
```

```
{ XDR stream }  
XDRPtr = ^XDR;  
XDR = record  
    op : u_int;  
    size : u_int;  
    base : Pointer;  
    curr : Pointer;  
end;
```

```
type des_block=Array[0..7] of u_char;
```

```
type
  { authentication }
  opaqueauthPtr = ^opaque_auth;
  opaque_auth = record
    kind : u_int;
    base : Pointer;
    len : u_int;
  end;
```

```
type
  { authentication }
  authPtr = ^auth;
  auth = record
    kind : u_int;
    base : Pointer;
    len : u_int;
  end;
```

```
type auth_ops=record
  ah_nextverf:pointer;
  ah_marshall:pointer;
  ah_validate:pointer;
  ah_refresh:pointer;
  ah_destry:pointer;
end;
```

```
type pauth_ops=^auth_ops;
```

```
type _AUTH=record
  ah_cred:auth;
  ah_verf:auth;
  ah_key:des_block;
  ah_ops:pauth_ops;
  ah_private:pointer;
end;
```

```
type _AUTHPtr=^_AUTH;
```

```
type
  { UNIX style authentication }
```

```

auth_unix = record
    time : u_long;
    machine : Array [0..(MAX_NAME_LEN - 1)] of shortint;
    uid : Longint;
    gid : Longint;
    len : u_long;
    gids : Array [0..(MAX_GIDS - 1)] of Longint;
end;

```

```

type authunix_parms=auth_unix;
type pauthunix_parms=^authunix_parms;
type pauth_unix=^auth_unix;

```

```

type authdes_fullname=record
    name:pointer;
    key:des_block;
    window:u_long;
end;

```

```

type authdes_cred=record
    adc_namekind:integer;
    adc_fullname:authdes_fullname;
    adc_nickname:u_long;
end;
type pauthdes_cred=^authdes_cred;

```

```

type authdes_verf=record
    adv_time:des_block;
    adv_int_u:u_long;
end;
type pauthdes_verf=^authdes_verf;

```

```

type
    { error info }
    ReVers = record
        low : u_long;
        high : u_long;
    end;

```

```
ReLb = record
    s1 : Longint;
    s2 : Longint;
end;

_ru = record
    case integer of
        0 : (RE_errno : integer);
        1 : (RE_why : u_long);
        2 : (RE_vers : ReVers);
        3 : (RE_lb : ReLb);
    end;
end;

PCLrpcError = ^CLrpcError;
CLrpcError = record
    re_status : Longint;
    ru : _ru;
end;

type
    { Client Handle }
    PHCLIENT = ^HCLIENT;
    HCLIENT = record
        Client : Pointer;
        cl_auth : authPtr;
    end;

type
    { port map entry }
    pmap = record
        prog : Longint;
        vers : Longint;
        prot : Longint;
        port : Longint;
    end;

type
    { port map list element }
    pmaplistPtr = ^pmaplist;
```

```
pmaplist = record
    map : pmap;
    next : THandle;
end;
```

```
type
```

```
{ port map callit procedure arguments structure }
call_args = record
    prog : u_long;
    vers : u_long;
    proc : u_long;
    count : u_int;
    buf : Pointer;
end;
```

```
type
```

```
{ port map callit procedure results structure }
call_results = record
    port : u_long;
    count : u_int;
    buf : Pointer;
end;
```

```
type
```

```
{ service request information }
svc_requestPtr = ^svc_request;
svc_request = record
    prog : u_long;
    vers : u_long;
    proc : u_long;
    verify : auth;
    buf : Pointer;
    hSvc : THandle;
end;
```

```
type INT=integer;
```

```
type _FILE=record
```

```
    _ptr:Pointer;
    _cnt:integer;
```

```
    _base:Pointer;
    _flag:integer;
    _file:integer;
    _charbuf:integer;
    _bufsiz:integer;
    _tmpfname:Pointer;
end;

//type BOOL=boolean;
type LARGE_INTEGER=int64;
type ULARGE_INTEGER=int64;
type LONGLONG=Int64;
type ULONGLONG=int64;
type long_double=double;

type rpcb=record
    r_prog:u_long;
    r_vers:u_long;
    r_netid :Pointer;
    r_addr:Pointer;
    r_owner:Pointer;
end;

type rpcbPtr=^rpcb;

type
    rpcblist_ptr=^rp__list;
    rp__list=record
        rpcb_map:rpcb;
        rpcb_next:rpcblist_ptr;
    end;
type prpcblist_ptr=^rpcblist_ptr;

type rpcb_rmtcallargs=record
    prog:u_long;
    vers:u_long;
    proc:u_long;
    args:record
        args_len:u_int;
```

```
        args_val:pointer;
    end;
end;
type prpcb_rmtcallargs=^rpcb_rmtcallargs;

type rpcb_rmtcallres=record
    addr:Pointer;
    results:record
        results_len:u_int ;
        results_val:Pointer;
    end;
end;
type prpcb_rmtcallres=^rpcb_rmtcallres;

type rpcb_entry=record
    r_maddr:pointer;
    r_nc_netid:pointer;
    r_nc_semantics:u_long;
    r_nc_protofmly:pointer;
    r_nc_proto:pointer;
end;
type prpcb_entry=^rpcb_entry;

type    rpcb_entry_list_ptr=^rpcb_entry_list;
        rpcb_entry_list=record
            rpcb_entry_map:rpcb_entry;
            rpcb_entry_next:rpcb_entry_list_ptr;
end;

type prpcb_entry_list_ptr=^rpcb_entry_list_ptr;

type
    rpcbs_addrlist_ptr=^rpcbs_addrlist;
    rpcbs_addrlist=record
        prog:u_long;
        vers:u_long;
        success:integer;
        failure:integer;
```

```
        netid:pointer;
        next:rpcbs_addrlist_ptr;
end;

type prpcbs_addrlist_ptr=^rpcbs_addrlist_ptr;

type
    rpcbs_rmtcalllist_ptr=^rpcbs_rmtcalllist;
    rpcbs_rmtcalllist=record
        prog:u_long;
        vers:u_long;
        proc:u_long;
        success:integer;
        failure:integer;
        indirect:integer;
        netid:pointer;
        next:rpcbs_rmtcalllist_ptr;
end;

type prpcbs_rmtcalllist_ptr=^rpcbs_rmtcalllist_ptr;

type rpcbs_proc=array[0..(RPCBSTAT_HIGHPROC-1)] of integer;

type rpcb_stat=record
    info:rpcbs_proc;
    setinfo:integer;
    unsetinfo:integer;
    addrinfo:rpcbs_addrlist_ptr;
    rmtinfo:rpcbs_rmtcalllist_ptr;
end;

type prpcb_stat=^rpcb_stat;

type rpcb_stat_byvers=Array[0..(RPCBVERS_STAT-1)] of rpcb_stat;

type
    netbuf=record
        maxlen:u_int;
        buf:record
            buf_len:u_int;
```

```
        buf_val:pointer;
    end;
end;

type pnetbuf=^netbuf;

type pdes_block=^des_block;

{ XDR primitives }
function xdr_void (x : XDRPtr; var v : Pointer) : Bool; stdcall;
function xdr_int (x : XDRPtr; var v : Integer) : Bool; stdcall;
function xdr_u_int (x : XDRPtr; var v : u_int) : Bool; stdcall;
function xdr_long (x : XDRPtr; var v : Longint) : Bool; stdcall;
function xdr_u_long (x : XDRPtr; var v : u_long) : Bool; stdcall;
function xdr_short (x : XDRPtr; var v : Smallint) : Bool; stdcall;
function xdr_u_short (x : XDRPtr; var v : Word) : Bool; stdcall;
function xdr_bool (x : XDRPtr; var v : Bool) : Bool; stdcall;
function xdr_enum (x : XDRPtr; var v : Integer) : Bool; stdcall;
function xdr_opaque (x : XDRPtr; buf : Pointer; len : u_int) : Bool; stdcall;
function xdr_union (x : XDRPtr; var kind : Integer; u : Pointer;
    var choice : xdr32_discrim; dflt : xdr_proc) : Bool; stdcall;
function xdr_char (x : XDRPtr; var c : shortint) : Bool; stdcall;
function xdr_u_char (x : XDRPtr; var c : byte) : Bool; stdcall;
function xdr_float (x : XDRPtr; var v : Single) : Bool; stdcall;
function xdr_double (x : XDRPtr; var v : Double) : Bool; stdcall;
function xdr_array (x : XDRPtr; var arr : Pointer; var cnt : u_int;
    max : u_int; el_size : u_int; el_proc : xdr_proc) : Bool; stdcall;
function xdr_bytes (x : XDRPtr; var buf : Pointer; var size : u_int;
    max : u_int) : Bool; stdcall;
function xdr_string (x : XDRPtr; var buf : Pointer; max : u_int) : Bool; stdcall;
function xdr_reference (x : XDRPtr; var buf : PPtr; size : u_int;
    proc : xdr_proc) : Bool; stdcall;
function xdr_pointer (x : XDRPtr; var buf : Pointer; size : u_int;
    proc : xdr_proc) : Bool; stdcall;
function xdr_wrapstring (x : XDRPtr; var buf : Pointer) : Bool; stdcall;
function xdr_vector (x : XDRPtr; buf : Pointer; cnt : u_int; size : u_int;
    proc : xdr_proc) : Bool; stdcall;
procedure xdr_free (proc : xdr_proc; buf : Pointer); stdcall;
```

```
function xdr_LARGE_INTEGER (x: XDRptr; var v:LARGE_INTEGER): Bool; stdcall;
function xdr_u_LARGE_INTEGER (x: XDRptr; var v:ULARGE_INTEGER): Bool; stdcall;
function xdr_u_hyper (x: XDRptr; var v:ULONGLONG): Bool; stdcall;
function xdr_u_longlong_t (x: XDRptr; var v:ULONGLONG): Bool; stdcall;
function xdr_longlong_t (x: XDRptr; var v:LONGLONG): Bool; stdcall;
function xdr_hyper (x: XDRptr; var v:LONGLONG): Bool; stdcall;
function xdr_quadruple (x: XDRptr; var v:long_double): Bool; stdcall;
```

```
{xdr_admin}
```

```
function xdrmem_create (buf: Pointer; size: u_int; op: u_int):XDRPtr;stdcall;
function xdrstdio_create (f: _FILE; op:u_int):XDRPtr;stdcall;
function xdrrec_create (sendsz:u_int;recvsz: u_int; tcphandle:
pointer;readit:pointer;writeit:pointer):XDRPtr;stdcall;
function xdrrec_skiprecord (xdr:XDRPtr):BOOL;stdcall;
function xdrrec_eof (xdr:XDRPtr):BOOL;stdcall;
function xdrrec_endofrecord (xdr:XDRPtr;b:BOOL):BOOL;stdcall;
function xdrrec_readbytes (xdr:XDRPtr; c:pointer; u:u_int):integer;stdcall;
```

```
{ XDR stream routines }
```

```
function xdr_create (buf : Pointer; size : u_int; op : u_int) : XDRPtr; stdcall;
function xdr_put_long (x : XDRPtr; v : u_long) : Bool; stdcall;
function xdr_get_long (x : XDRPtr; var v : u_long) : Bool; stdcall;
function xdr_put_bytes (x : XDRPtr; buf : Pointer; len : u_int) : Bool; stdcall;
function xdr_get_bytes (x : XDRPtr; buf : Pointer; len : u_int) : Bool; stdcall;
function xdr_set_pos (x : XDRPtr; pos : u_int) : Bool; stdcall;
function xdr_get_pos (x : XDRPtr; var pos : u_int) : Bool; stdcall;
procedure xdr_destroy (x : XDRPtr); stdcall;
```

```
{ XDR routines for authentication }
```

```
function xdr_auth (x : XDRPtr; var v : auth) : Bool; stdcall;
function xdr_auth_unix (x : XDRPtr; var v : auth_unix) : Bool; stdcall;
```

```
function authnone_create : _AUTHPtr; stdcall;
function authunix_create (var buf : shortint; x : u_long; y : u_long;
    z : u_long; var v : u_long ) : _AUTHPtr; stdcall;
function authsys_create (var buf : shortint; x : u_long; y : u_long;
    z : u_long; var v : u_long ) : _AUTHPtr; stdcall;
function authdes_create (clientnetname:pointer;password:pointer;
```

```

    servernetname:pointer;window:u_int):_AUTHPtr;stdcall;

function xdr_opaque_auth (xdr:XDRPtr;a:authPtr):BOOL;stdcall;
function xdr_authunix_parms (xdr:XDRPtr;a: pauthunix_parms):BOOL;stdcall;
function xdr_authdes_cred (xdr:XDRPtr;ac:pauthdes_cred):BOOL;stdcall;
function xdr_authdes_verf (xdr:XDRPtr;av:pauthdes_verf):BOOL;stdcall;

procedure auth_destroy (var v : auth); stdcall;

function xdr_des_block (xdr:XDRPtr;db: des_block):BOOL;stdcall;
procedure bin2hex(bindata: pointer; hexstar:pointer; len:integer);stdcall;
procedure hex2bin(hexstar: pointer; bindata:pointer; len:integer);stdcall;
procedure passwd2des(passwd: pointer; pdb:pdes_block);stdcall;
function xencrypt(p1:pointer; p2: pointer; p3:pointer):BOOL;stdcall;
function xdecrypt(p1:pointer; p2: pointer; p3:pointer):BOOL;stdcall;
function getpublickey(p1:pointer; p2: pointer):BOOL;stdcall;
function getsecretkey(p1:pointer; p2: pointer; p3:pointer):BOOL;stdcall;
function getcommonkey(p1:pointer; p2: pointer; p3:pointer;
db:pdes_block):BOOL;stdcall;

{ XDR routines for port mapper structures }
function xdr_pmap (x : XDRPtr; var v : pmap) : Bool; stdcall;
function xdr_pmaplist (x : XDRPtr; var v : pmaplist) : Bool; stdcall;
function xdr_broadcast_args (x : XDRPtr; var v : call_args) : Bool; stdcall;
function xdr_broadcast_results (x : XDRPtr; var v : call_results) : Bool;
stdcall;
function xdr_call_args (x : XDRPtr; var val : call_args) : Bool; stdcall;
function xdr_call_results (x : XDRPtr; var val : call_results) : Bool; stdcall;

function IsPortMapperRunning():bool;stdcall
function StartPortMapper():bool;stdcall

function xdr_rpcb(x:XDRPtr; r:rpcbPtr):bool;stdcall;
function xdr_rp__list(x:XDRPtr; l:rpcblist_ptr):bool;stdcall;
function xdr_rpcblist_ptr(x:XDRPtr; p:prpcblist_ptr):bool;stdcall;
function xdr_rpcb_rmtcallargs(x: XDRPtr; a:prpcb_rmtcallargs):bool;stdcall;
function xdr_rpcb_rmtcallres(x: XDRPtr; r:prpcb_rmtcallres):bool;stdcall;
function xdr_rpcb_entry(x: XDRPtr; r:prpcb_entry):bool;stdcall;
function xdr_rpcb_entry_list(x: XDRPtr; l:rpcb_entry_list_ptr):bool;stdcall;
function xdr_rpcb_entry_list_ptr(x: XDRPtr; l:
prpcb_entry_list_ptr):bool;stdcall;

```

```

function xdr_rpcbs_addrlist(x: XDRPtr; a:rpcbs_addrlist):bool;stdcall;
function xdr_rpcbs_addrlist_ptr(x: XDRPtr; a:prpcbs_addrlist_ptr):bool;stdcall;
function xdr_rpcbs_rmtcalllist(x: XDRPtr; l:rpcbs_rmtcalllist):bool;stdcall;
function xdr_rpcbs_rmtcalllist_ptr(x: XDRPtr;
l:prpcbs_rmtcalllist_ptr):bool;stdcall;
function xdr_rpcbs_proc(x: XDRPtr;p: rpcbs_proc):bool;stdcall;
function xdr_rpcb_stat(x: XDRPtr; s: prpcb_stat):bool;stdcall;
function xdr_rpcb_stat_byvers(x: XDRPtr; b: rpcb_stat_byvers ):bool;stdcall;
function xdr_netbuf(x: XDRPtr; b: pnetbuf):bool;stdcall;

{ port mapper functions }
function pmap_getport (host : PSockAddrIn; prog : u_long; vers : u_long;
    proc : u_long) : u_int; stdcall;
function pmap_getmaps (host : PSockAddrIn) : pmaplistPtr;
function pmap_set (prog : u_long; vers : u_long; prot : u_long;
    port : u_long) : Bool; stdcall;
function pmap_unset (prog : u_long; vers : u_long) : Bool; stdcall;
function pmap_rmtcall (host : PSockAddrIn; prog : u_long; vers : u_long;
    proc : u_long; in_proc : xdr_proc; in_arg : Pointer; out_proc : xdr_proc;
    out_arg : Pointer; t : TTimeVal; var port : u_long) : Integer; stdcall;

{ "blocking" RPC client functions }
function clnttcp_create (host : PSockAddrIn; prog : u_long; vers : u_long;
    var x : integer; y : u_int; z : u_int) : PHCLIENT; stdcall;
function clntudp_create (host : PSockAddrIn; prog : u_long; vers : u_long;
    t : TTimeVal; var y : integer) : PHCLIENT; stdcall;
procedure clnt_destroy (hClnt : PHCLIENT); stdcall;
function clnt_call (hClnt : PHCLIENT; proc : u_long; in_proc : xdr_proc;
    in_arg : Pointer; out_proc : xdr_proc; out_arg : Pointer;
    t : TTimeVal) : Integer; stdcall;
function callrpc (host : Pointer; prog : u_long; vers : u_long;
    proc : u_long; in_proc : xdr_proc; in_arg : Pointer; out_proc : xdr_proc;
    out_arg : Pointer) : Integer; stdcall;
function rpc_call (host : Pointer; prog : u_long; vers : u_long;
    proc : u_long; in_proc : xdr_proc; in_arg : Pointer; out_proc : xdr_proc;
    out_arg : Pointer; uid : Pointer) : Integer; stdcall;
procedure clnt_seterror (hClnt : PHCLIENT; x : u_int); stdcall;
function clnt_geterror (hClnt : PHCLIENT) : Integer; stdcall;
procedure clnt_geterr (hClnt : THandle; x : PClRpcError); stdcall;
function clnt_broadcast (prog : u_long; vers : u_long; proc : u_long;
    in_proc : xdr_proc; in_arg : Pointer; out_proc : xdr_proc; out_arg : Pointer;

```

```

    upcall : TFarProc) : Integer; stdcall;
function clnt_create (host : Pointer; y : u_long; z : u_long;
    prot : Pointer) : PHCLIENT; stdcall;
function rpc_broadcast (prog : u_long; vers : u_long; proc : u_long;
    in_proc : xdr_proc; in_arg : Pointer; out_proc : xdr_proc; out_arg : Pointer;
    upcall : TFarProc; var v : shortint) : Integer; stdcall;
procedure clnt_perror (hClnt : PHCLIENT; var x : shortint); stdcall;
function clnt_sperror (hClnt : PHCLIENT) : Pointer; stdcall;
function clnt_control (Client : PHCLIENT; Request : u_int; Info : Pointer) :
    BOOL; stdcall;

{ asynchronous RPC client functions }
function rac_send (hClnt : PHCLIENT; prog : u_long; in_proc : xdr_proc;
    in_arg : Pointer; out_proc : xdr_proc; out_arg : Pointer;
    t : TTimeVal) : HASYNCRPC; stdcall;
function rac_recv (hClnt : PHCLIENT; x : HASYNCRPC) : Integer; stdcall;
function rac_poll (hClnt : PHCLIENT; x : HASYNCRPC) : Integer; stdcall;
procedure rac_drop (hClnt : PHCLIENT; x : HASYNCRPC); stdcall;

{ RPC server functions }
function registerrpc (prog : u_long; vers : u_long; proc : u_long;
    service : TFarProc; in_proc : xdr_proc; out_proc : xdr_proc) : BOOL; stdcall;
function svctcp_create (sock : Integer; port : smallint; sendsz : u_long;
    recvsz : u_long) : HSERVER; stdcall;
function svcudp_create (sock : Integer; port : smallint) : HSERVER; stdcall;
function svc_register (hSvc : HSERVER; prog : u_long; vers : u_long;
    service : TFarProc; prot : u_long) : BOOL; stdcall;
function svc_unregister (prog : u_long; vers : u_long) : BOOL; stdcall;
procedure svc_destroy (hSvc : HSERVER); stdcall;
procedure svc_run ; stdcall;
procedure svc_run_ex (numthreads : Integer); stdcall;
procedure svc_exit ; stdcall;
function svc_getargs (hSvc : HSERVER; proc : xdr_proc; x : Pointer) : BOOL;
    stdcall;
function svc_freeargs (hSvc : HSERVER; proc : xdr_proc; x : Pointer) : BOOL;
    stdcall;
function svc_getcaller (hSvc : HSERVER) : TSocketAddrIn; stdcall;
function svc_sendreply (hSvc : HSERVER; proc : xdr_proc; x : Pointer) : BOOL;
    stdcall;
procedure svcerr_auth (hSvc : HSERVER; x : u_int); stdcall;
procedure svcerr_decode (hSvc : HSERVER); stdcall;
procedure svcerr_noprogram (hSvc : HSERVER); stdcall;

```

```
procedure svcerr_noproc (hSvc : HSERVER); stdcall;
procedure svcerr_systemerr (hSvc : HSERVER); stdcall;
procedure svcerr_weakauth (hSvc : HSERVER); stdcall;

procedure svc_fdset (pfd_set : PFDSets); stdcall;
procedure svc_getreqset (pfd_set : PFDSets); stdcall;
function svc_update_threads (NumOfThreads: INT):integer; stdcall;
function svcudp_bufcreate (sock:INT; port:SHORT; sendsz:ULONG;
recvsh:ULONG):HSERVER;stdcall;
procedure svc_set_timeout (TimeOut: TIMEVAL);stdcall;
function svctcp_create_secure (sock:INT; port:SHORT; sendsz:ULONG; recvsh:ULONG;
netname:Pointer; passwd:Pointer):HSERVER;stdcall;
function svcudp_create_secure (sock:INT; port:SHORT; netname:pointer;
passwd:pointer):HSERVER;stdcall;
```

implementation

```
function xdr_void; external Lib;
function xdr_int; external Lib;
function xdr_u_int; external Lib;
function xdr_long; external Lib;
function xdr_u_long; external Lib;
function xdr_short; external Lib;
function xdr_u_short; external Lib;
function xdr_bool; external Lib;
function xdr_enum; external Lib;
function xdr_opaque; external Lib;
function xdr_union; external Lib;
function xdr_char; external Lib;
function xdr_u_char; external Lib;
function xdr_float; external Lib;
function xdr_double; external Lib;
function xdr_array; external Lib;
function xdr_bytes; external Lib;
function xdr_string; external Lib;
function xdr_reference; external Lib;
function xdr_pointer; external Lib;
function xdr_wrapstring; external Lib;
function xdr_vector; external Lib;
function xdr_LARGE_INTEGER; external Lib;
function xdr_u_LARGE_INTEGER; external Lib;
```

```
function xdr_u_hyper; external Lib;
function xdr_u_longlong_t; external Lib;
function xdr_longlong_t; external Lib;
function xdr_hyper; external Lib;
function xdr_quadruple; external Lib;

procedure xdr_free; external Lib;
function xdr_create; external Lib;
function xdr_put_long; external Lib;
function xdr_get_long; external Lib;
function xdr_put_bytes; external Lib;
function xdr_get_bytes; external Lib;
function xdr_set_pos; external Lib;
function xdr_get_pos; external Lib;
procedure xdr_destroy; external Lib;
function xdr_auth; external Lib;
function xdr_auth_unix; external Lib;
function authnone_create; external Lib;
function authunix_create; external Lib;
function authsys_create; external Lib;
function authdes_create; external Lib;

function xdr_des_block; external Lib;
procedure bin2hex; external Lib;
procedure hex2bin; external Lib;
procedure passwd2des; external Lib;
function xencrypt; external Lib;
function xdecrypt; external Lib;
function getpublickey; external Lib;
function getsecretkey; external Lib;
function getcommonkey; external Lib;

function xdr_opaque_auth; external Lib;
function xdr_authunix_parms; external Lib;
function xdr_authdes_cred; external Lib;
function xdr_authdes_verf; external Lib;

procedure auth_destroy; external Lib;
function xdr_pmap; external Lib;
function xdr_pmaplist; external Lib;
```

```
function IsPortMapperRunning; external Lib;
function StartPortMapper; external Lib;
function xdr_broadcast_args; external Lib;
function xdr_broadcast_results; external Lib;
function xdr_call_args; external Lib;
function xdr_call_results; external Lib;
function pmap_getport; external Lib;
function pmap_getmaps; external Lib;
function pmap_set; external Lib;
function pmap_unset; external Lib;
function pmap_rmtcall; external Lib;
function clnttcp_create; external Lib;
function clntudp_create; external Lib;
procedure clnt_destroy; external Lib;
function clnt_call; external Lib;
function callrpc; external Lib;
function rpc_call; external Lib;
procedure clnt_seterror; external Lib;
function clnt_geterror; external Lib;
procedure clnt_geterr; external Lib;
function clnt_broadcast; external Lib;
function clnt_create; external Lib;
function rpc_broadcast; external Lib;
procedure clnt_perror; external Lib;
function clnt_sperror; external Lib;
function clnt_control; external Lib;
function rac_send; external Lib;
function rac_recv; external Lib;
function rac_poll; external Lib;
procedure rac_drop; external Lib;

function xdrmem_create; external Lib;
function xdrrec_create; external Lib;
function xdrrec_skiprecord; external Lib;
function xdrrec_eof; external Lib;
function xdrrec_endofrecord; external Lib;
function xdrstdio_create; external Lib;
function xdrrec_readbytes; external Lib;

function xdr_rpcb; external Lib;
```

```
function xdr_rp__list; external Lib;
function xdr_rpcblist_ptr; external Lib;
function xdr_rpcb_rmtcallargs; external Lib;
function xdr_rpcb_rmtcallres; external Lib;
function xdr_rpcb_entry; external Lib;
function xdr_rpcb_entry_list; external Lib;
function xdr_rpcb_entry_list_ptr; external Lib;
function xdr_rpcbs_addrlist; external Lib;
function xdr_rpcbs_addrlist_ptr; external Lib;
function xdr_rpcbs_rmtcalllist; external Lib;
function xdr_rpcbs_rmtcalllist_ptr; external Lib;
function xdr_rpcbs_proc; external Lib;
function xdr_rpcb_stat; external Lib;
function xdr_rpcb_stat_byvers; external Lib;
function xdr_netbuf; external Lib;
```

```
{Server-Side functions}
```

```
function registerrpc; external LibSrv;
function svctcp_create; external LibSrv;
function svcudp_create; external LibSrv;
function svc_register; external LibSrv;
function svc_unregister; external LibSrv;
procedure svc_destroy; external LibSrv;
procedure svc_run; external LibSrv;
procedure svc_run_ex; external LibSrv;
procedure svc_exit; external LibSrv;
function svc_getargs; external LibSrv;
function svc_freeargs; external LibSrv;
function svc_getcaller; external LibSrv;
function svc_sendreply; external LibSrv;
procedure svcerr_auth; external LibSrv;
procedure svcerr_decode; external LibSrv;
procedure svcerr_noprogram; external LibSrv;
procedure svcerr_noproc; external LibSrv;
procedure svcerr_systemerr; external LibSrv;
procedure svcerr_weakauth; external LibSrv;

procedure svc_fdset; external LibSrv;
procedure svc_getreqset; external LibSrv;
```

```
function svc_update_threads; external LibSrv;  
function svcudp_bufcreate; external LibSrv;  
procedure svc_set_timeout; external LibSrv;  
function svctcp_create_secure; external LibSrv;  
function svcudp_create_secure; external LibSrv;  
  
end.
```


Getting Started with Distinct ONC RPC/XDR

The following section gives a step by step description of how to get started with the Distinct ONC RPC/XDR Toolkit. It assumes that the developer already has a basic knowledge of RPC programming.

Before you Start

If you have a previous version of the Distinct ONC RPC/XDR Toolkit on your system you must uninstall this before installing version 4.0 or higher. Now install the toolkit following the on-screen instructions. The installation program should create shortcuts in the start menu for the RPCInfo and RPCBIND applications under the Distinct folder. If these are not present, your installation was not successful and you need to reinstall the toolkit.

Using the Toolkit in the Microsoft C++ Environment

Once the toolkit has been successfully installed:

1. At the end of your installation you will be asked to enter the product serial number. If you are evaluating choose Trial, otherwise enter the serial number and keycode that came with your toolkit. The serial number came with the product when you purchased it. You also received an ID, you need to register your software on the Distinct registration site to receive your keycode if you have not already done this, do this now by clicking this link <http://www.distinct.com/sales/register.asp> When you have both the serial number and keycode you can enter them in the serial number dialog box to activate your toolkit license.
2. Create a blank workspace, console application or MFC application depending upon your needs. For example if you wish to create an application called abc, create this project under a directory called abc.
3. Create a new source file with the name abc.x within the same directory and write the IDL definitions (constants, data structures and function calls) of your RPC service to this file, save it and close it.
4. Open a command prompt and go to the abc folder and type following command:

```
C:\>rpcgen abc.x
```

This is assuming you are creating a client and a server application. If you are only creating a client application add the `-l` parameter to `rpcgen`. In this case the command would be

```
C:\>rpcgen -l abc.x
```

5. This should result in creating 3 or 4 files in the same directory namely-

```
abc.h
abc_clnt.c
abc_svc.c
abc_xdr.c      (This file will not be generated if you are not creating composite data types such as
structures and unions. This depends upon the contents of abc.x file)
```

Note that as described above command line switches of the `rpcgen` utility can be used to limit the files generated to just the client or server stub files as needed. See the section entitled RPC Protocol Compiler for details on the `RPCGEN` options available.

6. If the above files were generated, your XDR data definition is ready and you are ready to write the application which has two basic parts: the server part which actually does the required job, and the client part which calls it. In this explanation, we will call them abc server and abc client respectively.

7. Your abc.h file and it should have a line

```
#include <d32-rpc.h>
```

 This has been added by the Distinct RPC generator, and it calls the RPC header file.
8. In the VC++ interface go to Tools->Options->Directories and add the following folder:

```
%RPC Installation Directory% \Header\C
```

 If you have installed the RPC toolkit in default directory, then the above directory path should look like,

```
C:\Program Files\Distinct\RPC-32\Header\C
```
9. Create the client source file for abc client and include following files to the project.

```
abc.h  
abc_clnt.c  
[abc_xdr.c]      (if available)
```

 In the VC++ interface go to Project->Settings->Link and add the following file for linking

```
d32-rpc.lib
```

 Compile and link the abc client project.
10. To build the server application, create the server source file for abc server and include following files to the project.

```
abc.h  
abc_svc.c  
[abc_xdr.c]      (if available)
```

 In the VC++ interface go to Project->Settings->Link and add the following files for linking:

```
d32-rpc.lib  
drpcsrvr.lib
```

 Compile and link the abc server project.
 Now run the server from the command prompt. You will see the "RPCBind" icon in the Windows system tray. You can check if the server is running properly by running RPCInfo, which comes with the Distinct RPC toolkit, to query the PC running the server for a list of registered services.
11. Now run the client and you should get the required service back from the server.

Using the Toolkit in the Borland C++ Environment

Follow the same procedures as above. Following are some additional steps for the Borland C++ Builder environment:

1. To include the header files in your project. In the Borland C++ Builder interface go to Project->Options->Directories and in the Include path section add the following directory

```
%RPC Installation Directory% \Header\C
```

If you have installed the RPC toolkit in default directory, then the above directory structure should look like,

```
C:\Program Files\Distinct\RPC-32\Header\C
```

2. If you get a Link error at compilation time, include the following library to the project.
 1. d32-rpc.lib (For server and for client)
 2. drpcsrvr.lib (Only for RPC server)

These file will be stored in

%RPC Installation Directory% \Lib\Bc\

If you have installed the RPC toolkit in default directory, then the above directory structure should look like:

C:\Program Files\Distinct\RPC-32\Lib\Bc

Then recompile your project.

Facts about the RPC Deployment Licenses

Before you can deploy the Distinct RPC libraries as part of your application, you must purchase the appropriate licenses from Distinct. If you are expecting to deploy only a few copies you may opt to use the respective Distinct run time installation for the client or server license involved. On the other hand if this is a commercial or high deployment application, it may be easier to integrate the libraries with your own installation. The following provides you with the information needed to do this.

Modules required when deploying an RPC Client license

When deploying an RPC client built with the Distinct ONC RPC/XDR Toolkit for C/C++ you must include the following files:

DSTNCT32.DLL
GHOST32.EXE
D32-RPC.DLL
RPC
RPCINFO.EXE

Modules required when deploying an RPC Server license

When deploying an RPC server built with the Distinct ONC RPC/XDR Toolkit for C/C++ you must include the following files:

DSTNCT32.DLL
GHOST32.EXE
DRPCSRVR.DLL
D32-RPC.DLL
RPCBIND.EXE
RPC
RPCINFO.EXE

Reference

auth_destroy ()

Description

Destroy authentication information.

```
#include <windows.h>
```

```
#include <d32-rpc.h>
```

```
void WINAPI auth_destroy (authority)
```

AUTH **authority*; Authority structure previously created with **authnone_create**, **authsys_create**, **authunix_create** or **authdes_create**

Remarks

The **auth_destroy** function destroys the authentication information specified by *authority*. The authentication information must have been created previously using **authnone_create**, **authsys_create**, **authunix_create** or **authdes_create**.

Return Value

The **auth_destroy** function does not return a value.

authdes_create ()

Description

Create system authentication information.

```
#include <windows.h>
```

```
#include <d32-rpc.h>
```

```
AUTH * WINAPI authdes_create (clientnetname, password, servernetname, window)
```

```
char *clientnetname;          RPC client's network name
```

```
char *password;              Password for RPC client's secret key
```

```
char *servernetname;        RPC server's network name
```

```
unsigned int window;        Valid period of the client credential, given in seconds
```

Remarks

The **authdes_create** function is used on the client side to return an authentication handle that will enable the use of the secure authentication system. The first parameter *clientnetname* and the third *servernetname* are the network name for RPC client and server respectively. A password, the second parameter is also needed to decrypt the client's secret key stored in the public key database. The last parameter is *window* on the validity of the client credential, given in seconds. If the difference in time between the client's clock and the server's clock exceeds *window*, the server will reject the client's credentials.

Warning

The **authdes_create** function requires the inclusion of a DLL library "libey32.dll" which is subject to US export restrictions for encryption software of its type. Please read and ensure compliance with the US export regulations on DES encryption before including this API call in your application. This library is available from Distinct. Please contact our technical support staff to obtain a copy. We will require your full name, company address and list of countries in which you intend to deploy your resulting RPC application. This feature comes with the Distinct ONC RPC/XDR Toolkit – ENC.

Return Value

The **authdes_create** function returns an authentication handle in case of success and NULL if an authentication handle could not be created.

authnone_create ()

Description

Create empty authentication information.

```
#include <windows.h>
```

```
#include <d32-rpc.h>
```

```
AUTH * WINAPI authnone_create (void)
```

Remarks

The **authnone_create** function creates an authentication handle that can be used by the client to pass empty authentication information to the server.

Return Value

The **authnone_create** function returns an authentication handle in case of success and NULL if an authentication handle could not be created.

authsys_create ()

Description

Create system authentication information.

```
#include <windows.h>
```

```
#include <d32-rpc.h>
```

```
AUTH * WINAPI authsys_create (host, uid, gid, len, aux_gids)
```

char * <i>host</i> ;	Name of local system
unsigned long <i>uid</i> ;	User id
unsigned long <i>gid</i> ;	Users group id
unsigned long <i>len</i> ;	Number of elements in <i>aux_gids</i> array
unsigned long * <i>aux_gids</i> ;	Array of group ids of user

Remarks

The **authsys_create** function creates an authentication handle which can be used by the client to pass authentication information to the server. Currently the **authsys_create** function uses UNIX style authentication and therefore provides the same functionality as the **authunix_create** function.

Return Value

The **authsys_create** function returns an authentication handle in case of success and NULL if an authentication handle could not be created.

authunix_create ()

Description

Create UNIX style authentication information.

```
#include <windows.h>
```

```
#include <d32-rpc.h>
```

```
AUTH * WINAPI authunix_create (host, uid, gid, len, aux_gids)
```

char * <i>host</i> ;	Name of local system
unsigned long <i>uid</i> ;	User id
unsigned long <i>gid</i>	Users group id
unsigned long <i>len</i> ;	Number of elements in <i>aux_gids</i> array
unsigned long * <i>aux_gids</i>	Array of group ids of user

Remarks

The **authunix_create** function creates an authentication handle which can be used by the client to pass authentication information to the server. Currently the **authunix_create** function provides the same functionality as the **authsys_create** function.

Return Value

The **authunix_create** function returns an authentication handle in case of success and NULL if an authentication handle could not be created.

callrpc ()

Description

Issue remote procedure call to specified server using UDP protocol.

```
#include <windows.h>
```

```
#include <d32-rpc.h>
```

```
int WINAPI callrpc (host, prog, vers, proc, in_proc, in_arg, out_proc, out_arg)
```

char *host;	Name of server
unsigned long prog;	RPC program number
unsigned long vers;	Version of RPC program
unsigned long proc;	Procedure number requested
xdr_proc in_proc;	XDR encode procedure
void *in_arg;	Argument
xdr_proc out_proc;	XDR decode procedure
void *out_arg;	Return value

Remarks

The **callrpc** function makes a remote procedure call to a specified program, version, and procedure number on the specified host with AUTH_NONE authentication. The XDR routine pointed to by *in_proc* is used to encode the arguments in the buffer *in_arg*. Upon successful completion of the call, the XDR routine pointed to by *out_proc* is used to decode the result into the buffer *out_arg*. It is the responsibility of the caller to ensure that the buffer pointed to by *out_arg* has been allocated large enough to hold the result. The **callrpc** function will retry the request 5 times, every five seconds, returning with a time-out error if the 25 seconds has elapsed with no reply. RPCs made this way use UDP/IP transport with no control of time-outs or authentication. The maximum size of the packet allowed is the one defined in registry (see section Registry Entries for details).

Return Value

The **callrpc** function returns RPC_SUCCESS if successful, or one of the error return values listed in section '1.12 - RPC Error Return Values'.

clnt_broadcast ()

Description

Broadcast a procedure request to all servers on the network.

```
#include <windows.h>
```

```
#include <winsock.h>
```

```
#include <d32-rpc.h>
```

```
int WINAPI clnt_broadcast (prog, vers, proc, in_proc, in_arg, out_proc, out_arg, result)
```

unsigned long <i>prog</i> ;	RPC program number
unsigned long <i>vers</i> ;	Version of RPC program
unsigned long <i>proc</i> ;	Procedure number requested
xdr_proc <i>in_proc</i> ;	XDR encode procedure
void * <i>in_arg</i> ;	Argument
xdr_proc <i>out_proc</i> ;	XDR decode procedure
void * <i>out_arg</i> ;	Return value
BOOL WINAPI (* <i>result</i>) (void * <i>out</i> ; SOCKADDR_IN * <i>addr</i>);	

Remarks

The **clnt_broadcast** function is a broadcast version of **callrpc** function. The call is broadcast over the network and each time a response is received the callback function *result* is called. The function pointed to by *result* must have the following prototype.

```
BOOL WINAPI result (void *out; SOCKADDR_IN *addr);
```

On every call to the *result* function, *out* will point to the decoded result of the remote call and *addr* will point to a SOCKADDR_IN structure (as defined in the Windows Sockets include file) specifying the address of the responding host. If the *result* function returns FALSE then **clnt_broadcast** waits for more replies, otherwise it returns with the appropriate status.

The broadcast packets are limited in size by the underlying data link transfer unit. For Ethernet, this limits the caller arguments portion of the request packet to 1400 bytes. Refer to **callrpc** function for further details.

Return Value

The **clnt_broadcast** function returns RPC_SUCCESS if successful or one of the error return values listed in section '1.12- RPC Error Return Values'.

clnt_call ()

Description

Issue remote procedure call on established UDP or TCP connection.

```
#include <windows.h>
#include <winsock.h>
#include <d32-rpc.h>
```

```
int WINAPI clnt_call (hClnt, proc, in_proc, in_arg, out_proc, out_arg, timeout)
```

HCLIENT *hClnt;	Client structure handle
unsigned long proc;	Procedure number
xdr_proc in_proc;	XDR encode procedure
void *in_arg;	Argument
xdr_proc out_proc;	XDR decode procedure
void *out_arg;	Return value
TIMEVAL timeout;	Time to wait for completion

Remarks

The **clnt_call** function issues a remote procedure call on the connection identified by *hClnt*. The client structure handle *hClnt* must have been created previously using **clnttcp_create** or **clntudp_create** or one of the other client creation functions. The parameter *proc* specifies the procedure number of the remote service to be called. The XDR routine pointed to by *in_proc* is used to encode the arguments in the buffer *in_arg*. Upon successful completion of the call, the XDR routine pointed to by *out_proc* is used to decode the result into the buffer *out_arg*. It is the responsibility of the caller to ensure that the buffer pointed to by *out_arg* has been allocated large enough to hold the result. The number of seconds after which the call should time out is specified by the *timeout* structure (the *timeout.tv_usec* field is not used and should be set to 0).

Return Value

The **clnt_call** function returns **RPC_SUCCESS** if successful, or one of the error return values listed in section '[1.12 - RPC Error Return Values](#)'.

clnt_control ()**Description**

Change or retrieve information about client handle.

```
#include <windows.h>
```

```
#include <d32-rpc.h>
```

```
BOOL WINAPI clnt_control (hClnt, request, info)
```

```
HCLIENT *hClnt;           Client structure handle
```

```
unsigned long request;     Type of operation
```

```
void *info;               Pointer to information
```

Remarks

The **clnt_control** function sets or retrieves various parameters about the client handle *hClnt*. For both the TCP and the UDP protocol the supported requests and their arguments are listed below.

Request	Type	Description
CLSET_TIMEOUT	TIMEVAL	Set total timeout
CLGET_TIMEOUT	TIMEVAL	Get total timeout
CLGET_SVC_ADDR	SOCKADDR	Get servers address
CLSET_FD_CLOSE	NULL	Close file descriptor when destroying client handle

In addition the following requests are valid for clients using the UDP transport protocol.

Request	Type	Description
CLSET_RETRY_TIMEOUT	TIMEVAL	Set retransmission time
CLGET_RETRY_TIMEOUT	TIMEVAL	Get retransmission time

Return Value

The **clnt_control** function returns TRUE if successful and FALSE in case of failure.

clnt_create ()

Description

Create RPC client structure using either the TCP or the UDP protocol.

```
#include <windows.h>
```

```
#include <d32-rpc.h>
```

```
HCLIENT * WINAPI clnt_create (host, prog, vers, type)
```

char * <i>host</i> ;	Internet name or address of server
unsigned long <i>prog</i> ;	RPC program number
unsigned long <i>vers</i> ;	Version number
char * <i>type</i> ;	TCP or UDP

Remarks

The **clnt_create** function creates an RPC client structure which can then be used to issue one or more remote procedure calls to the server specified by *host*. This can be either the name of the server or the address of the server in the dotted form. The *prog* and *vers* parameters specify the RPC program number and version of the server to be contacted on the host.

The *type* parameter must point to the character string "tcp" or "udp" to specify whether the client requests the TCP or the UDP protocol.

Return Value

The **clnt_create** function returns the handle of the new client structure or NULL to indicate an error condition.

clnt_destroy ()

Description

Free all resources allocated for established connection.

```
#include <windows.h>
```

```
#include <d32-rpc.h>
```

```
void WINAPI clnt_destroy (hClnt)
```

```
HCLIENT *hClnt;           Client structure handle
```

Remarks

The **clnt_destroy** function frees all resources associated with a client structure. The client structure handle *hClnt* must have been created previously using **clnttcp_create** or **clntudp_create** or other client creation routines. The *hClnt* handle can not be used after a call to **clnt_destroy**. Calling **clnt_destroy** while an asynchronous RPC call is in progress will produce unpredictable results.

Return Value

The **clnt_destroy** function does not return a value.

clnt_freeres ()

Description

Free any resources allocated by the client when it decoded the results of an RPC call.

```
#include <windows.h>
```

```
#include <d32-rpc.h>
```

BOOL WINAPI clnt_freeres (hClnt, outproc, out)

HCLIENT *hClnt; Handle to the client structure.

xdr_proc outproc; The XDR routine used to free the structure.

void *out; Address of the results.

Remarks

The **clnt_freeres** function frees any resources allocated by the D32-RPC.DLL to decode the results of an RPC call. This function has effect if the client structure was previously created using **clnttcp_create**, **clntudp_create** or one of the other client create functions. This function assumes that the XDR routine *outproc* points to was used to decode the results and that the buffer used to decode the results is *out*.

Return Value

The **clnt_freeres** function returns TRUE. If the resources are successfully freed. Otherwise it returns FALSE.

clnt_geterr ()

Description

Get error structure from client handle.

```
#include <windows.h>
```

```
#include <d32-rpc.h>
```

```
void WINAPI clnt_geterr (hClnt, RpcError)
```

```
HCLIENT *hClnt;           Client structure handle
```

```
CIRpcError *RpcError;     Structure in which error information is returned
```

Remarks

The **clnt_geterr** function retrieves the complete error structure from the last performed client operation. The client structure handle *hClnt* must have been created previously using **clnttcp_create** or **clntudp_create** or other client creation routines.

Return Value

The **clnt_geterr** function does not return a value.

clnt_geterror ()

Description

Get last error in failed client operation.

```
#include <windows.h>
```

```
#include <d32-rpc.h>
```

```
int WINAPI clnt_geterror (hClnt)
```

```
HCLIENT *hClnt;           Client structure handle or NULL if client creation  
                           failed
```

Remarks

The **clnt_geterror** function retrieves the error value from the last performed client operation. The client structure handle *hClnt* must have been created previously using **clnttcp_create** or **clntudp_create** or other client creation routines. If the client creation routine failed then this routine can still be called with *hClnt* set to NULL to retrieve the error.

Return Value

The **clnt_geterror** function returns the error which occurred during the last operation.

clnt_perror ()

Description

Display message box with error from last performed client operation.

```
#include <windows.h>
```

```
#include <d32-rpc.h>
```

```
void WINAPI clnt_perror (hClnt, s)
```

```
HCLIENT *hClnt;           Client structure handle or NULL if client creation  
                           failed
```

```
char *s;                   User error message
```

Remarks

The **clnt_perror** function displays a message box with the error message generated by the last client operation. The actual message is prepended with the string pointed to by *s* and a colon. The client structure handle *hClnt* must have been created previously using **clnttcp_create** or **clntudp_create** or other client creation routines. If the client creation routine failed then this routine can still be called with *hClnt* set to NULL to display the error message box.

Return Value

The **clnt_perror** function does not return a value.

clnt_sperror ()

Description

Return string corresponding to last error in failed client operation.

```
#include <windows.h>
```

```
#include <d32-rpc.h>
```

```
char * WINAPI clnt_sperror (hClnt)
```

```
HCLIENT *hClnt;           Client structure handle or NULL if client creation  
                           failed.
```

Remarks

The **clnt_sperror** function returns a string corresponding to the last failed client operation. The client structure handle *hClnt* must have been created previously using **clnttcp_create** or **clntudp_create** or other client creation routines. If the client creation routine failed then this routine can still be called with *hClnt* set to NULL.

Return Value

The **clnt_sperror** function returns the string corresponding to the last error.

clnttcp_create ()

Description

Create connection with specified server using TCP protocol.

```
#include <windows.h>
#include <winsock.h>
#include <d32-rpc.h>
```

HCLIENT * WINAPI clnttcp_create (*addr, prog, vers, sockp, sendsz, recvsz*)

SOCKADDR_IN * <i>addr</i> ;	Internet address of server
unsigned long <i>prog</i> ;	RPC program number
unsigned long <i>vers</i> ;	Version number
int * <i>sockp</i> ;	Socket identifier
unsigned int <i>sendsz</i> ;	Send buffer size
unsigned int <i>recvsz</i> ;	Receive buffer size

Remarks

The **clnttcp_create** function creates an RPC client structure which can then be used to issue one or more remote procedure calls with **clnt_call** over the TCP transport protocol to the server specified by *addr*. The server address *addr* should be given as a **SOCKADDR_IN** (as defined in the Windows Sockets include file) and must be in network byte order. The *prog* and *vers* parameters specify the RPC program number and version of the server to be contacted on the host. If the *addr->port* field is set to 0 (i.e. the port of the remote service is not known), then the port mapper on the remote machine is called to look up the port on which the service is registered. If port number is known it must be given in network byte order. If *sockp* points to **RPC_ANYSOCK**, then a new socket is created and the buffer pointed to by *sockp* will be updated to contain the socket number. If *sockp* points to a value other than **RPC_ANYSOCK**, then that value is expected to be a valid TCP socket. Because the TCP protocol is able to transmit argument and return structures of virtually any size, the maximum send and receive buffer sizes in bytes must be given in *sendsz* and *recvsz*. If *sendsz* or *recvsz* are set to 0, then the default values from the registry are used.

Return Value

The **clnttcp_create** function returns the handle of the new client structure or **NULL** to indicate an error condition.

clntudp_bufcreate ()

Description

Create connection with specified server with specified send and receive buffer sizes using UDP protocol.

```
#include <windows.h>
#include <winsock.h>
#include <d32-rpc.h>
```

HCLIENT * WINAPI clntudp_create (*addr, prog, vers, wait, sockp, sendsz, recvsz*)

SOCKADDR_IN * <i>addr</i> ;	Internet address of server
unsigned long <i>prog</i> ;	RPC program number
unsigned long <i>vers</i> ;	Version number
TIMEVAL <i>wait</i> ;	Resend time
int * <i>sockp</i> ;	Socket identifier
unsigned int <i>sendsz</i> ;	Send buffer size
unsigned int <i>recvsz</i>	Receive buffer size

Remarks

The **clntudp_bufcreate** function creates an RPC client structure which can then be used to issue one or more remote procedure calls with **clnt_call** over the UDP transport protocol to the server specified by *addr*. The server address *addr* should be given as a **SOCKADDR_IN** (as defined in the Windows Sockets include file) and must be in network byte order. The *prog* and *vers* parameters specify the RPC program number and version of the server to be contacted on the host. If the *addr->port* field is set to 0 (i.e. the port of the remote service is not known), then the port mapper on the remote machine is called to look up the port on which the service is registered. If port number is known it must be given in network byte order. The number of seconds after which the call should be resent if no answer arrived is specified by the *wait* structure (the *wait.usec* field is not used and should be set to 0). The total timeout in seconds is specified when calling **clnt_call**. If *sockp* points to **RPC_ANYSOCK**, then a new socket is created and the buffer pointed to by *sockp* will be updated to contain the socket number. If *sockp* points to a value other than **RPC_ANYSOCK**, then that value is expected to be a valid UDP socket which has been created with a call to the underlying Windows Sockets library. The *sendsz* and *recvsz* parameters specify the send and receive buffer sizes respectively to be used.

Return Value

The **clntudp_create** function returns the handle of the new client structure or **NULL** to indicate an error condition.

clntudp_create ()

Description

Create connection with specified server with default send and receive buffer sizes using UDP protocol.

```
#include <windows.h>
#include <winsock.h>
#include <d32-rpc.h>
```

HCLIENT * WINAPI clntudp_create (*addr, prog, vers, wait, sockp*)

SOCKADDR_IN * <i>addr</i> ;	Internet address of server
unsigned long <i>prog</i> ;	RPC program number
unsigned long <i>vers</i> ;	Version number
TIMEVAL <i>wait</i> ;	Resend time
int * <i>sockp</i> ;	Socket identifier

Remarks

The **clntudp_create** function creates an RPC client structure which can then be used to issue one or more remote procedure calls with **clnt_call** over the UDP transport protocol to the server specified by *addr*. The server address *addr* should be given as a **SOCKADDR_IN** (as defined in the Windows Sockets include file) and must be in network byte order. The *prog* and *vers* parameters specify the RPC program number and version of the server to be contacted on the host. If the *addr->port* field is set to 0 (i.e. the port of the remote service is not known), then the port mapper on the remote machine is called to look up the port on which the service is registered. If port number is known it must be given in network byte order. The number of seconds after which the call should be resent if no answer arrived is specified by the *wait* structure (the *wait.usec* field is not used and should be set to 0). The total timeout in seconds is specified when calling **clnt_call**. If *sockp* points to **RPC_ANYSOCK**, then a new socket is created and the buffer pointed to by *sockp* will be updated to contain the socket number. If *sockp* points to a value other than **RPC_ANYSOCK**, then that value is expected to be a valid UDP socket which has been created with a call to the underlying Windows Sockets library.

Return Value

The **clntudp_create** function returns the handle of the new client structure or NULL to indicate an error condition.

pmap_getmaps ()

Description

Query a specified server for list of available services.

```
#include <windows.h>
```

```
#include <winsock.h>
```

```
#include <d32-rpc.h>
```

```
pmaplist * WINAPI pmap_getmaps (host)
```

```
SOCKADDR_IN *host;           Internet address of server
```

Remarks

The **pmap_getmaps** function queries the server identified by *host* for a list of all services (independent of the transport protocol) registered with its RPCBIND or port mapper. The server address is specified in network byte order. The information is obtained in the form of a linked list and the caller must free each element (the `pmaplist` structure itself and the `pmap` structure it contains) of the linked list by calling **xdr_free**.

Return Value

The **pmap_getmaps** function returns a pointer to the first element of a linked list of port mapper (`pmaplist`) structures or NULL to indicate an error condition.

pmap_getport ()

Description

Retrieve the UDP or TCP port number of specified service.

```
#include <windows.h>
```

```
#include <winsock.h>
```

```
#include <d32-rpc.h>
```

```
unsigned int WINAPI pmap_getport (host, prog, vers, prot)
```

```
SOCKADDR_IN *host;           Internet address of server
```

```
unsigned long prog;          RPC program number
```

```
unsigned long vers;          RPC version number
```

```
unsigned long prot;          network protocol
```

Remarks

The **pmap_getport** function calls the port mapper service on the remote machine specified by *host* to determine the port number on which the service with program number *prog*, version number *vers* and transport protocol *prot* is registered.

Return Value

The **pmap_getport** function returns the port number (in host byte order) on which the remote service is registered, or 0 to indicate an error condition.

pmap_rmtcall ()**Description**

Request remote port mapper to execute specified command.

```
#include <windows.h>
#include <winsock.h>
#include <d32-rpc.h>
```

```
int WINAPI pmap_rmtcall (addr, prog, vers, proc, in_proc, in_arg, out_proc, out_arg, wait,
port)
```

SOCKADDR_IN *addr;	Address of remote machine
unsigned long prog;	RPC program number
unsigned long vers;	Version of RPC program
unsigned long proc;	Procedure number requested
xdr_proc in_proc;	XDR encode procedure
void *in_arg;	Argument
xdr_proc out_proc;	XDR decode procedure
void *out_arg;	Return value
TIMEVAL wait;	Time to wait for response
unsigned long *port;	Remote port number

Remarks

The **pmap_rmtcall** function calls the port mapper on the remote machine specified by *addr* to execute the given RPC procedure. The three values given by *prog*, *vers* and *proc* uniquely identify the remote procedure to be executed. The XDR routine pointed to by *in_proc* is used to encode the arguments in the buffer *in_arg*. Upon successful completion of the call, the XDR routine pointed to by *out_proc* is used to decode the result into the buffer *out_arg*. It is the responsibility of the caller to ensure that the buffer pointed to by *out_arg* has been allocated large enough to hold the result. The number of seconds after which the call should time out is specified by the *wait* structure (the *wait.tv_usec* field is not used and should be set to 0). Upon successful completion, *port* contains the port number on which the service is registered on the remote machine.

Return Value

The **pmap_rmtcall** function returns `RPC_SUCCESS` if successful, or one of the error return values listed in section '[1.12 - RPC Error Return Values](#)'.

pmap_set ()

Description

Register given service with local port mapper.

```
#include <windows.h>
```

```
#include <winsock.h>
```

```
#include <d32-rpc.h>
```

```
BOOL WINAPI pmap_set (prog, vers, prot, port)
```

```
unsigned long prog;           RPC program number
```

```
unsigned long vers;          Version number
```

```
unsigned long prot;          Protocol
```

```
unsigned long port;          Port
```

Remarks

The **pmap_set** function calls the port mapper service on the local machine to register an RPC program *prog* with version *vers* and transport protocol *prot* on the local port specified by *port*.

The *prot* parameter must be set to IPPROTO_UDP or IPPROTO_TCP accordingly.

Return Value

The **pmap_set** function returns TRUE if the registration was successful and FALSE if the registration failed.

pmap_unset ()

Description

Remove specified service from local machine.

```
#include <windows.h>
```

```
#include <d32-rpc.h>
```

```
BOOL WINAPI pmap_unset (prog, vers)
```

```
unsigned long prog;           RPC program number
```

```
unsigned long vers;          Version number
```

Remarks

The **pmap_unset** function removes the mapping of the program number *prog* and the version *vers* from the list of services registered with the port mapper on the local machine.

Return Value

The **pmap_unset** function returns TRUE if successful, or FALSE to indicate an error condition.

rac_drop ()

Description

Terminate asynchronous remote procedure call in progress.

```
#include <windows.h>
```

```
#include <d32-rpc.h>
```

```
void WINAPI rac_drop (hClnt, AsyncHandle)
```

HCLIENT *hClnt; Client structure

HASYNCRPC AsyncHandle; Asynchronous handle returned by **rac_send**

Remarks

The **rac_drop** function is used to terminate the processing of a previous **rac_send** call. It does not stop the execution on the remote machine but any further replies will be discarded. The *hClnt* parameter specifies the client structure handle created by calling **clnttcp_create** or **clntudp_create** or another client creation routine and *AsyncHandle* specifies a valid handle returned by the **rac_send** call.

Return Value

Returns back after notifying the remote machine.

rac_poll ()

Description

Poll status of asynchronous remote procedure call in progress.

```
#include <windows.h>
```

```
#include <d32-rpc.h>
```

```
int WINAPI rac_poll (hCln, AsyncHandle)
```

HCLIENT *hCln; Client structure

HASYNCRPC AsyncHandle; Asynchronous handle returned by **rac_send**

Remarks

The **rac_poll** function allows an application to check if a reply to a **rac_send** request has been received. The *hCln* parameter specifies the client structure handle created by calling **clnttcp_create** or **clntudp_create** or another client creation routine and *AsyncHandle* specifies a valid handle returned by the **rac_send** call.

The **rac_poll** function returns immediately.

Return Value

The **rac_poll** function returns **RPC_SUCCESS** if the remote call has succeeded (i.e. a reply has been received) or one of the error return values listed in section '[1.12 - RPC Error Return Values](#)'.

rac_recv ()

Description

Obtain results of asynchronous remote procedure call.

```
#include <windows.h>
```

```
#include <d32-rpc.h>
```

```
int WINAPI rac_recv (hClnt, AsyncHandle)
```

HCLIENT *hClnt; Client structure

HASYNCRPC AsyncHandle; Asynchronous handle returned by **rac_send**

Remarks

The **rac_recv** function is used to retrieve a decoded reply to an asynchronous RPC call issued with **rac_send**. If no reply has been received, then the call blocks until a reply is available. The **rac_poll** function can be used to check if a reply has been received before calling **rac_recv**.

Return Value

The **rac_recv** function returns **RPC_SUCCESS** if the remote call has succeeded (i.e. a reply has been received) or one of the error return values listed in section '1.12 - RPC Error Return Values'.

rac_send ()**Description**

Issue asynchronous remote procedure call.

```
#include <windows.h>
#include <winsock.h>
#include <d32-rpc.h>
```

HASYNCRPC WINAPI rac_send (*hClnt, proc, in_proc, in_arg, out_proc, out_arg, timeout*)

HCLIENT * <i>hClnt</i> ;	Client structure
unsigned long <i>proc</i> ;	Procedure number
xdr_proc <i>in_proc</i> ;	XDR encode procedure
void * <i>in_arg</i> ;	Argument
xdr_proc <i>out_proc</i> ;	XDR decode procedure
void * <i>out_arg</i> ;	Return value
TIMEVAL <i>timeout</i> ;	Time in seconds to wait

Remarks

The **rac_send** function initiates an asynchronous RPC call over the connection identified by *hClnt*. The client must have already been created using the **clnttcp_create** or **clntudp_create** function. The procedure of the remote service to be called is given by *proc*. The XDR routine pointed to by *in_proc* is used to encode the arguments in the buffer *in_arg*. The procedure returns immediately after sending the RPC request. It does not wait for a response from the other side. The **rac_poll** function can be used to check if a reply to the request has been received.

When a response arrives, it is decoded with the XDR routine pointed to by *out_proc* into the buffer *out_arg*. The *out_arg* buffer must be allocated large enough by the calling program. The **rac_rcv** function can be used to retrieve a decoded reply of the **rac_send** call.

If a response is not received within the time specified by the *timeout* structure, then the connection is dropped by the RPC/XDR-32 library. Only the *timeout.tv_sec* field is used and the *timeout.tv_usec* field should be set to zero.

The **rac_drop** function can be used to stop the processing of a **rac_send** call.

Return Value

The **rac_send** function returns the handle of the newly created asynchronous client structure or NULL to indicate an error condition.

registerrpc ()**Description**

Register service using UDP protocol with local port mapper.

```
#include <windows.h>
```

```
#include <d32-rpc.h>
```

```
BOOL WINAPI registerrpc (prog, vers, proc, req, inproc, outproc)
```

```
unsigned long prog;           RPC program number
```

```
unsigned long vers;          Version number
```

```
unsigned long proc;          Procedure number
```

```
void * WINAPI (*req) (void *ptr);
```

```
xdr_proc inproc;             XDR decode procedure
```

```
xdr_proc outproc;           XDR encode procedure
```

Remarks

The **registerrpc** function registers the RPC service identified by the RPC program number *prog*, the version *vers* and the procedure number *proc* with the local port mapper. The service is registered for the UDP transport protocol only (**svc_register** can be used to register services for either the UDP or the TCP transport protocol). The service will be available on a default port selected by the system. The *req* parameter specifies the address of the function to be called whenever a service request arrives. The function pointed to by *req* must have the following prototype.

```
void * WINAPI req (void *ptr);
```

For each service request, the *req* function will be called with a pointer to the argument structure which will have already been decoded using the XDR routine pointed to by *inproc*. After performing the requested service, the function pointed to by *req* returns the address of a buffer it has allocated which contains the return structure. The system then encodes this return structure using the XDR routine pointed to by *outproc*, transmits it back to the client and frees the pointer returned by *req*.

Return Value

The **registerrpc** function returns TRUE to indicate success and FALSE to indicate failure.

rpc_broadcast ()

Description

Broadcast remote procedure call to all servers on local network.

```
#include <windows.h>
#include <winsock.h>
#include <d32-rpc.h>
```

```
int WINAPI rpc_broadcast (prog, vers, proc, in_proc, in_arg, out_proc, out_arg, result, type)
```

unsigned long prog;	RPC program number
unsigned long vers;	Version of RPC program
unsigned long proc;	Procedure number requested
xdr_proc in_proc;	XDR encode procedure
void *in_arg;	Argument
xdr_proc out_proc;	XDR decode procedure
void *out_arg;	Return value
BOOL WINAPI (*result) (void *out, struct sockaddr_in *addr);	
char *type;	Only UDP allowed

Remarks

The **rpc_broadcast** function is similar to the **clnt_broadcast** function. Refer to **clnt_broadcast** for details. The *type* parameter must point to the character string "udp" to specify the UDP protocol.

Return Value

The **rpc_broadcast** function returns **RPC_SUCCESS** if successful or one of the error return values listed in section '[1.12 - RPC Error Return Values](#)'.

rpc_call ()**Description**

Issue remote procedure call to specified server using TCP or UDP protocol.

```
#include <windows.h>
```

```
#include <d32-rpc.h>
```

```
int WINAPI rpc_call (host, prog, vers, proc, in_proc, in_arg, out_proc, out_arg, type);
```

char * <i>host</i> ;	Name or internet address of server
unsigned long <i>prog</i> ;	RPC program number
unsigned long <i>vers</i> ;	Version number
unsigned long <i>proc</i> ;	Procedure number
xdr_proc <i>in_proc</i> ;	XDR encode procedure
void * <i>in_arg</i> ;	Argument
xdr_proc <i>out_proc</i> ;	XDR decode procedure
void * <i>out_arg</i> ;	Return value
char * <i>type</i> ;	TCP or UDP

Remarks

The **rpc_call** function issues a remote procedure call to the server specified by *host*. This can be either the name of the server or the address of the server in the dotted form. The three values given by *prog*, *vers* and *proc* uniquely identify the remote procedure to be executed. The XDR routine pointed to by *in_proc* is used to encode the arguments in the buffer *in_arg*. Upon successful completion of the call, the XDR routine pointed to by *out_proc* is used to decode the result into the buffer *out_arg*. It is the responsibility of the caller to ensure that the buffer pointed to by *out_arg* has been allocated large enough to hold the result.

The *type* parameter must point to the character string "tcp" or "udp" to specify whether the client requests the TCP or the UDP protocol.

Return Value

The **rpc_call** function returns `RPC_SUCCESS` if successful, or one of the error return values listed in section '1.12 - RPC Error Return Values'.

svc_destroy ()

Description

Signals that the specified service is to be destroyed and if possible free all resources allocated for given service.

```
#include <windows.h>
```

```
#include <d32-rpc.h>
```

```
void WINAPI svc_destroy (hSvc)
```

```
HSERVER hSvc;           Service structure handle
```

Remarks

The **svc_destroy** function destroys the service associated with the service structure handle *hSvc* which was returned by the **svctcp_create** or **svcudp_create** function. The service is removed from the local port mapper. If function **svc_run** or **svc_run_ex** is used for servicing the client requests, service is marked for deletion and the allocated resources are freed once it is determined that there are no pending client requests to be serviced. If function **svc_getreqset** is used for servicing the requests, the service is deleted and all the allocated resources are freed instantly. Hence, in this case, this function should be called only if no request is in progress. Any subsequent client requests for the service will fail. A server application must destroy all of its services by calling **svc_destroy** before it quits.

Return Value

Returns after successful deletion of the service.

svc_exit ()

Description

Signals that server is to be shut down.

```
#include <windows.h>
#include <d32-rpc.h>

void WINAPI svc_exit ()
```

Remarks

The **svc_exit** function destroys and unregisters with portmapper all the services registered by calling the function **svctcp_create** or **svcudp_create**. Notification is sent to the function **svc_run** or **svc_run_ex** to stop accepting any more requests from the clients and shut down after all the pending requests are serviced. This function, like **svc_destroy**, only sets the flag that the server is no longer available. Freeing of resources is done only after it is determined that there are no outstanding client requests to be serviced. This implies that the RPC server may be active even after this function returns. Though it will not accept any new requests. This function is equivalent to the function **svc_destroy** with the difference that it destroys all the registered services. If function **svc_getreqset** is used for servicing the requests than this function cannot be used. Use **svc_destroy** instead. A server application must call either **svc_exit** or **svc_destory** for proper cleanup before exiting.

Return Value

Returns immediately after marking shut down.

svc_fdset ()**Description**

Returns all the server's socket descriptor set created by **svc_XXX** create functions

```
#include <windows.h>
```

```
#include <d32-rpc.h>
```

```
void WINAPI svc_fdset (SvcRdSet)
```

```
FD_SET *SvcRdSet;           File descriptor set
```

Remarks

The **svc_fdset** function fills the caller provided *SvcRdSet* with all the service descriptors created by **svc_XXX** create functions. It is useful to applications doing their own asynchronous request processing.

Return Value

Returns after filling all the descriptors.

svc_freeargs ()

Description

Free argument buffer of remote service request.

```
#include <windows.h>
```

```
#include <d32-rpc.h>
```

```
BOOL WINAPI svc_freeargs (hSvc, proc, out)
```

```
HSERVER hSvc;           Service structure handle
```

```
xdr_proc proc;         XDR routine
```

```
void *out;             Argument buffer
```

Remarks

The **svc_freeargs** function calls the XDR routine specified by *proc* to free the memory buffer pointed to by *out*. This function should only be called from a service function after it has successfully decoded the arguments by calling **svc_getargs**.

Return Value

The **svc_freeargs** function returns TRUE if successful, or FALSE to indicate that the buffer could not be freed.

svc_getargs ()

Description

Get arguments for service supplied by remote client.

```
#include <windows.h>
```

```
#include <d32-rpc.h>
```

```
BOOL WINAPI svc_getargs (hSvc, proc, in)
```

```
HSERVER hSvc;           Service structure handle
```

```
xdr_proc proc;         XDR routine
```

```
void *in;              Argument buffer
```

Remarks

The **svc_getargs** function calls the XDR routine specified by *proc* to decode the call arguments into the buffer pointed to by *in*. It is the caller's responsibility to ensure that the buffer pointed to by *in* has been allocated large enough to hold the decoded arguments.

Return Value

The **svc_getargs** function returns TRUE if successful, or FALSE to indicate that the arguments could not be decoded.

svc_getcaller ()**Description**

Get internet address of current client.

```
#include <windows.h>
```

```
#include <winsock.h>
```

```
#include <d32-rpc.h>
```

```
SOCKADDR_IN WINAPI svc_getcaller (hSvc)
```

```
HSERVER hSvc;           Service structure handle
```

Remarks

The **svc_getcaller** function returns the internet address of the current client.

Return Value

The **svc_getcaller** function returns the internet address of the client which is currently being serviced.

svc_getreqset ()

Description

Handle the requests on the service descriptors described in the `FD_SET` passed.

```
#include <windows.h>
```

```
#include <d32-rpc.h>
```

```
void WINAPI svc_getreqset (SvcRdSet)
```

```
FD_SET *SvcRdSet;           File descriptor set
```

Remarks

The `svc_getreqset` function services the requests on the descriptors described by *SvcRdSet*. It is used by applications doing their own asynchronous request processing.

Return Value

Returns when the read bits on all the descriptors are cleared.

svc_register ()**Description**

Register service using UDP or TCP protocol with local port mapper.

```
#include <windows.h>
```

```
#include <d32-rpc.h>
```

```
BOOL WINAPI svc_register (hSvc, prog, vers, req, prot)
```

```
HSERVER hSvc;           Service structure handle
```

```
unsigned long prog;      RPC program number
```

```
unsigned long vers;     Version number
```

```
void WINAPI (*req) (svc_request *ptr, void *hSvc);
```

```
unsigned long prot;     Protocol
```

Remarks

The **svc_register** function registers the RPC service identified by the RPC program number *prog* and the version *vers* with the local port mapper. The service structure handle *hSvc* must have been created previously by calling either **svcudp_create** or **svctcp_create** to allocate the necessary resources for the UDP or the TCP transport protocol. The *prot* parameter must be set to IPPROTO_UDP or IPPROTO_TCP accordingly. If a service with the same *prog*, *vers* and *prot* values is already registered then **svc_register** will fail. The *req* parameter specifies the address of the function to be called whenever a service request arrives. The function pointed to by *req* must have the following prototype.

```
void WINAPI req (svc_request *ptr, void *hSvc);
```

For each service request, the *req* function will be called with a pointer to a service request structure. The service structure handle *hSvc* identifies the requested service. At any given time, more than one client may be serviced. The *req* function should call **svc_getargs** to decode the argument buffer and **svc_freeargs** to free the argument buffer. After completing the service request, the function should either call **svc_sendreply** to transmit the return buffer or one of the service error functions (**svcerr_noproc**, **svcerr_noprogram**, **svcerr_progvers**, or **svcerr_decode**) to report an error to the client. If the same service is registered for both the UDP and the TCP transport protocol, then the same service function *req* can be used. The *verify* field of the service request structure can be checked for client authentication parameters.

Return Value

The **svc_register** function returns TRUE if successful, or FALSE to indicate an error.

svc_run ()**Description**

Start accepting service requests from clients.

```
#include <windows.h>
```

```
#include <d32-rpc.h>
```

```
void WINAPI svc_run (void)
```

Remarks

The **svc_run** function enables the server to run so that it can start accepting client requests. By default, for each incoming request it creates a thread to handle it which exits after sending the results to the client. This function causes memory leak on Windows 95. To avoid the situation either use the function **svc_run_ex** or disable multi-threading using registry. **svc_run** keeps on accepting the requests until either function **svc_exit** is called or all the registered services are destroyed by calling function **svc_destroy**. Before quitting, it waits until all the pending requests are serviced. Then it free all the allocated resources and perform cleanup.

Return Value

The **svc_run** function returns after all the services are stopped and cleanup is performed.

svc_run_ex ()

Description

Start accepting service requests from clients.

```
#include <windows.h>
```

```
#include <d32-rpc.h>
```

```
void WINAPI svc_run_ex (NumOfThreads)
```

```
int prog;           Number of worker threads to be created
```

Remarks

The **svc_run_ex** function enables the server to run so that it can start accepting client requests. The parameter *NumOfThreads* specify the number of worker threads to create for servicing the requests, which can be modified by calling function **svc_update_threads** while server is running. Set this parameter to 0 to disable the creation of threads. In this case, function **svc_update_threads** is unavailable to change the number while server is running. **svc_run_ex** keeps on accepting the requests until either function **svc_exit** is called or all the registered services are destroyed by calling function **svc_destroy**. Before quitting, it waits until all the pending requests are serviced. Then it free all the allocated resources and perform cleanup.

Return Value

The **svc_run_ex** function returns after all the services are stopped and cleanup is performed.

svc_sendreply ()

Description

Transmit reply data structure to remote client.

```
#include <windows.h>
```

```
#include <d32-rpc.h>
```

```
BOOL WINAPI svc_sendreply (hSvc, proc, buf)
```

HSERVER <i>hSvc</i> ;	Service structure handle
xdr_proc <i>proc</i> ;	XDR encoding procedure
void * <i>buf</i> ;	Reply buffer

Remarks

The **svc_sendreply** function encodes the data in the buffer pointed to by *buf* by calling the XDR routine specified by *proc*. If the result buffer is successfully encoded then it is transmitted to the client.

Return Value

The **svc_sendreply** function returns TRUE if successful, or FALSE to indicate that the reply could not be encoded or transmitted.

svc_set_timeout ()

Description

Set the time server waits before it checks for the exit flag.

```
#include <windows.h>
```

```
#include <d32-rpc.h>
```

```
BOOL WINAPI svc_sendreply (TimeOut)
```

```
TIMEVAL TimeOut           New time out value
```

Remarks

The **svc_set_timeout** function set the time for which the RPC server waits before checking for the exit flag. Set this value to small time if server is taking too much time to shut down.

Return Value

The **svc_set_timeout** function returns after setting the requested timeout.

svc_unregister ()

Description

Remove entry for given service from local port mapper table.

```
#include <windows.h>
```

```
#include <d32-rpc.h>
```

```
BOOL WINAPI svc_unregister (prog, vers)
```

```
unsigned long prog;           RPC program number
```

```
unsigned long vers;          Version number
```

Remarks

The **svc_unregister** function removes all services with the RPC program number *prog* and the version *vers* from the local port mapper. The function will remove services for both the UDP and the TCP transport protocol.

Return Value

The **svc_unregister** function returns TRUE if at least one service was successfully removed, or FALSE to indicate that the service could not be removed or that the service did not exist.

svc_update_threads ()**Description**

Update the number of server's worker thread.

```
#include <windows.h>
```

```
#include <d32-rpc.h>
```

```
BOOL WINAPI svc_update_threads (NumOfThreads)
```

```
int NumOfThreads           Number of threads wanted.
```

Remarks

The **svc_update_threads** function creates or deletes the number of worker threads to have the specified number of threads, i.e. *NumOfThreads*. This function is available only if **svc_run_ex** is used to service the requests and it was not called with 0 threads..

Return Value

The **svc_update_threads** function returns TRUE on success.

svcerr_auth ()

Description

Return "Authentication error " to remote client.

```
#include <windows.h>
```

```
#include <d32-rpc.h>
```

```
void WINAPI svcerr_auth (hSvc, why)
```

```
HSERVER hSvc;           Service structure handle
```

```
unsigned int why;       Reason for rejecting authentication
```

Remarks

The **svcerr_auth** function is used to pass the "Authentication error " code back to the client instead of transmitting a result structure with **svc_sendreply**. This error indicates that the server has refused to perform the requested operation due to an authentication error.

The *why* parameter must be set to one of the following values.

AUTH_BAD_CRED	Bad credentials
AUTH_REJECTED_CRED	Client must begin a new session
AUTH_BAD_VERF	Bad verifier
AUTH_REJECTED_VERF	Verifier rejected
AUTH_TOO_WEAK	Rejected for security reasons

Return Value

The **svcerr_auth** function does not return a value.

svcerr_decode ()**Description**

Return "Unable to decode arguments" error to client.

```
#include <windows.h>
```

```
#include <d32-rpc.h>
```

```
void WINAPI svcerr_decode (hSvc)
```

```
HSERVER hSvc;           Service structure handle
```

Remarks

The **svcerr_decode** function is used to pass the "Unable to decode arguments" error code back to the client instead of transmitting a result structure with **svc_sendreply**. This error indicates an XDR decode problem or a low memory condition.

Return Value

The **svcerr_decode** function does not return a value.

svcerr_noproc ()

Description

Return "Procedure not available" error to client.

```
#include <windows.h>
```

```
#include <d32-rpc.h>
```

```
void WINAPI svcerr_noproc (hSvc)
```

```
HSERVER hSvc;           Service structure handle
```

Remarks

The **svcerr_noproc** function is used to pass the "Procedure not available" error code back to the client instead of transmitting a result structure with **svc_sendreply**. This error indicates that an invalid procedure was requested from a registered program.

Return Value

The **svcerr_noproc** does not return a value.

svcerr_noprog ()

Description

Return "Program not available" error to client.

```
#include <windows.h>
```

```
#include <d32-rpc.h>
```

```
void WINAPI svcerr_noprog (hSvc)
```

```
HSERVER hSvc;           Service structure handle
```

Remarks

The **svcerr_noprog** function is used to pass the "Program not available" error code back to the client instead of transmitting a result structure with **svc_sendreply**. This error indicates that an unknown program was requested (program not registered).

Return Value

The **svcerr_noprog** function does not return a value.

svcerr_progrvers ()

Description

Return "Program version not available" error to client.

```
#include <windows.h>
```

```
#include <d32-rpc.h>
```

```
void WINAPI svcerr_progrvers (hSvc)
```

```
HSERVER hSvc;           Service structure handle
```

Remarks

The **svcerr_progrvers** function is used to pass the "Program version not available" error code back to the client instead of transmitting a result structure with **svc_sendreply**. This error indicates that an unsupported version of a registered program was requested.

Return Value

The **svcerr_progrvers** function does not return a value.

svcerr_systemerr ()**Description**

Return "System error" to client.

```
#include <windows.h>
```

```
#include <d32-rpc.h>
```

```
void WINAPI svcerr_systemerr (hSvc)
```

```
HSERVER hSvc;           Service structure handle
```

Remarks

The **svcerr_systemerr** function is used to pass the "System error" code back to the client instead of transmitting a result structure with **svc_sendreply**. This error return indicates that a serious local error occurred on the server (for example, when the service can no longer allocate storage).

Return Value

The **svcerr_systemerr** function does not return a value.

svcerr_weakauth ()

Description

Return "Refuse to perform remote procedure" error to remote client.

```
#include <windows.h>
```

```
#include <d32-rpc.h>
```

```
void WINAPI svcerr_weakauth (hSvc)
```

```
HSERVER hSvc;           Service structure handle
```

Remarks

The **svcerr_weakauth** function is used to pass the "Refuse to perform remote procedure" error code back to the client instead of transmitting a result structure with **svc_sendreply**. This error indicates correct but insufficient authentication. This routine is equivalent to calling **svcerr_auth** with the AUTH_TOO_WEAK parameter.

Return Value

The **svcerr_weakauth** function does not return a value.

svctcp_create ()

Description

Allocate resources for RPC service using TCP transport protocol.

```
#include <windows.h>
```

```
#include <d32-rpc.h>
```

```
HSERVER WINAPI svctcp_create (sock, port, sendsz, recvsz)
```

int <i>sock</i> ;	Socket
int <i>port</i> ;	Port number
unsigned long <i>sendsz</i> ;	Send buffer size
unsigned long <i>recvsz</i> ;	Receive buffer size

Remarks

The **svctcp_create** function creates an RPC service structure and allocates the necessary resources for an RPC service over the TCP transport protocol. The new service can then be registered with the local port mapper by calling **svc_register**. If *sock* is set to `RPC_ANYSOCK`, then a new socket is created. If *sock* is set to a value other than `RPC_ANYSOCK`, then that value is expected to be a valid TCP socket which has been created with a call to the underlying Windows Sockets library. The socket will be bound to the local port specified by *port* (in host byte order). If *port* is set to 0, then a default value is used. A port number should only be specified if a service needs to listen on a specific port (for example, the port mapper). Normally, a client will first call the port mapper to find out on which port the service is running and the port is therefore arbitrary.

The maximum send and receive buffer sizes in bytes should be given in *sendsz* and *recvsz*. If *sendsz* or *recvsz* are set to 0, then the default values from registry are used.

The parameter *port* is maintained for compatibility only and may be removed in future versions. If a service needs to use a specific port then the application should create its own socket and bind it to the specific port and then call **svctcp_create** with that socket.

Return Value

The **svctcp_create** function returns the handle of the new service structure or NULL to indicate an error condition.

svctcp_create_secure ()

Description

Allocate resources for Secure RPC service using TCP transport protocol.

```
#include <windows.h>
```

```
#include <d32-rpc.h>
```

```
HSERVER WINAPI svctcp_create_secure (sock, port, sendsz, recvsz, netname, passwd)
```

int <i>sock</i> ;	Socket
int <i>port</i> ;	Port number
unsigned long <i>sendsz</i> ;	Send buffer size
unsigned long <i>recvsz</i> ;	Receive buffer size
char * <i>netname</i> ;	RPC server's network name
char * <i>passwd</i> ;	Password to decrypt server's secret key in database

Remarks

The **svctcp_create_secure** function is the same as the **svctcp_create** function, except that this function creates a **Secure RPC** service. So two more parameters are needed: *netname* is the RPC server's network name, and *passwd* is the password used to decrypt the server's secret key in the publickey database.

Warning

The **svctcp_create_secure** function requires the inclusion of a DLL library "libey32.dll" which is subject to US export restrictions for encryption software of its type. Please read and ensure compliance with the US export regulations on DES encryption before including this API call in your application. This library is available from Distinct. Please contact our technical support staff to obtain a copy. We will require your full name, company address and list of countries in which you intend to deploy your resulting RPC application. This feature comes with the Distinct ONC RPC/XDR Toolkit – ENC

Return Value

The **svctcp_create_secure** function returns the handle of the new service structure or NULL to indicate an error condition.

svculdp_bufcreate ()

Description

Allocate resources for RPC service with specified send and receive buffer sizes using UDP transport protocol.

```
#include <windows.h>
```

```
#include <d32-rpc.h>
```

HSERVER WINAPI svculdp_bufcreate (*sock*, *port*, *sendsz*, *recvsz*)

int <i>sock</i> ;	Socket
int <i>port</i> ;	Port number
unsigned long <i>sendsz</i> ;	Send buffer size
unsigned long <i>recvsz</i> ;	Receive buffer size

Remarks

The **svculdp_bufcreate** function creates an RPC service structure and allocates the necessary resources for an RPC service over the UDP transport protocol. The new service can then be registered with the local port mapper by calling **svc_register**. If *sock* is set to `RPC_ANYSOCK`, then a new socket is created. If *sock* is set to a value other than `RPC_ANYSOCK`, then that value is expected to be a valid UDP socket which has been created with a call to the underlying Windows Sockets library. The socket will be bound to the local port specified by *port* (in host byte order). If *port* is set to 0, then a default value is used. A port number should only be specified if a service needs to listen on a specific port (for example, the port mapper). Normally, a client will first call the port mapper to find out on which port the service is running and the port is therefore arbitrary.

The parameter *port* is maintained for compatibility only and may be removed in future versions. If a service needs to use a specific port then the application should create its own socket and bind it to the specific port and then call **svculdp_bufcreate** with that socket.

Return Value

The **svculdp_bufcreate** function returns the handle of the new service structure or NULL to indicate an error condition.

svcdp_create ()

Description

Allocate resources for RPC service with default send and receive buffer sizes using UDP transport protocol.

```
#include <windows.h>
```

```
#include <d32-rpc.h>
```

```
HSERVER WINAPI svcdp_create (sock, port)
```

```
int sock;                Socket
```

```
int port;                Port number
```

Remarks

The **svcdp_create** function creates an RPC service structure and allocates the necessary resources for an RPC service over the UDP transport protocol. The new service can then be registered with the local port mapper by calling **svc_register**. If *sock* is set to `RPC_ANYSOCK`, then a new socket is created. If *sock* is set to a value other than `RPC_ANYSOCK`, then that value is expected to be a valid UDP socket which has been created with a call to the underlying Windows Sockets library. The socket will be bound to the local port specified by *port* (in host byte order). If *port* is set to 0, then a default value is used. A port number should only be specified if a service needs to listen on a specific port (for example, the port mapper). Normally, a client will first call the port mapper to find out on which port the service is running and the port is therefore arbitrary.

The parameter *port* is maintained for compatibility only and may be removed in future versions. If a service needs to use a specific port then the application should create its own socket and bind it to the specific port and then call **svcdp_create** with that socket.

Refer to **callrpc** for a discussion on buffer sizes and the effect on data sizes.

Return Value

The **svcdp_create** function returns the handle of the new service structure or `NULL` to indicate an error condition.

svcadp_create_secure ()

Description

Allocate resources for secure RPC service with default send and receive buffer sizes using UDP transport protocol.

```
#include <windows.h>
```

```
#include <d32-rpc.h>
```

```
HSERVER WINAPI svcadp_create_secure (sock, port, netname, passwd)
```

int <i>sock</i> ;	Socket
int <i>port</i> ;	Port number
char * <i>netname</i> ;	RPC server's network name
char * <i>passwd</i> ;	Password to decrypt server's secret key in database

Remarks

The **svcadp_create_secure** function is same as **svcadp_create** function, except that this function creates an **Secure RPC** service. This requires two more parameters: *netname* is the RPC server's network name, and *passwd* is the password used to decrypt the server's secret key in the publickey database.

Warning

The **svcadp_create_secure** function requires the inclusion of a DLL library "libey32.dll" which is subject to US export restrictions for encryption software of its type. Please read and ensure compliance with the US export regulations on DES encryption before including this API call in your application. This library is available from Distinct. Please contact our technical support staff to obtain a copy. We will require you full name, company address and list of countries in which you intend to deploy your resulting RPC application.

Return Value

The **svcadp_create_secure** function returns the handle of the new service structure or NULL to indicate an error condition.

Database Functions

The database functions are used to get the entries from the RPC database. The RPC Database file is a text file called RPC which is present in the directory from which the D32-RPC.DLL was loaded.

All these functions use the following structure which is defined in **d32-rpc.h**.

```
struct rpcent {
    char *r_name;
    char **r_aliases;
    int r_number;
};
```

Warning: The *rpcent* structure returned by **getrpcbyname**, **getrpcbynumber** and **getrpcnt** is a static entry and therefore multiple threads must synchronize access to this element.

getrpcbyname ()

Description

Get the *rpcent* structure corresponding to name of a RPC program.

#include <windows.h>

#include <d32-rpc.h>

struct rpcent *getrpcbyname (*name*);
char *name; The name of the RPC Program.

Remarks

The **getrpcbyname** function retrieves *rpcent* structure for the RPC program for which the name is specified.

Return Value

The **getrpcbyname** returns a pointer to the *rpcent* structure if the name is found in the database. Otherwise it returns NULL.

getrpcbynumber ()

Description

Get the *rpcent* structure corresponding to number of a RPC program.

#include <windows.h>

#include <d32-rpc.h>

struct rpcent *getrpcbynumber (*number*);
int number; The number of the RPC Program.

Remarks

The **getrpcbynumber** function retrieves *rpcent* structure for the RPC program for which the number is specified.

Return Value

The **getrpcbynumber** returns a pointer to the *rpcent* structure if the number is found in the database. Otherwise it returns NULL.

getrpcnt ()

Description

Get the *rpcent* structure from the next line of the RPC database file opening the file, if necessary.

```
#include <windows.h>
#include <d32-rpc.h>

struct rpcent *getrpcent (void);
```

Remarks

The **getrpcent** function retrieves *rpcent* structure from the next line in the RPC database opening the file, if necessary.

Return Value

The **getrpcent** returns a pointer to the *rpcent* structure corresponding to the next available line. If there is an error or EOF is encountered it returns NULL.

setrpcent ()**Description**

Open and rewind the RPC database file.

```
#include <windows.h>
#include <d32-rpc.h>

void setrpcent (stayopen);
BOOL stayopen;          Close the file after each open ?.
```

Remarks

The **setrpcent** function opens and rewinds the RPC database file. If *stayopen* is TRUE then the file is not closed after each call to **getrpcent** or other calls.

Return Value

This function does not return a value.

endrpcent ()**Description**

Close the RPC database file.

```
#include <windows.h>
#include <d32-rpc.h>

void endrpcent (void);
```

Remarks

The **endrpcent** function closes the RPC database file which was opened using **setrpcent**.

Return Value

This function does not return a value.

Advanced Topics

Broadcast RPC

Two functions (**clnt_broadcast** and **rpc_broadcast**) provide RPC broadcast call. When an RPC broadcast is issued, a message is sent to all `rpcbind` daemons on the network. An `rpcbind` daemon with which the requested service is registered forwards the request to the server. The main differences between broadcast RPC and normal RPC calls are:

- Normal RPC expects one answer, broadcast RPC expects many answers (one or more answer from each responding machine).
- Broadcast RPC works only on UDP.
- With broadcast RPC, all unsuccessful responses are filtered out; so, if there is a version mismatch between the broadcaster and a remote service, the broadcaster never hears from the service.
- Only UDP services registered with `rpcbind` are accessible through broadcast RPC; service addresses may vary from one host to another, so **clnt_broadcast** or **rpc_broadcast** sends messages to `rpcbind`'s network address.
- The size of broadcast requests is limited by the maximum transfer unit (MTU) of the local network; the MTU for Ethernet is 1400 bytes.

The BROADCAST example demonstrates the use of **clnt_broadcast**. The example can be used as a utility to ask the network whether a service is available or not. The following is the source of *broadcast.c*:

```
#include <windows.h>
#include <d32-rpc.h>
#include <stdio.h>

/*
 * replyProc collects replies from the broadcast.
 */
static BOOL replyProc(PVOID res, SOCKADDR_IN *who)
{
    register struct hostent *hp;

    hp = gethostbyaddr((char *)&who->sin_addr, sizeof(who->sin_addr), AF_INET);
    printf("%s %s\n", inet_ntoa(who->sin_addr), (hp == NULL)?"(unknown)":hp->h_name);
    return (FALSE);
}

void main(int argc, char *argv[])
{
    int          rpc_stat;
    u_long       prognum, versnum, procnum;

    if (argc != 4) {
        fprintf(stderr, "Usage   : %s prognum versnum procnum\n", argv[0]);
        fprintf(stderr, "example : %s 100000 2 0\n", argv[0]);
        exit(1);
    }
    prognum = (u_long) atoi(argv[1]);
    versnum = (u_long) atoi(argv[2]);
    procnum = (u_long) atoi(argv[3]);
    /*
     * See if anybody is out there. Note procnum should expect no request
     * arguments and send nothing back or we'll have some en/decode problems,
     * potentially hosing the server. NULLPROC does this.
     */
    rpc_stat = clnt_broadcast(prognum, versnum, procnum, xdr_void, NULL,
                             xdr_void, NULL, replyProc);
    if (rpc_stat != RPC_SUCCESS && rpc_stat != RPC_TIMED_OUT) {
        fprintf(stderr, "%s: broadcast failed: %d\n", argv[0], rpc_stat);
    }
}
```

```
        exit(1);
    }
    exit(0);
}
```

In this example, **replyProc** responds to the reply by displaying the IP address and host name of the server that has responded. Since the function returns `FALSE` at all the times, it will continue to collect replies and the RPC client code will continue to resend the broadcast until it times out. If **replyProc** returns `TRUE`, the broadcasting stops, and **clnt_broadcast** returns successfully.

Batching

Batching allows a client to send a large sequence of messages to the server without waiting for the server to reply. The server does not send replies to batch calls. Batching uses TCP for its transport.

A sequence of batch calls is usually terminated by a legitimate remote procedure call operation in order to flush the pipeline and get positive acknowledgement.

Because the server does not respond to each call, the client can send new calls while the server is processing previous calls. This decreases interprocess communication overhead and the total time of a series of calls. The client should end with a non-batched call to flush the pipeline.

In our example BATCH, the batching server (batchsvc.exe) has three routines: RESET (set the sum to zero), ADDNUM (add a number to sum with no reply, this is a batching call), GETSUM (send sum to client). Batching client (batchcl.exe) sends numbers specified on command line to the batch server one by one and asks for the sum at the end.

Authentication

The RPC protocol provides the fields necessary for a client to identify itself to a service, and vice-versa, in each call and reply message. The Distinct ONC RPC/XDR toolkit currently supports the following authentication flavors:

AUTH_NONE	Null authentication.
AUTH_UNIX	An authentication flavor based on UNIX operation system, process permissions authentication.
AUTH_SHORT	An alternative flavor of AUTH_UNIX used by some servers for efficiency. Client programs using AUTH_UNIX authentication can receive AUTH_SHORT response verifiers from some servers.
AUTH_DES	An authentication flavor based on DES encryption techniques.

The default authentication flavor for RPC applications is AUTH_NONE, which is created by **authnone_create** by default. The UNIX authentication credential and the DES authentication credential are created by **authunix_create** by **authdes_create** respectively.

UNIX/SYS Authentication

UNIX authentication was used by most of the Sun's original network services. The credentials contain the client's machine-name, uid, gid, and group-access-list. The verifier is not required.

```

/*
 * AUTH_UNIX flavor credentials.
 */
struct authunix_parms {
    u_long      time;
    char *      machine;
    long        uid;
    long        gid;
    u_int len;
    long *      gids;
#define aup_time      time
#define aup_machname machine
#define aup_uid       uid
#define aup_gid       gid
#define aup_len       len
#define aup_gids      gids
}

```

The sample AUTH contains a pair of UNIX Authentication client and server, demonstrating how UNIX Authentication works. On client side, **authunix_create** is used to create the credential:

```
clnt->cl_auth = authunix_create (clienthost, uid, gid, gids_len, gids);
```

On server side, the credential can be checked in the procedure dispatch routine (and usually NULLPROC is passed by authentication check by default, as it is used to check whether the RPC service is available or not):

```

// authentication here if not NULLPROC
if (rqstp->proc != NULLPROC) {
    if (rqstp->cred.oa_flavor == AUTH_UNIX) {
        cred = (struct authunix_parms *) rqstp->clntcred;
        .....
        // check here
    } else {
        printf ("Client doesn't use AUTH_UNIX. exit.");
        svcerr_weakauth (transp);
        return;
    }
}
}

```

Setting up DES Authentication for Secure RPC

In order to use Secure RPC (AUTH_DES authentication) you must first obtain a copy of the libeay32.dll built for single DES support, which is available from Distinct with the -ENC version of this toolkit.

The "libeay32.dll" is subject to US export restrictions for encryption software of its type. Please read and ensure compliance with the US export regulations on DES encryption before including this API call in your application. This library is available from Distinct. Please contact our technical support staff to obtain a copy. We will require your full name, company address and list of countries in which you intend to deploy your resulting RPC application.

Specifically if you are located in a country or are shipping to a country or government to which we are able to export Distinct RPC BUT to which the export of DES encryption is prohibited, our license agreement prohibits you from using the Secure RPC feature of this toolkit.

Basic Steps Required to Build your AUTHDES Project

The Public Key Database - publickey

Secure RPC requires a database for Diffie Hellman public key pairs. In a UNIX system, there is a key server responding to public key related requests. On the Windows system a plain text database file called publickey must be created in the same folder where the RPC client resides. The publickey database must contain the public and private key pairs for users who will use the AUTHDES feature.

The format of the publickey file is the same as that of the file "/etc/publickey" on a Solaris system.

A typical entry in "publickey" is:

```
osname.userid@domain xxx:yyy
```

where:

osname.userid @domain is the name of user and system associated with the key. Note that in Windows the machine name is not required. However, if the key on the remote system includes this, then you must use the same entry here too.

xxx is the DH public key for the user *osname.userid@domain*

yyy is the DH secret key for the same user. The DH secret key is encrypted using a password known to the user only.

The **NEWKEY.EXE** application that comes with this toolkit is used to generate public key pairs for RPC applications. The generated key pairs need to be copied to the "publickey" file as shown in the entry above. Both the client and server must have exactly the same entry in the publickey file (on UNIX, this may be handled by NIS instead).

The following describes the step to build your application:

On the Client Side

To use AUTH_DES authentication, a client must set its authentication handle appropriately. For example:

```
clnt->cl_auth = authdes_create (clntname, passwd, srvname, window);
```

where "clntname" and "srvname" are the network names for the RPC client and server respectively. "passwd" is the password for RPC client's DH secret key. And "window" is the time length in seconds during which the authentication credential is valid.

The public key database "publickey" must be placed in the folder where the RPC client executable resides. There must be entries for both RPC client and RPC service in the database at the same time.

On the Server Side

To use AUTH_DES authentication, a server must use one or both of two functions for server side des authentication:

HSERVER WINAPI svctcp_create_secure (INT, SHORT, ULONG, ULONG, PCHAR, PCHAR);

HSERVER WINAPI svcudp_create_secure (INT, SHORT, PCHAR, PCHAR);

Which adds two more arguments to svctcp_create() and svcudp_create(). The first one is the network name of the RPC server, and the second is the password for RPC server's DH secret key.

In order to authenticate an RPC client, the server side public key database must contain an entry for RPC client.

If the server wants to report which client is connecting, the following can be added in the server's main callback routine (authdes_prog_1() in the sample):

```

if (rqstp->proc != NULLPROC) // pass by NULLPROC
{
    if (rqstp->cred.oa_flavor == AUTH_DES)
    {
        des_cred = (struct authdes_cred *) rqstp-
>clntcred;
        printf ("Client netname = %s\n", des_cred-
>adc_fullname.name);
    }
    else
    {
        printf ("Client doesn't use AUTH_DES. exit.");
        svcerr_weakauth (transp);
        return;
    }
}

```

Please refer to the AUTH sample for an example of what the files look like.

RPC Callback and RPC Transient Program Numbers

Sometime it can be useful to have a server become a client and make a Remote Procedure Call (RPC) back to the process client. For example, when the client submits a request to the server it would like the server to return immediately allowing the client to continue with other operations. Then when the server finishes processing the request it can send a callback to the client (for this type of programming both client and server libraries must be incorporated in the client application)

The sample `CALLBACK` illustrates how to let the server call back to the client. An RPC callback requires a program number to make the remote procedure call on. Since this will be a dynamically generated program number, it should be in the transient range, 0x40000000 to 0x5fffffff.

The `gettransient` routine returns a valid program number in transient range, and registers it with the port mapper. This routine only talks to the port mapper running on the same machine as the `gettransient` routine itself. The call to the `pmap_set` routine is a test-and-set operation. That is, it individually tests whether a program number has already been registered, and reserves the number if it has not. On return, the `sockp` argument contains a socket that can be used as the argument to an `svcudp_create` or `svctcp_create` routine.

The client makes a remote procedure call to the server, passing it a transient program number. Then the client waits around to receive a callback from the server at that program number. The server registers the `CALLBACKPROG` program so that it can receive the remote procedure call informing it of the callback program number. Then, when receiving the client call with the callback program number, the server sends a callback remote procedure call, using the program number it received earlier.

Multithreaded RPC Programming

The Distinct ONC RPC/XDR toolkit provides **sun_run_ex** in addition to **sun_run**. The main difference between these two functions is the multi-thread behavior when the RPC server receives a remote procedure call. If **sun_run** is used, the UDP server is running in single-thread mode, all UDP clients talk to this single-thread server; while TCP server creates a new thread for each TCP client, a specific TCP client talks to the specific TCP server thread. While UDP server and TCP server behave in the same way, if **sun_run_ex** is used. The RPC server first creates some threads waiting for clients' call, the number of threads is specified by **sun_run_ex**. These threads are always there to accept calls from clients; the calls can be from the same client or different clients.

On the client side, each thread should have its own HCLIENT object when using **clnt_call** to send remote procedure calls to the server. If multi-threading is required for one HCLIENT object, asynchronous RPC call functions (**rac_send**, **rac_poll**, **rac_recv**, **rac_drop**) can be used instead.

NOTE: It is very important to initialize the return variables (fields to zero, pointers to NULL, etc.) when multi-thread safe mode is used for generated RPC codes.

Asynchronous RPC Client Call

An asynchronous RPC client call is different from the **clnt_call**, which returns only when requested server replies; **rac_send** returns immediately after sending out the remote procedure call to the server. Later **rac_poll** can be used to check whether the reply has arrived or not. If reply comes back, **rac_recv** can be used to receive the reply. Also **rac_drop** can be used to cancel the remote procedure call.

The following is an example of an asynchronous call:

```
HASYNCRPC hAsync;
hAsync = rac_send(clnt, PROC, (xdr_proc)xdr_in, in, (xdr_proc)xdr_out, out, TIMEOUT);
if (hAsync == NULL) {
    printf("client call rac_send() failed.\n");
    exit(1);
}
// do anything here before check rpc call result
for (;;) { // checking for reply
    retval = rac_poll(clnt, hAsync);
    if (retval != RPC_CALL_INPROGRESS)
        break;
}
if (retval != RPC_SUCCESS) {
    printf("client call rac_poll() failed.\n");
} else {
    rac_recv(clnt, hAsync); // get the reply
    // deal with out value here.
}
```