# Legal Information

## Microsoft RPC Programmer's Guide and Reference

# About This Guide

This guide explains the Microsoft RPC programming model, standards, and tools in detail. Part I, the Overviews portion of the guide, consists of a sequence of topics that will help you understand distributed application programming and Microsoft RPC as follows:

Microsoft RPC Model provides an overview of the client-server programming model, standards for distributed application programming, and a description of how Microsoft RPC works.

Installing The RPC Programming Environment tells how to install the files and tools needed to develop distributed applications with Microsoft RPC.

Tutorial provides an overview of the development of a small distributed application. This example demonstrates all the steps in developing a distributed application, the tools you use, and the components that make up the executable programs.

The following topics deal with the underlying mechanisms that pass data from the calling application to the remote procedure.

Building RPC Applications describes the MIDL compiler and the necessary environment for building distributed applications with Microsoft RPC.

IDL and ACF Files describes the IDL and ACF files used to specify the interface to the remote procedure call and the MIDL compiler switches that control how these files are processed.

Data and Language Features demonstrates the use of standard data types.

Arrays and Pointers explains how to pass arrays pointers as parameters.

Binding and Handles describes the binding handle – the data structure that allows the developer to bind the calling application to the remote procedure.

Memory Management offers ideas about how to manage memory on the client and server when performing remote procedure calls.

Encoding Services describes the methods for encoding or decoding data.

Run-time RPC Functions describe the Microsoft RPC run-time libraries and the functions in the run-time library that applications use to manage their own client and server binding.

Security describes the methods for implementing security features in your distributed applications.

Installing and Configuring RPC Applications discusses installing your client and server applications in the MS-DOS, Microsoft Windows 3.x, Windows 95, and Windows NT environments, describes how to configure the name service provider and the security service. This section also contains network transport information for RPC.

Part II, The RPC Programmers' Reference provides alphabetical listings of RPC Data Types and Structures, and RPC Functions.

The Appendix contains a listing of RPC Error Codes and sample code that demonstrates a variety of RPC concepts.

For detailed information on using the MIDL Compiler, and for descriptions of MIDL language attributes, see the MIDL Programmers' Reference. For information about Microsoft Windows 3.x, see the Microsoft Windows 3.x software development kit. For information about the Microsoft Windows NT and Windows 95 operating systems, see the Microsoft Win32 software development kit. For information about C and C++,

see your C/C++ programming documentation.

# Microsoft RPC Model

Microsoft Remote Procedure Call (RPC) for the C and C++ programming languages is designed to help meet the needs of developers working on the next generation of software for the Microsoft® MS-DOS®, Microsoft® Windows®, Microsoft® Windows® 95, and Microsoft® Windows NT® family of operating systems.

Microsoft RPC represents the convergence of three powerful programming models: the familiar model of developing C applications by writing procedures and libraries; the model that uses powerful computers as network servers to perform specific tasks for their clients; and the client-server model, in which the client usually manages the user interface while the server handles data storage, queries, and manipulation.

This section explains the convergence of these three powerful models in distributed computing, which delivers the ability to share computational power among the computers on a network. This section also describes the industry standard for RPC and provides an overview of Microsoft RPC components and their operation.

## The Programming Model

In the early days of computing, each program was written as a large monolithic chunk, filled with **goto** statements. Each program had to manage its own input and output to different hardware devices. As the programming discipline matured, this monolithic code was organized into procedures, with the commonly used procedures packed in libraries for sharing and reuse. Today's RPC is the next step in the development of procedure libraries. Now, procedure libraries can run on other remote computers.

{ewc msdncd, EWGraphic, bsd23535 0 /a "SDK_A06.BMP"}

The C programming language supports procedure-oriented programming. In C, the main procedure relates to all other procedures as black boxes. For example, the main procedure cannot find out how procedures A, B, and X do their work. The main procedure only calls another procedure; it has no information about how that procedure is implemented.

{ewc msdncd, EWGraphic, bsd23535 1 /a "SDK_A08.BMP"}

Procedure-oriented programming languages provide simple mechanisms for specifying and writing procedures. For example, the ANSI standard C function prototype is a construct used to specify the name of a procedure, the type of the result it returns, if any, and the number, sequence, and type of its parameters. Using the function prototype is a formal way to specify an interface between procedures.

In this guide, the term *procedure* is synonymous with the terms *subroutine* and *subprocedure* and refers to any sequence of computer instructions that accomplishes a functional purpose. In this documentation, the term *function* refers to a procedure that returns a value.

Related procedures are often grouped in libraries. For example, a procedure library can include a set of procedures that performs tasks common to a single domain such as floating-point math operations, formatted input and output, and network functions.

The procedure library is another level of packaging that makes it easy to develop applications. Procedure libraries can be shared among many applications. Libraries developed in C are usually accompanied by header files. Each program that uses the library is compiled with the header files that formally define the interface to the library's procedures.

The Microsoft RPC tools represent a general approach in which procedure libraries written in C can run on other computers. In fact, an application can link with libraries implemented using RPC without indicating to the user that the application is using RPC.

## The Client-Server Model

Client-server architecture is an effective and popular design for distributed applications. In the client-server model, an application is split into two parts: a front-end client that presents information to the user, and a back-end server that stores, retrieves, and manipulates data, and generally handles the bulk of the computing tasks for the client. In this model, the server is usually a more powerful computer than the client and serves as a central data store for many client computers, thus making the system easy to administer.

Typical examples of client-server applications include shared databases, remote file servers, and remote printer servers. The following figure illustrates the client-server model.

{ewc msdncd, EWGraphic, bsd23535 2 /a "SDK_A04.BMP"}

Network systems support the development of client-server applications through an interprocess communication (IPC) facility in which the client and server can communicate and coordinate their work. You can use NetBIOS NCBs (network control blocks), mailslots, or named pipes to transfer information between two or more computers.

For example, the client can use an IPC mechanism to send an opcode and data to the server requesting that a particular procedure be called. The server receives and decodes the request and calls the appropriate procedure. The server then performs all the computations needed to satisfy the request and returns the result to the client. Client-server applications are usually designed to minimize the amount of data transmitted over the network.

Using NetBIOS, mailslots, or named pipes to implement interprocess communication means learning specific details relating to network communication. Each application must manage the network-specific conditions. To write this network-specific level of code, you must:

- Learn details relating to network communications and how to handle error conditions.
- Translate data to different internal formats, when the network includes different kinds of computers.
- Support communications using multiple transport interfaces.

In addition to all the possible errors that can occur on a single computer, the network has its own error conditions. For example, a connection can be lost, a server can disappear from the network, the network security service can deny access to system resources, or users can compete for and tie up system resources. Because the state of the network is always changing, an application can fail in new and interesting ways that are difficult to reproduce. For these reasons, each application must rigorously handle all possible error conditions.

When you write a client-server application, you must provide the layer of code that manages network communication. The advantage of using Microsoft RPC is that the RPC tools provide this layer for you. RPC virtually eliminates the need to write network-specific code, thus making it easier to develop distributed applications.

Using the remote procedure call model, RPC tools manage many of the details relating to network protocols and communication. This allows you to focus on the details of the application rather than the details of the network.

## The Compute-Server Model

Networking software for personal computers has been built on the model of a powerful computer – the server – that provides specialized services to workstations, or client computers. In this model, servers are designated as file servers, print servers, or communications (modem) servers, depending on whether they are assigned to file sharing or are connected to printers or modems.

RPC represents an evolutionary step in this model. In addition to its traditional roles, a server using RPC can be designated as a computational server or a compute server. In this role, the server shares its own computational power with other computers on the network. A workstation can ask the compute server to perform computations and return the results. The client not only uses files and printers, it also uses the central processing units of other computers.

## How RPC Works

The RPC tools make it appear to users as though a client directly calls a procedure located in a remote server program. The client and server each have their own address spaces; that is, each has its own memory resource that is allocated to data used by the procedure. The following figure illustrates the RPC architecture.

{ewc msdncd, EWGraphic, bsd23535 3 /a "SDK_A11.BMP"}

As the illustration shows, the client application calls a local stub procedure instead of the actual code implementing the procedure. Stubs are compiled and linked with the client application. Instead of containing the actual code that implements the remote procedure, the client stub code:

- Retrieves the required parameters from the client address space.
- Translates the parameters as needed into a standard network data representation (NDR) format for transmission over the network.
- Calls functions in the RPC client run-time library to send the request and its parameters to the server.

The server performs the following steps to call the remote procedure:

- The server RPC run-time library functions accept the request and call the server stub procedure.
- The server stub retrieves the parameters from the network buffer and converts them from the network transmission format to the format the server needs.
- The server stub calls the actual procedure on the server.

The remote procedure then runs, possibly generating output parameters and a return value. When the remote procedure is done, a similar sequence of steps returns the data to the client:

- The remote procedure returns its data to the server stub.
- The server stub converts output parameters to the format required for transmission over the network and returns them to the RPC run-time library functions.
- The server RPC run-time library functions transmit the data on the network to the client computer.

The client completes the process by accepting the data over the network and returning it to the calling function:

- The client RPC run-time library receives the remote-procedure return values and returns them to the client stub.
- The client stub converts the data from its network data representation to the format used by the client computer. The stub writes data into the client memory and returns the result to the calling program on the client.
- The calling procedure continues as if the procedure had been called on the same computer.

For Microsoft Windows 3.*x*, Windows 95, and Windows NT, the run-time libraries are provided in two parts: an import library, which is linked with the application and the RPC run-time library, which is implemented as a dynamic-link library (DLL).

The server application contains calls to the server run-time library functions which register the server's interface and allow the server to accept remote procedure calls. The server application also contains the application-specific remote procedures that are called by the client applications.

# OSF Standards for RPC

The design and technology behind Microsoft RPC is just one part of a complete environment for distributed computing defined by the Open Software Foundation (OSF), a consortium of companies formed to define that environment. The OSF requests proposals for standards, accepts comments on the proposals, votes on whether to accept the standards, and then promulgates them. The components of the OSF distributed computing environment (DCE) are shown in the following figure.

{ewc msdncd, EWGraphic, bsd23535 4 /a "SDK_A05.BMP"}

In selecting the RPC standard, the OSF cited the following rationale:

- The three most important properties of a remote procedure call are simplicity, transparency, and performance.
- The selected RPC model adheres to the local procedure model as closely as possible. This requirement minimizes the amount of time developers spend learning the new environment.
- The selected RPC model permits interoperability; its core protocol is well defined and cannot be modified by the user.
- The selected RPC model allows applications to remain independent of the transport and protocol on which they run, while supporting a variety of transports and protocols.
- The selected RPC model can be easily integrated with other components of the DCE.

The OSF-DCE remote procedure call standards define not only the overall approach, but the language and the specific protocols to use for communications between computers as well, down to the format of data as it is transmitted over the network.

Microsoft's implementation of RPC is compatible with the OSF standard with the exception of some minor differences. Client or server applications written using Microsoft RPC will interoperate with any DCE RPC client or server whose run-time libraries run over a supported protocol. For a list of supported protocols, see **Building RPC Applications**.

## Microsoft RPC Components

The Microsoft RPC product includes the following major components:

- MIDL compiler
- Run-time libraries and header files
- Transport interface modules
- Name service provider
- Endpoint supply service

In the RPC model, you can formally specify an interface to the remote procedures using a language designed for this purpose. This language is called the Interface Definition Language, or IDL. The Microsoft implementation of this language is called the Microsoft Interface Definition Language, or MIDL.

After you create an interface, you must pass it through the MIDL compiler. This compiler generates the stubs that translate local procedure calls into remote procedure calls. Stubs are placeholder functions that make the calls to the run-time library functions, which manage the remote procedure call. The advantage of this approach is that the network becomes almost completely transparent to your distributed application. Your client program calls what appear to be local procedures; the work of turning them into remote calls is done for you automatically. All the code that translates data, accesses the network, and retrieves results is generated for you by the MIDL compiler and is invisible to your application.

## Summary: RPC Extends Client-Server Computing

Microsoft RPC is an evolution of the procedural programming model familiar to all developers. It also represents a new category of specialized server and extends the model of client-server computing.

Microsoft RPC is a tool developers use to leverage the power of the single personal computer by increasing its computational capacity far beyond its own resources. With RPC, you can harness all the CPU horsepower available on the network.

Microsoft RPC allows a process running in one address space to make a procedure call that is executed in another address space. The call looks like a standard local procedure call but is actually made to a stub that interacts with the run-time library and performs all the steps necessary to execute the call in the remote address space.

As a tool for creating distributed applications, Microsoft RPC provides the following benefits:

- The RPC programming model is already familiar. You can easily turn functions into remote procedures, which simplify the development and test cycles.
- RPC hides many details of the network interface from the developer. You do not have to understand specific network functions or low-level network protocols to implement powerful distributed applications.
- RPC solves the data-translation problems that crop up in heterogeneous networks; individual applications can ignore this problem.
- The RPC approach is scalable. As a network grows, applications can be distributed to more than one computer on the network.
- The RPC model is an industry standard. The Microsoft implementation is compatible with both client and server.

# Installing the RPC Programming Environment

You develop RPC distributed applications, for all supported platforms, on the 32-bit Windows NT platform. Although the 16-bit MIDL compiler is no longer supported, you can develop 16-bit code by doing the following:

1. Use the 32-bit MIDL compiler installed as part of Win32 SDK Setup.
2. Select the MS-DOS or Windows 3.*x* option of the MIDL **/env** command line switch.
3. Compile your MS-DOS or Windows 3.*x* application and RPC stubs using your 16-bit development environment.

When the Win32 SDK is installed, the RPC development environment and the run-time libraries are automatically installed. For the 32-bit Windows platform, no additional installation is required.

**Note** See **Building RPC Applications** for information about various build environments.

## Developing 32-bit Windows Applications

The Microsoft Win32 SDK contains the Microsoft® Windows NT® and Microsoft® Windows® 95 APIs. When you install the Win32® SDK, you install the following RPC tools and files:

- C/C++ language header (.H) files for the RPC run-time libraries and run-time library(.LIB and .DLL) files for 32-bit Windows platforms
- 32-bit sample programs
- RPC reference Help files
- The **uuidgen** utility

When you install Windows NT or Windows 95, you install the following:

- RPC Run-time .DLLs
- RPC Locator (NT only) and RPC Endpoint-mapping services

## Developing 16-bit Windows and MS-DOS Client Applications

To develop client-side distributed applications for MS-DOS and Microsoft Windows 3.*x*, you must install the Microsoft Windows 3.*x*/MS-DOS version of the RPC SDK, which is contained in a disk image in the Win32 SDK directory \rpc_sdk. Run SETUP.EXE from Disk 1 to install the header and library files for MS-DOS and/or 16-bit Windows.

When you install the 16-bit SDK, you install the following:

- Header files and libraries needed to build RPC applications for MS-DOS and Windows 3.*x*.
- Sample RPC programs for MS-DOS and Windows 3.*x*.
- Run-time RPC and .DLL files for MS-DOS and Windows 3.*x*.
- 32-bit MIDL compiler.

Microsoft RPC development for 16-bit Windows and MS-DOS requires:

- Microsoft Visual C++ or other C/C++ compiler.
- One of the following platforms, for testing your 16-bit application:
  - Microsoft Windows version 3.*x* with Microsoft LAN Manager version 2.2
  - Microsoft Windows 3.*x* with a Windows Sockets-compliant TCP/IP stack
  - Microsoft Windows 3.*x* with Workgroup Connection 3.1
  - Microsoft Windows for Workgroups 3.11 with NetBEUI or the Microsoft TCP/IP-32 stack
  - Microsoft Windows 3.*x* with NetWare 3.*x* or 4.*x*
  - Microsoft Windows for Workgroups 3.11 with NetWare 3.*x* or 4.*x* software

## Developing Macintosh Client Applications

### To develop client-side applications for the Macintosh, you must have the following:

- Visual C++ for the Macintosh. The RPC runtime has been compiled using Visual C++ cross-development tools. In order to use rpc.lib, you must link against the C run-time and swapper library (swap.lib) provided with Visual C++, version 2.0 or later.
- The Macintosh RPC SDK, which is contained in a disk image in the Win32 SDK directory \rpc_sdk. Run SETUP.EXE from Disk 1 to install the Macintosh header and library files. Note that the current rpc.lib is native 68K. We currently do not provide a native Power Mac library. RPC runs in emulation on Power Macs.
- The target computer must have a microprocessor of 68020 or later, and it must be running System 7.0 or later.

### To connect to the Windows NT or Windows 95 server

- Current Windows NT-supported protocols for the Macintosh are ADSP and TCP/IP. In order to use ADSP, the Windows NT server must have both the AppleTalk protocol and Services for Macintosh. Windows 95 supports only the TCP/IP protocol for the Macintosh.

### To write an RPC client

1. If you use **atexit** to perform cleanup during shutdown, do not call any RPC APIs in your exit processing function.
2. If a yielding function is not registered, an RPC will not yield on the Macintosh. Register a yielding function by calling **RpcMacSetYieldInfo**.

```
void RPC_ENTRY MacCallbackFunc(short *pStatus)
{
  MSG msg;
  while (*pStatus == 1)
  {
    if(PeekMessage(&msg,NULL,0,0,PM_REMOVE))
    {
    TranslateMessage(&msg);
    DispatchMessage(&msg);
    }
  }
}
```

3. Most client-side APIs that are supported by Windows 3.*x* are also supported by the Macintosh. The Macintosh does not support the following APIs:
   - **RpcNs*** APIs
   - **RpcMgmt*** APIs
   - **RpcWinSetYieldInfo** (replaced by **RpcMacSetYieldInfo**)

The only authentication service currently supported for the Macintosh is RPC_C_AUTHN_WINNT.

The following protocol sequences are supported:

- ADSP:ncacn_at_dsp
- TCP:ncacn_ip_tcp

# Tutorial

This tutorial takes you through the steps required to create a simple, single-client, single-server distributed application from an existing stand-alone application. These steps are:

- Create interface definition and application configuration files.
- Use the MIDL compiler to generate C-language client and server stubs and headers from those files.
- Write a client application that manages its connection to the server.
- Write a server application that contains the actual remote procedures.
- Compile and link these files to the RPC run-time library to produce the distributed application.

The client application passes a character string to the server in a remote procedure call and the server prints the string ("Hello, World") to its standard output.

The complete source files for this example application, with additional code to handle command-line input and to output various status messages to the user, are in the Win32 SDK directory \mstools\samples\rpc\hello and in the Code Samples, RPC section of the Win32 SDK documentation.

## The Stand-alone Application

This stand-alone application, which consists of a call to a single function, forms the basis of our distributed application. The function, HelloProc, is defined in its own source file so that it can be compiled and linked with either a stand-alone application or a distributed application.

```
/* file hellop.c */
#include <stdio.h>
void HelloProc(unsigned char * pszString)
{
    printf("%s\n", pszString);
}

/* file: hello.c, a stand-alone application */
#include "hellop.c"
void main(void)
{
    unsigned Char * pszString = "Hello, World";
    HelloProc(pszString);
}
```

## Defining the Interface

The interface definition is a formal specification for how the client application and the server application communicate with each other. The interface defines how the client and server "recognize" each other, the remote procedures that the client application can call, the data types for those procedures' parameters and return values, and how the data is transmitted between client and server.

You define this interface in the Microsoft Interface Definition Language (MIDL) which consists of C-language definitions augmented with keywords, called attributes, which describe how the data is transmitted over the network.

The interface definition (.IDL) file contains type definitions, attributes and function prototypes that describe how data is transmitted on the network. The application configuration (.ACF) file contains attributes that configure your application for a particular operating environment without affecting its network characteristics.

## Generating the UUID

The first step in defining the interface is to use the **uuidgen** utility to generate a universally unique identifier (UUID) that lets the client and server applications recognize each other.

The **uuidgen** utility (UUIDGEN.EXE) is automatically installed when you install the Win32 SDK. The following command generates a UUID and creates a template file called hello.idl:

```
C:\>uuidgen /i /ohello.idl
```

Your hello.idl template will look like this (with a different UUID, of course):

```
[
uuid(7a98c250-6808-11cf-b73b-00aa00b677a7),
version(1.0)
]
interface INTERFACENAME
{

}
```

## The IDL File

The IDL file consists of one or more interface definitions, each of which has a header and a body. The header contains information that applies to the entire interface, such as the UUID. This information is enclosed in square brackets and is followed by the keyword **interface** and the interface name. The body contains C-style data type definitions and function prototypes, augmented with attributes that describe how the data is transmitted over the network.

In our example, the interface header contains only the UUID and the version number. The version number ensures that, when there are multiple versions of an RPC interface, only compatible versions of the client and server will be connected.

The interface body contains the function prototype for **HelloProc**. The function parameter *pszString* has the attributes **in** and **string**. The **in** attribute tells the run-time library that the parameter is passed only from the client to the server. The **string** attribute specifies that the stub should treat the parameter as a C-style character string.

We want the client application to be able to shut down the server application, so we add a prototype for another remote function, **Shutdown**, that we will implement later in this tutorial.

```
//file hello.idl
[
uuid(7a98c250-6808-11cf-b73b-00aa00b677a7),
version(1.0)
]
interface hello
{
void HelloProc([in, string] unsigned char * pszString);
void Shutdown(void);
}
```

## The ACF File

The ACF file allows you to customize your client and/or server applications' RPC interface without affecting the network characteristics of the interface. For example, if your client application contains a complex data structure that only has meaning on the local machine, you can specify in the ACF file how the data in that structure can be represented in a machine-independent form for remote procedure calls.

Our example demonstrates another use of the ACF file – to specify the type of binding handle that represents the connection between client and server. The **implicit_handle** attribute in the ACF header allows the client application to select a server for its remote procedure call. We have defined the handle to be of the type **handle_t** (a MIDL primitive data type). The binding handle name that we specified, *hello_IfHandle,* will be defined in the MIDL-generated header file. This handle will be used in calls to the client run-time library function.

Notice that this particular ACF file has an empty body.

```
//file: hello.acf
[implicit_handle (handle_t hello_IfHandle)
] interface hello
{
}
```

The MIDL compiler has an option, **/app_config**, that lets you include certain ACF attributes, such as **implicit_handle**, in the IDL file, rather than creating a separate ACF file. Consider using this option if your application doesn't require a lot of special configuration, and if strict OSF compatibility is not an issue.

## Generating the Stub Files

After defining the client/server interface, you usually develop your client and server source files, and then use a single makefile to generate the stub and header files and compile and link the client and server applications. However, if this is your first exposure to the distributed computing environment, you may want to invoke the MIDL compiler now to see what MIDL generates before you continue.

The MIDL compiler (MIDL.EXE) is automatically installed when you install the Win32 SDK. Make sure that hello.idl and hello.acf are in the same directory. The following command will generate the header file hello.h, and the client and server stubs, hello_c.c and hello_s.c:

```
C:\> midl hello.idl
```

Notice that hello.h contains function prototypes for HelloProc and Shutdown, as well as forward declarations for two functions, MIDL_user_allocate and MIDL_user_free. You will provide those two memory management functions in the server application. If data were also being transmitted from the server to the client (via an **out** parameter) you would also need to provide these two memory management functions in the client application.

Also note the definitions for our global handle variable, *hello_IfHandle*, and the client and server interface handle names, *hello_v1_0_c_ifspec* and *hello_v1_0_s_ifspec*, which the client and server applications will use in run-time calls.

You don't need to do anything with the stub files hello_c.c and hello_s.c.

```
/*file: hello.h */
/* this ALWAYS GENERATED file contains the definitions for the interfaces */
/* File created by MIDL compiler version 3.00.06
/* at Tue Feb 20 11:33:32 1996 */
/* Compiler settings for hello.idl:
    Os, W1, Zp8, env=Win32, ms_ext, c_ext
    error checks: none */
//@@MIDL_FILE_HEADING(  )
#include "rpc.h"
#include "rpcndr.h"

#ifndef __hello_h_
#define __hello_h_

#ifdef __cplusplus
extern "C"{
#endif

/* Forward Declarations */

void __RPC_FAR * __RPC_USER MIDL_user_allocate(size_t);
void __RPC_USER MIDL_user_free( void __RPC_FAR * );

#ifndef __hello_INTERFACE_DEFINED__
#define __hello_INTERFACE_DEFINED__

/****************************************
 * Generated header for interface: hello
 * at Tue Feb 20 11:33:32 1996
 * using MIDL 3.00.06
```

```
 ************************************/
/* [implicit_handle][version][uuid] */

            /* size is 0 */
void HelloProc(
    /* [string][in] */ unsigned char __RPC_FAR *pszString);
            /* size is 0 */
void Shutdown( void);
extern handle_t hello_IfHandle;

extern RPC_IF_HANDLE hello_v1_0_c_ifspec;
extern RPC_IF_HANDLE hello_v1_0_s_ifspec;
#endif /* __hello_INTERFACE_DEFINED__ */

/* Additional Prototypes for ALL interfaces */
/* end of Additional Prototypes */
#ifdef __cplusplus
}
#endif
#endif
```

## The Client Application

The helloc.c source file contains a directive to include the MIDL-generated header file, hello.h. Within hello.h are directives to include rpc.h and rpcndr.h, which contain the definitions for the RPC runtime routines and data types that our client and server applications use.

Because the client is managing its connection to the server, the client application calls run-time functions to establish a handle to the server and to release this handle after the remote procedure calls are complete. The function **RpcStringBindingCompose** combines the components of the binding handle into a string representation of that handle and allocates memory for the string binding. The function **RpcBindingFromStringBinding** creates a server binding handle, *hello_IfHandle*, for the client application from that string representation.

In the call to **RpcStringBindingCompose**, we have not specified the UUID because we have just one implementation of the interface "hello". We also have not specified a network address because we want the default, which is the local host machine. The protocol sequence is a character string that represents the underlying network transport and the endpoint is a name that is specific to the protocol sequence. We are using named pipes (a native Windows NT protocol) for our network transport, so the protocol sequence is "ncacn_np" and we have named our endpoint "\pipe\hello".

The actual remote procedure calls, **HelloProc** and **Shutdown**, take place within the RPC exception handler – a set of macros that let you control exceptions that occur outside the application code. If the RPC runtime module reports an exception, control passes to the **RpcExcept** block, which is where you would insert code to do any needed cleanup and then exit gracefully. In our example, we need only inform the user that an exception ocurred.

After the remote procedure calls are completed the client first calls **RpcStringFree** to free the memory that was allocated for the string binding, and then calls **RpcBindingFree** to release the handle.

```
/* file: helloc.c */
#include <stdlib.h>
#include <stdio.h>
#include <ctype.h>
#include "hello.h"

void main()
{
    RPC_STATUS status;
    unsigned char * pszUuid             = NULL;
    unsigned char * pszProtocolSequence = "ncacn_np";
    unsigned char * pszNetworkAddress   = NULL;
    unsigned char * pszEndpoint    = "\\pipe\\hello";
    unsigned char * pszOptions          = NULL;
    unsigned char * pszStringBinding    = NULL;
    unsigned char * pszString      = "hello, world";
    unsigned long ulCode;

    status = RpcStringBindingCompose(pszUuid,
                                     pszProtocolSequence,
                                     pszNetworkAddress,
                                     pszEndpoint,
                                     pszOptions,
                                     &pszStringBinding);
    if (status) {
        exit(status);
```

```
    }
    status = RpcBindingFromStringBinding(pszStringBinding,
                                         &hello_IfHandle);

    if (status) {
        exit(status);
    }


    RpcTryExcept {
        HelloProc(pszString);
        Shutdown();
    }
    RpcExcept(1) {
        ulCode = RpcExceptionCode();
        printf("Runtime reported exception 0x%lx = %ld\n", ulCode, ulCode);
    }
    RpcEndExcept

    status = RpcStringFree(&pszStringBinding);

    if (status) {
        exit(status);
    }

    status = RpcBindingFree(&hello_IfHandle);

    if (status) {
        exit(status);
    }

    exit(0);

}  // end main()
```

## The Server Application

The server side of the distributed application informs the system that its services are available and then waits for client requests.

Depending on the size of your application and your coding preferences, you can choose to implement remote procedures in one or more separate files. In our example, the main server routine is in the source file hellos.c, and the remote procedure remains in the file hellop.c, which we created for the stand-alone program.

The benefit of organizing the remote procedures in separate files is that the procedures can be linked with a stand-alone program to debug the code before it is converted to a distributed application. After the program works as a stand-alone program, you can compile and link the remote-procedure source files with the server application.

As with the client-application source file, the server-application source file must include the hello.h header file to obtain definitions for the RPC data and functions and for the interface-specific data and functions.

The server calls the RPC runtime functions **RpcServerUseProtseqEp** and **RpcServerRegisterIf** to make binding information available to the client. Since we have only one implentation of our remote procedures, we only pass the interface handle name to **RpcServerRegisterIf**. The other parameters are set to NULL. The server then calls the **RpcServerListen** function to indicate that it is waiting for client requests.

The server application must also include the two memory management functions that the server stub calls – **midl_user_allocate** and **midl_user_free**. These functions allocate and free memory on the server when a remote procedure passes parameters to the server. In our example, **midl_user_allocate** and **midl_user_free** are simply wrappers for the C-library functions **malloc** and **free**. (Note that, in the MIDL compiler- generated forward declarations, "midl" is uppercase. The header file rpcndr.h defines midl_user_free and midl_user_allocate to be MIDL_user_free and MIDL_user_allocate, respectively.)

```
/* file: hellos.c */
#include <stdlib.h>
#include <stdio.h>
#include <ctype.h>
#include "hello.h"

void main()
{
    RPC_STATUS status;
    unsigned char * pszProtocolSequence = "ncacn_np";
    unsigned char * pszSecurity     = NULL; /*Security not implemented */
    unsigned char * pszEndpoint    = "\\pipe\\hello";
    unsigned int    cMinCalls           = 1;
    unsigned int    cMaxCalls           = 20;
    unsigned int    fDontWait           = FALSE;

    status = RpcServerUseProtseqEp(pszProtocolSequence,
                                   cMaxCalls,
                                   pszEndpoint,
                                   pszSecurity);

    if (status) {
        exit(status);
    }
```

```
    status = RpcServerRegisterIf(hello_v1_0_s_ifspec,
                                 NULL,
                                 NULL);

    if (status) {
        exit(status);
    }

    status = RpcServerListen(cMinCalls,
                             cMaxCalls,
                             fDontWait);

    if (status) {
        exit(status);
    }

 }  // end main()

/****************************************************/
/*         MIDL allocate and free                   */
/****************************************************/

void __RPC_FAR * __RPC_USER midl_user_allocate(size_t len)
{
    return(malloc(len));
}

void __RPC_USER midl_user_free(void __RPC_FAR * ptr)
{
    free(ptr);
}
```

## Stopping the Server Application

A robust server application should stop listening for clients, and clean up after itself before shutting down. The two core server functions that accomplish this are **RpcMgmtStopServerListening** and **RpcServerUnregisterIf**.

The server function **RpcServerListen** doesn't return to the calling program until an exception occurs, or until a call to **RpcMgmtStopServerListening** occurs. By default, only another server thread is allowed to halt the RPC server by using **RpcMgmtStopServerListening**. Clients who try to halt the server will receive the error RPC_S_ACCESS_DENIED. However, it is possible to configure RPC to allow some or all clients to stop the server. See the reference page for **RpcMgmtStopServerListening** for details.

You can also have the client application make a remote procedure call to a shutdown routine on the server. The shutdown routine calls **RpcMgmtStopServerListening** and **RpcServerUnregisterIf**. Our example application uses this approach by adding a new remote function, **Shutdown**, to the file hellop.c.

In the Shutdown function, the single null parameter to **RpcMgmtStopServerListening** indicates that the local application should stop listening for remote procedure calls. The two null parameters to **RpcServerUnregisterIf** are wildcards, indicating that all interfaces should be unregistered. The FALSE parameter indicates that the interface should be removed from the registry immediately, rather than waiting for pending calls to complete.

```
/* add this function to hellop.c */
void Shutdown(void)
{
    RPC_STATUS status;

    status = RpcMgmtStopServerListening(NULL);

    if (status) {
       exit(status);
    }

    status = RpcServerUnregisterIf(NULL, NULL, FALSE);

    if (status) {
       exit(status);
    }
} //end Shutdown
```

## Compiling and Linking

The following makefile shows the dependencies among the files needed to compile the client and server applications and link them to the RPC runtime library and the standard C run-time library.

This makefile was used to build client and server applications from the source code in this tutorial. The stubs and headers were generated with MIDL version 2.0. The build environment was Microsoft Visual C++ 4.0, running on Windows NT 3.51. The compiler and linker commands and arguments may be different for your computer configuration. See your compiler documentation for more information.

```
#makefile for helloc.exe and hellos.exe
#link refers to the linker
#conflags refers to flags for console applications
#conlibs refers to libraries for console applications

!include <ntwin32.mak>

all : helloc hellos

# Make the client side application helloc
helloc : helloc.exe
helloc.exe : helloc.obj hello_c.obj
    $(link) $(linkdebug) $(conflags) -out:helloc.exe \
        helloc.obj hello_c.obj \
        rpcrt4.lib $(conlibs)

# helloc main program
helloc.obj : helloc.c hello.h
    $(cc) $(cdebug) $(cflags) $(cvars) $*.c

# helloc stub
hello_c.obj : hello_c.c hello.h
    $(cc) $(cdebug) $(cflags) $(cvars) $*.c

# Make the server side application
hellos : hellos.exe
hellos.exe : hellos.obj hellop.obj hello_s.obj
    $(link) $(linkdebug) $(conflags) -out:hellos.exe \
        hellos.obj hello_s.obj hellop.obj \
        rpcrt4.lib $(conlibsmt)

# hello server main program
hellos.obj : hellos.c hello.h
    $(cc) $(cdebug) $(cflags) $(cvarsmt) $*.c

# remote procedures
hellop.obj : hellop.c hello.h
    $(cc) $(cdebug) $(cflags) $(cvarsmt) $*.c

# hellos stub file
hello_s.obj : hello_s.c hello.h
    $(cc) $(cdebug) $(cflags) $(cvarsmt) $*.c

# Stubs and header file from the IDL file
```

```
hello.h hello_c.c hello_s.c : hello.idl hello.acf
    midl hello.idl
```

## Running the Application

To run the application on a single Windows NT machine, open two console windows. In the first window, type

```
C:\> hellos
```

and in the second window, type

```
C:\> helloc
```

Because our distributed application uses named pipes as the transport protocol, the server-side application will not run on Windows 95. To experiment with different protocol sequences, endpoints, and other options, build the sample hello application from the source files in \mstools\samples\rpc\hello on the Win32 SDK CD.

# Building RPC Applications

The procedure for building a distributed RPC application varies slightly, depending on the operating-system platform you are developing on and the target platform, the version of the MIDL and C or C++ compiler, and the API libraries you use. However, the basic procedure is the same in all cases: develop the MIDL and C source files, compile the MIDL source files, compile the C source files, and then link with the RPC and other API libraries.

## Environment, Compiler, and API Set Choices

You can develop RPC applications for different target environments: MS-DOS, Microsoft® Windows 3.*x*, Windows® 95, and Windows NT®. You can also choose to develop the executable applications for these target environments using different build environments. Accordingly, you can choose among several development environments, MIDL and C compilers, and API sets.

Available tools and libraries are described in the following table:

| Development tool | Description |
|---|---|
| MIDL 3.0 for 32-bit environment | Produces C source code for 16-bit or 32-bit environment. |
| C and MSVC for 16-bit environment | Produces 16-bit object files only. |
| C and MSVC for 32-bit environment (Win32 SDK) | Produces 32-bit object files only. |
| Win32 API | Provided for 32-bit environment only (RPC functions are provided as 32-bit DLLs). |
| Windows 3.*x* API | Provided for 16-bit environment only (RPC functions are provided as 16-bit Windows DLLs). |

## General Build Procedure

Use the following procedure to develop your distributed application:

1. Install the RPC SDK for your platform. For more information on how to install RPC, see Installing the RPC Programming Environment.
2. Develop the IDL file (and optional ACF) that specifies the interface.
3. Develop the C-language source files that implement and call the interface.
4. Generate C-language stub files by compiling the IDL file and optional ACF with the MIDL compiler.
5. Compile the C-language source and stub files with the C compiler.
6. Link the object files with the RPC import libraries for your platform.
7. Run the client and server distributed applications.

## Developing IDL Files

This section includes the following topics:

- A description of the **uuidgen** utility.
- A discussion on importing system header files.
- A discussion on importing other IDL files.

## The uuidgen Utility

The UUID is assigned to an interface to distinguish that interface from other interfaces. The UUID is generated from a command-line utility program, **uuidgen**, which creates unique identifiers in the required format using both a time identifier and a machine identifier. It guarantees that any two UUIDs produced on the same machine are unique because they are produced at different times, and that any two UUIDs produced at the same time are unique because they are produced on different machines. The **uuidgen** utility generates the UUID in IDL file format or C-language format.

The textual representation of a UUID is a string consisting of 8 hexadecimal digits followed by a hyphen, followed by three hyphen-separated groups of 4 hexadecimal digits, followed by a hyphen, followed by 12 hexadecimal digits. The following example is a valid UUID string:

```
6B29FC40-CA47-1067-B31D-00DD010662DA
```

When you run the **uuidgen** utility from the command line, you can use the following command switches:

| **uuidgen switch** | **Description** |
|---|---|
| **/i** | Outputs UUID to an IDL interface template. |
| **/s** | Outputs UUID as an initialized C structure. |
| **/o**<*filename*> | Redirects output to a file; specified immediately after the **/o** switch. |
| **/n**<*number*> | Specifies the number of UUIDs to generate. |
| **/v** | Displays version information about **uuidgen**. |
| **/h or ?** | Displays command-option summary. |

## Importing System Header Files

While it is often possible to use the **#include** directive to include header files in your IDL file, it is not recommended that you do so. The MIDL compiler will generate stubs for all functions defined in the .IDL file being compiled. Usually a header file contains a number of prototypes that you neither need nor want to include in your stub files, and a **#include** effectively puts all those definitions into your main IDL file. Furthermore, if there are nonremotable types defined in the header file, your IDL file may not compile. There are two ways to include typedefs from header files in your IDL file:

- Use the **import** directive to include data types defined in a header file. Unlike the C-language **#include** directive, the **import** directive only picks up type and constant definitions and ignores procedure prototypes. This approach will work as long as your main .IDL file does not reference any nonremotable types defined in the header file.
- Create a helper .IDL file with a dummy interface that includes the header files. Then, use the **import** directive to include the helper file. In this way, only the typedefs will appear in the compiled stubs. For example:

```
//in helper.IDL:
interface dummy
{ #include "kitchensink.h"
  #include "system.h"
}
//in main.IDL:
import "helper.IDL";
```

## Importing Other IDL Files

When you import IDL files using the **import** attribute, you reuse software. You can also port existing applications to RPC.

Microsoft RPC offers several extensions to the MIDL compiler that affect:

- Pointer-attribute type inheritance among imported IDL files.
- How many support routines are generated.
- Where support routines are located.

Note that an interface without attributes can be imported into a base IDL file. However, the interface must contain only datatypes with no procedures. If even one procedure is contained in the interface, a local or UUID attribute must be specified.

## The MIDL 3.0 Compiler

You use the MIDL compiler to generate C-language client and server stub files and a header file for your distributed application. The *MIDL Programmer's Guide and Reference* contains the following information:

- C Compiler and C Preprocessor Requirements
- Link Libraries
- Files Generated for an RPC Interface
- MIDL Command-line Reference
- MIDL Language Reference
- MIDL Compiler Errors and Warnings

## Developing C Source Files

Your C-language source files must include the header file that will be generated by the MIDL compiler. By default, the generated header file has the same name as the IDL file. You can specify the name of the generated header file with the MIDL compiler command-line option **midl /header**. Whatever filename you choose, include the generated header file in your C source code.

The generated header file contains directives to include in the following RPC header files:

| Header files | Description |
|---|---|
| RPC.H | RPC types and run-time function prototypes. |
| RPCNDR.H | **byte**, **boolean**, and **small** types and data-conversion function prototypes. |

# The IDL and ACF Files

The MIDL design specifies two distinct files, the Interface Definition Language (IDL) file and the application configuration file (ACF). These files contain attributes which direct the generation of the C-language stub files that manage the remote procedure call. The purpose of distinguishing the files is to keep the network interface separate from characteristics that affect only the operating environment.

The IDL file specifies a network contract between the client and server – that is, the IDL file specifies what is transmitted between client and server. Keeping this information distinct from the information about the operating environment makes the IDL file portable to other environments. The IDL file consists of two parts: an interface header and an interface body.

The ACF specifies attributes that affect only local performance rather than the network contract. Microsoft RPC allows you to combine the ACF and IDL attributes in a single IDL file. You can also combine multiple interfaces in a single IDL file (and its ACF).

The syntax of MIDL is based on the syntax of the C programming language. Whenever a language concept in this description of MIDL is not fully defined, the C-language definition of that term is implied.

This section is organized by topic and summarizes the attributes that are specified in the IDL and ACF files, and in the output files generated by the MIDL compiler. The same material is alphabetized and presented in more detail in the reference topics. For more information, see the MIDL Language Reference and the MIDL Command-Line Reference.

## The IDL Interface Header

The interface header specifies information about the interface as a whole. It must contain the **uuid** or **local** attribute and, whichever one you choose, must occur only once. The **version** attribute may occur at-most-once. The interface header can also contain the attributes **pointer_default** and **endpoint**.

Interface attributes for imported files are optional. However, the top-level importing interface (also called the base interface) must have at least one **uuid** or **local** attribute. MIDL explicitly checks for one of these attributes.

### The uuid Attribute

The **uuid** attribute designates a UUID that distinguishes one interface from other interfaces. The textual representation of a UUID is a string consisting of 8 hexadecimal digits followed by a hyphen, followed by three, hyphen-separated groups of 4 hexadecimal digits, followed by a hyphen, followed by 12 hexadecimal digits. For example:

```
12345678-1234-ABCD-1234-0123456789AB
```

Use the command-line utility **uuidgen** to generate unique identifiers.

### The version Attribute

The **version** attribute identifies a particular version of an interface in cases where multiple versions of the interface exist. The **version** keyword is followed by either a pair of decimal integers separated by a period, or a single decimal integer. The first of the two integers represents the major version number, and the second represents the minor version number of an interface. If a single integer is used, it represents the major version number. Both major and minor version numbers are unsigned short integers in the range between zero and 65535, inclusive.

Leading zeros in a major or minor version-number specification are not significant. A version specification of 1.0001 is the same as 0001.001 and 1.1.

### The endpoint Attribute

The **endpoint** attribute specifies a well-known port or ports (communication end-points) on which servers of the interface listen. Well-known port values are typically assigned by the central authority that owns the protocol.

### The local Attribute

The **local** attribute, when used as an interface attribute, specifies that you want to use the MIDL compiler to generate header files only. Stubs are not generated and checks for transmissibility are omitted.

### The pointer_default Attribute

The **pointer_default** attribute specifies the pointer attribute that is applied to an unattributed pointer specification in the IDL file, including unattributed pointers nested in structure and union fields and arrays. It is not applied to unattributed top-level pointer parameters, which default to **ref**.

Failing to supply a **pointer_default** attribute on an interface that contains an unattributed pointer results in a compile-time warning.

# The IDL Interface Body

The interface body contains data types used in remote procedure calls and the function prototypes for the procedures to be executed remotely. The interface body can contain imports, pragmas, constant declarations, type declarations, and function declarations. In Microsoft-extensions mode, the MIDL compiler also allows implicit declarations in the form of variable definitions.

## Base Types

Base types are the fundamental data types of MIDL. Other types in the interface must be derived from base types or from predefined types.

For example, MIDL restricts the use of the **void *** data type because the size of this type is not fixed. The automatically generated stubs must know the exact size of every data item to be transmitted.

The **boolean** type is an 8-bit data item that is implemented by the MIDL compiler as an **unsigned char**. In keeping with commonly followed programming practices, MIDL implements FALSE as zero and TRUE as 1. When you use the Microsoft-extensions mode of the MIDL compiler, **boolean** initializations using zero and 1 are allowed in addition to TRUE and FALSE. However, in DCE-compatibility mode, only the values TRUE and FALSE are allowed.

The **byte** type consists of 8 bits. The **byte** type is considered opaque data and, consequently, the value is not converted on transmission.

The **char** type is an unsigned 8-bit entity that maps to the **unsigned char** in C. MIDL translates all **char** types in the IDL file to **unsigned char** types in the generated header file. The user can change the default sign of **char** on the target system with the **/char** switch.

The **handle_t** type is used to declare a primitive handle in a type declaration or in a parameter list. Objects of type **handle_t** are not transmitted on the network.

The **void** keyword is valid in a function declaration or in a pointer declaration. In a function declaration, it designates a procedure with no arguments or a procedure that does not return a result. In a pointer declaration, **void** can only be used with the **context_handle** attribute.

The keyword **int** designates a 32-bit integer on 32-bit platforms. On 16-bit platforms you must use one of the following integer modifiers and the **int** keyword is optional. The **hyper** keyword designates a 64-bit integer, the **long** keyword a 32-bit integer, the **short** keyword a 16-bit integer, and the **small** keyword an 8-bit integer. For more information, see **signed** and **unsigned**.

The **float** keyword designates a 32-bit floating-point number and the **double** keyword designates a 64-bit floating-point number. For more information, see **signed** and **unsigned**.

## Predefined Types

The predefined types **error_status_t** and **wchar_t** are derived from the MIDL base types. The predefined type **wchar_t** is a wide-character type and is defined as an **unsigned short**. The predefined type **error_status_t** is the data type returned by the stubs when the stubs encounter a run-time error.

Attributes specify how data is managed on the network. For example, when the function parameter represents a pointer, array, or union, the attributes direct the generation of stub code that packages the data for network transmission.

## The import Directive

The **import** directive is closely related to the **#include** C-preprocessor macro. It directs the compiler to include, at the point of import, the data types defined in the imported files and to make them available for use in the interface. In contrast to the C **#include** macro, the **import** directive ignores procedure

prototypes defined in imported files.

## Pragmas

C-preprocessing directives such as **#define** are expanded by the C preprocessor during MIDL compilation and are not available at C-compile time. To avoid losing these C-preprocessor macro definitions, use the **cpp_quote** or **pragma** **midl_echo** directive. These directives take quoted strings as parameters and instruct the MIDL compiler to emit the parameter string into the generated header file in the same lexical position relative to other interface components.

## Constant Declarations

Constant declarations allow you to associate a constant value with an identifier and use the identifier as part of an expression. Constant declarations are generated as **#define** statements in the header file. The MIDL compiler does not perform any range checking on integral expressions.

Constant declarations are limited to integral **char**, **boolean**, **wchar_t**, **wchar_t ***, **char ***, and **void *** types. The constant value is an expression whose operands are all constant integer literals, boolean expressions that are computable at compile time, or single characters or strings, depending on the **const** type. For more information, see **const**.

# IDL Attributes

Attributes are keywords that specify characteristics of the data in the remote procedure calls and characteristics of the interface. Most attributes appear within square brackets in the IDL and ACF files. The following table briefly describes categories of MIDL attributes that can appear in the IDL file:

| Attribute category | Attributes | Description |
|---|---|---|
| Array attributes | **max_is**, **size_is**, **first_is**, **last_is**, **length_is** | Apply to the first dimension of an array. |
| Directional attributes | **in**, **out** | Describe the direction in which the parameter is transmitted on the network; either or both **in** and **out** can be applied. |
| Field attributes | **switch_is**, array attributes, pointer attributes, **string**, **ignore** | Apply to **struct** or **union** members. |
| Function attributes | **callback**, **call_as**, **idempotent**, **local**, **maybe**, **optimize**, pointer attributes, usage attributes | Apply to the return type and characteristics of the function. |
| Interface attributes | **uuid**, **object**, **local**, **version**, **pointer_default**, **endpoint** | Apply to the interface as a whole. |
| Parameter attributes | Directional attributes, array attributes, pointer attributes, **switch_is**, **string**, **context_handle** | Describe the network-transmission characteristics of function parameters. |
| Pointer attributes | **ref**, **unique**, **iid_is**, **ptr** | Describe characteristics of the pointer and its data. |
| Type attributes | **handle**, **ms_union**, **v1_enum**, **transmit_as**, **switch_type**, **represent_as** pointer attributes, field attributes | Apply to a type definition. |
| Usage attributes | **string**, **ignore**, **context_handle** | Describe how the data object is used. |

## The ACF File

The application configuration file (**ACF**) has two parts: an interface header similar to the interface header in the IDL file, and a body, containing configuration attributes that apply to types and functions defined in the interface body of the IDL file.

## The ACF Header

The ACF header contains attributes that apply to the interface as a whole. Attributes applied to individual types and functions in the ACF body override the attributes in the ACF header. No attributes are required in the ACF header.

The ACF header can include one of the following attributes: **auto_handle**, **implicit_handle**, or **explicit_handle**. These handle attributes specify the type of handle used for implicit binding when a remote function does not have an explicit binding-handle parameter. When the ACF is not present or does not specify an auto, implicit **handle**, or explicit attribute, MIDL uses **auto_handle** for implicit binding.

Either **code** or **nocode** can appear in the interface header, but the one you choose can appear only once. When neither attribute is present, the compiler uses **code** as a default.

## The ACF Body

The ACF body contains configuration attributes that apply to types and functions defined in the interface body of the IDL file. The ACF body can contain ACF **include**, **typedef**, function, and parameter attributes. All of these items are optional. The body of the ACF can be empty. Attributes applied to individual types and functions in the ACF body override attributes in the ACF header.

The ACF specifies behavior on the local computer and does not affect the data transmitted over the network. It is used to specify details of a stub to be generated. In DCE-compatibility mode (**/osf**), the ACF does not affect interaction between stubs, but between the stub and application code.

A parameter specified in the ACF must be one of the parameters specified in the IDL file. The order of specification of the parameter in the ACF is not significant because the matching is by name, not by position. The parameter list in the ACF can be empty, even when the parameter list in the corresponding IDL signature is not. Abstract declarators (unnamed parameters) in the IDL file cause the MIDL compiler to report errors while processing the ACF because the parameter is not found.

The ACF **include** directive specifies the header files to appear in the generated header as part of a standard C-preprocessor **#include** statement. The ACF keyword **include** differs from a **#include** directive. The ACF keyword **include** causes the line "**#include** *filename*" to appear in the generated header file, while the C-language directive "**#include** *filename*" causes the contents of that file to be placed in the ACF.

The ACF **typedef** statement lets you apply ACF type attributes to types previously defined in the IDL file. The ACF **typedef** syntax differs from the C **typedef** syntax.

The ACF function attributes let you specify attributes that apply to the function as a whole. For more information, see **code**, **optimize**, and **nocode**.

The ACF parameter attributes let you specify attributes that apply to individual parameters of the function. For more information, see **byte_count**.

### See Also

**/app_config**, **/osf**, **auto_handle**, **code**, **explicit_handle**, **IDL**, **implicit_handle**, **include**, **midl**, **nocode**, **optimize**, **represent_as**, **typedef**

# ACF Attributes

## The include Declaration

The **include** statement specifies one or more header files to be included in generated stub code via the C-preprocessor **#include** statement. The user must supply the C header file when compiling the stubs. The ACF **include** statement provides some flexibility in distributed application design. The **include** statement is necessary for certain types, such as **implicit_handle** types that are not defined in the IDL or its closure under **#include** and **import** directives.

## Implicit Binding Handles

When an interface contains one or more functions whose first parameters are not an explicit handle and do not have an **in** or an **in**, **out** context handle bound to a remote address space, an implicit handle is needed. The **implicit_handle** and **auto_handle** attributes provide this capability.

The **implicit_handle** attribute specifies a global variable that is used as the RPC binding handle for all calls without a binding parameter.

The **auto_handle** attribute indicates that any function needing implicit handles is automatically bound. When no binding handle to a server exists just prior to calling the function for the first time, the stub automatically establishes a binding handle for the call.

Either **auto_handle or implicit_handle** can appear, but not both. When a function in the interface requires an implicit handle and no ACF is supplied, or the supplied ACF does not specify either **implicit_handle** or **auto_handle**, the MIDL compiler uses **auto_handle** and issues an informational message.

## The code and nocode Attributes

If **code** appears in the interface attribute list, client stub code is generated for any function in the interface that does not appear in the ACF with a **nocode** in its function attribute list and which does not have a **local** attribute.

If **nocode** appears in the interface attribute list, stub code is generated only for functions in the interface that appear in the ACF with a **code** in their function attribute lists and which do not have a **local** attribute.

The **nocode** attribute is ignored when server stubs are generated. Applying **nocode** when generating server stubs in DCE-compatibility mode is an error. Either **code** or **nocode** can appear in an function attribute list, but not both.

## The allocate Attribute

The **allocate** attribute allows you to customize the allocation and deallocation patterns used by the application and stubs. It can be applied to pointer types as a type attribute or as an interface attribute. When it occurs as an interface attribute, it affects all pointer parameters and types in the interface.

| allocate attribute | Description |
|---|---|
| **allocate(single_node)** | Storage for each node on both the caller and callee side is allocated separately by calling **midl_user_allocate**. |
| **allocate(all_nodes)** | The size of the total graph (or tree) is precomputed by the stub and **midl_user_allocate** is called once to allocate sufficient memory for all nodes in the graph upon return from a remote call. In this case, application code has to release this storage by making a single call to |

| | |
|---|---|
| | midl_user_free. |
| **allocate(free)** | Storage allocated for nodes on the callee side is freed by stubs upon return from the manager code. |
| **allocate(dont_free)** | Storage allocated for nodes on the server side is not deallocated by the server stub. This feature is useful for maintaining persistent pointer structures as part of the server application. |

## The byte_count Attribute

The **byte_count** ACF attribute associates a pointer parameter with another parameter that specifies the size in bytes of the memory area indicated by the pointer. Memory referenced by the pointer parameter is contiguous and is not allocated or freed by the client stubs. This feature of the **byte_count** attribute lets the developer create a persistent buffer area in client memory that can be reused across multiple calls.

The parameter providing the buffer must be an **out** pointer parameter and the parameter providing the length in bytes must be an **in** parameter of integral type. The **byte_count** attribute cannot be specified on a parameter that has the size attributes (**size_is**, **max_is**) applied to it.

## Using ACF Attributes in the IDL File

The Microsoft RPC MIDL compiler offers an operating mode that makes it possible to provide one file containing both the IDL attributes and selected ACF attributes. You can supply the ACF attributes **auto_handle** and **implicit_handle** in the IDL file when you use the MIDL compiler switch **/app_config**.

## MIDL Compiler Output

With the IDL and ACF files as input, the MIDL compiler generates up to five C-language source files. By default, the MIDL compiler uses the base filename of the IDL file as part of the generated stub files. When more than six characters are present in the base filename, some file systems may not accept the full stub name. The following conventions are used:

| File | Default portion of base filename | Example |
|---|---|---|
| IDL file | --- | ABCDEFGH.IDL |
| Header | .H | ABCDEF.H |
| Client stub | _C.C | ABCDEF_C.C |
| Server stub | _S.C | ABCDEF_S.C |

# Data and Language Features

The Microsoft® Interface Definition Language (MIDL) provides the set of features that extend the C programming language to support remote procedure calls. MIDL is not a variation of C; it is a strongly typed formal language through which you can control the data transmitted over the network. MIDL is designed to be similar to C so developers familiar with C can learn it quickly.

This topic discusses three language features: strong typing, directional attributes, and data transmission.

**Strong Typing:**

MIDL enforces strong typing by mandating the use of keywords that unambiguously define the size and type of data. The most visible effect of strong typing is that MIDL does not allow variables of the type **void ***.

**Directional Attributes:**

Directional attributes describe whether the data is transmitted from client to server, server to client, or both.

**Data Transmission:**

The **transmit_as** attribute lets you convert one data type to another data type for transmission over the network. The **represent_as** attribute lets you control the way data is presented to the application.

## Strong Typing

C is a weakly typed language. In a weakly typed language, the compiler allows operations such as assignment and comparison among variables of different types. For example, C allows the value of a variable to be cast to another type. The ability to use variables of different types in the same expression promotes flexibility as well as efficiency.

A strongly typed language imposes restrictions on operations among variables of different types. In those cases, the compiler issues an error prohibiting the operation. These strict guidelines regarding data types are designed to avoid potential errors.

The difficulty with using a weakly typed language such as C for remote procedure calls is that distributed applications can run on several different computers with different C compilers and different architectures.

When an application runs on only one computer, you don't have to be concerned with the internal data format because the data is handled in a consistent manner. However, in a distributed computing environment, different computers can use different definitions for their base data types. For example, some computers define the **int** type so its internal representation is 16 bits, while other computers use 32 bits. One computer architecture, known as "little endian," assigns the least significant byte of data to the lowest memory address and the most significant byte to the highest address. Another architecture, known as "big endian," assigns the least significant byte to the highest memory address associated with that data.

Remote procedure calls require strict control over parameter types. To handle data transmission and conversion over the network, MIDL strictly enforces type restrictions for data transferred over the network. For this reason, MIDL includes a set of well-defined base types.

## Base Types

To prevent the problems that implementation-dependent data types can cause on varying computer architectures, MIDL defines its own base data types:

| Base type | Description |
|---|---|
| **boolean** | Data item that can have the value TRUE or FALSE. |
| **byte** | An 8-bit data item guaranteed to be transmitted without any change. |
| **char** | An 8-bit unsigned character data item. |
| **double** | A 64-bit floating-point number. |
| **float** | A 32-bit floating-point number. |
| **handle_t** | Primitive handle that can be used for RPC binding or data serializing. |
| **hyper** | A 64-bit integer that can be declared as either signed or unsigned. (Can also be referred to as **_int64**.) |
| **int** | A 32-bit integer that can be declared as either signed or unsigned. |
| **long** | A 32-bit integer that can be declared as either signed or unsigned. |
| **short** | A 16-bit integer that can be declared as either signed or unsigned. |
| **small** | An 8-bit integer that can be declared as either signed or unsigned. |
| **wchar_t** | Wide-character type that is supported as a Microsoft extension to IDL. Therefore, this type is not available if you compile using the /**osf** switch. |

The header file RPCNDR.H provides definitions for most of these base data types. The keyword **int** is recognized and is remoteable on 32-bit platforms. On 16-bit platforms, the **int** data type requires a modifier, such as **short** or **long**, to specify its length.

Although **void *** is recognized as a generic pointer type by the ANSI C standard, MIDL restricts its usage. Each pointer used in a remote or serializing operation must point to either base types or types constructed from base types. (There is an exception: context handles are defined as **void *** types. For more information see Context Handles.)

## Signed and Unsigned Types

Compilers that use different defaults for signed and unsigned types can cause software errors in your distributed application.

You can avoid these problems by explicitly declaring your character types as signed or unsigned.

MIDL defines the **small** type to take the same default sign as the **char** type in the target C compiler. If the compiler assumes that **char** is unsigned, **small** will also be defined as unsigned. Many C compilers let you change the default as a command-line option. For example, the Microsoft C compiler **/J** command-line option changes the default sign of **char** from signed to unsigned.

You can also control the sign of variables of type **char** and **small** with the MIDL compiler command-line switch **/char**. This switch allows you to specify the default sign used by your compiler. The MIDL compiler explicitly declares the sign of all **char** types that do not match your C-compiler default type in the generated header file.

## Wide-Character Types

Microsoft RPC supports the wide-character type **wchar_t**. The wide-character type uses 2 bytes for each character. The ANSI C-language definition allows you to initialize long characters and long strings as:

```
wchar_t wcInitial = L'a';
wchar_t * pwszString = L"Hello, world";
```

## Structures

Normal C semantics apply to the fields of base types. Fields of more complex types, such as pointers, arrays, and other constructed types, can be modified by type or **field_attributes**. For more information, see **struct**.

# Unions

Some features of the C language, such as unions, require special MIDL keywords to support their use in remote procedure calls.

A union in the C language is a variable that holds objects of different types and sizes. The developer usually creates a variable to keep track of the types stored in the union. To operate correctly in a distributed environment, the variable that indicates the type of the union, or the "discriminant," must also be available to the remote computer. MIDL provides the **switch_type** and **switch_is** keywords to identify the discriminant type and name.

MIDL requires that the discriminant be transmitted with the union in one of two ways:

- The union and the discriminant must be provided as parameters.
- The union and the discriminant must be packaged in a structure.

Two fundamental types of discriminated unions are provided by MIDL: **non-encapsulated_union** and **encapsulated_union**. The discriminant of a nonencapsulated union is another parameter if the union is a parameter. It is another field if the union is a field of a structure. The definition of an encapsulated union is turned into a structure definition whose first field is the discriminant and whose second and last fields are the union.

The following example demonstrates how to provide the union and discriminant as parameters:

```
typedef [switch_type(short)] union {
    [case(0)]    short     sVal;
    [case(1)]    float     fVal;
    [case(2)]    char      chVal;
    [default]    ;
} DISCRIM_UNION_PARAM_TYPE;

short UnionParamProc(
    [in, switch_is(sUtype)] DISCRIM_UNION_PARAM_TYPE Union,
    [in]                    short                    sUtype);
```

The union in the preceding example can contain a single value: either **short**, **float**, or **char**. The type definition for the union includes the MIDL **switch_type** attribute which specifies the type of the discriminant. Here, [switch_type(short)] specifies that the discriminant is of type **short**. The switch must be an integer type.

If the union is a member of a structure, then the discriminant must be a member of the same structure. If the union is a parameter, then the discriminant must be another parameter. The prototype for the function **UnionParamProc** shows the discriminant *sUtype* as the last parameter of the call. (The discriminant can appear in any position in the call.) The type of the parameter specified in the **switch_is** attribute must match the type specified in the **switch_type** attribute.

The following example demonstrates the use of a single structure that packages the discriminant with the union:

```
typedef struct {
    short utype;  /* discriminant can precede or follow union */
    [switch_is(utype)] union {
       [case(0)]   short     sVal;
       [case(1)]   float     fVal;
       [case(2)]   char      chVal;
```

```
        [default]   ;
    } u;
} DISCRIM_UNION_STRUCT_TYPE;

short UnionStructProc(
    [in] DISCRIM_UNION_STRUCT_TYPE u1);
```

The Microsoft RPC MIDL compiler allows union declarations outside of **typedef** constructs. This feature is an extension to DCE IDL. For more information, see **union**.

## Enumerated Types

The **enum** declaration is not translated into **#define** statements as it is by some DCE compilers, but is reproduced as a C-language **enum** declaration in the generated header file.

## Arrays

See **arrays**.

## Directional (Parameter) Attributes

All parameters in the function prototype must be associated with directional attributes. The three possible combinations of directional attributes are: 1) **in**, 2) **out**, and 3) **in, out**. These describe the way parameters are passed between calling and called procedures. When you compile in the default (Microsoft-extended mode) and you omit a directional attribute for a parameter, the MIDL compiler assumes a default value of **in**.

An **out** parameter must be a pointer. In fact, the **out** attribute is not meaningful when applied to parameters that do not act as pointers because C function parameters are passed by value. In C, the called function receives a private copy of the parameter value; it cannot change the calling function's value for that parameter. If the parameter acts as a pointer, however, it can be used to access and modify memory. The **out** attribute indicates that the server function should return the value to the client's calling function, and that memory associated with the pointer should be returned in accordance with the attributes assigned to the pointer.

The following interface demonstrates the three possible combinations of directional attributes that can be applied to a parameter. The function **InOutProc** is defined in the IDL file as:

```
void InOutProc ([in]       short     s1,
                [in, out]  short *  ps2,
                [out]      float *  pf3);
```

The first parameter, *s1*, is **in** only. Its value is transmitted to the remote computer, but is not returned to the calling procedure. Although the server application can change its value for *s1*, the value of *s1* on the client is the same before and after the call.

The second parameter, *ps2*, is defined in the function prototype as a pointer with both **in** and **out** attributes. The **in** attribute indicates that the value of the parameter is passed from the client to the server. The **out** attribute indicates that the value pointed to by *ps2* is returned to the client.

The third parameter is **out** only. Space is allocated for the parameter on the server, but the value is undefined on entry. As mentioned above, all **out** parameters must be pointers.

The remote procedure changes the value of all three parameters, but only the new values of the **out** and **in, out** parameters are available to the client.

```
#define MAX 257

void InOutProc(short    s1,
               short * ps2,
               float * pf3)
{
    *pf3 = (float) s1 / (float) *ps2;
    *ps2 = (short) MAX - s1;
    s1++;  // in only; not changed on the client side
    return;
}
```

On return from the call to **InOutProc**, the second and third parameters are modified. The first parameter, which is **in** only, is unchanged.

{ewc msdncd, EWGraphic, bsd23533 0 /a "SDK_A22.BMP"}

{ewc msdncd, EWGraphic, bsd23533 1 /a "SDK_A23.BMP"}

{ewc msdncd, EWGraphic, bsd23533 2 /a "SDK_A21.BMP"}

## Function Attributes

The **callback** and **local** attributes can be applied as function attributes.

Callbacks are a special kind of remote call from server to client that executes as part of a conceptual single-execution thread. A callback is always issued in the context of a remote call (or callback) and is executed by the thread that issued the original remote call (or callback).

It is often desirable to place a local procedure declaration in the IDL file, since this is the logical place to describe interfaces to a package. The **local** attribute indicates that a procedure declaration is not actually a remote function, but a local procedure. The MIDL compiler does not generate any stubs for that function. For more information, see **callback** and **local**.

## Field Attributes

Field attributes are the attributes that can be applied to fields of an array, structure, or union: **ignore**, **size_is**, **max_is**, **length_is**, **first_is**, **last_is**, **switch_is**, and **string**, and pointer attributes. For example, field attributes are used in conjunction with array declarations to specify either the size of the array or the portion of the array that contains valid data. This is done by associating another parameter, structure field, or a constant expression with the array.

The **ignore** attribute designates pointer fields to be ignored during the marshalling process. Such an ignored field is set to NULL on the receiver side.

## Conformant Arrays (size_is, max_is Attributes)

An array is called conformant if its bounds are determined at run time. The **size_is** attribute designates the upper bound on the allocation size of the array and the **max_is** attribute designates the upper bound on the value of a valid array index. For more information, see **arrays**.

## Varying and Open Arrays (length_is, first_is, last_is Attributes)

An array is called "varying" if its bounds are determined at compile time, but the range of transmitted elements is determined at run time. An open array (also called a conformant varying array) is an array whose upper bound and range of transmitted elements are determined at run time. To determine the range of transmitted elements of an array, the array declaration must include a **length_is**, **first_is**, or **last_is** attribute.

The **length_is** attribute designates the number of array elements to be transmitted and the **first_is** attribute designates the index of the first array element to be transmitted. The **last_is** attribute designates the index of the last array element to be transmitted.

## The switch_is Attribute

The **switch_is** attribute designates a union discriminator. When the union is a procedure parameter, the union discriminator must be another parameter of the same procedure. When the union is a field of a structure, the discriminator must be another field of the structure at the same level as the union field. The discriminator must be a **boolean**, **char**, **integral**, or **enum** type, or a type that resolves to one of these types. For more information, see **non-encapsulated_union**.

## The string Attribute

The **string** attribute designates that a one-dimensional character or byte array, or a pointer to a zero-terminated character or byte stream, is to be treated as a string. The string attribute applies only to one-dimensional arrays and pointers. The element type is limited to **char**, **byte**, **wchar_t**, or a named type that resolves to these types.

# Type Attributes

Type attributes are the MIDL attributes that can be applied to type declarations: **transmit_as**, **represent_as**, **user_marshal**, **wire_marshal**, **handle**, **context_handle**, **switch_type**, and the pointer type attributes.

The **switch_type** attribute designates the type of a union discriminator. This attribute applies only to a nonencapsulated union.

A context handle is a pointer with a **context_handle** attribute. The **context_handle** attribute allows you to write procedures that maintain state information between remote procedure calls. A context handle with a non-null value represents saved context and serves two purposes. On the client side, it contains the information needed by the RPC run-time library to direct the call to the server. On the server side, it serves as a handle on active context.

The **handle** attribute specifies that a type can occur as a user-defined, (generic) handle. This feature permits the design of handles that are meaningful to the application. The user must provide binding and unbinding routines to convert between the user-defined handle type and the RPC primitive handle type, **handle_t**. A primitive handle contains destination information meaningful to the RPC run-time libraries. A user-defined handle can only be defined in a type declaration, not in a function declaration. A parameter with the **handle** attribute has a double purpose. It is used to determine the binding for the call, and it is transmitted to the called procedure as a normal data parameter.

The **transmit_as** and **represent_as** attributes instruct the compiler to associate a transmissible type which the stub passes between client and server, with a user type which the client and server applications use. You must supply the routines that carry out the conversion between the user type and the transmissible type, and the routines to release the memory that was used to hold the converted data. Using the **transmit_as** IDL attribute or the **represent_as** ACF attribute instructs the stub to call these conversion routines before and after transmission.

The **wire_marshal** and **user_marshal** attributes are Microsoft extensions to the OSF-DCE IDL. Their syntax and functionality are similar to that of the DCE-specified **transmit_as** and **represent_as** attributes, respectively. The difference is that, instead of converting the data from one type to another, you marshal the data directly.To do this, you must supply the external routines for sizing the data buffer on the client and server sides, marshaling and unmarshaling the data on the client and server sides, and freeing the data on the server side. The MIDL compiler generates format codes that instruct the NDR engine to call these external routines when needed.

The **wire_marshal** and **user_marshal** attributes make it possible to marshal data types that otherwise could not be transmitted across process boundaries. Also, because there is less overhead associated with the type conversion, **wire_marshal** and **user_marshal** provide improved performance at run time, when compared to **transmit_as** and **represent_as**.

The **wire_marshal** and **user_marshal** attributes are mutually exclusive in respect to each other and with the **transmit_as** and **represent_as** attributes for a given type.

## The transmit_as Attribute

The **transmit_as** attribute offers a way to control data marshalling without worrying about marshalling data at a low level – that is, without worrying about data sizes or byte swapping in a heterogeneous environment. By lettting you reduce the amount of data transmitted over the network, the **transmit_as** attribute can make your application more efficient.

You use the **transmit_as** attribute to specify a data type that will be used for transmission instead of using the data type provided. You supply routines that convert the data type to and from the type that is used for transmission. You must also supply routines to free the memory used for the data type and the transmitted type. For example, the following defines *xmit_type* as the transmitted data type for an application-presented *type* specified as *type_spec*:

```
typedef [transmit_as (xmit_type)] type_spec type;
```

The following table describes the four user-supplied routine names. *Type* is the data type known to the application, and *xmit_type* is the data type used for transmission:

| Routine | Description |
|---|---|
| *type*_**to_xmit** | Allocates an object of the transmitted type and converts from presented type to transmitted type (caller and callee). |
| *type*_**from_xmit** | Converts from transmitted type to presented type (caller and callee). |
| *type*_**free_inst** | Frees resources used by the presented type (callee only). |
| *type*_**free_xmit** | Frees storage returned by the *type*_**to_xmit** routine (caller and callee). |

Other than by these four user-supplied functions, the transmitted type is not manipulated by the application. The transmitted type is defined only to move data over the network. After the data is converted to the type used by the application, the memory used by the transmitted type is freed.

These user-supplied routines are provided by either the client or the server application based on the directional attributes.

If the parameter is **in** only, the client transmits to the server. The client needs the *type*_**to_xmit** and *type*_**free_xmit** functions. The server needs the *type*_**from_xmit** and *type*_**free_inst** functions.

For an **out**-only parameter, the server transmits to the client. The server needs the *type*_**to_xmit** and *type*_**free_xmit** functions, while the client needs the *type*_**from_xmit** function.

For the temporary *xmit_type* objects, the stub will call *type*_**free_xmit** to free any memory allocated by a call to *type*_**to_xmit**.

Certain guidelines apply to the presented type instance. If the presented type is a pointer or contains a pointer, then the *type*_**from_xmit** routine must allocate pointees of the pointers (the presented type object itself is manipulated by the stub in the usual way).

For **out** and **in, out** parameters, or one of their components, of a type that contains the **transmit_as** attribute, the *type*_**free_inst** routine is automatically called for the data objects that have the attribute. For **in** parameters, the *type*_**free_inst** routine is called only if the **transmit_as** attribute has been applied to the parameter. If the attribute has been applied to the components of the parameter, the *type*_**free_inst** routine is not called. There are no freeing calls for the embedded data and at-most-one call (related to the top-level attribute) for an **in** only parameter.

Effective with MIDL version 2.0, both client and server must supply all four functions. For example, a linked list can be transmitted as a sized array. The *type***_to_xmit** routine walks the linked list and copies the ordered data into an array. The array elements are ordered so the many pointers associated with the list data structure do not have to be transmitted. The *type***_from_xmit** routine reads the array and puts its elements into a linked-list data structure.

The double-linked list (DOUBLE_LINK_LIST) includes data and pointers to the previous and next list elements:

```
typedef struct _DOUBLE_LINK_LIST {
    short sNumber;
    struct _DOUBLE_LINK_LIST * pNext;
    struct _DOUBLE_LINK_LIST * pPrevious;
} DOUBLE_LINK_LIST;
```

Rather than shipping the complex data structure, the **transmit_as** attribute can be used to send it over the network as an array. The sequence of items in the array retains the ordering of the elements in the list at a lower cost:

```
typedef struct _DOUBLE_XMIT_TYPE {
    short sSize;
    [size_is(sSize)] short asNumber[];
} DOUBLE_XMIT_TYPE;
```

The **transmit_as** attribute appears in the IDL file:

```
typedef [transmit_as(DOUBLE_XMIT_TYPE)]  DOUBLE_LINK_LIST
                                         DOUBLE_LINK_TYPE;
```

In the following example, **ModifyListProc** defines the parameter of type DOUBLE_LINK_TYPE as an **in, out** parameter:

```
void ModifyListProc([in, out] DOUBLE_LINK_TYPE * pHead)
```

The four user-defined functions use the name of the type in the function names and use the presented and transmitted types as parameter types, as required:

```
void __RPC_USER DOUBLE_LINK_TYPE_to_xmit (
     DOUBLE_LINK_TYPE __RPC_FAR * pList,
     DOUBLE_XMIT_TYPE __RPC_FAR * __RPC_FAR * ppArray);

void __RPC_USER DOUBLE_LINK_TYPE_from_xmit (
     DOUBLE_XMIT_TYPE __RPC_FAR * pArray,
     DOUBLE_LINK_TYPE __RPC_FAR * pList);

void __RPC_USER DOUBLE_LINK_TYPE_free_inst (
     DOUBLE_LINK_TYPE __RPC_FAR * pList);

void __RPC_USER DOUBLE_LINK_TYPE_free_xmit (
     DOUBLE_XMIT_TYPE __RPC_FAR * pArray);
```

## The type_to_xmit Function

The stubs call the *type***_to_xmit** function to convert the type that is presented by the application into the transmitted type. The function is defined as:

```
void __RPC_USER <type>_to_xmit (
     <type> __RPC_FAR *, <xmit_type> __RPC_FAR *      __RPC_FAR *);
```

The first parameter is a pointer to the presented data. The second parameter is set by the function to point to the transmitted data. The function must allocate memory for the transmitted type.

In the following example, the client calls the remote procedure that has an **in, out** parameter of type DOUBLE_LINK_TYPE. The client stub calls the *type***_to_xmit** function, here named **DOUBLE_LINK_TYPE_to_xmit**, to convert double-linked list data into a sized array.

The function determines the number of elements in the list, allocates an array large enough to hold those elements, then copies the list elements into the array. Before the function returns, the second parameter, *ppArray*, is set to point to the newly allocated data structure.

```
void __RPC_USER DOUBLE_LINK_TYPE_to_xmit (
    DOUBLE_LINK_TYPE __RPC_FAR * pList,
    DOUBLE_XMIT_TYPE __RPC_FAR * __RPC_FAR * ppArray)
{
    short cCount = 0;
    DOUBLE_LINK_TYPE * pHead = pList;  // save pointer to start
    DOUBLE_XMIT_TYPE * pArray;

    /* count the number of elements to allocate memory */
    for (; pList != NULL; pList = pList->pNext)
        cCount++;

    /* allocate the memory for the array */
    pArray = (DOUBLE_XMIT_TYPE *) MIDL_user_allocate
         (sizeof(DOUBLE_XMIT_TYPE) + (cCount * sizeof(short)));
    pArray->sSize = cCount;

    /* copy the linked list contents into the array */
    cCount = 0;
    for (i = 0, pList = pHead; pList != NULL; pList = pList->pNext)
        pArray->asNumber[cCount++] = pList->sNumber;

    /* return the address of the pointer to the array */
    *ppArray = pArray;
}
```

## The type_from_xmit Function

The stubs call the *type*_**from_xmit** function to convert data from its transmitted type to the type that is presented to the application. The function is defined as:

```
void __RPC_USER <type>_from_xmit (
    <xmit_type> __RPC_FAR *,
    <type> __RPC_FAR *);
```

The first parameter is a pointer to the transmitted data. The function sets the second parameter to point to the presented data.

The *type*_**from_xmit** function must manage memory for the presented type. The function must allocate memory for the entire data structure that starts at the address indicated by the second parameter, except for the parameter itself (the stub allocates memory for the root node and passes it to the function). The value of the second parameter cannot change during the call. The function can change the contents at that address.

In this example, the function **DOUBLE_LINK_TYPE_from_xmit** converts the sized array to a double-linked list. The function retains the valid pointer to the beginning of the list, frees memory associated with the rest of the list, then creates a new list that starts at the same pointer. The function uses a utility function, **InsertNewNode**, to append a list node to the end of the list and to assign the *pNext* and *pPrevious* pointers to appropriate values.

```
void __RPC_USER DOUBLE_LINK_TYPE_from_xmit(
     DOUBLE_XMIT_TYPE __RPC_FAR * pArray,
     DOUBLE_LINK_TYPE __RPC_FAR * pList)
{
    DOUBLE_LINK_TYPE *pCurrent;
    int i;

    if (pArray->sSize <= 0) {  // error checking
        return;
    }

    if (pList == NULL) // if invalid, create the list head
        pList = InsertNewNode(pArray->asNumber[0], NULL);
    else {
        DOUBLE_LINK_TYPE_free_inst(pList);  // free all other nodes
        pList->sNumber = pArray->asNumber[0];
        pList->pNext = NULL;
    }

    pCurrent = pList;
    for (i = 1; i < pArray->sSize; i++)
        pCurrent = InsertNewNode(pArray->asNumber[i], pCurrent);

    return;
}
```

## The type_free_xmit Function

The stubs call the *type*_**free_xmit** function to free memory associated with the transmitted data. After the *type*_**from_xmit** function converts the transmitted data to its presented type, the memory is no longer needed. The function is defined as:

```
void __RPC_USER <type>_free_xmit(<xmit_type> __RPC_FAR *);
```

The parameter is a pointer to the memory that contains the transmitted type.

In this example, the memory contains an array that is in a single structure. The function **DOUBLE_LINK_TYPE_free_xmit** uses the user-supplied function **midl_user_free** to free the memory:

```
void __RPC_USER DOUBLE_LINK_TYPE_free_xmit(
     DOUBLE_XMIT_TYPE __RPC_FAR * pArray)
{
     midl_user_free(pArray);
}
```

## The type_free_inst Function

The stubs call the *type*_**free_inst** function to free memory associated with the presented type. The function is defined as:

```
void __RPC_USER <type>_free_inst(<type> __RPC_FAR *)
```

The parameter points to the presented type instance. This object should not be freed. For a discussion on when to call the function, see the **transmit_as** Attribute.

In the following example, the double-linked list is freed by walking the list to its end, then backing up and freeing each element of the list.

```
void __RPC_USER DOUBLE_LINK_TYPE_free_inst(
     DOUBLE_LINK_TYPE __RPC_FAR * pList)
{
    while (pList->pNext != NULL)  // go to end of the list
        pList = pList->pNext;

    pList = pList->pPrevious;
    while (pList != NULL) {  // back through the list
        midl_user_free(pList->pNext);
        pList = pList->pPrevious;
    }
}
```

## The represent_as Attribute

The **represent_as** attribute lets you specify how a particular remotable data type is represented for the application. This is done by specifying the name of the represented type for a known transmittable type and supplying the routines to convert the data type to and from the other data type. You must also supply the routines to free the memory used by the data type objects.

You use the **represent_as** attribute to present an application with a different and perhaps nonremotable data type, instead of the type that is actually transmitted between the client and server. Also, the type the application manipulates can be unknown at the time of MIDL compilation. When you choose a well-defined transmittable type, you need not be concerned about data representation in the heterogenic environment. The **represent_as** attribute can make your application more efficient by reducing the amount of data transmitted over the network.

The **represent_as** attribute is similar to the **transmit_as** attribute. However, while **transmit_as** lets you specify a data type that will be used for transmission, **represent_as** lets you specify how a data type is represented for the application. The represented type need not be defined in the MIDL processed files; it can be defined at the time the stubs are compiled with the C compiler. To do this, use the include directive in the ACF to compile the appropriate header file. For example, the following ACF defines a local represented *repr_type* for the given transmittable *named_type*:

```
typedef [represent_as(repr_type) [, type_attribute_list] named_type;
```

The following table describes the four user-supplied routines:

| Routine | Description |
|---|---|
| *named_type*_**from_local** | Allocates an instance of the network type and converts from the local type to the network type. |
| *named_type*_**to_local** | Converts from the network type to the local type. |
| *named_type*_**free_local** | Frees memory allocated by a call to the *named_type*_**to_local** routine, but not the type itself. |
| *named_type*_**free_inst** | Frees storage for the network type (both sides). |

Other than by these four user-supplied routines, the named type is not manipulated by the application and the only type visible to the application is the represented type. The represented type name is used instead of the named type name in the prototypes and stubs generated by the compiler. You must supply the set of routines for both sides.

For temporary *named_type* objects, the stub will call *named_type*_**free_inst** to free any memory allocated by a call to *named_type*_**from_local**.

If the represented type is a pointer or contains a pointer, the *named_type*_**to_local** routine must allocate pointees of the pointers (the represented type object itself is manipulated by the stub in the usual way). For **out** and **in, out** parameters of a type that contain **represent_as** or one of its components, the *named_type*_**free_local** routine is automatically called for the data objects that contain the attribute. For **in** parameters, the *named_type*_**free_local** routine is only called if the **represent_as** attribute has been applied to the parameter. If the attribute has been applied to the components of the parameter, the *_**free_local** routine is not called. Freeing routines are not called for the embedded data and at-most-once call (related to the top-level attribute) for an **in** only parameter.

**Note** It is possible to apply both the **transmit_as** and **represent_as** attributes to the same type. When marshalling data, the **represent_as** type conversion is applied first and then the **transmit_as** conversion is applied. The order is reversed when unmarshalling data. Thus, when marshalling, ***_from_local** allocates an instance of a named type and translates it from a local type object to the temporary named type object. This object is the presented type object used for the ***_to_xmit** routine. The ***_to_xmit** routine then allocates a transmitted type object and translates it from the presented (named) object to the transmitted object.

An array of long integers can be used to represent a linked list. In this way, the application manipulates the list and the transmission uses an array of long integers when a list of this type is transmitted. You can begin with an array, but using a construct with an open array of long integers is more convenient. The following example shows how to do this:

```
/* IDL definitions */

typedef struct_lbox {
    long        data;
    struct_lbox *        pNext
} LOC_BOX, * PLOC_BOX;

/* The definition of the local type visible to the application,
as shown above, can be omitted in the IDL file. See the include
in the ACF file. */

typedef struct_xmit_lbox {
    short        Size;
    [size_is(Size)] long DaraArr[];
} LONGARR;

void WireTheList( [in,out] LONGARR * pData );

/* ACF definitions */

/* If the IDL file does not have a definition for PLOC_BOX, you
can still ready it for C compilation with the following include
statement (notice that this is not a C include):
include "local.h";*/

typedef [represent_as(PLOC_BOX)] LONGARR;
```

Note that the prototypes of the routines that use the LONGARR type are actually displayed in the STUB.H files as PLOC_BOX in place of the LONGARR type. The same is true of the appropriate stubs in the STUB_C.C file.

You must supply the folllowing four functions:

```
void __RPC_USER
LONGARR_from_local(
    PLOC_BOX __RPC_FAR * pList,
    LONGARR __RPC_FAR * _RPC_FAR * ppDataArr );

void __RPC_USER
LONGARR_to_local(
    LONGARR __RPC_FAR * _RPC_FAR * ppDataArr,
```

```
    PLOC_BOX __RPC_FAR * pList );

void __RPC_USER
LONGARR_free_inst(
    LONGARR __RPC_FAR * pDataArr);

void __RPC_USER
LONGARR_free_local(
    PLOC_BOX __RPC_FAR * pList );
```

The routines shown above do the following:

- The **LONGARR_from_local** routine counts the nodes of the list, allocates a LONGARR object with the size **sizeof**(**LONGARR**) + Count***sizeof**(**long**), sets the *Size* field to Count, and copies the data to the *DataArr* field.
- The **LONGARR_to_local** routine creates a list with Size nodes and transfers the array to the appropriate nodes.
- The **LONGARR_free_inst** routine frees nothing in this case.
- The **LONGARR_free_local** routine frees all the nodes of the list.

## The wire_marshal Attribute

The **wire_marshal** attribute is an IDL type attribute similar in syntax to **transmit_as**, but provides a more efficient way to marshal data across a network.

You use the **wire_marshal** attribute to specify a data type that will be transmitted in place of the application-specific data type. Each application-specific type has a corresponding transmissible type that defines the wire representation (the way it is transmitted).The application-specific type need not be remotable, but it must be a type that MIDL recognizes. To marshal a type unknown to MIDL, use the ACF attribute **user_marshal**.

Your application-specific type can be a simple, composite, or pointer type. The main restriction is that the type object must have a fixed, well-defined memory size. If the size of your type object needs to change, use a pointer field rather than a conformant array. Alternatively, you can define a pointer to the changeable type.

You supply the routines for the sizing, marshaling, unmarshaling and freeing passes. The following table describes the four user-supplied routine names. The <type> is the *userm-type* specified in the **wire_marshal** type definition.

| Routine | Description |
|---|---|
| <*type*>_UserSize | Sizes the rpc data buffer before marshaling on the client or server side. |
| <*type*>_UserMarshal | Marshals the data on the client or server side. |
| <*type*>_UserUnmarshal | Unmarshals the data on the client or server side. |
| <*type*>_UserFree | Frees the data on the server side. |

These user-supplied routines are provided by either the client or the server application based on the directional attributes.

If the parameter is **in** only, the client transmits to the server. The client needs the <*type*>_UserSize and <*type*>_UserMarshal functions. The server needs the <*type*>_UserUnmarshal and <*type*>_UserFree functions.

For an **out**-only parameter, the server transmits to the client. The server needs the <*type*>_UserSize and <*type*>_UserMarshal functions, while the client needs the <*type*>_UserMarshal function.

## See Also

**The user_marshal Attribute**, **Marshaling Rules for user_marshal and wire_marshal**, **wire_marshal**, **user_marshal**

## The user_marshal Attribute

The **user_marshal** attribute is an ACF type attribute similar in syntax to **represent_as**. As with the IDL attribute, **wire_marshal**, it offers a more efficient way to marshal data across a network. As an ACF attribute, **user_marshal** lets you custom marshal data types that are unknown to MIDL. Each application-specific type has a corresponding transmissible type that defines the wire representation.

Your application-specific type can be a simple, composite, or pointer type. The main restriction is that the type object must have a fixed, well-defined memory size. If the size of your type object needs to change, use a pointer field rather than a conformant array. Alternatively, you can define a pointer to the changeable type.

As with the **wire_marshal** attribute, you supply routines for the sizing, marshaling, unmarshaling and freeing passes. The following table describes the four user-supplied routine names. The <type> is the *userm-type* specified in the **user_marshal** type definition.

| Routine | Description |
|---|---|
| <*type*>_UserSize | Sizes the rpc data buffer before marshaling on the client or server side. |
| <*type*>_UserMarshal | Marshals the data on the client or server side. |
| <*type*>_UserUnmarshal | Unmarshals the data on the client or server side. |
| <*type*>_UserFree | Frees the data on the server side. |

These user-supplied routines are provided by either the client or the server application, based on the directional attributes.

If the parameter is **in** only, the client transmits to the server. The client needs the <*type*>_UserSize and <*type*>_UserMarshal functions. The server needs the <*type*>_UserUnmarshal and <*type*>_UserFree functions.

For an **out**-only parameter, the server transmits to the client. The server needs the <*type*>_UserSize and <*type*>_UserMarshal functions, while the client needs the <*type*>_UserMarshal function.

## See Also

**The wire_marshal Attribute**, **Marshaling Rules for user marshal and wire_marshal**, **user_marshal**, **wire_marshal**

## The type_UserSize Function

The <*type*>**_UserSize** function is a helper function for the **wire_marshal** and **user_marshal** attributes.The stubs call this function to size the rpc data buffer for the user data object before data is marshaled on the client or server side. The function is defined as:

```
unsigned long __RPC_USER  <type>_UserSize(
    unsigned long __RPC_FAR * pFlags,
    unsigned long             StartingSize,
    <type>  __RPC_FAR *       pMyObj);
```

The <*type*> in the function name means the *userm-type*, as specified in the **wire_marshal** or **user_marshal** type definition. This type may be nonremotable or even, when used with the **user_marshal** attribute, a type unknown to the MIDL compiler. The wire type name (the name of the transmissible type) is not used in the function prototype. Note, however, that the wire type defines the wire layout for the data as specified by OSF DCE.

The *pFlags* argument is a pointer to an unsigned long flag field.The upper word of the flag contains NDR data representation flags as defined by OSF DCE for floating point, endianess, and character representations. The lower word contains a marshaling context flag as defined by the COM channel. The exact layout of the flags within the field is:

| Bits | Flag | Value |
|---|---|---|
| 31-24 | Floating-point representation | 0 = IEEE<br>1 = VAX<br>2 = Cray<br>3 = IBM |
| 23-20 | Integer and floating-point byte order | 0 = Big-endian<br>1 = Little-endian |
| 19-16 | Character representation | 0 = ASCII<br>1 = EBCDIC |
| 15-0 | Marshaling context flag | 0 = MSHCTX_LOCAL<br>1 = MSHCTX_NOSHAREDMEM<br>2 = MSHCTX_DIFFERENTMSCHINE<br>3 = MSHCTX_INPROC |

The marshaling context flag makes it possible to alter the behavior of your routine depending on the context for the RPC call. For example, if you have a handle (**long**) to a block of data, you could send the handle for an in-process call, but you would send the actual data for a call to a different machine.

The marshaling context flag and its values are defined in the wtypes.h and wtypes.idl files in the Win32 SDK.

**Note** When the wire type is properly defined, you do not have to use the NDR Data Representation flags, as the NDR engine performs the necessary conversions. The flags and their values are shown here solely for interest.

The *StartingSize* argument is the current buffer offset. The starting size indicates the buffer offset for the user object and it may or may not be aligned properly. Your routine should account for whatever padding is necessary.

The *pMyObj* argument is a pointer to a user type object.

The return value is the new offset or buffer position. The function should return the cumulative size, which is the starting size plus possible padding plus the data size.

The <*type*>_UserSize function can return an overestimate of the size needed. The actual size of the sent buffer is defined by the data size, not by the buffer allocation size.

The <*type*>_UserSize function is not called if the wire size can be computed at compile time. Note that for most unions, even if there are no pointers, the actual size of the wire representation can be determined only at run time.

## See also

**Marshaling rules for user_marshal and wire_marshal**, **user_marshal**, **wire_marshal**

### the type_UserMarshal Function

The <*type*>**_UserMarshal** function is a helper function for the **wire_marshal** and **user_marshal** attributes.The stubs call this function to marshal data on the client or server side. The function is defined as:

```
unsigned char __RPC_FAR * __RPC_USER  <type>_UserMarshal(
    unsigned long __RPC_FAR * pFlags,
    unsigned char __RPC_FAR * pBuffer,
    <type>  __RPC_FAR *       pMyObj);
```

The <*type*> in the function name means the *userm-type* specified in the **wire_marshal** or **user_marshal** type definition. This type may be nonremotable or even, when used with the **user_marshal** attribute, a type unknown to the MIDL compiler. The wire type name (the name of transmissible type) is not used in the function prototype. Note, however, that the wire type defines the wire layout for the data as specified by OSF DCE.

The *pFlags* argument is a pointer to an unsigned long flag field.The upper word of the flag contains NDR data representation flags as defined by OSF DCE for floating point, endianess, and character representations. The lower word contains a marshaling context flag as defined by the COM channel. The exact layout of the flags within the field is described in The type_UserSize Function.

The *pBuffer* argument is the current buffer pointer. This pointer may or may not be aligned on entry. Your <*type*>_UserMarshal function should align the buffer pointer appropriately, marshal the data, and return the new buffer position, which is the address of the first byte after the marshaled object. Keep in mind that the wire type specification determines the actual layout of the data in the buffer.

The *pMyObj* argument is a pointer to a user type object.

The return value is the new buffer position, which is the address of the first byte after the unmarshaled object.

## Checking for Buffer Overflow

Buffer overflow can occur when you incorrectly calculate the size of the data and attempt to marshal more data than intended. You should be careful to avoid this situation and to check against it where possible, using the pointer that <*type*>_UserMarshal returns. Otherwise, you risk having the NDR engine raise a buffer overflow exception later.

## See Also

**Marshaling Rules for user_marshal and wire_marshal**, **wire_marshal**, **user_marshal**

## The type_UserUnmarshal Function

The <*type*>**_UserUnmarshal** function is a helper function for the **wire_marshal** and **user_marshal** attributes.The stubs call this function to unmarshal data on the client or server side. The function is defined as:

```
unsigned char __RPC_FAR * __RPC_USER  <type>_UserUnmarshal(
    unsigned long __RPC_FAR * pFlags,
    unsigned char __RPC_FAR * pBuffer,
    <type>  __RPC_FAR *       pMyObj);
```

The <*type*> in the function name means the *userm-type* specified in the **wire_marshal** or **user_marshal** type definition. This type may be nonremotable or even, when used with the **user_marshal** attribute, a type unknown to the MIDL compiler. The wire type name (the name of transmissible type) is not used in the function prototype. Note, however, that the wire type defines the wire layout for the data as specified by OSF DCE.

The *pFlags* argument is a pointer to an unsigned long flag field.The upper word of the flag contains NDR data representation flags as defined by OSF DCE for floating point, endianess, and character representations. The lower word contains a marshaling context flag as defined by the COM channel. The exact layout of the flags within the field is described in The type_UserSize Function.

The *pBuffer* argument is the current buffer pointer. This pointer may or may not be aligned on entry. Your <*type*>_UserUnmarshal function should align the buffer pointer appropriately, unmarshal the data, and return the new buffer position, which is the address of the first byte after the unmarshaled object.

The *pMyObj* argument is a pointer to a user-defined type object.

## Data Conversion

In a hetergeneous environment, the NDR engine performs any data conversion necessary prior to calling the <*type*>_UserUnmarshal function. Note that the NDR engine carries out this data conversion according to the wire-type definition supplied for this user data type. The flag indicates the data representation of the sender.

## See Also

**Marshaling Rules for user_marshal and wire_marshal**, **wire_marshal**, **user_marshal**

## The type_UserFree Function

The <*type*>**_UserFree** function is a helper function for the **wire_marshal** and **user_marshal** attributes.The stubs call this function to free the data on the server side. The function is defined as:

```
void __RPC_USER  <type>_UserFree(
    unsigned long __RPC_FAR * pFlags,
    <type_name>  __RPC_FAR *  pMyObj );
```

The <*type*> in the function name means the *userm-type* specified in the **wire_marshal** or **user_marshal** type definition.

The *pFlags* argument is a pointer to an unsigned long flag field.The upper word of the flag contains NDR data representation flags as defined by OSF DCE for floating point, endianess, and character representations. The lower word contains a marshaling context flag as defined by the COM channel. The exact layout of the flags within the field is described in The type_UserSize Function.

The *pMyObj* argument is a pointer to a user type object.The NDR engine frees the top level object. You are responsible for freeing any objects that the top-level object may point to.

## See Also

**Marshaling Rules for user_marshal and wire_marshal**, **wire_marshal**, **user_marshal**

## Marshaling Rules for user_marshal and wire_marshal

The OSF-DCE specification for marshaling embedded pointer types requires that you observe the following restrictions when you implement the <*type*>_UserSize, <*type*>_UserMarshal, and <*type*>_UserUnMarshal functions. (The rules and examples given here are for marshaling. However, your sizing and unmarshaling routines must follow the same restrictions):

- If the *wire-type* is a flat type with no pointers, your marshaling routine for the corresponding *userm-type* should simply marshal the data according to the layout of the *wire-type*. For example:

  ```
  typedef [wire_marshal (long)] void * HANDLE_HANDLE
  ```

  Note that the wire type, **long**, is a flat type. Your HANDLE_HANDLE_UserMarshal function marshals a long whenever a HANDLE_HANDLE object is passed to it.
- If the *wire-type* is a pointer to another type, your marshaling routine for the corresponding *userm-type* should marshal the data according to the layout for the type that the *wire-type* points to. The NDR engine takes care of the pointer. For example:

  ```
  typedef struct HDATA{
  long size;
  [size_is(size) long * pData;
  } HDATA;
  typedef HDATA * WIRE_TYPE;
  typedef [wire_marshal(WIRE_TYPE)] void * HANDLE_DATA
  ```

  Note that the wire type, WIRE_TYPE, is a pointer type. Your HANDLE_DATA_UserMarshal function marshals the data related to the handle, using the HDATA layout, rather than the HDATA * layout.
- A *wire-type* must be either a flat data type or a pointer type. If your transmissible type must be something else (a struct with pointers, for example), use a pointer to your desired type as the *wire-type*.

The effect of these restrictions is that the types defined with the **wire_marshal** or **user_marshal** attributes can be freely embedded in other types.

## See Also

**wire_marshal**, **user_marshal**, **The type_UserSize Function**, **The type_UserMarshal Function**, **The type_UserUnMarshal Function**, **The type_UserFree Function**

## Three Pointer Types

MIDL supports three types of pointers to accommodate a wide range of applications. The three different levels are called reference, unique, and full pointers, and are indicated by the attributes **ref**, **unique**, and **ptr**, respectively. The pointer classes described by these attributes are mutually exclusive.

Pointer attributes can be applied to pointers in type definitions, function return types, function parameters, members of structures or unions, or array elements.

Embedded pointers are pointers that are members of structures or unions or elements of arrays. Embedded pointers can differ from top-level pointers, depending upon directional attributes. In the **in** direction, embedded **ref** pointers are assumed to be pointing to valid storage and must not be null. This situation is recursively applicable to any **ref** pointers they are pointing to. In the **in** direction, embedded unique and full pointers may or may not be null.

Any pointer attribute placed on a parameter in the syntax of a function declaration affects only the rightmost pointer declarator for that parameter. To affect other pointer declarators, intermediate named types must be used.

Functions that return a pointer can have a pointer attribute as a function attribute. The **unique** and **ptr** attributes must be applied to function return types. Member declarations that are pointers can specify a pointer attribute as a field attribute. A pointer attribute can also be applied as a type attribute in **typedef** constructs.

When no pointer attribute is specified as a field or type attribute, pointer attributes are applied according to the rules for unattributed pointer declaration as follows:

In DCE-compatibility mode, pointer attributes are determined in the defining IDL file. If there is a **pointer_default** attribute specified in the defining interface, that attribute is used. If no **pointer_default** attribute is present, all unattributed pointers are full pointers.

In Microsoft-extensions mode, pointer attributes can be determined by importing IDL files and are applied in the following order:

1. An explicit pointer attribute applied at the use site.
2. The **ref** attribute, when the unattributed pointer is a top-level pointer parameter.
3. A **pointer_default** attribute specified in the defining interface.
4. A **pointer_default** attribute specified in the base interface.
5. The **unique** attribute.

The **pointer_default** interface attribute specifies the default pointer attributes to be applied to a pointer declarator in a type, parameter, or return type declaration when that declaration does not have an explicit pointer attribute applied to it. The **pointer_default** interface attribute does not apply to an unattributed top-level pointer of a parameter, which is assumed to be **ref**.

# Arrays and Pointers

Because RPC is designed to be transparent, you can expect a remote procedure call to behave just like a local procedure call. When a pointer is a parameter, the remote procedure can access the data object the pointer refers to in the same way a local procedure accesses it.

To achieve this transparency, the client stub transmits to the server both the pointer and the data object that it points to. If the remote procedure changes the data, the server must transmit the new data back to the client so the client can copy the new data over the original data.

The number of MIDL attributes relating to arrays and pointers demonstrates the flexibility that C affords. MIDL offers several attributes that extend C arrays and pointers to the distributed environment.

## Array Attributes

There is a close relationship between arrays and pointers in the C language. When passed as a parameter to a function, an array name is treated as a pointer to the first element of the array, as shown in the following example:

```
/* fragment */
extern void f1(char * p1);

void main(void)
{
    char chArray[MAXSIZE];

    fLocal1(chArray);
}
```

In a local call, you can use the pointer parameter to march through memory and examine the contents of other addresses:

```
/* dump memory (fragment) */
void fLocal1(char * pch1)
{
    int i;

    for (i = 0; i < MAXSIZE; i++)
       printf("%c ", *pch1++);
}
```

When a client passes a pointer to a remote procedure in C, the client stub transmits both the pointer and the data it points to. Unless the pointer is restricted to its corresponding data, all the client's memory must be transmitted with every remote call. By enforcing strong typing in the interface definition, MIDL limits client stub processing to the data that corresponds with the specified pointer.

The size of the array and the range of array elements transmitted to the remote computer can be constant or variable. When these values are variable, and thus determined at run time, you must use attributes in the IDL file to tell the stubs how many array elements to transmit. The following MIDL attributes support array bounds:

| Attribute | Description | Default |
|---|---|---|
| first_is | Index of the first array element transmitted. | 0 |
| last_is | Index of the last array element transmitted. | - |
| length_is | Total number of array elements transmitted. | - |
| max_is | Highest valid array index value. | - |
| min_is | Lowest valid array index value. | 0 |
| size_is | Total number of array elements allocated for the array. | - |

**Note** The **min_is** attribute is not implemented in Microsoft® RPC. The minimum array index is always treated as zero.

# The size_is Attribute

The **size_is** attribute is associated with an integer constant, expression, or variable that specifies the allocation size of the array. Consider a character array whose length is determined by user input:

```
/* IDL file */
[ uuid(20B309B1-015C-101A-B308-02608C4C9B53),
  version(2.0)
]
interface arraytest
{
void fArray2([in] short sSize,
             [in, out, size_is(sSize)] char achArray[*]);
}
```

The asterisk (*) that marks the placeholder for the variable-array dimension is optional.

The server stub must allocate memory on the server that corresponds to the memory on the client for that parameter. The variable that specifies the size must always be at least an **in** parameter. The **in** directional attribute is required so that the size value is defined on entry to the server stub. This size value provides information that the server stub requires to allocate the memory. The size parameter can also be **in, out**.

## See Also

Multiple Levels of Pointers

# The length_is Attribute

The **size_is** attribute lets you specify the maximum size of the array. When this is the only attribute, all elements of the array are transmitted. Instead of sending all elements of the array, you can specify the transmitted elements using the **length_is** attribute as:

```
/* IDL file */
[ uuid(20B309B1-015C-101A-B308-02608C4C9B53),
  version(3.0)
]
interface arraytest
{
void fArray3([in, out, size_is(sSize), length_is(sLen)] char achArray[],
             [in] short sSize,
             [in] short SLength);
}
```

Size describes allocation while length describes transmission. The number of elements transmitted must always be less than or equal to the number of elements allocated. The value associated with **length_is** is always less than or equal to **size_is**.

# The first_is and last_is Attributes

You can determine the number of transmitted elements by specifying the first and last elements. Use the **first_is** and **last_is** attributes as shown:

```
/* IDL file */
[ uuid(20B309B1-015C-101A-B308-02608C4C9B53),
  version(4.0)
]
interface arraytest
{

void fArray4([in, out,
              size_is(sSize),
              first_is(sFirst),
              last_is(sLast)]   char achArray[],
             [in] short sSize,
             [in] short sLast,
             [in] short sFirst) ;
}
```

# The max_is Attribute

You can specify the valid bounds of the array using the **max_is** attribute.

```
/* IDL file */
[ uuid(20B309B1-015C-101A-B308-02608C4C9B53),
  version(5.0)
]
interface arraytest
{
void fArray5([in] short sMax,
             [in, out, max_is(sMax)]  char achArray[]);
}
```

Field attributes can be supplied in various combinations as long as the stub can use the information to determine the size of the array and the number of bytes to transmit to the server. The relationships between the attributes are defined using the following formulas:

```
size_is = max_is + 1;
length_is = last_is - first_is + 1;
```

The values associated with the attributes must obey several common-sense rules based on those formulas. These are:

- The **first_is** index value cannot be smaller than zero and **last_is** cannot be greater than **max_is**.
- Do not specify a negative size for an array. Define the first and last elements so they result in a length value of zero or greater. Define the **max_is** value so the size is zero or greater. If MIDL was invoked with the /error bounds_check option, then the stub raises an exception when the size is less than zero, or the transmitted length is less than zero.
- You cannnot use the **length_is** and **last_is** attributes at the same time, nor can you use the **size_is** and **max_is** attributes at the same time.

Because of the close relationship in C between arrays and pointers, MIDL also lets you declare arrays in parameter lists using pointer notation. MIDL treats a parameter that is a pointer to a type as an array of that type if the parameter has any of the attributes commonly associated with arrays.

```
/* IDL file */
[ uuid(20B309B1-015C-101A-B308-02608C4C9B53)
  version(6.0)
]
interface arraytest
{
void fArray6([in, out, size_is(sSize)] char * p1,
             [in] short sSize);
void fArray7([in, out, size_is(sSize)] char achArray[],
             [in] short sSize);
}
```

In the preceding example, the array and pointer parameters in the functions **fArray6** and **fArray7** are equivalent.

# The string Attribute in Arrays

You can use the **string** attribute for one-dimensional character arrays, wide-character arrays, and byte arrays that represent text strings.

If you use the **string** attribute, the client stub uses the C-library functions **strlen** or **wstrlen** to count the number of characters in the string. To avoid possible inconsistencies, MIDL does not let you use the **string** attribute at the same time as the **first_is**, **last_is**, and **size_is** attributes.

As always with null-terminated strings in C, you must allow space for the null character at the end of the string. For example, when declaring a string that will hold up to 80 characters, allocate 81 characters.

```
/* IDL file */
[ uuid(20B309B1-015C-101A-B308-02608C4C9B53),
  version(8.0)
]
interface arraytest
{
void fArray8([in, out, string] char achArray[]);
}
```

## Multi-Dimensional Arrays

Array attributes can also be used with multidimensional arrays. However, be careful to ensure that every dimension of the array has a corresponding attribute. For example:

```
/* IDL file */
[ uuid(20B309B1-015C-101A-B308-02608C4C9B53),
  version(2.0)
]
void arr2d( [in] short        dlsize,
    [in] short                d2len,
    [in,
      size_is( dlsize, ),
      length_is ( , d2len) ] long    array2d[*][30] ) ;
```

The array shown above is a conformant array (of size *dlsize*) of 30 element arrays (with *d2len* elements shipped for each).

The **string** attribute can also be used with multidimensional arrays. The attribute applies to the least-significant dimension such as a conformant array of strings. You can also use multidimensional pointer attributes, but if you do so, the order of the attributes will be reserved because of the right-to-left behavior associated with pointers. For example:

```
/* IDL file */
[ uuid(20B309B1-015C-101A-B308-02608C4C9B53),
  version(2.0)
]
void arr2d( [in] short        d1len,
    [in] short                d2len,
       [in] size_is(d1len, d2len) ] long  ** ptr2d) ;
```

In the example above, the variable *ptr2d* is d1len pointers to d2len pointers to long.

Be sure that a multidimensional array is not equivalent to multiple levels of pointers. A multidimensional array is a single, large block of memory and should not be confused with an array of pointers. Also, ANSI C syntax allows only the most significant (leftmost) array dimension to be unspecified in a multidimensional array. Therefore, the following is a valid statement:

```
long a1[] [20]
```

Compare this to the following invalid statement:

```
long a1[20] []
```

## Pointers

It is very efficient to use pointers as C function parameters. The pointer costs only a few bytes and can be used to access a large amount of memory. However, in a distributed application, the client and server procedures can reside in different address spaces on different computers that may not have access to the same memory.

When one of the remote procedure's parameters is a pointer to an object, the client must transmit a copy of that object and its pointer to the server. If the remote procedure modifies the object through its pointer, the server returns the pointer and its modified copy.

MIDL offers pointer attributes to minimize the amount of required overhead and the size of your application. For example, you can specify a binary tree using the following definition:

```
typedef struct _treetype {
    long               lValue;
    struct _treetype * left;
    struct _treetype * right;
} TREETYPE;

TREETYPE * troot;
```

The contents of a tree node can be accessed by more than one pointer, thus making it more complicated for the RPC support code to manage the data and the pointers. The underlying stub code must resolve the various pointers to the addresses and determine whose copy of the data represents the most recent version.

The amount of processing can be reduced if you guarantee that your pointer is the only way the application can access that area of memory. The pointer can still have many of the features of a C pointer. For example, it can change between null and non-null values or stay the same. However, as long as the data referenced by the pointer is unique to the pointer, you can reduce the amount of processing by the stubs. To do this, designate such a pointer by using the **unique** attribute.

You can further reduce the complexity if you specify that the non-null pointer to an address of valid memory will not change during the remote call. However, the contents of memory can change and the data returned from the server will be written into this area on the client. To do this, designate such a pointer, known as a reference pointer, by using the **ref** attribute.

## Reference Pointers

Reference pointers are the simplest pointers and require the least amount of processing by the client stub. These pointers are mainly used to implement reference semantics and allow for **out** parameters in C.

In the following example, the value of the pointer does not change during the call, although the contents of the data at the address indicated by the pointer can change.

{ewc msdncd, EWGraphic, bsd23532 0 /a "SDK_A07.BMP"}

A reference pointer has the following characteristics:

- It always points to valid storage and never has the value NULL.
- It never changes during a call and always points to the same storage before and after the call.
- Data returned from the callee is written into the existing storage.
- The storage pointed to by a reference pointer cannot be accessed by any other pointer or any other name in the function.

## Unique Pointers

Unique pointers can change in value, but as with reference pointers, they do not cause aliasing of data – that is, the data that is accessible through the pointer is not accessible through any other name in the remote operation. This constraint saves a significant amount of processing.

The pointer itself can change from a null to a non-null value or from a non-null to a null value during the call. In the following example, the pointer is null before the call and points to a valid string after the call:

{ewc msdncd, EWGraphic, bsd23532 1 /a "SDK_A01.BMP"}

By default, the **unique** pointer attribute is applied to all pointers that are not parameters. In Microsoft-extensions mode, this default setting can be changed with the **pointer_default** attribute.

A unique pointer has the following characteristics:

- It can have the value NULL.
- It can change from null to non-null during the call. When the value changes to non-null, new memory is allocated on return.
- It can change from non-null to null during the call. When the value changes to NULL, the application is responsible for freeing the memory.
- If the value changes from one non-null value to another non-null value, the change is ignored.
- The storage that a unique pointer points to cannot be accessed by any other pointer or name in the operation.
- Return data is written into existing storage if the pointer does not have the value NULL.

## Full Pointers

Full pointers have all the properties of unique pointers. In addition, full pointers support aliasing. This means that multiple pointers can refer to the same data, as shown in the following figure:

{ewc msdncd, EWGraphic, bsd23532 2 /a "SDK_A02.BMP"}

## Pointers and Memory Allocation

The ability to change memory through pointers often requires that the server and the client allocate enough memory for the elements in the array.

Whenever a stub must allocate or free memory, it calls run-time library functions that in turn call the functions **midl_user_allocate** and **midl_user_free**. These functions are not included as part of the run-time library. You need to write your own versions of these functions and link them with your application. In this way, you can decide how to manage memory. (The exception is if you are compiling your IDL file in OSF-compatibility (/**osf**) mode, in which case you do not need to implement these functions)

You must write these functions to the following prototypes:

```
void __RPC_FAR * __RPC_API midl_user_allocate(size_t len)

void __RPC_API midl_user_free(void __RPC_FAR * ptr)
```

For example, the versions of these functions for an application can simply call standard library functions:

```
void __RPC_FAR * __RPC_API midl_user_allocate(size_t len)
{
    return(malloc(len));
}

void __RPC_API midl_user_free(void __RPC_FAR * ptr)
{
    free(ptr);
}
```

## Program Efficiency Using Pointer Parameters

The **in**, **out**, and **in, out** directional attributes significantly affect the amount of stub code when they are applied to pointer parameters.

## Default Pointer Types for Pointers

The MIDL compiler offers three different default cases for pointers that do not have pointer attributes at definition time:

- Function parameters that are top-level pointers default to **ref** pointers.
- Pointers embedded in structures and pointers to other pointers (non-top-level pointers) default to the type specified by the **pointer_default** attribute.
- When no **pointer_default** attribute is supplied, non-top-level pointers default to **unique** pointers in MIDL's default (Microsoft-extensions) mode, and to **ptr** in DCE-compatible mode.

A pointer returned by a function must be a unique or full pointer. The MIDL compiler reports an error if a function result is, either explicitly or by default, a reference pointer. The returned pointer always indicates new storage because there is not enough information for the stubs to determine whether there is existing storage in the caller address space for pointer-valued function results.

Functions that return a pointer value can specify a pointer attribute as a function attribute. If a pointer attribute is not present, the function return pointer uses the value provided by the pointer_default attribute.

**Note** To ensure unambiguous pointer-attribute behavior, always use explicit pointer attributes when defining a pointer.

## Pointer-Attribute Type Inheritance

According to the DCE specification, each IDL file must define attributes for its pointers. If an explicit attribute is not assigned to a pointer, the pointer uses the value specified by the **pointer_default** keyword. Some DCE implementations do not allow unattributed pointers. If a pointer does not have an explicit attribute, the IDL file must have a **pointer_default** specification so the pointer attribute can be set.

In default (Microsoft-extensions) mode, you can specify a pointer's attribute in the IDL file that imports the defining IDL file. Pointers defined in one IDL file can inherit attributes that are specified in other IDL files.

Also in default mode, IDL files can include unattributed pointers. If neither the base nor the imported IDL files specify a pointer attribute or **pointer_default**, unattributed pointers are interpreted as unique pointers.

The MIDL compiler assigns pointer attributes to pointers using the following priority rules (1 is highest):

1. Explicit pointer attributes explicitly applied to the pointer at the definition or use site.
2. **Pointer_default** attribute in the IDL file that defines the type.
3. **Pointer_default** attribute in the IDL file that imports the type.
4. **Ptr** (DCE-compatibility mode); **unique** (Microsoft-extensions mode).

# Using Arrays, Strings, and Pointers

The Doctor program (see \samples\rpc\doctor in the Win32 SDK) is a good example of an application that demonstrates the design trade-offs involved with array and string attributes. The Doctor program is a small sample version of a better-known program, Eliza, developed by artificial intelligence researcher Joseph Weizenbaum at MIT in the 1960s. The program mimics a therapist by using simple pattern-matching to respond to natural language input.

The Eliza program recorded the following session:

| User | Eliza |
|---|---|
| Men are all alike. | In what way? |
| They're always bugging us about something or other. | Can you think of a specific example? |
| Well, my boyfriend made me come here. | Your boyfriend made you come here. |
| He says I'm depressed much of the time. | I am sorry to hear that you are depressed. |
| It's true, I am unhappy. | Do you think coming here will help you not to be unhappy? |
| I need some help; that much seems certain. | What would it mean to you if you got some help? |
| Perhaps I could learn to get along with my mother. | Tell me more about your family. |
| My mother takes care of me. | Who else in your family takes care of you? |
| My father. | Your father. |
| You are like my father in some ways. | What resemblance do you see? |

The Doctor program can be split into client-side and server-side applications. The client side prompts the patient for input and displays the doctor's response. The server side processes the patient input and generates the doctor's response. This is a classic example of a client-server application: the client is responsible for user interaction while the server handles the extensive computational load. Not much data is passed to and returned by the function but, because the data can require a significant amount of processing, it is processed by the server.

The Doctor program uses a character array for input and returns another character array as output. The following topics demonstrate the design trade-offs between the various interfaces that can manage these parameters.

The table below lists four ways to pass character arrays between the client and server, and the attributes and functions needed to implement each approach.

| Approach | Attributes or functions |
|---|---|
| Counted character arrays | **size_is**, **length_is**, **ref** |
| Stub-managed strings | **string**, **ref**, **midl_user_allocate** on server |
| Stub-managed strings | **string**, **unique**, **midl_user_allocate** on client and server |

Function that returns a **unique** string

Within the constraints associated with these combinations of attributes, there are alternative ways of sending one character array from client to server and of returning another character array from server to client.

## Counted Character Arrays

The **size_is** attribute indicates the upper bound of the array while the **length_is** attribute indicates the number of array elements to transmit. In addition to the array, the remote procedure prototype must include any variables representing length or size that determine the transmitted array elements (they can be separate parameters or bundled with the string in a structure). These attributes can be used with wide-character or single-byte character arrays just as they would be with arrays of other types.

# in, out, size_is Prototype

The following function prototype uses a single-counted character array that is passed both ways: from client to server and from server to client:

```
#define STRSIZE 500 //maximum string length

void Analyze(
    [in, out, length_is(*pcbSize), size_is(STRSIZE)] char  achInOut[],
    [in, out]  long *pcbSize);
```

As an **in** parameter, achInOut must point to valid storage on the client side. The developer allocates memory associated with the array on the client side before making the remote procedure call.

The stubs use the **size_is** parameter STRSIZE to allocate memory on the server and then use the **length_is** parameter pcbSize to transmit the array elements into this memory. The developer must make sure the client code sets the **length_is** variable before calling the remote procedure:

```
/* client */
char achInOut[STRSIZE];
long cbSize;
...
gets(achInOut);                  // get patient input
cbSize = strlen(achInOut) + 1;   // transmit '\0' too
Analyze(achInOut, &cbSize);
```

In the previous example, the character array achInOut is also used as an **out** parameter. In C, the name of the array is equivalent to the use of a pointer. By default, all pointers are reference pointers – they do not change in value and they point to the same area of memory on the client before and after the call. All memory accessed by the remote procedure must fit the size specified on the client before the call or the stubs will generate an exception.

Before returning, the **Analyze** function on the server must reset the pcbSize variable to indicate the number of elements that the server will transmit to the client as shown:

```
/* server */
Analyze(char * str, long * pcbSize)
{
   ...
   *pcbSize = strlen(str) + 1; // transmit '\0' too
   return;
}
```

Instead of using a single string for both input and output, you may find it more efficient and flexible to use separate parameters.

# in, size_is and out, size_is Prototype

The following function prototype uses two counted strings. The developer must write code on both client and server to keep track of the character array lengths and pass parameters that tell the stubs how many array elements to transmit.

```
void Analyze(
    [in,  length_is(cbIn), size_is(STRSIZE)]    char  achIn[],
    [in]                                        long  cbIn,
    [out, length_is(*pcbOut), size_is(STRSIZE)] char  achOut[],
    [out]                                       long *pcbOut);
```

Note the parameters that describe the array length are transmitted in the same direction as the arrays: cbIn and achIn are **in** parameters while pcbOut and achOut are **out** parameters. As an **out** parameter, the parameter pcbOut must follow C convention and be declared as a pointer.

The client code counts the number of characters in the string, including the trailing zero, before calling the remote procedure as shown:

```
/* client */
char achIn[STRSIZE], achOut[STRSIZE];
long cbIn, cbOut;
...
gets(achIn);                    // get patient input
cbIn = strlen(achIn) + 1;       // transmitted elements
Analyze(achIn, cbIn, achOut, &cbOut);
```

The remote procedure on the server supplies the length of the return buffer in cbOut as shown:

```
/* server */
void Analyze(char *pchIn,
             long cbIn,
             char *pchOut,
             long *pcbOut)
{
   ...
   *pcbOut = strlen(pchOut) + 1; // transmitted elements
   return;
}
```

Knowing that the parameter is a string allows us to use the **string** attribute. This attribute directs the stub to calculate the string size, thus eliminating the overhead associated with the **size_is** parameters.

## Strings

The **string** attribute indicates the parameter is a pointer to an array of type **char**, **byte**, or **w_char**. As with a conformant array, the size of a **string** parameter is determined at run time. Unlike a conformant array, the developer does not have to provide the length associated with the array – the **string** attribute tells the stub to determine the array size by calling **strlen**. A **string** attribute cannot be used at the same time as the **length_is** or **last_is** attributes.

The **in, string** attribute combination directs the stub to pass the string from client to server only. The amount of memory allocated on the server is the same as the transmitted string size plus one.

The **out, string** attributes direct the stub to pass the string from server to client only. The call-by-value design of the C language insists that all **out** parameters must be pointers. (The key idea is that by passing the value of the address, the function can indirectly change the value stored at that address. If the value itself were passed, the function would only be able to modify its local copy of the value. For a more extensive explanation of the difference between call by value and call by reference, see any C language programming book published by Microsoft Press.)

The **out** parameter must be a pointer and, by default, all pointer parameters are reference pointers. The reference pointer does not change during the call – it points to the same memory as before the call. For string pointers, the additional constraint of the reference pointer means the client must allocate sufficient valid memory before making the remote procedure call. The stubs transmit the string indicated by the **out, string** attributes into the memory already allocated on the client side.

# in, out, string Prototype

The following function prototype uses a single **in, out, string** parameter for both the input and output strings. The string first contains patient input and is then overwritten with the doctor response as shown:

```
void Analyze([in, out, string, size_is(STRSIZE)] char  achInOut[]);
```

This example is similar to the one that employed a single-counted string for both input and output. As with that example, the **size_is** attribute determines the number of elements allocated on the server. The **string** attribute directs the stub to call **strlen** to determine the number of transmitted elements.

The client allocates all memory before the call as:

```
/* client */
char achInOut[STRSIZE];
...
gets(achInOut);            // get patient input
Analyze(achInOut);
printf("%s\n", achInOut);  // display doctor response
```

Note that the **Analyze** function no longer must calculate the length of the return string as it did in the counted-string example where the **string** attribute was not used. Now the length is calculated by the stubs as shown:

```
/* server */
void Analyze(char *pchInOut)
{
   ...
   Respond(response, pchInOut); // don't need to call strlen
   return;                      // stubs handle size
}
```

# in, string and out, string Prototype

The following function prototype uses two parameters: an **in, string** parameter and an **out, string** parameter.

```
void Analyze(
    [in, string]                        *pszInput,
    [out, string, size_is(STRSIZE)]     *pszOutput);
```

The first parameter is **in** only. This input string is only transmitted from the client to the server and is used as the basis for further processing by the server. The string is not modified and is not required again by the client, so it does not have to be returned to the client.

The second parameter, representing the doctor's response, is **out** only. This response string is only transmitted from the server to the client. The allocation size is provided so the server stubs can allocate memory for it. Because pszOutput is a **ref** pointer, the client must have sufficient memory allocated for the string before the call. The response string is written into this area of memory when the remote procedure returns.

## Multiple Levels of Pointers

You can use multiple pointers such as a **ref** pointer to another **ref** pointer that points to the character array as shown:

```
void Analyze(
   [in, string]                       char  *pszInput,
   [out, string, size_is(STRSIZE)]    char **ppszOutput);
```

When there are multiple levels of pointers, the attributes are associated with the pointer closest to the variable name. The client is still responsible for allocating any memory associated with the response.

The following example allows the stub to call the server without knowing in advance how much data will be returned:

```
[uuid( ...),
version(3.3),
pointer_default(unique)]  //required whenever you
                          // have pointers to pointers
                          // pointer has to be unique so
                          // that it can be NULL if
                          // necessary
HRESULT GetBars([out] long * pSize,
         [out, size_is( , *pSize)]
          BAR ** ppBar);//BAR type defined elsewhere
```

In this example, the stub passes the server a unique pointer, which the server initializes to NULL. The server then allocates a block of BARs, sets the pointer, sets the size argument and returns. Note that in order for the server to have an effect on the caller you must pass a [**ref**] pointer to a [**unique**] pointer to your data. Also note the comma in **size_is**( , *pSize ), which says that the top level pointer is *not* a sized pointer, but the lower level pointer *is* a sized pointer.

On the client side, the stub allocates the block, asigns the address to the ppBar argument and unmarshals BAR objects. The size of the block (and the number of unmarshaled BARs) is indicated by the size argument.

## See Also

**size_is**

# Pipes

The **pipe** type constructor is a highly efficient mechanism for passing large amounts of data, or any quantity of data that is not all available in memory at one time, during a remote procedure call. When you use a pipe, the RPC runtime handles the actual data transfer, thus you eliminate the overhead associated with repeated remote procedure calls.

You define a pipe type in the interface definition and then use that type as a parameter in remote procedure calls. You use the pipe with the **in** attribute parameter to *pull*, or transfer data from the client to the server. You use the pipe with the **out** parameter to *push*, or transfer data from the server to the client.

In relation to the input and output parameters, this *push* and *pull* terminology may seem counter-intuitive. It is helpful to keep in mind that the server application and the client stub are the active agents, working in concert to pull the data from the client application for an input parameter, and to push the data back to the client application for an output parameter. The server stub and the client application are passive parties to this exchange. The terms *push* and *pull*, then, describe the server application's action on the data.

The following topics discuss how to use the pipe type constructor in the IDL interface, the client application, and the server application. For more information on pipe syntax and restrictions, see pipe in the MIDL Language Reference.

## The Pipe Interface

When you define a pipe in an IDL file, the MIDL compiler generates a pipe control structure whose members are pointers to *push*, *pull*, and *alloc* procedures and a *state* variable that coordinates these procedures. The *state* variable is local to each side – that is, the client and server each maintain their own pipe state, by which their application code and stub code communicate.

The client application initializes the fields in the pipe control structure, maintains its *state* variable, and manages the data transfer with its own *push*, *pull*, and *alloc* functions, as described in the next topic. The client stub code calls these application functions in loops during data transfer. For an input pipe the client stub marshals the transfer data and transmits it to the server stub. For an output pipe, the client stub unmarshals the data into a buffer and passes a pointer to that buffer back to the client application.

The server stub code initializes the fields of the pipe control structure to a *state* variable and *push*, and *pull* routines. The server stub maintains the state and manages its private storage for the transfer data. The server application calls the *pull* and *push* routines in loops during the remote procedure call as it receives and unmarshals data from the client stub, or marshals and transmits data to the client stub.

In the following example, we define a pipe type LONG_PIPE, whose element size is defined as **long**. We also declare function prototypes for the remote procedure calls InPipe and OutPipe, to send and receive data, respectively.

### Example

```
//file: pipedemo.idl
typedef pipe long LONG_PIPE;
void InPipe( [in] LONG_PIPE pipe_data );
void OutPipe( [out] LONG_PIPE *pipe_data );
//end pipedemo.idl

//file: pipedemo.h (fragment)
typedef struct pipe_LONG_PIPE
    {
    void (__RPC_FAR * pull) (
        char __RPC_FAR * state,
        long __RPC_FAR * buf,
        unsigned long esize,
        unsigned long __RPC_FAR * ecount );
    void (__RPC_FAR * push) (
        char __RPC_FAR * state,
        long __RPC_FAR * buf,
        unsigned long ecount );
    void (__RPC_FAR * alloc) (
        char __RPC_FAR * state,
        unsigned long bsize,
        long __RPC_FAR * __RPC_FAR * buf,
        unsigned long __RPC_FAR * bcount );
    char __RPC_FAR * state;
    } LONG_PIPE;

void InPipe(
    /* [in] */ LONG_PIPE pipe_data);
void OutPipe(
    /* [out] */ LONG_PIPE __RPC_FAR *pipe_data);
//end pipedemo.h
```

## See Also

pipe, /Oi

## Client-Side Pipe Implementation

The client application must implement the following procedures, which the client stub will call during data transfer:

- A *pull* procedure (for an input pipe).
- A *push* procedure (for an output pipe).
- An *alloc* procedure to allocate a buffer for the transfer data.

All of these procedures must use the arguments specified by the MIDL-generated header file.

In addition, the client application must have a state variable to keep track of where to locate or place data.

The state variable can be as simple as a file handle, if you are transferring data from one file to another, or an integer that points to an element in an array, as shown in our example below. Or you can define a fairly complex state structure to perform additional tasks, such as coordinating the *push* and *pull* routines on an [**in**, **out**] parameter.

The *alloc* procedure can also be as simple or as complex as your application requires. For example, it can return a pointer to the same buffer every time every time the stub calls the function, or it can allocate a different amount of memory each time. If your data is already in the proper form (an array of pipe elements, for example) you can coordinate the *alloc* procedure with the *pull* procedure to allocate a buffer that already has the data in it. In that case, your *pull* procedure could be an empty routine.

Note that the the buffer allocation must be in bytes. The *push* and *pull* procedures, on the other hand, manipulate elements, whose size in bytes depends on how they were defined.

The *pull* procedure must find the data to be transferred, read the data into the buffer, and set the number of elements to send. When there is no more data to send, the procedure sets this argument to zero. When all the data is sent the *pull* procedure should do any needed cleanup before returning. For a parameter that is an **in**, **out** pipe, the *pull* procedure must reset the state variable after all the data has been transmitted, so that the *push* procedure can use it to receive data.

The *push* procedure takes a pointer to a buffer and an element count from the client stub and, if the element count is greater than 0, processes the data. For example, it could copy the data from the stub's buffer to its own own memory. When the element count equals zero, the *push* procedure completes any needed cleanup tasks before returning.

In the following example, the client functions SendLongs and ReceiveLongs each allocate a pipe structure and a global memory buffer, initialize the structure, make the remote procedure call and free the memory.

### Example:

```
//file: client.c (fragment)
#include "pipedemo.h"
long *  global_pipe_data;
long    global_buffer[BUF_SIZE];

ulong   pipe_data_index; /* state variable */

void SendLongs()
{
 LONG_PIPE   in_pipe;
 int        i;
 global_pipe_data =
```

```
        (long *)malloc( sizeof(long) * PIPE_SIZE );

 for (i=0; i<PIPE_SIZE; i++)
     global_pipe_data[i] = IN_VALUE;

 pipe_data_index = 0;
 in_pipe.state =  (rpc_ss_pipe_state_t )&pipe_data_index;
 in_pipe.pull  = PipePull;
 in_pipe.alloc = PipeAlloc;

 InPipe( in_pipe ); /* Make the rpc */

 free( (void *)global_pipe_data );

 return;
}//end SendLongs

void PipeAlloc( rpc_ss_pipe_state_t state_info,
                     ulong requested_size,
                     long **allocated_buf,
                     ulong *allocated_size )
{
 ulong *state = (ulong *)state_info;
if ( requested_size > (BUF_SIZE*sizeof(long)) )
   *allocated_size = BUF_SIZE * sizeof(long);
 else
   *allocated_size = requested_size;
 *allocated_buf = global_buffer;
return;
}//end PipeAlloc

void PipePull( rpc_ss_pipe_state_t state_info,
                    long *input_buffer,
                    ulong max_buf_size,
                    ulong *size_to_send )
{
 ulong   current_index;
 ulong        i;
 ulong        elements_to_read;
 ulong *state = (ulong *)state_info;

 current_index = *state;
 if (*state >=  PIPE_SIZE )
    {
     *size_to_send = 0; /* end of pipe data */
     *state = 0; /* Reset the state = global index */
     return;
    }

 if ( current_index + max_buf_size > PIPE_SIZE )
    elements_to_read = PIPE_SIZE - current_index;
 else
    elements_to_read = max_buf_size;

 for (i=0; i < elements_to_read; i++)
```

```
     /*client sends data */
     input_buffer[i] = global_pipe_data[i + current_index];

 *state +=   elements_to_read;
 *size_to_send = elements_to_read;

 return;
}//end PipePull


void ReceiveLongs()
{
 LONG_PIPE    *out_pipe;
 idl_long_int    i;

 global_pipe_data =
        (long *)malloc( sizeof(long) * PIPE_SIZE );

    pipe_data_index = 0;
    out_pipe.state =  (rpc_ss_pipe_state_t )&pipe_data_index;
    out_pipe.push  = PipePush;
    out_pipe.alloc = PipeAlloc;

    OutPipe( &out_pipe ); /* Make the rpc */

    free( (void *)global_pipe_data );

    return;
}//end ReceiveLongs()

void PipePush( rpc_ss_pipe_state_t state_info,
                    long *buffer,
                    ulong num_elts )
{
 ulong   elts_to_copy, i;
 ulong *state = (ulong *)state_info;

 if (num_elts == 0)/* end of data */
    {
     *state = 0; /* Reset the state = global index */
     return;
    }
 if (*state + num_elts > PIPE_SIZE)
     elts_to_copy = PIPE_SIZE - *state;
 else
     elts_to_copy = num_elts;

 for (i=0; i <elts_to_copy; i++)
     { /*client receives data */
     global_pipe_data[*state] = buffer[i];
     (*state)++;
     }
 return;
}//end PipePush
```

## See Also

pipe, /Oi

## Server-Side Pipe Implementation

The server application performs pipe transfer by calling the server stub's *pull* or *push* routine in a loop. Normal termination of the loop occurs when a zero-sized chunk of data is passed.

```
//file: server.c (fragment)
uc_server.c
#define PIPE_TRANSFER_SIZE    100 /* Transfer 100 pipe elements at one time
*/

void InPipe(LONG_PIPE     long_pipe )
{
    long   local_pipe_buf[PIPE_TRANSFER_SIZE];
    ulong   actual_transfer_count = PIPE_TRANSFER_SIZE;

    while(actual_transfer_count > 0) /* Loop to get all
                                        the pipe data elements */
    {
        long_pipe.pull(    long_pipe.state,
                           local_pipe_buf,
                           PIPE_TRANSFER_SIZE,
                           &actual_transfer_count);
        /* process the elements */
    } // end while
    return;
} //end InPipe

void OutPipe(LONG_PIPE *long_pipe )
{
 long   *long_pipe_data;
 ulong   index = 0;
 ulong   elts_to_send = PIPE_TRANSFER_SIZE;

/* Allocate memory for the data to be passed back in the pipe */
 long_pipe_data = (long *)malloc( sizeof(long) * PIPE_SIZE );

 while(elts_to_send >0) /* Loop to send pipe data elements */
 {
  if (index >= PIPE_SIZE)
     elts_to_send = 0;
  else
    {
     if ( (index + PIPE_TRANSFER_SIZE) > PIPE_SIZE )
          elts_to_send = PIPE_SIZE - index;
     else
          elts_to_send = PIPE_TRANSFER_SIZE;
    }

  long_pipe->push( long_pipe->state,
                   &(long_pipe_data[index]),
                   elts_to_send );
  index += elts_to_send;

  } //end while
```

```
  free((void *)long_pipe_data);

  return;
}
```

## See Also

**pipe**, **/Oi**

## Rules for Multiple Pipes

You can combine **in**, **out**, and **in**,**out** pipe parameters in any combination in a single call but you must process the pipes in a specific order, as shown in the following example:

- Get the data from every input pipe, starting with the first (leftmost) **in** parameter, and continuing in order, draining each pipe before beginning to process the next.
- After every input pipe has been completely processed, send the data for the output pipes, again starting with the first **out** parameter, and continuing in order, filling each pipe before beginning to process the next.

```
//in .IDL file:
void InOutUCharPipe( [in,out] UCHAR_PIPE *uchar_pipe_1,
                     [out] UCHAR_PIPE * uchar_pipe_2,
                     [in] UCHAR_PIPE uchar_pipe_3);

//remote procedure:
void InOutUCharPipe( UCHAR_PIPE *param1,
                     UCHAR_PIPE *param2,
                     UCHAR_PIPE  param3)
{
while(!END_OF_PIPE1){
  param1->pull (. . .)
. . .
};
while(!END_OF_PIPE3){
  param3.pull (. . .)
. . .
};
while(!END_OF_PIPE1){
  param1->push (. . .)
. . .
};
while(!END_OF_PIPE2){
  param2->push(. . .)
. . .
};
return;
} //end InOutUCharPipe
```

## Combining Pipe and Non-pipe Parameters

When you combine pipe types and other types in a remote procedure call, the data is transmitted differently depending on the direction of the parameter:

- In the **in** direction, the data for all non-pipe arguments will be transmitted first, followed by pipe data.
- In the **out** direction, the server sends the pipe data first. After the manager routine returns, the server transmits the non-pipe data.
- When there are **in**,**out** pipe arguments combined with **in**,**out** non-pipe arguments, first the input data is transmitted in its entirety, as described above. Then the output data is transmitted as described above.

The following restriction applies to this (MIDL 3.0) implementation of pipes: When you combine pipe types and other types in a single remote procedure call, the non-pipe parameters must have a well-defined size in order to allow the MIDL compiler to calculate the buffer size needed. For example, you cannot combine pipe parameters with a **unique** pointer or a conformant structure, since their sizes cannot be determined at compile time.

### See Also

**pipe**, **/Oi**

# Binding and Handles

Binding is the process of creating a logical connection between a client and a server that the client uses to make remote procedure calls to that server. The binding between client and server is represented by a data structure called a binding handle.

A binding handle is analogous to a file handle returned by the **fopen** C run-time library function or a window handle returned by the function **CreateWindow**. As with these handles, the binding handle is opaque; your application cannot use it to directly access and manipulate data about that binding. This handle is a pointer or index into a data structure that is available only to the RPC run-time libraries. You provide the handle and the run-time libraries access the appropriate data.

The client obtains a handle by calling RPC run-time functions that bind to the server, or by supplying a name or UUID to a service that provides the corresponding handle.

The handles managed by an application can be classified into two broad categories: context handles and binding handles. Context handles are used to maintain state information while binding handles contain only information about the binding. Note that a serialization application also manages serialization handles, but these are not binding handles. See Encoding Services for additional information on serialization handles.

Every binding handle is either primitive or user defined, according to its data type. In addition to being primitive or user defined, every handle is either implicit or explicit, according to the way your application specifies the handle for each remote procedure call. These types combine to specify four kinds of binding handles:

{ewc msdncd, EWGraphic, bsd23534 0 /a "SDK_A12.BMP"}

This section defines the characteristics of RPC binding handles and demonstrates their use in sample applications.

**Note** In addition to binding handles, Microsoft RPC also supports serialization handles used to encode or decode data. These are used for serialization on a local computer and do not involve remote binding. For additional information on serialization handles, see Encoding Services.

## Binding

The server registers its interface and then listens for requests from clients. Clients bind to the server by making calls to the RPC run-time functions. The most significant distinction between handle types is whether the application or the stub makes the calls to the RPC run-time functions to manage the binding handle.

There are two basic types of binding:

- Auto binding
- Application-managed binding

When you use auto binding and auto handles, the stubs automatically call the correct sequence of functions and the application will not be able to access the handle at all.

When you use application-managed binding, the client application explicitly calls a sequence of run-time functions to obtain a valid handle. Besides auto handles, the application-managed binding category includes all the other types of handles: primitive, user-defined, and context handles.

The following figure shows this categorization of binding handles:

{ewc msdncd, EWGraphic, bsd23534 1 /a "SDK_A03.BMP"}

## Application-Initiated Binding

Applications bind to the server and obtain a handle that is used by the stubs to make remote procedure calls. When the client is finished making remote calls, the application can unbind from the server and invalidate the handle. A client application that manages its own binding and handles can obtain a handle in two ways:

- Call **RpcBindingFromStringBinding**
- Call the name-service functions **RpcNsBindingImportBegin**, **RpcNsBindingImportNext**, and **RpcNsBindingImportDone**

When the client explicitly calls **RpcBindingFromStringBinding**, the client must supply the following information to identify the server:

- The globally unique identifier (GUID) or UUID of the object.
- The transport type to communicate over, such as named pipes or TCP/IP.
- The network address, which is the server name for the named-pipe transport.
- The endpoint, which contains the pipe name for the named-pipe transport.

(The object UUID and the endpoint information are optional.)

The client or client stub communicates this identifying information to the RPC run-time library by means of a data structure called the string binding, which combines these elements using a specified syntax.

In the following examples, the pszNetworkAddress parameter and other parameters that include embedded backslashes can appear strange at first glance. Because the backslash is an escape character in the C programming language, two backslashes are needed to represent each single literal backslash character. The string-binding data structure must contain four backslash characters to represent the two literal backslash characters that precede the server name. The following example shows eight backslashes so four literal backslash characters will appear in the string-binding data structure after processing by the **sprintf** function. For example:

```
/* client application */

char * pszUuid = "6B29FC40-CA47-1067-B31D-00DD010662DA";
char * pszProtocol = "ncacn_np";
char * pszNetworkAddress = "\\\\\\\\servername";
char * pszEndpoint = "\\\\pipe\\\\pipename";
char * pszString;

int len = 0;

len  = sprintf(pszString, "%s", pszUuid);
len += sprintf(pszString + len, "@%s:", pszProtocolSequence);
if (pszNetworkAddress != NULL)
    len += sprintf(pszString + len, "%s", pszNetworkAddress);
len += sprintf(pszString + len, "[%s]", pszEndpoint);
```

In the following example, the string binding appears as:

```
6B29FC40-CA47-1067-B31D-00DD010662DA@ncacn_np:\\\\servername[\\pipe\\
pipename]
```

The client then obtains the binding handle by calling **RpcBindingFromStringBinding**:

```
RPC_BINDING_HANDLE hBinding;

status = RpcBindingFromStringBinding(pszString, &hBinding);
...
```

A convenience function, **RpcStringBindingCompose**, assembles the object UUID, protocol sequence, network address, and endpoint into the correct syntax for the call to **RpcBindingFromStringBinding**. You do not have to worry about putting the ampersand, colon, and the various components for each protocol sequence in the right place; you just supply the strings as parameters to the function. The run-time library even allocates the memory needed for the string binding. For example:

```
char * pszNetworkAddress = "\\\\server";
char * pszEndpoint = "\\pipe\\pipename";
status = RpcStringBindingCompose(
            pszUuid,
            pszProtocolSequence,
            pszNetworkAddress,
            pszEndpoint,
            pszOptions,
            &pszString);
...
status = RpcBindingFromStringBinding(
            pszString,
            &hBinding);
...
```

Another convenience function, **RpcBindingToStringBinding**, takes a binding handle as input and produces the corresponding string binding.

## String Bindings

The string binding is a character string that consists of several sub-strings. The strings in a string binding represent the object UUID, the protocol sequence, the network address, the endpoint, and the endpoint options.

The object UUID is a unique identifier. The protocol sequence is a string that represents the RPC network-communications protocol. The protocol sequence also determines network-address and endpoint-naming conventions. For example, the protocol sequence **ncacn_ip_tcp** indicates a connection-based NCA connection over TCP/IP. For more information about protocol sequences, see Specifying the Protocol Sequence or the reference entry for **PROTSEQ**.

The network address indicates the server name and the endpoint indicates a communication port at that server.

The client application can itself combine these substrings into the correct string-binding syntax, or it can call the function **RpcStringBindingCompose**. After a client calls **RpcStringBindingCompose**, it calls **RpcBindingFromStringBinding** to obtain the binding handle. For a complete description of the required syntax, see **String Binding**.

Most distributed applications should use the name-service functions instead of the string binding to obtain the binding handle. The name-service functions allow your server application to register its interface and object UUIDs, network address, and endpoint under a single logical name. These functions provide location independence and ease of administration.

## Binding-Handle Types

MIDL provides several types of handles. In this way, you can select the handle type that is best suited for your application. (See Encoding Services for additional information on using a primitive handle as a serializing handle.)

Handle characteristics:

- Handles can be parameters that are passed to the remote procedure, or they can be global data structures that do not appear in the remote function prototype.
- You can declare handles of the primitive handle type **handle_t**, or you can declare a handle type packaged in structures with other data.
- Some handles are invisible to the client application and are completely managed by the stubs while others are declared, defined, and managed by the application.
- A special type of handle, the context handle, allows you to maintain state information on the server in addition to acting as a binding handle.

The following table summarizes MIDL handle types:

| Handle type | Characteristics |
|---|---|
| Primitive | A handle of the predefined type **handle_t**. Note that serializing handles (which are not binding handles) are also of the type **handle_t**. See Serialization Handles for more information. |
| Explicit | A handle used as a parameter to the remote procedure. The explicit handle usually appears as the first parameter for compatibility with DCE. |
| Implicit | A handle defined in the generated header file as a global variable that is available to the stubs. The developer defines the handle in the ACF only and does not include the handle as a parameter to the remote procedure call. |
| User-defined | A handle of the primitive type **handle_t** that is created by a user-supplied function that converts the user-defined data to the handle. |
| Auto | A handle that is automatically generated by the MIDL compiler and managed by the client run-time library. The client stub manages the binding and the handle; the client application does not require any explicit code to manage the binding or the handle. |
| Context | A handle that includes information about the state of the server. The context handle is automatically associated with specific user-defined functions on the server. |

Handle characteristics can be combined in several ways to produce such types as explicit primitive, explicit user-defined, implicit primitive, and implicit user-defined handles, depending upon your application needs.

## Primitive Handles

A primitive handle is a handle with the data type **handle_t**. Ultimately, every handle is mapped to a primitive handle by the stubs.

As with a file handle or a window handle, a primitive handle is opaque; it contains information that is meaningful to the RPC run-time library but not to your application.

The primitive handle is defined in the client source code as a handle of the base type **handle_t** using a statement such as:

```
handle_t hMyHandle;     // primitive handle
```

For encoding or decoding data, the handle_t data type is used for a serialization handle. For additional information on serialization handles, see Encoding Services.

## Explicit Handles

An explicit handle is a handle that the client application specifies explicitly as a parameter to each remote procedure call. To conform to the OSF standard, the handle must be specified as the first parameter on each remote procedure. You create an explicit handle by declaring the handle as a parameter to the remote operations in the IDL file. The Hello, World example can be redefined to use an explicit handle as shown:

```
/* IDL file for explicit handles */

[ uuid(20B309B1-015C-101A-B308-02608C4C9B53),
  version(1.0)
]
interface hello
{
void HelloProc([in] handle_t h1,
               [in, string] char *  pszString);
}
```

## Implicit Handles

An implicit handle is a handle that is stored in a global variable. You usually initialize the handle and then do not refer to it again until you destroy the binding. Each remote procedure call with an explicit binding-handle parameter uses the implicit handle. You create an implicit handle by specifying the **implicit_handle** attribute in the ACF for an interface as:

```
/* ACF file (complete) */

[implicit_handle(handle_t hHello)
]
interface hello
{
}
```

The application uses the implicit handle only as a parameter to the RPC functions. The implicit handle is not used as a parameter to the remote procedure call. For example:

```
status = RpcBindingFromStringBinding(
            pszStringBinding,
            &hHello);
...
status = RpcBindingFree(hHello);
...
```

## User-Defined Handles

A user-defined handle, also called a customized or generic handle, is a handle of a user-defined data type. You create a user-defined handle when you specify the **handle** attribute on a type definition in your IDL file.

You must also supply bind and unbind routines that the client stub calls at the beginning and end of each remote procedure call. The bind and unbind routines use the following function prototypes:

| Function prototype | Description |
|---|---|
| **handle_t** *type*_**bind**(*type*) | Binding routine |
| **void** *type*_**unbind**(*type*, **handle_t**) | Unbinding routine |

The following example shows how the user-defined handle is defined in the IDL file:

```
/* usrdef.idl */
[uuid(20B309B1-015C-101A-B308-02608C4C9B53),
version(1.0),
pointer_default(unique)
]
interface usrdef
{
typedef struct _DATA_TYPE {
    unsigned char * pszUuid;
    unsigned char * pszProtocolSequence;
    unsigned char * pszNetworkAddress;
    unsigned char * pszEndpoint;
    unsigned char * pszOptions;
} DATA_TYPE;

typedef [handle] DATA_TYPE * DATA_HANDLE_TYPE;
void UsrdefProc(
                [in] DATA_HANDLE_TYPE  hBinding,
                [in, string] unsigned char *   pszString);

void Shutdown([in] DATA_HANDLE_TYPE hBinding);
}
```

The user-defined bind and unbind routines appear in the client application. In the following example, the bind routine converts the string-binding information to a binding handle by calling **RpcBindingFromStringBinding**. The unbind routine frees the binding handle by calling **RpcBindingFree**.

The name of the user-defined binding handle, DATA_HANDLE_TYPE, appears as part of the name of the functions and appears as the parameter type in the function parameters as:

```
/* This _bind routine is called by the client stub at the */
/* beginning of each remote procedure call                */

RPC_BINDING_HANDLE __RPC_USER DATA_HANDLE_TYPE_bind(DATA_HANDLE_TYPE dh1)
{
    RPC_BINDING_HANDLE hBinding;
    RPC_STATUS status;
```

```
unsigned char *pszStringBinding;

    status = RpcStringBindingCompose(
          dh1.pszUuid,
          dh1.pszProtocolSequence,
          dh1.pszNetworkAddress,
          dh1.pszEndpoint,
          dh1.pszOptions,
          &pszStringBinding);
          ...

    status = RpcBindingFromStringBinding(
          pszStringBinding,
          &hBinding);
          ...

    status = RpcStringFree(&pszStringBinding);
    ...

    return(hBinding);
}

/* This _unbind routine is called by the client stub at the end */
/* after each remote procedure call.                            */
void __RPC_USER DATA_HANDLE_TYPE_unbind(DATA_HANDLE_TYPE dh1,
                                        RPC_BINDING_HANDLE h1)
{
    RPC_STATUS status;
    status = RpcBindingFree(&h1);
    ...
}
```

## Auto Handles

Auto handles are useful when the application does not require a specific server and when it does not need to maintain any state information between the client and server. When you use an auto handle, you do not have to write any client application code to deal with binding and handles – you simply specify the use of the auto handle in the ACF. The stub then defines the handle and manages the binding.

For example, a time-stamp operation can be implemented using an auto handle. It makes no difference to the client application which server provides it with the time stamp because it can accept the time from any available server.

You specify the use of auto handles by including the **auto_handle** attribute in the ACF. The time-stamp example uses the following ACF:

```
/* ACF file */
[auto_handle]
interface autoh
{
}
```

**Note** Auto handles are not supported for the Macintosh platform.

The auto handle is used by default when the ACF does not include any other handle attribute and when the remote procedures do not use explicit handles. The auto handle is also used by default when the ACF is not present.

The remote procedures are specified in the IDL file. The auto handle must not appear as an argument to the remote procedure. For example:

```
/* IDL file */
[ uuid (6B29FC40-CA47-1067-B31D-00DD010662DA),
  version(1.0),
  pointer_default(unique)
]
interface autoh
{
void GetTime([out] long * time);
void Shutdown(void);
}
```

The benefit of the auto handle is that the developer does not have to write any code to manage the handle; the stubs manage the binding automatically. This is significantly different from the Hello, World example, where the client manages the implicit primitive handle defined in the ACF and must call several run-time functions to establish the binding handle.

Here, the stubs do all the work and the client only needs to include the generated header file AUTO.H to obtain the function prototypes for the remote procedures. The client application calls to the remote procedures appear just as if they were calls to local procedures as shown:

```
/* auto handle client application (fragment) */

#include <stdio.h>
#include <time.h>
#include "auto.h"    // header file generated by the MIDL compiler
```

```
void main(int argc, char **argv)
{
    time_t t1;
    time_t t2;
    char * pszTime;
    ...

    RpcTryExcept {
        GetTime(&t1);  // GetTime is a remote procedure
        GetTime(&t2);

        pszTime = ctime(&t1);
        printf("time 1= %s\n", pszTime);

        pszTime = ctime(&t2);
        printf("time 2= %s\n", pszTime);

        Shutdown();    // Shutdown is a remote procedure
    }
    RPCExcept(1)    {
    ...
    }
    RPCEndExcept

    exit(0);
}
```

The client application does not have to make any explicit calls to the client run-time functions. Those calls are managed by the client stub.

The server side of the application that uses auto handles must call the function **RpcNsBindingExport** to make binding information about the server available to clients. The auto handle requires a location service running on a server that is accessible to the client. The Microsoft implementation of the name service, the Microsoft Locator, manages auto handles. The server calls the following run-time functions:

```
/* auto handle server application (fragment) */

#include "auto.h" //header file generated by the MIDL compiler

void main(void)
{
    RpcUseProtseqEp(...);
    RpcServerRegisterIf(...);
    RpcServerInqBindings(...);
    RpcNsBindingExport(...);
    ...
}
```

The calls to the first two functions are similar to the Hello, World example; these functions make information about the binding available to the client. The calls to the **RpcServerInqBindings** and **RpcNsBindingExport** functions put the information in the name-service database. The call to **RpcServerInqBindings** fills the vector with valid data before the call to the export function. After the data has been exported to the database, the client (or client stubs) can call **RpcNsBindingImportBegin** and

**RpcNsBindingImportNext** to obtain this information.

The calls to **RpcServerInqBindings** and **RpcNsBindingExport** and their associated data structures appear as:

```
RPC_BINDING_VECTOR * pBindingVector;
RPCSTATUS status;

status = RpcServerInqBindings(&pBindingVector);

status = RpcNsBindingExport(
                fNameSyntaxType,      // name syntax type
                pszAutoEntryName,     // nsi entry name
                autoh_ServerIfHandle, // if server handle
                pBindingVector,       // set in previous call
                NULL);                // UUID vector
```

Note that the **RpcServerInqBindings** parameter &pBindingVector is a pointer to a pointer to **RPC_BINDING_VECTOR**.

The previous example shows the parameters to the **RpcNsBindingExport** function that should be used with the Microsoft Locator. As already mentioned, this locator is the Microsoft implementation of the name-service functions provided with Microsoft RPC. For more information about the Microsoft Locator, see **Run-time RPC Functions**.

To remove the exported interface from the name-service database completely, the server calls **RpcNsBindingUnexport** as shown:

```
status = RpcNsBindingUnexport(
                fNameSyntaxType,
                pszAutoEntryName,
                auto_ServerIfHandle,
                NULL);              // unexport handles only
```

The **unexport** function should be used only when the service is being permanently removed. It should not be used when the service is temporarily disabled, such as when the server is shut down for maintenance. A service can be registered with the name-service database, yet be unavailable because the server is temporarily offline. The client application should contain exception-handling code for such a condition.

The calls to the remote procedures are surrounded by the exception-handling code. For more information about exception handling, see Run-time RPC Functions.

## Microsoft RPC Binding-Handle Extensions

The Microsoft extensions to the IDL language support multiple handle parameters and handle parameters that appear in positions other than the first, leftmost, parameter.

The following table describes the sequence of steps that the MIDL compiler goes through to resolve the binding-handle parameter in DCE-compatibility mode (/**osf**) and in default (Microsoft-extended) mode:

| | **DCE-compatibility mode** | | **default mode** |
|---|---|---|---|
| 1. | Binding handle that appears in first parameter position | 1. | Leftmost explicit binding handle |
| 2. | Leftmost **in**, **context_handle** parameter | 2. | Implicit binding handle specified by **implicit_handle** or **auto_handle** |
| 3. | Implicit binding handle specified by **implicit_handle** or **auto_handle** | 3. | If no ACF present, default to use of **auto_handle** |
| 4. | If no ACF present, default to use of **auto_handle** | | |

DCE IDL compilers look for an explicit binding handle as the first parameter. When the first parameter is not a binding handle and one or more context handles are specified, the leftmost context handle is used as the binding handle. When the first parameter is not a handle and there are no context handles, the procedure uses implicit binding using the ACF attribute **implicit_handle** or **auto_handle**.

The Microsoft extensions to the IDL allows the binding handle to be in a position other than the first parameter. The leftmost **in** explicit-handle parameter, whether it is a primitive, user-defined, or context handle, is the binding handle. When there are no handle parameters, the procedure uses implicit binding using the ACF attribute **implicit_handle** or **auto_handle**.

The following rules apply to both DCE-compatibility (/**osf**) mode and default mode:

- Auto-handle binding is used when no ACF is present.
- Explicit **in** or **in**, **out** handles for an individual function pre-empt any implicit binding specified for the interface.
- Multiple **in** or **in**, **out** primitive handles are not supported.
- Multiple **in** or **in**, **out** explicit context handles are allowed.
- All user-defined handle parameters except the binding-handle parameter are treated as transmissible data.

The following table contains examples and describes how the binding handles are assigned in each compiler mode:

| **Example** | **Description** |
|---|---|
| `void proc1( void );` | No explicit handle is specified. The implicit binding handle, specified by **implicit_handle** or **auto_handle**, is used. When no ACF is present, an auto handle is used. |
| `void proc2( [in] handle_t H,` | An explicit handle of type **handle_t** is specified. The |

| | |
|---|---|
| `[in] short s );` | parameter H is the binding handle for the procedure. |
| `void proc3( [in] short s,`<br>`[in] handle_t H );` | The first parameter is not a handle.<br>In default mode, the leftmost handle parameter, H, is the binding handle. In /**osf** mode, implicit binding is used. An error is reported because the second parameter is expected to be transmissible, and **handle_t** cannot be transmitted. |
| `typedef [handle] short * MY_HDL;`<br><br>`void proc1([in] short s,`<br>`[in] MY_HDL H );` | The first parameter is not a handle. In default mode, the leftmost handle parameter, H, is the binding handle. The stubs call the user-supplied routines MY_HDL_bind and MY_HDL_unbind. In/**osf** mode, implicit binding is used. The user-defined handle parameter H is treated as transmissible data. |
| `typedef [handle] short * MY_HDL;`<br><br>`void proc1([in] MY_HDL H,`<br>`[in] MY_HDL p );` | The first parameter is a binding handle. The parameter H is the binding-handle parameter. The second user-defined handle parameter is treated as transmissible data. |
| `typedef [context_handle] void * CTXT_HDL;`<br><br>`void proc1([in] short s,`<br>`[in] long l,`<br>`[in] CTXT_HDL H ,`<br>`[in] char c);` | The binding handle is a context handle. The parameter H is the binding handle. |

## Binding-Handle Use by Function

The following table contains the list of RPC run-time routines that operate on binding handles and specifies the type of binding handle allowed:

| Routine | Input argument | Output argument |
|---|---|---|
| RpcBindingCopy | Server | Server |
| RpcBindingFree | Server | None |
| RpcBindingFromStringBinding | None | Server |
| RpcBindingInqAuthClient | Client | None |
| RpcBindingInqAuthInfo | Server | None |
| RpcBindingInqObject | Server or client | None |
| RpcBindingReset | Server | None |
| RpcBindingSetAuthInfo | Server | None |
| RpcBindingSetObject | Server | None |
| RpcBindingToStringBinding | Server or client | None |
| RpcBindingVectorFree | Server | None |
| RpcNsBindingExport | Server | None |
| RpcNsBindingImportNext | None | Server |
| RpcNsBindingLookupNext | None | Server |
| RpcNsBindingSelect | Server | Server |
| RpcServerInqBindings | None | Server |

## Context Handles

A context handle contains context information created and returned by the server. Every application that uses a context handle must also specify an alternate method of binding because an initial binding must be used before the server can return a context handle.

You can create a context handle by specifying the **context_handle** attribute on a data-type definition in the IDL file. A context handle can also be associated with a special function called the context rundown routine, which is called by the server run-time library whenever an active binding to a client is broken unexpectedly.

In an interface that uses a context handle, if you do not also specify a primary implicit handle to contain the initial binding, the MIDL compiler generates an auto handle for you. It also generates the code in the client stub to perform auto binding.

For example, a file handle represents state information; it keeps track of the current location in the file. The file-handle parameter to a remote procedure call is packaged as a context handle. To begin, define a structure that contains the file name and the file handle as shown:

```
/* cxhndlp.c (fragment) */
typedef struct {
     FILE * hFile;
     char   achFile[256];
} FILE_CONTEXT_TYPE;
```

The IDL file defines the handle as a **void** * type and casts it to the required type on the server:

```
/* cxhndl.idl (fragment) */
typedef [context_handle] void * PCONTEXT_HANDLE_TYPE;
typedef [ref] PCONTEXT_HANDLE_TYPE * PPCONTEXT_HANDLE_TYPE;
```

The first remote procedure call initializes the handle and sets it to a non-null value. You must define the context with an **out** directional attribute in the IDL file:

```
/* cxhndl.idl (fragment) */
short RemoteOpen([out] PPCONTEXT_HANDLE_TYPE pphContext,
                 [in, string] unsigned char * pszFile);
```

The remote procedure **RemoteOpen** opens a file on the server:

```
/* cxhndlp.c (fragment)*/
short RemoteOpen(PPCONTEXT_HANDLE_TYPE pphContext,
         unsigned char         *pszFileName)
{
    FILE               *hFile;
    FILE_CONTEXT_TYPE  *pFileContext;

    if ((hFile = fopen(pszFileName, "r")) == NULL) {
        *pphContext = (PCONTEXT_HANDLE_TYPE) NULL;
    return(-1);
    }
    else {
        pFileContext = (FILE_CONTEXT_TYPE *)
                       midl_user_allocate(sizeof(FILE_CONTEXT_TYPE));
        pFileContext->hFile = hFile;
```

```
        strcpy(pFileContext->achFile, pszFileName);
        *pphContext = (PCONTEXT_HANDLE_TYPE) pFileContext;
    return(0);
    }
}
```

After the client calls **RemoteOpen**, the context handle contains valid data and is used as the binding handle. The client can free the explicit handle used to launch the context handle:

```
/* cxhndlc.c  (fragment)*/
printf("Calling the remote procedure RemoteOpen\n");
if (RemoteOpen(&phContext, pszFileName) < 0) {
    printf("Unable to open %s\n", pszFileName);
    Shutdown();
    exit(2);
}

/* Now the context handle also manages the binding. */
status = RpcBindingFree(&hStarter);
printf("RpcBindingFree returned 0x%x\n", status);
if (status)
    exit(status);
```

After the **RemoteOpen** function returns a valid, non-null context handle, subsequent calls use the context handle as an **in** pointer:

```
/* cxhndl.idl (fragment)*/
short RemoteRead(
                [in] PCONTEXT_HANDLE_TYPE phContext,
                [out] unsigned char  achBuf[BUFSIZE],
                [out] short * pcbBuf);

short RemoteClose([in, out] PPCONTEXT_HANDLE_TYPE pphContext);
```

The client application reads the file until it encounters the end of the file; it then closes the file. The context handle appears as a parameter in the **RemoteRead** and **RemoteClose** functions as:

```
/* cxhndlc.c (fragment)*/
printf("Calling the remote procedure RemoteRead\n");
while (RemoteRead(phContext, pbBuf, &cbRead) > 0) {
    for (i = 0; i < cbRead; i++)
        putchar(*(pbBuf+i));
}

printf("Calling the remote procedure RemoteClose\n");
if (RemoteClose(&phContext) < 0 ) {
    printf("Close failed on %s\n", pszFileName);
    exit(2);
}
```

## See also

**context_handle**

## Server Context Rundown Routine

If communication breaks down while the server is maintaining context on behalf of the client, a cleanup routine may be needed to reset the context information. This cleanup routine is called a "context rundown routine."

The context rundown routine is optional and, when supplied, is called when the client terminates without requesting that the server free the context. This can occur when the client does not close the context handle, or when the client terminates abnormally.

When no context rundown routine is needed, the **context_handle** attribute can be applied to parameters. When a context rundown routine is needed, the **context_handle** attribute must be used in a type definition.

The type name determines the name of the context rundown routine. Given a context handle of type *type-id*, the server application must supply the context rundown routine named *type-id*_**rundown**. The signature of the routine is shown below:

**void** __RPC_USER *type-id*_**rundown** (*type-id*);

When the server terminates the context and fails to return a null context handle, the context rundown routine is not called and memory allocated by the run-time library for the maintenance of the context is not released.

## Client Context Reset

When the server becomes unavailable and the client application wants to reset its context data, the client calls the RPC function **RpcSsDestroyClientContext**.

## Multi-threaded Clients and Context Handles

When you have a multi-threaded client where multiple threads are using the same context handle, the calls will be serialized at the server. This saves the server manager from having to guard against another thread from the same client changing the context or from the context running down while a call is dispatched. However, in certain cases serialization may create deadlock. For example, consider the following sequence:

Thread 1 : Gets a context handle and makes a call. This call blocks on some synchronization event sitting on the server.

Thread 2 : Makes a call to the same server, using the same context handle. This call is intended to trigger the event thread 1 is blocking on. Because the calls are serialized, the event is never triggered.

If you have a situation like this you can use the **RpcSsDontSerializeContext** function to disable serialization. Be aware, however, that a call to this routine affects the entire process and is unrevertable.

## Summary of Binding and Handles

Binding is the process of making a logical connection from a client to a server and a handle is a data structure that represents a binding. It is analogous to a file handle or a window handle.

There are two principal types of binding: automatic and application managed. Auto binding requires a locator service on the server and does not maintain state information between client and server. Application-managed binding is controlled using the string-binding data structure or the name service to obtain a handle.

Context handles maintain state information on the server. The server can supply a context rundown routine which is called whenever an active binding to a client is broken unexpectedly.

If you use a context handle and do not specify a primary implicit handle, the MIDL compiler generates an auto handle to be used for the initial binding. It also generates the code in the client stub to perform auto binding.

Serialization handles are primitive handles used for data serialization. They cannot be used for binding.

# Memory Management

With RPC, a single conceptual execution thread can be processed by two or more processing threads. These processing threads can run on the same computer or on different computers. RPC depends on the ability to simulate the client thread's address space in the server thread's address space and to return data, including new and changed data, from the server to the client memory.

Memory management in the context of RPC involves:

- How the memory needed to simulate a single conceptual address space is allocated and deallocated in the different address spaces of the client and server's threads.
- Which software component is responsible for managing memory – the application or the MIDL-generated stub.
- MIDL attributes that affect memory management: directional attributes, pointer attributes, array attributes, and the ACF attributes **byte_count**, **allocate**, and **enable_allocate**.

As a developer, you can choose among several methods for selecting the way that memory is allocated and freed. Consider a complex data structure, such as a linked list or a tree, that consists of nodes connected with pointers. You can apply attributes that select the following models:

- Node-by-node allocation and deallocation.
- A single, linear buffer for the entire tree allocated by the stub.
- A single, linear buffer for the entire tree allocated by the client application.
- Persistent storage on the server.

Each of these models is described in detail in this chapter.

This section does not describe the use of different Intel-architecture memory models. For information about using different Intel-architecture memory models, see Building RPC Applications.

## How Memory Is Allocated and Deallocated

Typically, stub code generated by the MIDL compiler calls user-supplied functions to allocate and free memory. These functions, named **midl_user_allocate** and **midl_user_free**, must be supplied by the developer and linked with the application.

All applications must supply implementations of **midl_user_allocate** and **midl_user_free**, even though the names of these functions may not appear explicitly in the stubs. The only exception is if you are compiling in OSF-compatibility (/**osf**) mode.

These user-supplied functions must match a specific, defined, function prototype, but otherwise can be implemented in any way that is convenient or useful for the application.

## midl_user_allocate

**void __RPC_FAR * __RPC_USER midl_user_allocate (size_t** *cBytes***);**

## Parameters

*cBytes*

Specifies the count of bytes to allocate.

Both client applications and server applications must implement the **midl_user_allocate** function, unless you are compiling in OSF-compatibility (/**osf**) mode. Applications and generated stubs call **midl_user_allocate** directly or indirectly to manage allocated objects. For example:

- The client and server applications should call **midl_user_allocate** to allocate memory for the application, such as when creating a new node.
- The server stub calls **midl_user_allocate** when unmarshalling data into the server address space.
- The client stub calls **midl_user_allocate** when unmarshalling data from the server that is referenced by an **out** pointer. Note that for **in**, **out**, **unique** pointers, the client stub calls **midl_user_allocate** only if the **unique** pointer value was NULL on input and changes to a non-null value during the call. If the **unique** pointer was non-null on input, the client stub writes the associated data into existing memory.

If **midl_user_allocate** fails to allocate memory, it should return a null pointer or raise a user-defined exception.

The **midl_user_allocate** function should return a pointer as shown:

- For Windows NT running on Intel platforms, the pointer is 4 bytes aligned.
- For Windows NT running on MIPS and Alpha platforms, the pointer is 8 bytes aligned.
- For Windows 95, the pointer is 4 bytes aligned.
- For Windows 3.*x* and MS-DOS platforms, the pointer is 2 bytes aligned.

For example, the sample programs provided with the Win32 SDK implement **midl_user_allocate** in terms of the C function **malloc**:

```
void __RPC_FAR * __RPC_USER midl_user_allocate(size_t cBytes)
{
    return((void __RPC_FAR *) malloc(cBytes));
}
```

**Note** If the **Rpcss** package is enabled (for example, as the result of using the **enable_allocate** attribute), **RpcSmAllocate** should be used to allocate memory on the server side. For additional information on **enable_allocate**, see MIDL Reference.

## midl_user_free

**void __RPC_USER midl_user_free(void __RPC_FAR** * *pBuffer*);

## Parameters

*pBuffer*

Specifies a pointer to the memory that is to be freed.

Both client application and server application must implement the **midl_user_free** function, unless you are compiling in OSF-compatibility (/**osf**) mode. The **midl_user_free** function must be able to free all storage allocated by **midl_user_allocate**.

Applications and stubs call **midl_user_free** when dealing with allocated objects. For example:

- The server application should call **midl_user_free** to free memory allocated by the application, such as when deleting a pointed-at node.
- The server stub calls **midl_user_free** to release memory on the server after marshalling all **out** arguments, **in**, **out** arguments, and the function return value.

For example, the RPC Win32 sample program that displays "Hello, world" implements **midl_user_free** in terms of the C function **free**:

```
void __RPC_USER midl_user_free(void __RPC_FAR * p)
{
    free(p);
}
```

**Note** If the **Rpcss** package is enabled (for example, as the result of using the **enable_allocate** attribute), **RpcSmFree** can be used to free memory. See **Rpcss Memory Management Model** for more information.

## Memory-Management Models

A developer can choose from among several methods that select how memory is allocated and freed. Consider a complex data structure, such as a linked list or tree, that consists of nodes connected with pointers. You can apply attributes that select the following models:

- Node-by-node allocation and deallocation.
- A single linear buffer allocated by the stub for the entire tree.
- A single linear buffer allocated by the client application for the entire tree.
- Persistent storage on the server.
- TheRpcss Memory Management Model.

Each of these models is described in detail in the following topics.

## Node-by-Node Allocation and Deallocation

Node-by-node allocation and deallocation by the stubs is the default method of memory management for all parameters on both the client and the server. On the client side, the stub allocates each node with a separate call to **midl_user_allocate**. On the server side, rather than calling **midl_user_allocate**, private memory is used whenever possible. If **midl_user_allocate** is called, the server stubs will call **midl_user_free** to free the data. In most cases, using node-by-node allocation and deallocation instead of using **allocate (all_nodes)** will result in increased performance of the server side stubs.

## Stub-Allocated Buffers

Rather than forcing a distinct call for each node of the tree or graph, you can direct the stubs to compute the size of the data and to allocate and free memory by making a single call to **midl_user_allocate** or **midl_user_free**. The ACF attribute **allocate(all_nodes)** directs the stubs to allocate or free all nodes in a single call to the user-supplied memory-management functions.

For example, consider the following binary tree data structure:

```
/* IDL file fragment */
typedef struct _TREE_TYPE {
    short sNumber;
    struct _TREE_TYPE * pLeft;
    struct _TREE_TYPE * pRight;
} TREE_TYPE;

typedef TREE_TYPE * P_TREE_TYPE;
```

The ACF attribute **allocate(all_nodes)** applied to this data type appears in the **typedef** declaration in the ACF as:

```
/* ACF file fragment */
typedef [allocate(all_nodes)] P_TREE_TYPE;
```

The **allocate** attribute can only be applied to pointer types. The **allocate** ACF attribute is a Microsoft extension to DCE IDL and, as such, is not available if you compile with the MIDL /**osf** switch. When **allocate(all_nodes)** is applied to a pointer type, the stubs generated by the MIDL compiler traverse the specified data structure to determine the allocation size. The stubs then make a single call to allocate the entire amount of memory needed by the graph or tree. A client application can free memory much more efficiently by making a single call to **midl_user_free**. However, server stub performance is generally increased when using node-by-node memory allocation because the server stubs can often use private memory that requires no allocations.

For additional information, see Node-by-Node Allocation and Deallocation.

## Application-Allocated Buffer

The ACF attribute **byte_count** directs the stubs to use a preallocated buffer that is not allocated or freed by the client support routines. The **byte_count** attribute is applied to a pointer or array parameter that points to the buffer. It requires a parameter that specifies the buffer size in bytes.

The client-allocated memory area can contain complex data structures with multiple pointers. Because the memory area is contiguous, the application does not have to make many calls to individually free each pointer and structure. Like the **allocate(all_nodes)** attribute, the memory area can be allocated or freed with one call to the memory-allocation routine or the free routine. However, unlike the **allocate(all_nodes)** attribute, the buffer parameter is not managed by the client stub but by the client application.

The buffer must be an **out**-only parameter andthe buffer length in bytes must be an **in**-only parameter.

The **byte_count** attribute can only be applied to pointer types. The **byte_count** ACF attribute is a Microsoft extension to DCE IDL and, as such, is not available if you compile using the MIDL /**osf** switch.

In the following example, the parameter *pRoot* uses byte count:

```
/* function prototype in IDL file (fragment) */
void SortNames(
    [in] short cNames,
    [in, size_is(cNames)] STRINGTYPE pszArray[],
    [in] short cBytes,
    [out, ref] P_TREE_TYPE pRoot  /* tree with sorted data */
);
```

The **byte_count** attribute appears in the ACF as:

```
/* ACF file (fragment) */
SortNames([byte_count(cBytes)] pRoot);
```

The client stub generated from these IDL and ACF files does not allocate or free the memory for this buffer. The server stub allocates and frees the buffer in a single call using the provided size parameter. If the data is too large for the specified buffer size, an exception is raised.

## Persistent Storage on the Server

You can optimize your application so the server stub does not free memory on the server at the conclusion of a remote procedure call. For example, when a context handle will be manipulated by several remote procedures, you can use the ACF attribute **allocate(dont_free)** to retain the allocated memory on the server.

The **allocate(dont_free)** attribute is added to the ACF **typedef** declaration in the ACF. For example:

```
/* ACF file fragment */
typedef [allocate(all_nodes, dont_free)] P_TREE_TYPE;
```

When the **allocate(dont_free)** attribute is specified, the tree data structure is allocated, but not freed, by the server stub. When you make the pointers to such persistent data areas available to other routines – for example, by copying the pointers to global variables – the retained data is accessible to other server functions. The **allocate(dont_free)** attribute is particularly useful for maintaining persistent pointer structures as part of the server state information associated with a context-handle type.

## Rpcss Memory Management Model

The Rpcss package is the recommended memory management model and provides the best overall stub performance for memory management. The default allocator/deallocator pair used by the stubs and run time when allocating memory on behalf of the application is **midl_user_allocate**/**midl_user_free**. However, you can choose the Rpcss package instead of the default by using the ACF attribute **enable_allocate**.

In **/osf** mode, the Rpcss package is enabled for MIDL-generated stubs automatically whenever full pointers are used, whenever the arguments require memory allocation, or as a result of using the **enable_allocate** attribute. In default (Microsoft extended) mode, the Rpcss package is enabled only when the **enable_allocate** attribute is used. The **enable_allocate** attribute enables the Rpcss environment by the server side stubs. The client side becomes alerted to the possibility that the Rpcss package may be enabled. In /**osf** mode, the client side is not affected.

When the Rpcss package is enabled, allocation of memory on the server side is accomplished with the private Rpcss memory management allocator and deallocator pair. You can allocate memory using the same mechanism by calling **RpcSmAllocate** (or **RpcSsAllocate**). Upon return from the server stub, all the memory allocated by the Rpcss package is automatically freed. The following example shows how to enable the Rpcss package:

```
/* ACF file fragment */

[ implicit_handle(handle_t GlobalHandle),
    enable_allocate
]
{
}

/*Server management routine fragment. Replaces p=midl_user_allocate(size);
*/

    p=RpcSsAllocate(size);                /*raises exception */
    p=RpcSmAllocate(size, &status);       /*returns error code */
```

You can also enable the memory management environment for your application by calling the **RpcSmEnableAllocate** routine (and can disable it by calling the **RpcSmDisableAllocate** routine). Once enabled, application code can allocate and deallocate memory by calling functions from the **RpcSs*** or **RpcSm*** package.

## Who Manages Memory?

Generally, the stubs are responsible for packaging and unpackaging data, allocating and freeing memory, and transferring the data to and from memory. In some cases, however, the application is responsible for allocating and freeing memory. The following factors determine which component is responsible for memory management:

- Whether the pointer is a top-level **ref** parameter or whether the pointer is embedded within another structure.
- Directional attributes applied to the parameter.
- Pointer attributes applied to the parameter.
- Function return values.

## Top-Level and Embedded Pointers

When discussing how pointers and their associated data elements are allocated in Microsoft RPC, you have to differentiate between top-level pointers and embedded pointers. It is also useful to refer to the set of all pointers that are not top-level pointers.

Top-level pointers are those that are specified as the names of parameters in function prototypes. Top-level pointers and their referents are always allocated on the server. Embedded pointers are pointers that are embedded in data structures such as arrays, structures, and unions.

When embedded pointers are **out**-only and null on input, the server application can change their values to non-null. In this case, the client stubs allocate new memory for this data.

If the embedded pointer is not null on the client before the call, the stubs do not allocate memory on the client on return. Instead, the stubs attempt to write the memory associated with the embedded pointer into the existing memory on the client associated with that pointer, overwriting the data already there.

**Out**-only embedded pointers are discussed in Combining Pointer and Directional Attributes.

The term *non-top-level pointers* refers to all pointers that are not specified as parameter names in the function prototype, including both embedded pointers and multiple levels of nested pointers.

## Directional Attributes Applied to the Parameter

The directional attributes **in** and **out** determine how the client and server allocate and free memory. The following table summarizes the effect of directional attributes on memory allocation:

| Directional attribute | Memory on client | Memory on server |
|---|---|---|
| **in** | Client application must allocate before call. | Server stub allocates. |
| **out** | Client stub allocates on return. | Server stub allocates top-level pointer only; server application must allocate all embedded pointers. Server also allocates new data as needed. |
| **in**, **out** | Client application must allocate initial data transmitted to server; client stub allocates additional data. | Server stub allocates; server application allocates new data as needed. |

The following table summarizes the effect of directional attributes on memory deallocation:

| Directional attribute | Memory on client | Memory on server |
|---|---|---|
| (all cases) | Not freed. | Freed by server stubs on return (subject to ACF attribute **allocate**). |

Note that for **out**-only parameters, MIDL allocates only the memory required for the top-level pointer parameter. The generated stub does not chase, or dereference, subsequent pointers that are part of the **out**-only data structure. The server application must allocate and initialize all such pointers.

## Length, Size, and Directional Attributes

The size-related attributes **max_is** and **size_is** determine how many array elements the server stub allocates on the server.

The length-related attributes **length_is**, **first_is**, and **last_is** determine how many elements are transmitted to both the server and the client.

Different directional attribute(s) can be applied to a declarator and the parameter specified by a field attribute. However, some combinations of different directional attributes can cause errors when they are applied to the declarator and to the field attribute parameter.

As an example, consider a procedure with two parameters, an array, and the transmitted length of the array. The italicized term *dir_attr* refers to the directional attribute applied to the parameter as:

```
Proc1(
    [dir_attr] short * plength;
    [dir_attr, length_is(pLength)] short array[MAX_SIZE]);
```

The MIDL compiler behavior for each combination of directional attributes is described below:

| Array | Length parameter | Stub actions during call from client to server | Stub actions on return from server to client |
|---|---|---|---|
| **in** | **in** | Transmit the length and the number of elements indicated by the parameter. | No data transmitted. |
| **in** | **out** | Not legal; MIDL compiler error. | Not legal; MIDL compiler error. |
| **in** | **in**, **out** | Transmit the length and the number of elements indicated by the length parameter. | Transmit the length only. |
| **out** | **in** | Transmit the length.<br>If array size is fixed, allocate the array size on the server, but transmit no elements.<br>If array size is not bound, not legal: MIDL compiler error. | Transmit the number of elements indicated by the length.<br>Note that the length can be changed and can have a different value from the value on the client. Do not transmit the length. |
| **out** | **out** | Allocate space for the length parameter on the server but do not transmit the parameter.<br>If the array size is fixed, allocate the array size on the server, but transmit no elements.<br>If array size is not fixed, not legal: MIDL compiler error. | Transmit the length and the number of elements indicated by the length as set by the server application. |
| **out** | **in**, **out** | Transmit the length parameter. | Transmit the length.<br>Transmit the number of |

| | | | |
|---|---|---|---|
| | | If the array size is bound, allocate the array size on the server, but transmit no elements.<br>If array size is not bound, not legal: MIDL compiler error. | array elements indicated by the length. |
| **in**, **out** | **in** | Transmit the length and the number of elements indicated by the parameter. | Do not transmit the length.<br>Transmit the number of elements indicated by the length.<br>Note that the length can be changed and can have a different value from the original value on the client. |
| **in**, **out** | **out** | Not legal; MIDL compiler error. | Not legal; MIDL compiler error. |
| **in**, **out** | **in**, **out** | Transmit the length and the number of elements indicated by the parameter. | Transmit the length and the number of elements indicated by the parameter. |

In general, you should not modify the length or size parameters on the server side. If you change the length parameter, you can orphan memory. For more information, see Memory Orphaning.

## Pointer Attributes Applied to the Parameter

Each pointer attribute (**ref**, **unique**, and **ptr**) has characteristics that affect memory allocation. The following table summarizes these characteristics:

| Pointer attribute | Client | Server |
|---|---|---|
| Reference (**ref**) | Client application must allocate. | Special handling needed for for **out**-only non-top-level pointers. |
| Unique (**unique**) | If a parameter, then client application must allocate; if embedded, can be null.<br>Changing from null to non-null causes client stub to allocate; changing from non-null to null can cause orphaning. | |
| Full (**ptr**) | If a parameter, client application must allocate; if embedded, can be null.<br>Changing from null to non-null causes client stub to allocate; changing from non-null to null can cause orphaning. | |

The **ref** attribute indicates that the pointer points to valid memory. By definition, the client application must allocate all the memory the reference pointers require.

The unique pointer can change from null to non-null. If the unique pointer changes from null to non-null, new memory is allocated on the client. If the unique pointer changes from non-null to null, orphaning can occur. For more information, see Memory Orphaning.

## Combining Pointer and Directional Attributes

A few caveats apply to certain combinations of directional attributes and pointer attributes.

## Embedded out-Only Reference Pointers

When you use **out**-only reference pointers in Microsoft RPC, the generated server stubs allocate only the first level of pointers accessible from the reference pointer. Pointers at deeper levels are not allocated by the stubs, but must be allocated by the server application layer.

For example, consider an **out**-only array of reference pointers:

```
/* IDL file (fragment) */
typedef [ref] short * PREF;

Proc1([out] PREF array[10]);
```

In the preceding example, the server stub allocates memory for ten pointers and sets the value of each pointer to null. The server application must allocate the memory for the ten **short** integers that are referenced by the pointers and must set the ten pointers to point to the integers.

When the **out**-only data structure includes nested reference pointers, the server stubs allocate only the first pointer accessible from the reference pointer. For example:

```
/* IDL file (fragment) */
typedef struct {
    [ref] small * psValue;
} STRUCT1_TYPE;

typedef struct {
    [ref] STRUCT1_TYPE * ps1;
} STRUCT_TOP_TYPE;

Proc2([out, ref] STRUCT_TOP_TYPE * psTop);
```

In the preceding example, the server stubs allocate the pointer psTop and the structure STRUCT_TOP_TYPE. The reference pointer ps1 in STRUCT_TOP_TYPE is set to null. The server stub does not allocate every level of the data structure, nor does it allocate the STRUCT1_TYPE or its embedded pointer, psValue.

## out-Only Unique or Full Pointer Parameters Not Accepted

**Out**-only unique or full pointers are not accepted by the MIDL compiler. Such specifications cause the MIDL compiler to generate an error message.

The automatically generated server stub has to allocate memory for the pointer referent so the server application can store data in that memory area. According to the definition of an **out**-only parameter, no information about the parameter is transmitted from client to server. In the case of a unique pointer, which can take the value NULL, the server stub does not have enough information to correctly duplicate the unique pointer in the server's address space, nor does the stub have any information about whether the pointer should point to a valid address or whether it should be set to NULL. Therefore, this combination is not allowed.

Rather than **out**, **unique** or **out**, **ptr** pointers, use **in**, **out**, **unique** or **in**, **out**, **ptr** pointers, or use another level of indirection such as a reference pointer that points to the valid unique or full pointer.

## Function Return Values

Function return values are similar to **out**-only parameters because their data is not provided by the client application. However they are managed differently. Unlike **out**-only parameters, they are not required to be pointers. The remote procedure can return any valid data type except ref pointers and nonencapsulated unions.

Function return values that are pointer types are allocated by the client stub with a call to **midl_user_allocate**. Accordingly, only the unique or full pointer attribute can be applied to a pointer function-return type.

## Memory Orphaning

When your distributed application uses an **in**, **out**, **unique** or **in**, **out**, **ptr** pointer parameter, the server side of the application can change the value of the pointer parameter to NULL. When the server subsequently returns the null value to the client, memory referenced by the pointer before the remote procedure call is still present on the client side, but is no longer accessible from that pointer (except in the case of an aliased full pointer). This memory is said to be *orphaned*.

Memory can also be orphaned whenever the server changes an embedded pointer to a null value. For example, if the parameter points to a complex data structure such as a tree, the server side of the application can delete nodes of the tree.

Another situation that can lead to a memory leak involves conformant, varying, and open arrays containing pointers. When the server application modifies the parameter that specifies the array size or transmitted range so that it represents a smaller value, the stubs use the smaller value(s) to free memory. The array elements with indices larger than the size parameter are orphaned. Your application must free elements outside the transmitted range.

Repeated orphaning of memory on the client without freeing the unused memory can lead to a situation where the client runs out of available memory resources.

## Summary of Memory Allocation Rules

The following table summarizes key rules regarding memory allocation:

| MIDL element | Description |
|---|---|
| Top-level **ref** pointers | Must be non-null pointers. |
| Function return value | New memory is always allocated for pointer return values. |
| **unique**, **out** or **ptr**, **out** pointer | Not allowed by MIDL. |
| Non-top-level **unique**, **in**, **out** or **ptr**, **in**, **out** pointer that changes from null to non-null | Client stubs allocate new memory on client on return. |
| Non-top-level **unique**, **in**, **out** pointer that changes from non-null to null | Memory is orphaned on client; client application is responsible for freeing memory and preventing leaks. |
| Non-top-level **ptr**, **in**, **out** pointer that changes from non-null to null | Memory will be orphaned on client if not aliased; client application is responsible for freeing and preventing memory leaks in this case. |
| **ref** pointers | Client-application layer usually allocates. |
| Non-null **in**, **out** pointer | Stubs attempt to write into existing storage on client. If **string** and size increases beyond size allocated on the client, it will cause a GP-fault on return. |

The following table summarizes the effects of key IDL and ACF attributes on memory management:

| MIDL feature | Client issues | Server issues |
|---|---|---|
| **allocate(single_node)**, **allocate(all_nodes)** | Determines whether one or many calls are made to the memory functions. | Same as client, except private memory can often be used for **allocate (single_node) [in]** and **[in**,**out]** data. |
| **allocate(free)** or **allocate(dont_free)** | (None; affects server) | Determines whether memory on the server is freed after each remote procedure call. |
| array attributes **max_is** and **size_is** | (None; affects server) | Determines size of memory to be allocated. |
| **byte_count** | Client must allocate buffer; not allocated or freed by client stubs. | ACF parameter attribute determines size of buffer allocated on server. |
| **enable_allocate** | Usually, none. However, the client may be using a different memory management environment. | Server uses a different memory management environment. **RpcSmAllocate** should be used for allocations. |

| | | |
|---|---|---|
| **in** attribute | Client application responsible for allocating memory for data. | Allocated on server by stubs. |
| **out** attribute | Allocated on client by stubs. | **out**-only pointer must be **ref** pointer; allocated on server by stubs. |
| **ref** attribute | Memory referenced by pointer must be allocated by client application. | Top-level and first-level reference pointers managed by stubs. |
| **unique** attribute | Non-null to null can result in orphaned memory; null to non-null causes client stub to call **midl_user_allocate**. | (Affects client) |
| **ptr** attribute | (See **unique**) | (See **unique**) |

# Encoding Services

Microsoft RPC supports two methods for encoding and decoding, or "serializing," data. You can serialize on a procedure or type basis. Serialization means that the data is marshalled to and unmarshalled from buffers that you control. This differs from the traditional usage of RPC in which the stubs and the RPC run-time library have full control of the marshalling buffers and the process is transparent to you. You can use the buffer for storage on a permanent media, encryption, and so on. When encoding, the data is marshalled to a buffer and the buffer is passed to you. When decoding, you supply a marshalling buffer with data in it and the data is unmarshalled from the buffer to memory.

When you use procedure serialization, MIDL generates a serialization stub for the procedure decorated with serialization attributes. When you call this routine, you execute a serialization call instead of a remote call. The procedure arguments are marshalled to or unmarshalled from a buffer in the usual way and you control the buffers.

In contrast, when type serialization occurs (a type is labelled with serialization attributes), MIDL generates routines to size, encode, and decode objects of that type. To serialize data, you must call these routines in the appropriate way. Type serialization is a Microsoft extension and, as such, is not available when you compile in DCE-compatibility (**/osf**) mode. By using the **encode** and **decode** attributes as interface attributes, RPC applies encoding to all the types and routines defined in the IDL file.

**Note** You must supply adequately aligned buffers when using encoding services. The beginning of the buffer must be aligned at 8. For procedure serialization, each procedure call must marshal into or unmarshal from a buffer position aligned at 8. For type serialization, sizing, encoding, and decoding must start at a position aligned at 8.

## Procedure Encoding and Decoding

When you use procedure encoding and decoding, a procedure, rather than a type, is labeled with the **encode** and/or **decode** attribute. Instead of generating the usual remote stub, the compiler generates a serialization stub for the routine.

Just as a remote procedure must use a binding handle to make a remote call, a serialization procedure must use an encoding handle to use encoding services. If an encoding handle is not specified, a default implicit encoding handle is used to direct the call. On the other hand, if the encoding handle is specified, either as an explicit **handle_t** argument of the routine or by using the **explicit_handle** attribute, the developer must pass a valid handle as an argument of the call. For additional information on how to create a valid serialization handle, see Serialization Handles, Examples of Fixed Buffer Encoding, and Examples of Incremental Encoding.

Microsoft RPC allows for remote and serialization procedures to be mixed in one interface. However, use caution when doing so. For implicit handles, the global implicit handle must be set to a valid binding handle before a remote call, and to a valid encoding or decoding handle before a serialization call.

## Type Encoding and Decoding

The MIDL compiler generates up to three functions for each type to which the **encode** or **decode** attribute is applied. For example, for a user-defined type named **MyType**, the compiler generates code for the **MyType_Encode**, **MyType_Decode**, and **MyType_AlignSize** functions. For these functions, the compiler writes prototypes to STUB.H and source code to STUB_C.C. Generally, you can encode a *MyType* object with **MyType_Encode** and decode an object from the buffer using **MyType_Decode**. **MyType_AlignSize** is used if you need to know the size of the marshalling buffer prior to allocating it.

The following encoding function is generated by the MIDL compiler. It serializes the data for the object pointed to by *pObject* and the buffer is obtained according to the method specified in the handle. After writing the serialized data to the buffer, you control the buffer. Note that the handle inherits the status from the previous calls and the buffers must be aligned at 8.

For an implicit handle:

```
void MyType_Encode (MyType __RPC_FAR * pObject);
```

For an explicit handle:

```
void MyType_Encode (handle_t Handle, MyType __RPC_FAR * pObject);
```

The following function deserializes the data from the application's storage into the object pointed to by *pObject*. You supply a marshalled buffer according to the method specified in the handle. Note that the handle may inherit the status from the previous calls and the buffers must be aligned at 8.

For an implicit handle:

```
void MyType_Decode (MyType __RPC_FAR * pObject);
```

For an explicit handle:

```
void MyType_Decode (handle_t Handle, MyType __RPC_FAR * pObject);
```

The following function returns the sum of the size in bytes of the type instance plus any padding bytes needed to align the data. This enables serializing a set of instances of the same or different types into a buffer while ensuring that the data for each object is appropriately aligned. **MyType_AlignSize** assumes that the instance pointed to by *pObject* will be marshalled into a buffer beginning at the offset aligned at 8.

For an implicit handle:

```
size_t MyType_AlignSize (MyType __RPC_FAR * pObject);
```

For an explicit handle:

```
size_t MyType_AlignSize (handle_t Handle, MyType __RPC_FAR * pObject);
```

## Serialization Handles

An application uses the serializing procedures or the serializing support routines generated by the MIDL compiler in conjunction with a set of library functions to manipulate an encoding-services handle. Together, these functions provide a mechanism for customizing the way an application serializes data. For example, instead of using several I/O operations to serialize a group of objects to a stream, an application can optimize performance by serializing several objects of different types into a buffer and then writing the entire buffer in a single operation. The functions that manipulate serialization handles are independent of the type of serialization you are using.

A serializing handle is required for any serializing operation and all serializing handles must be managed explicitly by you. To do this, you firstu create a valid handle with a call to one of the **Mes*HandleCreate** routines. Then, after the operation is complete, you release the handle with a call to **MesHandleFree**. Once the handle has been created or re-initialized, it represents a valid serialization context and can be used to encode or decode, depending on the type of the handle.

A serialization handle can be either an encoding or decoding handle. The encoding handles are available in three styles: incremental, fixed buffer and dynamic buffer. The decoding handles are available in two styles: incremental and (fixed) buffer. A serialization handle can be used for procedure or type serialization, regardless of the handle style.

## Implicit Versus Explicit Handles

You can declare a serialization handle with the primitive handle type, **handle_t**, and serialization handles can be explicit or implicit. An implicit handle must be specified in the ACF by using the **implicit_handle** attribute. Serializing procedures that do not have an explicit handle would then use the global variable corresponding to that handle in order to access a valid serializing context. When using type encoding, the generated routines supporting serialization of a particular type use the global implicit handle to access the serialization context. Note that remote routines may need to use the implicit handle as a binding handle. Be sure that the implicit handle is set to a valid serializing handle prior to making a serializing call.

An explicit handle is specified as a parameter of the serialization procedure prototype in the IDL file, or it can also be specified by using the **explicit_handle** attribute in the ACF. The explicit handle parameter is used to establish the proper serialization context for the procedure. To establish the correct context in the case of type serialization, the compiler generates the supporting routines that use explicit **handle_t** parameter as the serialization handle. You must supply a valid serializing handle when calling a serialization procedure or serialization type support routine.

## Serialization Styles

There are three styles you can use to manipulate serialization handles. These are: fixed buffer, dynamic buffer, and incremental. Regardless of the style you use, you must create either an encoding or decoding handle, serialize the data, and then free the handle. The style is set by creating the handle and defining the way a buffer is manipulated. The handle maintains the appropriate context associated with each of the three serialization styles.

# Fixed Buffer Serialization

When using the fixed buffer style, specify a buffer that is large enough to accommodate the encoding (marshalling) operations performed with the handle. When unmarshalling, you provide the buffer that contains all of the data to decode.

The fixed buffer style of serialization uses the following routines:

- **MesEncodeFixedBufferHandleCreate**
- **MesDecodeBufferHandleCreate**
- **MesBufferHandleReset**
- **MesHandleFree**

**MidlEncodeFixBufferHandleCreate** allocates the memory needed for the encoding handle and then initializes it. It has the following prototype:

```
RPC_STATUS RPC_ENTRY MesEncodeFixedBufferHandleCreate (
    char * Buffer,                  /* user-supplied buffer */
    unsigned long BufferSize,       /* size of the user-supplied
                                    /* buffer */
    unsigned long *pEncodedSize,    /* pointer to size of
                                    /* encoding */
    handle_t *pHandle);             /* pointer to the new
                                    /* handle */
```

The application can call the **MesBufferHandleReset** function to reinitialize the handle, or it can call the **MesHandleFree** function to free the handle's memory. To create a decoding handle corresponding to the fixed style encoding handle, you must use the **MesDecodeBufferHandleCreate** routine.

```
RPC_STATUS RPC_ENTRY MesDecodeBufferHandleCreate (
    char * Buffer,              /* buffer with data to
                                /* decode */
    unsigned long BufferSize,   /* number of bytes of
                                /* data to decode in buffer */
    handle_t *pHandle);         /* pointer to new handle */
```

The application calls **MesHandleFree** to free the encoding or decoding buffer handle.

```
RPC_STATUS RPC_ENTRY MesHandleFree (
    handle_t Handle);               /* handle to free */
```

# Examples of Fixed Buffer Encoding

The following section provides an example of how to use a fixed-buffer style, serializing handle for procedure encoding.

```
/*This is a fragment of the IDL file defining FooProc */

...
void __RPC_USER
FooProc( [in] handle_t Handle,     [in,out] FooType * pFooObject,
               [in, out] BarType * pBarObject);
...

/*This is an ACF file. FooProc is defined in the IDL file */

[    explicit_handle
]
interface regress
{
[ encode,decode ]    FooProc();
}
```

The following excerpt represents a part of an application.

```
if (MesEncodeFixedBufferHandleCreate (Buffer, BufferSize,
        pEncodedSize, &Handle) == RPC_S_OK)
{
...
/* Manufacture a FooObject and a BarObject */
...
/* The serialize works from the beginning of the buffer because the
    handle is in the initial state. The FooProc does the following
    when called with an encoding handle:
     - sizes all the parameters for marshalling,
     - marshalls into the buffer (and sets the internal state
    appropriately)
*/

FooProc ( Handle, pFooObject, pBarObject );
...
MesHandleFree ();
}
if (MesDecodeBufferHandleCreate (Buffer, BufferSize, &Handle) ==
    RPC_S_OK)
{

/* The FooProc does the following for you when called with a decoding
    handle:
     - unmarshalls the objects from the buffer into *pFooObject and
        *pBarObject
*/

FooProc ( Handle, pFooObject, pBarObject);
...
MesHandleFree ( Handle );
```

```
}
```

The following section provides an example of how to use a fixed-buffer style, serializing handle for type encoding.

```
/* This is an ACF file. FooType is defined in the IDL file */

[    explicit_handle
]
interface regress
{
typedef [ encode,decode ] FooType;
}
```

The following excerpt represents the relevant application fragments.

```
if (MesEncodeFixedBufferHandleCreate (Buffer, BufferSize,
    pEncodedSize, &Handle) == RPC_S_OK)
{
...
/* Manufacture a FooObject and a pFooObject */
...
FooType_Encode ( Handle, pFooObject );
...
MesHandleFree ();
}
if (MesDecodeBufferHandleCreate (Buffer, BufferSize, &Handle) ==
    RPC_S_OK )
{
FooType_Decode (Handle, pFooObject);
...
MesHandleFree ( Handle );
}
```

# Dynamic Buffer Serialization

When using the dynamic buffer style of serialization, the marshalling buffer is allocated by the stub and the data is encoded into this buffer and passed back to you. When unmarshalling, you supply the buffer that contains the data.

The dynamic buffer style of serialization uses the following routines:

- MesEncodeDynBufferHandleCreate
- MesDecodeBufferHandleCreate
- MesBufferHandleReset
- MesHandleFree

**MesEncodeDynBufferHandleCreate** allocates the memory needed for the encoding handle and then initializes it. It has the following prototype:

```
RPC_STATUS RPC_ENTRY MesEncodeDynBufferHandleCreate (
    char **pBuffer,                 /* pointer to buffer containing
                                    /* encoded data */
    unsigned long *pEncodedSize,    /* pointer to size of buffer
                                    /* containing encoded data */
    handle_t *pHandle);             /* pointer to the new handle */
```

The application can call the **MesBufferHandleReset** function to reinitialize the handle, or it can call the **MesHandleFree** function to free the handle's memory. To create a decoding handle corresponding to the dynamic buffer encoding handle, use the **MesDecodeBufferHandleCreate** routine. For prototypes of these routines, see Fixed Buffer Serialization.

# Incremental Serialization

When using the incremental style, you supply three routines to manipulate the buffer when required by the stub. These routines are: **Alloc**, **Read**, and **Write**. The **Alloc** routine allocates a buffer of the required size. The **Write** routine writes the data into the buffer, and the **Read** routine retrieves a buffer that contains marshalled data. A single serialization call can make several calls to these routines.

The incremental style of serialization uses the following routines:

- MesEncodeIncrementalHandleCreate
- MesDecodeIncrementalHandleCreate
- MesIncrementalHandleReset
- MesHandleFree

The prototypes for the **Alloc**, **Read**, and **Write** functions that you must provide are shown below:

```
void __RPC_USER Alloc (
   void *State,          /* application-defined pointer */
   char **pBuffer,       /* returns pointer to allocated buffer */
   unsigned int *pSize); /* inputs requested bytes; outputs
                         /* pBuffer size */
void __RPC_USER Write (
   void *State,          /* application-defined pointer */
   char *Buffer,         /* buffer with serialized data */
   unsigned int Size);   /* number of bytes to write from Buffer */
void __RPC_USER Read (
   void *State,          /* application-defined pointer */
   char **pBuffer,       /* returned pointer to buffer with data */
   unsigned int *pSize); /* number of bytes to read into pBuffer */
```

The *State* input argument for all three functions is the application-defined pointer that was associated with the encoding services handle. The application can use this pointer to access the data structure containing application-specific information such as a file handle or stream pointer. Note that the stubs do not modify the *State* pointer other than to pass it to the **Alloc**, **Read**, and **Write** functions. During encoding, **Alloc** is called to obtain a buffer into which the data is serialized. Then, **Write** is called to enable the application to control when and where the serialized data is stored. When decoding, **Read** is called to return the requested number of bytes of serialized data from wherever the application stored it.

An important feature of the incremental style is that the handle keeps the state pointer for you. This pointer maintains the state and is never touched by the RPC code, except when passing the pointer to **Alloc**, **Write**, or **Read** function. The handle also maintains an internal state that makes it possible to serialize and deserialize several type instances to the same buffer by adding padding as needed for alignment. The **MesIncrementalHandleReset** function resets a handle to its initial state to enable reading or writing from the beginning of the buffer.

The **Alloc** and **Write** functions, along with an application-defined pointer, are associated with an encoding-services handle by a call to the **MesEncodeIncrementalHandleCreate** function. **MesEncodeIncrementalHandleCreate** allocates the memory needed for the handle and then initializes it. It has the following prototype:

```
RPC_STATUS RPC_ENTRY MesEncodeIncrementalHandleCreate (
    void * UserState,        /* application-defined pointer */
    MIDL_ES_ALLOC Alloc,     /* pointer to Alloc function */
    MIDL_ES_WRITE Write,     /* pointer to Write function */
    handle_t *pHandle);      /* receives encoding services handle */
```

The application can call **MesDecodeIncrementalHandleCreate** to create a decoding handle, **MesIncrementalHandleReset** to reinitialize the handle, or **MesHandleFree** to free the handle's memory. The **Read** function, along with an application-defined parameter, is associated with a decoding handle by a call to the **MesDecodeIncrementalHandleCreate** routine. The function creates the handle and initializes it. It has the following prototype:

```
RPC_STATUS RPC_ENTRY MesDecodeIncrementalHandleCreate (
    void * UserState,        /* application-defined pointer */
    MIDL_ES_READ Read,       /* pointer to Read function */
    handle_t Handle);        /* handle to create and initialize */
```

The *UserState*, *Alloc*, *Write*, and *Read* parameters of **MesIncrementalHandleReset** can be NULL to indicate no change.

```
RPC_STATUS RPC_ENTRY MesIncrementalHandleReset (
    handle_t Handle,         /* handle to reinitialize */
    void * UserState,        /* application-defined pointer */
    MIDL_ES_ALLOC Alloc,     /* pointer to Alloc function */
    MIDL_ES_WRITE Write,     /* pointer to Write function */
    MIDL_ES_READ Read,       /* pointer to Read function */
    MIDL_ES_CODE OpCode);    /* operations allowed */

RPC_STATUS RPC_ENTRY MesHandleFree (
    handle_t Handle);            // handle to free
```

# Examples of Incremental Encoding

The following section provides an example of how to use the incremental style serializing handle for type encoding.

```
/* This is an acf file. FooType is defined in the idl file */

[    explicit_handle
]
interface regress
{
typedef [ encode,decode ] FooType;
}
```

The following excerpt represents the relevant application fragments.

```
if (MesEncodeIncrementalHandleCreate (State, AllocFn, WriteFn,
    &Handle) == RPC_S_OK)
{
...
/* The serialize works from the beginning of the buffer because
    the handle is in the initial state. The Foo_Encode does the
    following:
    - sizes *pFooObject for marshalling,
    - calls AllocFn with the size obtained,
    - marshalls into the buffer returned by Alloc, and
    - calls WriteFn with the filled buffer
*/

Foo_Encode ( Handle, pFooObject );
...
}
if (MesIncrementalHandleReset (Handle, NULL, NULL, NULL, ReadFn,
    MES_DECODE ) == RPC_OK)
{
/*The ReadFn is needed to reset the handle. The arguments
    that are NULL do not change. You can also call
    MesDecodeIncrementalHandleCreate (State, ReadFn, &Handle);
    The Foo_Decode does the following:
    - calls Read with the appropriate size of data to read and
        receives a buffer with the data
    - unmarshalls the object from the buffer into *pFooObject
*/

Foo_Decode ( Handle, pFooObject );
...
MesHandleFree ( Handle );
}
```

## Obtaining an Encoding Identity

An application that is decoding encoded data can obtain the identity of the routine used to encode the data, prior to calling a routine to decode it. The **MesInqProcEncodingId** routine provides this identity. It has the following prototype:

```
RPC_STATUS RPC_ENTRY MesInqProcEncodingId (
    handle_t Handle,                      /* decoding handle */
    PRPC_SYNTAX_IDENTIFIER pInterfaceId,  /* points to location
                                             the identity will be
                                             written to */
    unsigned long __RPC_FAR * pProcNum);  /* number of the routine
                                             used to encode data */
```

# Run-Time RPC Functions

The run-time RPC functions are those your distributed application calls to establish a binding handle that represents the logical connection between a client and a server. The binding handle enables the RPC run-time libraries to direct a client's remote procedure call to an instance of the specified interface on a server.

Obtaining the binding handle involves several data structures or strings:

- Protocol sequence and network address strings
- Endpoints
- Interface UUIDs and interface version numbers
- Object UUIDs
- Name-service database server entries

The following topics describe these data structures and strings and the RPC functions that allow your application to manipulate them.

The name-service functions allow a server to register its interface in a database. When a server registers its interface, any client in the domain can query the database, supplying a logical name and an optional object UUID, to obtain a binding handle to the server without knowing the host name of the server.

The RPC name service makes distributed applications easy to administer. When the server side of the distributed application is moved to another computer, clients do not have to be reconfigured. As long as the database entry name and object UUIDs remain the same, client applications can access the server application as they did before. When a client requests an interface that several servers have registered, the name service shuffles the binding handles before returning them to the client. This provides a measure of load balancing by preventing all the clients from using the same server.

You can provide more than one implementation of the remote procedure calls defined in an interface. RPC maps a remote procedure call to an implementation of the procedure through a table of function pointers known as the manager entry-point vector (EPV). You can add implementations of the procedure by supplying additional manager EPVs. The client's object UUID determines the appropriate implementation to use.

You can also add security to your distributed application in two ways: by installing a security package and calling the RPC functions related to security, or by using the security features built into Windows NT™ transport protocols. Most application writers will want to use the RPC security functions instead of transport-level security. Read the section "Using Authenticated RPC" for more details.

The set of RPC functions supported by Microsoft® RPC overlaps the OSF-DCE RPC functions. The Microsoft RPC functions are optimized for use with MS-DOS and Microsoft 16-bit and 32-bit Windows operating systems. They are fully compatible with other Microsoft naming and calling conventions.

For a complete description of each function and data structure in Microsoft RPC, see the RPC Function Reference.

# Naming Conventions for RPC Functions

RPC function names generally consist of the prefix "Rpc," an object name, and a verb that describes an operation on that object. The functions, with some exceptions, are named as shown:

**Rpc***ObjectOperation*

*Object*

Specifies a term that identifies an RPC object; a data structure defined by the RPC function.

*Operation*

Specifies an operation that is performed on the object specified by *Object*.

Functions that operate on UUID objects omit the prefix "Rpc" and start with the object name "Uuid."

The functions provided with this version of Microsoft RPC operate on the following objects:

| Object | Object in function name | Example |
|---|---|---|
| Binding handle | **Binding** | **RpcBindingFree** |
| Endpoint | **Ep** | **RpcEpRegister** |
| Interface | **If** | **RpcIfInqId** |
| Management | **Mgmt** | **RpcMgmtStopServerListening** |
| Name-service group entry | **NsGroup** | **RpcNsGroupDelete** |
| Name-service management | **NsMgmt** | **RpcNsMgmtEntryCreate** |
| Name-service profile entry | **NsProfile** | **RpcNsProfileEltAdd** |
| Name-service server entry | **NsBinding** | **RpcNsBindingExport** |
| Network | **Network** | **RpcNetworkInqProtseqs** |
| Object, type UUID mapping | **Object** | **RpcObjectSetType** |
| Protocol-sequence vector | **ProtseqVector** | **RpcProtseqVectorFree** |
| Server | **Server** | **RpcServerListen** |
| String | **String** | **RpcStringFree** |
| String binding | **StringBinding** | **RpcStringBindingCompose** |
| UUID | **Uuid** | **UuidCreate** |

**Note for OSF-DCE Programmers**: Microsoft RPC function names are derived by converting the first character of the OSF-DCE RPC function name, and every character that follows an underscore character, to uppercase and then removing underscore characters. For example, the OSF-DCE function **rpc_server_use_all_protseqs_if** is named **RpcServerUseAllProtseqsIf** in Microsoft RPC.

Microsoft data-structure names are derived from the OSF-DCE names by converting all characters to

uppercase and removing the trailing suffix **_t**. For example, the OSF-DCE data structure **rpc_binding_vector_t** is named **RPC_BINDING_VECTOR** in Microsoft RPC.

In the header files provided in Microsoft RPC, each RPC function that takes character-string parameters appears in two forms: followed by the suffix "A" and followed by the suffix "W." The "A" suffix represents the ASCII-character string version of the function and the "W" suffix represents the wide-character string version. The identifier UNICODE determines which version of the function is selected. The standard function name is mapped to either the ASCII or the wide-character string version.

Wide-character versions of the RPC functions are selected when you define the identifier UNICODE. You can define the identifier either with a **#define** preprocessor directive or with the **/D** option of the Microsoft C/C++ version 7.0 compiler. For example:

```
#define UNICODE
main()

cl /DUNICODE filename.c
```

You can use the wide-character version of a function on one side of the distributed application and the ASCII version on the other side. You do not need to use the same versions of the functions with both the client and server applications. You can use both versions in the same application.

## Macro Definitions

The RPC tools achieve model, calling, and naming-convention independence by associating data types and function-return types in the generated stub files and header files with definitions that are specific to each platform. These macro definitions ensure that any data types and functions that require the designation of **_ _far** are specified as far objects.

The following figure shows the macro definitions that the MIDL compiler applies to function calls between RPC components:

{ewc msdncd, EWGraphic, bsd23536 0 /a "SDK_A29.BMP"}

These are the macro definitions:

| Definition | Description |
|---|---|
| **_ _RPC_API** | Applied to calls made by the stub to the user application. Both functions are in the same executable program. |
| **_ _RPC_FAR** | Applied to the standard macro definition for pointers. This macro definition should appear as part of the signature of all user-supplied functions. |
| **_ _RPC_STUB** | Applied to calls made from the run-time library to the stub. These calls can be considered private. |
| **_ _RPC_USER** | Applied to calls made by the run-time library to the user application. These cross the boundary between a DLL and an application. |
| **_ _RPC_ENTRY** | Applied to calls made by the application or stub to the run-time library. This macro definition is applied to all RPC run-time functions. |

To link correctly with the Microsoft RPC run-time libraries, stubs, and support routines, some user-supplied functions must also include these macros in the function definition. Use the macro **_ _RPC_API** when you define the functions associated with memory management, user-defined binding handles, and the **transmit_as** attribute, and use the macro **_ _RPC_USER** when you define the context-rundown routine associated with the context handle. Specify the functions as:

**_ _RPC_USER midl_user_allocate(...)**

**_ _RPC_USER midl_user_free(...)**

**_ _RPC_USER** *handletype*__bind(...)__

**_ _RPC_USER** *handletype*__unbind(...)__

**_ _RPC_USER** *type*__to_local__

**_ _RPC_USER** *type*__from_local__

**_ _RPC_USER** *type*__to_xmit(...)__

**_ _RPC_USER** *type*__from_xmit(...)__

**_ _RPC_USER** type__free_local__

**_ _RPC_USER** *type*__free_inst(...)__

**_ _RPC_USER** *type*_**free_xmit(...)**

**_ _RPC_USER context_rundown(...)**

**Note** All pointer parameters in these functions must be specified using the macro **_ _RPC_FAR**.

These are the two approaches that can be used to select an application's memory model:

1. To use a single memory model for all files, compile all source files using the same memory-model compiler switches. For example, to develop a small-model application, compile both the application and the stub source code using the C-compiler switch **/AS**, as in the following:

   ```
   cl -c /AS myfunc.c
   cl -c /AS clstub_c.c
   ```

2. To use different memory models for the application source files and the support source files (stubs files), use the RPC macros when you define function prototypes in the IDL file. Compile the distributed-application source files using one compiler memory-model setting and compile the support files using another compiler memory-model setting. Use the same memory model for all of the files generated by the compiler.

## Data Structures

Obtaining the handle that represents the binding between clients and servers involves several key data structures:

- Binding handle
- Protocol sequence and network address string
- Endpoint
- Interface UUIDs and interface version number
- Object UUID
- Name-service database entries, including profile, group, and server entries.

## Endpoints

The endpoint specifies the communication port clients use to make remote procedure calls to a server.

The server application specifies endpoint information at the same time it specifies the protocol sequence by calling the RPC routine that starts with the prefix "RpcServerUseProtseq" or "RpcServerUseAllProtseqs."

A finite number of endpoints are available for any protocol sequence. Some of these are usually assigned by the authority responsible for the protocol. The syntax of the endpoint string depends on the protocol sequence you use. For example, the endpoint for TCP/IP is a port number, and the endpoint syntax for named pipes is a valid pipe name.

The major design decision you must make regarding the endpoint is whether it is well known or dynamic. Your choice of option determines whether the distributed application or the run-time library specifies the endpoint the application will use.

Most applications should use dynamic endpoints so the endpoint-mapping service can dynamically map a distributed application to an endpoint available for the protocol. In this way, this limited system resource can be assigned to a distributed service at run time as needed, instead of being dedicated to a distributed service when the service is developed.

## Well-Known Endpoints

A distributed application can specify an endpoint in a string that is used as a parameter to the function **RpcServerUseProtseqEp** or in a string that appears in the IDL file interface header as part of the **endpoint** interface attribute. Well-known endpoints are not recommended for most applications.

You can use two approaches to implement the well-known endpoint:

- Specify all information in a string binding.
- Store the well-known endpoint in the name-service database.

All the information needed to establish the binding can be written into a distributed application when you develop it. The client can specify the well-known endpoint directly in a string, call **RpcStringBindingCompose** to create a string that contains all the binding information, and obtain a handle by supplying this string to the function **RpcBindingFromStringBinding**. The client and server can be hard-coded to use a well-known endpoint, or written so that the endpoint information comes from the command line, a data file, or the IDL file.

When a server uses a well-known endpoint, the endpoint data is included as part of the name-service database server entry. When the client imports a binding handle from the server entry, the binding handle contains a complete server address that includes the well-known endpoint.

## Dynamic Endpoints

The number of communication ports for a particular server can be limited. For example, when you use the **ncacn_nb_nb** protocol sequence, indicating that RPC network communication occurs using NetBIOS over NetBEUI, less than 255 ports are available. The RPC run-time libraries allow you to assign endpoints dynamically as needed.

The application selects a dynamic endpoint in one of two ways: on the client side it uses a null string to indicate the endpoint when it composes a string binding, and on the server side it registers the server application in the name-service database, or it calls **RpcServerUseProtseq** or **RpcServerUseAllProtseqs** to explicitly select dynamic endpoints.

The dynamic endpoint is registered in an endpoint-map database. This a database that is managed by a specific service which creates and deletes elements for applications. In Windows NT™ and Windows® 95, the endpoint-mapping service is called RPCSS. The dynamic endpoint expires when the server instance stops running. To remove the old endpoint from the endpoint mapper database, call **RpcEpUnregister** at application termination.

## Fully and Partially Bound Handles

When you use dynamic endpoints, the run-time libraries obtain endpoint information as they need it. The run-time libraries make the distinction between a fully bound handle (one that includes endpoint information) and a partially bound handle (one that does not include endpoint information).

The client run-time library must convert the partially bound handle to a fully bound handle before the client can bind to the server. The client run-time library tries to convert the partially bound handle for the client application by obtaining the endpoint information as shown:

- From the client's interface specification.
- From an endpoint-mapping service running on the server.

If the client tries to use a partially bound handle when the endpoint information is not available in the interface specification and the server's endpoint-mapper does not know the server endpoint, the client will not have enough information to make its remote procedure call and that call will fail. To prevent this, you must register the endpoint in the endpoint mapper when your distributed application uses partially bound handles. For more information about the endpoint mapper, see Registering the Endpoint.

When a remote procedure call fails, the client application can call **RpcBindingReset** to remove out-of-date endpoint information. When the client tries to call the remote procedure, the client run-time library again tries to convert the fully bound handle to a partially bound handle. This is useful, for example, when the server has been stopped and restarted using a different dynamic endpoint.

## Using Datagram Protocols

Microsoft RPC supports datagram (connectionless) protocols as well as connection-oriented protocols. Some of the features available when using datagram protocols are shown below:

- Datagrams support the UDP and IPX connectionless transport protocols.
- Because it is not necessary to establish and maintain a connection, resource overhead is less using the datagram RPC protocol.
- Datagrams enable faster binding.
- As with connection-oriented RPC, datagram RPC calls are by default nonidempotent. This means the call is guaranteed not to be executed more than once. However, a function can be marked as idempotent in the IDL file telling RPC that it is harmless to execute the function more than once in response to a single, client request. This allows the run time to maintain less state on the server. Note that an idempotent call would be re-executed only in rare circumstances on an unstable network.
- Datagram RPC supports the broadcast IDL attribute. Broadcast enables a client to issue messages to multiple servers at the same time. This lets the client locate one of several available servers on the network, or control multiple servers simultaneously. Broadcast calls are implicitly idempotent. If the call contains [out] parameters, only the first server response is returned. Once a server responds, all future RPCs over that binding handle will be sent to that server only, including calls with the broadcast attribute. To send another broadcast, create a new binding handle or call RpcBindingReset on the existing handle.
- Datagram RPC supports the maybe IDL attribute. This lets the client send a call to the server without waiting for a response or confirmation. The call cannot contain [out] parameters. **Maybe** calls are implicitly idempotent.

## The RPC Name-Service Database

A name service is a service that maps names to objects, and usually maintains the (name, object) pairs in a database. Generally, the name is a logical name that is easy for users to remember and use. For example, a name service would allow a user to use the logical name "laserprinter." The name service maps this name to the network-specific name used by the print server.

To use a simplified explanation, the RPC name service maps a name to a binding handle and maintains the (name, binding handle) mappings in the RPC name-service database. The RPC name service allows client applications to use a logical name instead of a specific protocol sequence and network address. The use of the logical name makes it easier for network administrators to install and configure your distributed application.

An RPC name-service database entry has one of the following attributes: server, group, or profile. In the Microsoft implementation, entries can have only one attribute, so these entries are also known as server entries, group entries, and profile entries.

The server entry consists of interface UUIDs, object UUIDs (needed when the server implements multiple entry points), network address, protocol sequence, and any endpoint information associated with well-known endpoints. When a dynamic endpoint is used, the endpoint information is kept in the endpoint-map database rather than the name-service database, and the endpoint is resolved like any other dynamic endpoint. Server entries are managed by functions that start with the prefix "RpcNsBinding."

The group entry can contain server entries or other group entries. Group entries are managed by functions that start with the prefix "RpcNsGroup."

The profile entry can contain profile, group, or server entries. Profile entries are managed by the functions that start with the prefix "RpcNsProfile."

## Name-Service Application Guidelines

When you develop your distributed application, provide the application users with a method for specifying the name under which they can register the application in the name-service database. This method can consist of a data file, command-line input, or dialog box.

Though the RPC name-service architecture supports various methods for organizing an application's server entries, it is optimized for look-ups. As a result, frequent updates can hinder the performance of both the name service and the application. To avoid exporting information unnecessarily, choose a design that lets the server determine whether its information is in the name-service database. In addition, each server instance should export to its own entry name. Otherwise, it will be very difficult for an instance to change its supported object UUIDs or protocol sequences without disturbing another instance's information.

Following is a method that avoids these pitfalls and provides good performance, whether you use the Microsoft Locator or another name service.

To begin with, design your application so the first time a given server instance starts up, it picks a unique server-entry name and saves this name in an .INI file along with the application's other configuration information. Then, have it export its binding handles and object UUIDs, if any, to its name-service entry.

Subsequent invocations of the server instance should check that the name-service entry is present and contains the correct set of object UUIDs and binding handles. A missing entry may mean that an administrator removed it, or that a power outage caused the name-service information to be lost. It is important to verify that the binding handles in the entry are correct; if an administrator adds TCP/IP support to a computer, for example, RPC servers will listen on that protocol sequence when they call **RpcServerUseAllProtseqs**. However, if the server does not update the name-service entry, clients will not be informed that TCP is supported.

When the client imports, it should specify NULL as the entry name. Specifying NULL causes the Microsoft Locator to search for the interface in all name-service entries in the client machine's domain or workgroup, thus finding the information for every instance.

If you use object UUIDs to represent well-known objects such as printers, you can use a variation of this method. Instead of exporting bindings to one entry, design your application so each instance creates an entry for each supported object, such as "/.:/printers/Laser1" and "/.:/printers/Laser2." Then, have the server export its binding handles to each server entry, along with the object UUID relevant to that entry.

In this case, a client can look up a resource by name by importing from the relevant server entry; it does not require the object UUID of the resource. If it has the resource UUID but not the name, it can import from the null entry.

## An Overview of the Name Service Entry

The name service entry consists of three distinct sections. The first section is for interfaces (UUID + version), the second section contains the object UUIDs, and the third section is for binding handles. You provide a name for the entry that will serve as a way to identify it.

When calling **RpcNsBindingExport**, the server specifies the name of the entry in which to place the exported information. This newly exported interface is then added to the interface section of the name service entry. Any interfaces that are already present in the name service entry remain as well. This same process is followed for object UUIDs and binding handles.

The client calls **RpcNsBindingLookupBegin** (or **RpcNsBindingImportBegin**) to search for an appropriate binding handle. The entry name, interface handle, and an object UUID are extracted. These restrict the entries from which binding handles are returned. If an entry matches the search criteria, all the binding handles in that entry are returned from **RpcNsBindingImportNext**.

## Criteria for Name Service Entries

The following criteria are used when processing name service entries:

- If you provide a non-NULL entry name for **RpcNsBindingLookupBegin**, that entry will be the only entry searched for binding handles. If you pass NULL, all entries in your logon domain will be searched. Note that this does not include trusted domains.
- If you provide an interface handle, binding handles are returned from an entry only if the interface section of the entry contains a compatible version of that interface UUID. That is, the major version number must be the same as your interface UUID, while the minor version number must be equal to or greater than yours.
- If you provide an object UUID, binding handles are returned from an entry only if the object UUID section of the entry contains that particular object UUID.

If a name service entry survives the criteria described above, all the binding handles from those entries are gathered. Handles with a protocol sequence that is unsupported by the client are discarded and the remaining handles are returned to you as the output from **RpcNsBindingLookupNext**.

## Name Service Entry Cleanup

A name service entry should contain information that does not change frequently. For this reason, do not include dynamic endpoints in your exported binding handles because they will change at each invocation of the server and will clutter up your name service entry. To remove these binding handles, use **RpcBindingReset**. For example, a reasonable sequence of server operations would be:

For more than one transport:

```
RpcServerUseProtseq();
RpcServerUseProtseq();
```

To place bindings in the endpoint mapper:

```
RpcServerInqBindings(&Vector);
RpcEpRegister(Interface, Vector);
```

To remove endpoints from bindings:

```
for (i=0; i < Vector- > Count; + + i)
    {
    RpcBindingReset(Vector->BindingH[i];
    }
```

To add bindings to the name service:

```
RpcNsBindingExport(RPC_C_NS_SYNTAX_DEFAULT, EntryName, Interface
    Vector);
RpcServerListen();
```

Since the Microsoft Locator service does not use many resources to export information, the examples above work well. However, Microsoft RPC also supports Digital Equipment Corporation's Cell Directory Service (CDS), which is a more robust name service. When using CDS, **RpcNsBindingExport** or **RpcNsBindingUnexport** will create significant network traffic for replication and distribution. Thus, the server should determine if the information already has been exported and export only it if is has not.

## What Happens During a Query

This section describes how the network handles the query when a 32-bit client searches for a name in its own domain.

When your client application calls **RpcNsBindingImportBegin**, the locator residing on your client computer will try to satisfy this request. If there is nothing in the cache, it will forward the request by RPC to a master locator. If the master locator finds nothing in its cache, it sends the request to all the computers in the domain using a mail-slot broadcast. If there is a match, the locator on each computer will respond by a directed mail slot.(For example, if a process on that computer has exported the interface.) The responses are collated and the RPC is completed from the client's process locator, which will reply to the client process itself.

In a domain, the client locator searches for a master locator in the following places:

1. On the primary domain controller.
2. On each backup domain controller.

If a match is not found, the client locator declares itself to be the master locator. As such, it will broadcast queries if they cannot be satisfied locally.

In a workgroup, the client locator maintains a cache of the computers whose locators have broadcasted. It uses the one that has been running the longest as the master locator. If that computer is unavailable, the next, longest-broadcasting computer is used, and so on. If the client needs a master locator and the cache is empty, it replenishes the cache by sending a special mail-slot broadcast that requests master locators to respond. If there are no responses, the client locator declares itself to be the master locator and will broadcast queries if they cannot be satisfied locally.

This changes if your client application is a Windows 3.*x* or MS-DOS program. In this case, there is no locator running on the client computer, and rpcns1.dll or rpcnslm.rpc contains the code to find a master locator. All requests are forwarded directly to the master locator.

These guidelines are valid for names in the client's domain, such as names for "/.:/entryname". If the client requests a name from another domain through the use of "/.../DOMAIN/entryname;" the client locator forwards the request to the specified domain which will broadcast it if it does not have the answer. If the domain is down or is actually a workgroup, the request will fail.

**Note** Remember the following when working with entries in the name service:

- A client cannot use the "/.../DOMAIN/entryname" syntax to find an entry in its own domain. Use the syntax "/.:/entryname". However, you can use "/.../DOMAIN/entryname" to find an entry in another domain.
- The domain name in "/.../DOMAIN/entryname" must be uppercase. When looking for a match, the locator is case-sensitive.
- Locator entry names are also case-sensitive.When the client uses the "/.:/entryname" syntax, the locator will not search for entries in other domains, even if they have a trust relationship with the logon domain.
- Broadcasts do not cross LAN segments (for example, subnets). Thus, the usefulness of the locator is limited in an organization with multiple subnets.

## Using CDS

If you have CDS, you can use it instead of the Locator. Change the registry entries as shown:

```
HKEY_LOCAL_MACHINE
    Software
        Microsoft
            Rpc
                Name Service
                    NetworkAddress

HKEY_LOCAL_MACHINE
    Software
        Microsoft
            Rpc
                Name Service
                    Endpoint
```

Changing these entries will point to a gateway computer that is running the NSID. This will be used as the master locator. In the event of a crash, there will be no search for a replacement.

## Name Syntax

Microsoft RPC accepts names that conform to the following syntax:

*/***.:***/***name*[*/***name*...]
*/***...***/***domainname*/***name*[*/***name*...]

*name*

Specifies an identifier that can contain any character except the delimiting slash (/) character.

*domainname*

Specifies the name of the Windows NT domain.

A parameter that selects the name-syntax type and the string that specifies the name are supplied to many of the name-service interface (NSI) RPC functions.

Only one name-syntax type is supported by Microsoft RPC, as specified by the constant RPC_C_NS_SYNTAX_DCE. This constant is defined in the header file RPCNSI.H.

The name syntax specified by RPC_C_NS_SYNTAX_DCE is an extension of the OSF_DCE Cell Directory Service (CDS) name syntax. The ability to specify a domain name represents an extension to that syntax. There is no absolute limit on the number of names that can be separated by slash characters as long as the overall string is less than 256 characters.

The slashes allow you to specify a logical structure to the name, but they do not correspond to a logical structure in the objects themselves.

## Server Application RPC API Calls

For most distributed applications, you should write your server application to call the RPC functions in the following sequence:

1. Specify the protocol sequence(s). Call one of the following RPC functions: **RpcServerUseProtseq**, **RpcServerUseAllProtseqs**, **RpcServerUseProtseqIf**, **RpcServerUseAllProtseqsIf**, and **RpcServerUseProtseqEp**
   or the extended versions, which allow you to specify a policy for allocation of dynamic ports and allow allow multi-homed machines to selectively bind to Network Interface Cards (NICS): **RpcServerUseProtseqEx**, **RpcServerUseAllProtseqsEx**, **RpcServerUseProtseqIfEx**, **RpcServerUseAllProtseqsIfEx**, and **RpcServerUseProtseqEpEx**.
2. Call **RpcServerInqBindings** to obtain a vector containing all of the server's binding handles. You will use this binding vector for subsequent calls to **RpcEpRegister**, **RpcEpRegisterNoReplace**, and **RpcNsBindingExport**.
3. When you use dynamic endpoints, add the endpoints associated with the server to the endpoint-map database. Call **RpcEpRegister** or **RpcEpRegisterNoReplace**register the binding handles with the endpoint-mapping service.
   During implementation and debugging, you can use string bindings to communicate binding information to clients. This allows you to establish a client-server relationship without using the endpoint-map database or name-service database. To establish such a relationship, use **RpcBindingToStringBinding** to convert one or more binding handles in the binding-handle vector to a string binding and provide the string binding to the client.
4. Call **RpcBindingReset** on each of the dynamic bindings in the binding vecto to remove the dynamic endpoints from the bindings. Then export the binding vector to the name-service database. Call Page: 1
   **RpcNsBindingExport**to place the binding handles in the name-service database for access by any client.
5. Clean up data structures. Call the RPC function **RpcBindingVectorFree**. to free the vector of server binding handles.
6. Register the interface with the RPC run-time library. Call **RpcServerRegisterIfEx** or **RpcServerRegisterIf**. This is a required call.
7. Listen for clients. Call **RpcServerListen** or **RpcMgmtWaitServerListen**. **RpcServerListen** to begin receiving remote procedure call requests. This is a required call.

When the server application is no longer actively serving clients, you usually instruct it to call RPC functions in the following sequence:

1. Stop listening for clients. Call the RPC function **RpcMgmtStopServerListening**.
   If the server application is merely pausing, this is the only call that needs to be made. If the application is terminating:
2. Remove the interface. Call the RPC function **RpcServerUnregisterIf**.
3. Remove endpoint-map database entries. Call the RPC function **RpcEpUnregister**.
4. Remove the name-service entry. Call **RpcNsBindingUnexport**

### See Also

Specifying the Protocol Sequence, Registering the Endpoint, Exporting to the RPC Name-Service Database, Registering the Interface.

## Specifying the Protocol Sequence

One of the first acts of the server application is to specify the protocol sequences over which it can communicate with clients.

The protocol sequence is a character string that represents a valid combination of an RPC protocol (such as "ncacn"), a transport protocol (such as "tcp"), and a network protocol (such as "ip"). Microsoft RPC supports the following protocol sequences:

| Protocol sequence | Description | Supporting Platforms |
|---|---|---|
| **ncacn_nb_tcp** | Connection-oriented NetBIOS over TCP | Client only: MS-DOS, Windows 3.*x*<br>Client and server: Windows NT |
| **ncacn_nb_ipx** | Connection-oriented NetBIOS over IPX | Client only: MS-DOS, Windows 3.*x*<br>Client and server: Windows NT |
| **ncacn_nb_nb** | Connection-oriented NetBEUI | Client only: MS-DOS, Windows 3.*x*<br>Client and server: Windows NT, Windows 95 |
| **ncacn_ip_tcp** | Connection-oriented TCP/IP | Client only: MS-DOS,Windows 3.*x,* and Apple® Macintosh®<br>Client and server: Windows 95 and Windows NT |
| **ncacn_np** | Connection-oriented named pipes | Client only: MS-DOS, Windows 3.*x*, Windows 95<br>Client and server: Windows NT |
| **ncacn_spx** | Connection-oriented SPX | Client only: MS-DOS, Windows 3.*x*<br>Client and server: Windows NT, Windows 95 |
| **ncacn_dnet_nsp** | Connection-oriented DECnet | Client only: MS-DOS, Windows 3.*x* |
| **ncacn_at_dsp** | Connection-oriented AppleTalk DSP | Client: Apple Macintosh |
| **ncacn_vns_spp** | Connection-oriented Vines SPP transport | Client and Server: Windows NT |
| **ncadg_ip_udp** | Datagram (connectionless) UDP/IP | Client only: MS-DOS, Windows 3.*x*<br>Client and server: Windows NT |
| **ncadg_ipx** | Datagram (connectionless) IPX | Client only: MS-DOS, Windows 3.*x*<br>Client and server: Windows NT |
| **ncalrpc** | Local procedure call | Client and server: Windows NT and Windows 95 |

The server application specifies a single protocol sequence by calling one of the functions that starts with the prefix "RpcServerUseProtseq." The server specifies all supported protocol sequences by calling **RpcServerUseAllProtseqs**.

The function you choose to specify protocol sequences also specifies information about the endpoint. The endpoint can be specified explicitly (**RpcServerUseProtseqEp**), culled from the IDL file (**RpcServerUseProtseqIf**, **RpcServerUseAllProtseqsIf**), or selected for the application by the run-time library (**RpcServerUseProtseq**, **RpcServerUseAllProtseqs**). These are the choices:

| "Protseq" function | Description |
|---|---|
| RpcServerUseAllProtseqs | Registers all protocols using dynamic endpoints. |
| RpcServerUseAllProtseqsIf | Registers all protocols with endpoints from the IDL file. |
| RpcServerUseProtseq | Registers one protocol using a dynamic endpoint. |
| RpcServerUseProtseqEp | Registers one protocol with the specified endpoint. |
| RpcServerUseProtseqIf | Registers one protocol with the endpoint in the IDL file. |

The server application specifies endpoint information at the same time it specifies the protocol sequence by calling the RPC function that starts with the prefix "RpcServerUseProtseq" or "RpcServerUseAllProtseqs." The endpoint specifies the communication port through which clients make remote procedure calls to the server. For more information about endpoints, see Endpoints.

## Registering the Endpoint

When a server uses dynamic endpoints, the server application must also call **RpcEpRegister** or **RpcEpRegisterNoReplace** to register the endpoints.

Both functions add the server's interfaces and binding handles to the "endpoint mapper" database. When the client makes a remote procedure call using a partially bound handle, the client's run-time library asks the server machine's endpoint mapper for the endpoint of a compatible server instance. The client library supplies the interface UUID, protocol sequence, and, if present, the object UUID in the client binding handle. The endpoint mapper looks for a database entry that matches the client's information. The client/server interface UUID, the interface major version, and protocol sequence must all match exactly. In addition, the server's interface minor version must be greater than or equal to the client's minor version.

If all tests are successful, the endpoint mapper returns the valid endpoint and the client run-time library updates the endpoint in the binding handle.

## Exporting to the RPC Name-Service Database

After specifying the protocol sequence and endpoint and registering any dynamic endpoints in the endpoint-map database, the server application registers the binding handle for the interface with the RPC name-service provider by calling **RpcNsBindingExport**.

In the Microsoft environment, the server application should register itself with the name-service database every time the server application is run. In the OSF-DCE environment, the server application registers with the name-service database only once and that is when the application is installed. The Microsoft Locator maintains its database in transient memory on the server, while the OSF-DCE name service resides in permanent, replicated storage which is relatively expensive to update.

## Registering the Interface

After calling **RpcServerUseAllProtseqs**, registering dynamic endpoints in the endpoint-map database, and registering your distributed application in the name service, register the interface by calling **RpcServerRegisterIfEx** or **RpcServerRegisterIf** once for each implementation of the interface.

Where you provide a single implementation of each function prototype specified in the interface, supply the interface handle data structure generated by the MIDL compiler and supply null pointers for the manager type and the parameters of the manager entry-point vector (EPV).

**RpcServerRegisterIfEx** and **RpcServerRegisterIf** set values in the internal interface registry table. This table is used to map the interface UUID and object UUIDs to a manager EPV. The manager EPV is an array of function pointers that contains exactly one function pointer for each function prototype in the interface specified in the IDL file.

The run-time library uses the interface registry table (set by calls to the function **RpcServerRegisterIf***) and the object registry table (set by calls to the function **RpcObjectSetType**) to map interface and object UUIDs to the function pointer.

For information on supplying multiple EPVs to provide multiple implementations of the interface, see Multiple Interface Implementations.

## Entry-Point Vectors

The manager EPV is an array of function pointers that point to implementations of the functions specified in the IDL file. The number of elements in the array corresponds to the number of functions specified in the IDL file. Microsoft RPC supports multiple entry-point vectors representing multiple implementations of the functions specified in the interface.

The MIDL compiler automatically generates a manager EPV data type for use in constructing manager EPVs. The data type is named *if-name*_**SERVER_EPV**, where *if-name* specifies the interface identifier in the IDL file.

The MIDL compiler automatically creates and initializes a default manager EPV on the assumption that a manager routine of the same name exists for each procedure in the interface and is specified in the IDL file.

When a server offers multiple implementations of the same interface, the server must create one additional manager EPV for each implementation. Each EPV must contain exactly one entry point (address of a function) for each procedure defined in the IDL file. The server application declares and initializes one manager EPV variable of type *if-name*_**SERVER_EPV** for each additional implementation of the interface. It registers the EPVs by calling **RpcServerRegisterIfEx** or **RpcServerRegisterIf** once for each supported object type.

When the client makes a remote procedure call to the server, the EPV containing the function pointer is selected based on the interface UUID and the object type. The object type is derived from the object UUID by the object-inquiry function or the table-driven mapping controlled by **RpcObjectSetType**.

## Specifying the Manager EPV

If the routine names used by a manager correspond to those of the interface definition, you can register this manager using the default EPV of the interface generated by the MIDL compiler. You can also register a manager using a server-application-supplied EPV.

## The Default Manager EPV

By default, the MIDL compiler uses the procedure names from an interface's IDL file to generate a manager EPV, which the compiler places directly into the server stub. This default EPV is statically initialized using the procedure names declared in the interface definition.

To register a manager using the default EPV, specify NULL as the value of the *MgrEpv* argument (a null EPV).

## Server-Supplied Manager EPVs

A server can (and sometimes must) create and register a non-null manager EPV for an interface. To select a server-application-supplied EPV, pass a non-null EPV whose value has been declared by the server as the value of the *MgrEpv* argument. A non-null value for the *MgrEpv* argument always overrides a default EPV in the server stub.

The MIDL compiler automatically generates a manager EPV data type (RPC_MGR_EPV) for a server application to use in constructing manager EPVs. A manager EPV must contain exactly one entry point (function address) for each procedure defined in the IDL file.

A server must supply a non-null EPV in the following cases:

- When the names of manager routines differ from the procedure names declared in the interface definition.
- When the server uses the default EPV for registering another implementation of the interface.

A server declares a manager EPV by initializing a variable of type *if-name*_**SERVER_EPV** for each implementation of the interface.

## Registering Only One Manager of an Interface

When a server offers only one implementation of an interface, the server calls the **RpcServerRegisterIfEx** or **RpcServerRegisterIf** routine only once. In the simplest case, the server uses the default manager EPV. (The exception is when the manager uses routine names that differ from those declared in the interface.)

For the simple case, you supply the following values in the **RpcServerRegisterIf*** call:

- Manager EPVs

  To use the default EPV, you specify a null value for the *MgrEpv* argument.
- Manager type UUID

  When using the default EPV, you can register the interface with a nil manager type UUID by supplying either a null value or a nil UUID for the *MgrTypeUuid* argument. In this case, all remote procedure calls, regardless of the object UUID in their binding handle, are dispatched to the default EPV, assuming no **RpcObjectSetType** calls have been made.

  You can also provide a non-nil manager type UUID. In this case, you must also call the **RpcObjectSetType** routine.

## Registering Multiple Implementations of an Interface

You can supply more than one implementation of the remote procedure(s) specified in the IDL file. The server application calls **RpcObjectSetType** to map object UUIDs to type UUIDs and calls **RpcServerRegisterIfEx** or **RpcServerRegisterIf** to associate manager EPVs with a type UUID. When a remote procedure call arrives with its object UUID, the RPC server run-time library maps the object UUID to a type UUID. The server application then uses the type UUID and the interface UUID to select the manager EPV.

You can also specify your own function to resolve the mapping from object UUID to manager type UUID. You specify the mapping function when you call **RpcObjectSetInqFn**.

To offer multiple implementations of an interface, a server must register each implementation by calling the **RpcServerRegisterIf*** routine separately. For each implementation a server registers, it supplies the same *IfSpec* argument, but a different pair of *MgrTypeUuid* and *MgrEpv* arguments.

In the case of multiple managers, use the **RpcServerRegisterIf*** routine as follows:

- Manager EPVs

  To offer multiple implementations of an interface, a server must:
  - Create a non-null manager EPV for each additional implementation.
  - Specify a non-null value for the *MgrEpv* argument in the **RpcServerRegisterIf*** routine.

  Please note that the server can also register with the default manager EPV.
- Manager type UUID

  Provide a manager type UUID for each EPV of the interface. The nil type UUID (or null value) for the *MgrTypeUuid* argument can be specified for one of the manager EPVs. Each type UUID must be different.

## Rules for Invoking Manager Routines

The RPC run-time library dispatches an incoming remote procedure call to a manager that offers the requested RPC interface. When multiple managers are registered for an interface, the RPC run-time library must select one of them. To select a manager, the RPC run-time library uses the object UUID specified by the call's binding handle.

The run-time library applies the following rules when interpreting the object UUID of a remote procedure call:

- Nil object UUIDs

  A nil object UUID is automatically assigned the nil type UUID (it is illegal to specify a nil object UUID in the **RpcObjectSetType** routine). Therefore, a remote procedure call whose binding handle contains a nil object UUID is automatically dispatched to the manager registered with the nil type UUID, if any.
- Non-nil object UUIDs

  In principle, a remote procedure call whose binding handle contains a non-nil object UUID should be processed by a manager whose type UUID matches the type of the object UUID. However, identifying the correct manager requires that the server has specified the type of that object UUID by calling the **RpcObjectSetType** routine.

If a server fails to call the **RpcObjectSetType** routine for a non-nil object UUID, a remote procedure call for that object UUID goes to the manager EPV that services remote procedure calls with a nil object UUID (that is, the nil type UUID).

Remote procedure calls with a non-nil object UUID in the binding handle cannot be executed if the server assigned that non-nil object UUID a type UUID by calling the **RpcObjectSetType** routine, but did not also register a manager EPV for that type UUID by calling the **RpcServerRegisterIfEx** or **RpcServerRegisterIf** routine.

| Object UUID of call | Server set type for object UUID? | Server registered EPV type? | Dispatching action |
|---|---|---|---|
| Nil | Not applicable | Yes | Uses the manager with the nil type UUID. |
| Nil | Not applicable | No | Error (RPC_S_UNSUPPORTED_TYPE); rejects the remote procedure call. |
| Non-nil | Yes | Yes | Uses the manager with the same type UUID. |
| Non-nil | No | Ignored | Uses the manager with the nil type UUID. If no manager with the nil type UUID, error (RPC_S_UNSUPPORTEDTYPE); rejects the remote procedure call. |
| Non-nil | Yes | No | Error (RPC_S_UNSUPPORTEDTYPE); rejects the remote procedure call. |

The object UUID of the call is the object UUID found in a binding handle for a remote procedure call.

The server sets the type of the object UUID by calling **RpcObjectSetType** to specify the type UUID for an object.

The server registers the type for the manager EPV by calling **RpcServerRegisterIf*** using the same type UUID.

The nil object UUID is always automatically assigned the nil type UUID. It is illegal to specify a nil object UUID in the **RpcObjectSetType** routine.

## Dispatching a Remote Procedure Call to a Server-Manager Routine

The following tables show the steps taken by the RPC run-time library to dispatch a remote procedure call to a server-manager routine.

Assume a simple case where the server registers the default manager EPV, as described in the following tables:

**Interface registry table**

| Interface UUID | Manager type UUID | Entry-point vector |
|---|---|---|
| *uuid1* | Nil | Default EPV |

**Object registry table**

| Object UUID | Object type |
|---|---|
| Nil | Nil |
| (Any other object UUID) | Nil |

**Mapping the binding handle to an entry-point vector**

| Interface UUID (from client binding handle) | Object UUID (from client binding handle) | Object type (from object registry table) | Manager EPV (from interface registry table) |
|---|---|---|---|
| *uuid1* | Nil | Nil | Default EPV |
| Same as above | *uuidA* | Nil | Default EPV |

The following steps describe the actions taken by the RPC server run-time library:

1. The server calls **RpcServerRegisterIfEx** or **RpcServerRegisterIf** to associate an interface it offers with the nil manager type UUID and with the MIDL-generated default manager EPV. This call adds an entry in the interface registry table. The interface UUID is contained in the *IfSpec* argument.
2. By default, the object registry table associates all object UUIDs with the nil type UUID. In this example, the server does not call **RpcObjectSetType**.
3. The server run-time library receives a remote procedure code containing the interface UUID the call belongs to and the object UUID from the call's binding handle.

   See the following function reference entries for discussions of how an object UUID is set into a binding handle:

   - **RpcNsBindingImportBegin**
   - **RpcNsBindingLookupBegin**
   - **RpcBindingFromStringBinding**
   - **RpcBindingSetObject**

4. Using the interface UUID from the remote procedure call, the server's run-time library locates that interface UUID in the interface registry table.

   If the server did not register the interface using **RpcServerRegisterIf***, the remote procedure call returns to the caller with an RPC_S_UNKNOWN_IF status code.
5. Using the object UUID from the binding handle, the server's run-time library locates that object UUID in the object registry table. In this example, all object UUIDs map to the nil object type.

6. The server's run-time library locates the nil manager type in the interface registry table.
7. Combining the interface UUID and nil type in the interface registry table resolves to the default EPV, which contains the server-manager routines to be executed for the interface UUID found in the remote procedure call.

Assume that the server offers multiple interfaces and multiple implementations of each interface, as described in the following tables:

**Interface registry table**

| Interface UUID | Manager type UUID | Entry-point vector |
|---|---|---|
| *uuid1* | Nil | *epv1* |
| *uuid1* | *uuid3* | *epv4* |
| *uuid2* | *uuid4* | *epv2* |
| *uuid2* | *uuid7* | *epv3* |

**Object registry table**

| Object UUID | Object type |
|---|---|
| *uuidA* | *uuid3* |
| *uuidB* | *uuid7* |
| *uuidC* | *uuid7* |
| *uuidD* | *uuid3* |
| *uuidE* | *uuid3* |
| *uuidF* | *uuid8* |
| Nil | Nil |
| (Any other UUID) | Nil |

**Mapping the binding handle to an entry-point vector**

| Interface UUID (from client binding handle) | Object UUID (from client binding handle) | Object type (from object registry table) | Manager EPV (from interface registry table) |
|---|---|---|---|
| *uuid1* | Nil | Nil | *epv1* |
| *uuid1* | *uuidA* | *uuid3* | *epv4* |
| *uuid1* | *uuidD* | *uuid3* | *epv4* |
| *uuid1* | *uuidE* | *uuid3* | *epv4* |
| *uuid2* | *uuidB* | *uuid7* | *epv3* |
| *uuid2* | *uuidC* | *uuid7* | *epv3* |

The following steps describe the actions taken by the server's run-time library as shown in the preceding tables when called by a client with interface UUID *uuid2* and object UUID *uuidC*:

1. The server calls **RpcServerRegisterIfEx** or **RpcServerRegisterIf** to associate the interfaces it offers with the different manager EPVs. The entries in the interface registry table reflect four calls of **RpcServerRegisterIf*** to offer two interfaces, with two implementations (EPVs) for each interface.
2. The server calls **RpcObjectSetType** to establish the type of each object it offers. In addition to the default association of the nil object to a nil type, all other object UUIDs not explicitly found in the object registry table also map to the nil type UUID.

In this example, the server calls the **RpcObjectSetType** routine six times.

3. The server run-time library receives a remote procedure call containing the interface UUID the call belongs to and an object UUID from the call's binding handle.
4. Using the interface UUID from the remote procedure call, the server's run-time library locates the interface UUID in the interface registry table.
5. Using the object UUID from the binding handle, uuidC, the server's run-time library locates the object UUID in the object registry table and finds that it maps to type *uuid7*.
6. The server's run-time library locates the manager type by combining the interface UUID, *uuid2*, and type *uuid7* in the interface registry table. This resolves to *epv3*, which contains the server-manager routine to be executed for the remote procedure call.

The routines in *epv2* will never be executed because the server has not called the **RpcObjectSetType** routine to add any objects with a type UUID of *uuid4* to the object registry table.

A remote procedure call with interface UUID *uuid2* and object UUID *uuidF* returns to the caller with an RPC_S_UNKNOWN_MGR_TYPE status code because the server did not call the **RpcServerRegisterIf*** routine to register the interface with a manager type of *uuid8*.

## Return Values

| Value | Meaning |
|---|---|
| RPC_S_OK | Success |
| RPC_S_TYPE_ALREADY_REGISTERED | Type UUID already registered |

## See Also

**RpcBindingFromStringBinding**, **RpcBindingSetObject**, **RpcNsBindingExport**, **RpcNsBindingImportBegin**, **RpcNsBindingLookupBegin**, **RpcObjectSetType**, **RpcServerRegisterIfEx**, **RpcServerRegisterIf**, **RpcServerUnregisterIf**

## Supplying Your Own Object-Inquiry Function

Consider a server that manages thousands of objects of many different types. Whenever the server started, the server application would have to call the function **RpcObjectSetType** for every one of the objects, even though clients might refer to only a few of the objects (or take a long time to refer to them). The thousands of objects are likely to be on disk so that retrieving their types would be time consuming. Also, the internal table that is mapping the object UUID to the manager type UUID would essentially duplicate the mapping maintained with the objects themselves.

For convenience, the RPC function set includes the function **RpcObjectSetInqFn**. With this function, you provide your own object-inquiry function.

As an example, you can supply your own object-inquiry function when you map objects 100 - 199 to type number 1, 200 - 299 to type number 2, and so on. The object-inquiry function can also be extended to a distributed file system, where the server application does not "know" all the files (object UUIDs) available, or when files in the file system are named by object UUIDs and you do not want to preload all object-UUID-to-type-UUID mappings.

## Listening for Clients

After registering the protocol sequence, endpoint, and interface, and advertising the availability of the server application in the name-service database, the server calls **RpcServerListen** to indicate to the run-time library that it is ready to accept remote procedure calls from clients.

The OSF-DCE implementation of **RpcServerListen** does not return to the server application until another server thread calls **RpcMgmtStopServerListening**. The call to **RpcServerListen** ties up the server-manager thread.

Microsoft has extended the OSF-DCE definition of this function. You supply a flag that indicates whether to wait or to return immediately to the server application to allow further processing. When your server application uses this option, it can call a new function, **RpcMgmtWaitServerListen**, to perform the wait operation.

The wait functionality prevents the server from terminating an active client operation. When the server has selected the wait option by calling **RpcServerListen** or **RpcMgmtWaitServerListen**, the server waits until all client operations are complete before shutting down the server-manager application.

## Client Application RPC API Calls

To make the remote procedure call, the client must obtain a binding handle. There are two ways to obtain a binding handle:

- Importing from the name-service database. The client specifies the name of the name-service database entry and obtains a binding handle.
- Constructing individual strings that represent the client object UUID, server, protocol sequence, network address, endpoint, and options. Call the function **RpcStringBindingCompose** to assemble these strings into the correct syntax for a string binding, and then call **RpcBindingFromStringBinding** to obtain the binding handle.

For information see String Bindings.

Use the RPC name service in both client and server applications for ease of administration and maintenance.

## Importing from the Name-Service Database

When the server application is registered with the name-service database, the client can obtain binding handles by using one of two equivalent methods:

- Importing (call **RpcNsBindingImportBegin**, **RpcNsBindingImportNext**, and **RpcNsBindingImportDone**).
- Looking up and selecting (call **RpcNsBindingLookupBegin**, **RpcNsBindingLookupNext**, **RpcNsBindingSelect**, and **RpcNsBindingLookupDone**).

The import method returns a single binding handle while the look-up method returns a binding vector from which the application selects one binding handle using the function **RpcNsBindingSelect**.

The client queries the name service by supplying the logical name the client uses to refer to the server application. The name-service provider returns a binding handle.

The client can also choose to supply a null name (an empty string or a null pointer). In this case, the Microsoft Locator searches for name-service database entries that match the supplied interface UUID. The search varies slightly between the OSF-DCE CDS and the Microsoft Locator.

The OSF-DCE implementation of the name-service provider uses the DEFAULT_ENTRY environment variable, which is usually the name of a profile, to search for an entry that matches the interface ID specified in the import call. See NSID documentation for more details.

The Microsoft Locator implementation of the name-service provider does not use DEFAULT_ENTRY and does not support group or profile entries. Instead, all entries in the primary locator (at the domain controller) are combined to form a default profile. When no matches are found in that domain, the client application can search in another domain. For more information about specifying the domain name, see Name Syntax.

## Exception Handling

Microsoft RPC uses the same approach to exception handling as the Microsoft Win32 API.

With Microsoft Windows NT and Windows 95, the **RpcTryFinally** / **RpcFinally** / **RpcEndFinally** structure is equivalent to the Win32 **try-finally** statement. The RPC exception construct **RpcTryExcept** / **RpcExcept** / **RpcEndExcept** is equivalent to the Win32 **try-except** statement.

The exception-handler structures in Microsoft RPC are provided so they can also be supported on computers with MS-DOS and Windows 3.*x*. When you use the RPC exception handlers, your client-side source code is portable to Windows NT, Windows 95, Windows 3.*x*, and MS-DOS. The different RPC header files provided for each platform resolve the **RpcTry** and **RpcExcept** structures for each platform. In the Win32 environment, these macros map directly to the Win32 **try-finally** and **try-except** statements. In other environments, these macros map to other platform-specific implementations of exception handlers.

The RPC exception-handling macros provide consistent **try-except** support across MS-DOS, Windows 3.*x*, Windows 95, and Windows NT. With Windows NT and Windows 95, RPC **try-except** support expands into Win32 **try-except** support.

When you write distributed applications for Windows NT and Windows 95 only, use the Win32 **try-except** and **try-finally** statements. If you are writing for MS-DOS and Windows 3.*x*, use the RPC versions of these macros. Potential exceptions raised by these structures include the set of error codes returned by the RPC functions with the prefixes "RPC_S_" and "RPC_X" and the set of exceptions returned by Win32.

Exceptions that occur in the server application, server stub, and server run-time library (above the transport layer) are propagated to the client. This propagation feature includes multiple layers of callbacks. No exceptions are propagated from the server transport level. The following figure shows how exceptions are returned from the server to the client.

{ewc msdncd, EWGraphic, bsd23536 1 /a "SDK_A20.BMP"}

The RPC exception handlers differ slightly from the OSF-DCE exception-handling macros TRY, FINALLY, and CATCH. Various vendors provide include files that map the OSF-DCE RPC functions to the Microsoft RPC functions, including TRY, CATCH, CATCH_ALL, and ENDTRY. These header files also map the RPC_S_* error codes onto the OSF-DCE exception counterparts, rpc_s_*, and map RPC_X_* error codes to rpc_x_*. For OSF-DCE portability, use these include files.

For more information about the RPC exception handlers, see **RpcExcept** and **RpcFinally**. For more information about the Win32 exception handlers, see your Win32 API documentation.

# Security

Microsoft RPC supports two different methods for adding security to your distributed application:

- A security package that can be accessed using the RPC functions.
- The security features built into Windows NT transport protocols.

The transport-level security method is not the preferred method. We recommend you use RPC security because it works on all transports, across platforms, and provides a high levels of security, including privacy.

## Using Authenticated RPC

While previous versions of Microsoft RPC depended on the security built into the named-pipe transport, this version also implements the transport-independent security functions from OSF-DCE RPC, using the Windows NT Security Service as the default security provider. This higher-level security enables servers to filter client requests based on an authenticated identity associated with each request.

## An Overview of Authenticated RPC

To use authenticated RPC, a client passes its user security information to the run-time library. This security information is called the client credentials. The client run-time library forwards the credentials to the server run-time library which then passes it to the relevant security provider for verification. (In this version of Microsoft RPC, the NT Security Service is the only supported security provider. Other security providers may be added in the future.) When a call is made, the security provider ensures that the credentials are valid. If so, the server stub is called and the call proceeds. Otherwise, the client is denied access and the call fails.

Authenticated RPC involves a series of tasks performed by all servers every time a client tries to connect. The server must:

1. Extract binding information about the client from the incoming call.
2. Extract the authentication information from the binding handle and check the credentials with the NT Security Service.
3. Compare the client's authentication information with the access control list (ACL) on the security server's database.

## Writing a Secure Server

If your server registers with a security provider, client calls with invalid credentials will not be dispatched. However, calls with no credentials will be dispatched. There are three ways to keep this from happening:

- Register the interface using **RpcServerRegisterIfEx**, with a security callback function; this will cause the RPC runtime to automatically reject unauthenticated calls to that interface.
- Call **RpcBindingInqAuthClient** to determine the security level in use by the client; your stub can then return an error if the client is unauthenticated.
- Only allow calls using the RPC_C_AUTHN_PACKET_PRIVACY level. Then, all server replies will be encrypted during transmission.

**Note** If you are using the NT LAN Manager Security Support Provider (by means of the authentication-service constant RPC_C_AUTHN_WINNT), you should be aware that a client whose credentials specify an unknown user name will be given "guest" access permission. If you do not want this behavior, remove the "guest" account from your server.

The NTLMSSP provider also lets your server impersonate the client by calling **RpcImpersonateClient**. For more information on the NT security model, read Security Model.

If you need additional information on how to write a secure server, check with the manufacturer of your security provider.

## Implementing Security for Clients

To set up a binding handle for authenticated RPC, a client application calls **RpcBindingSetAuthInfo**. Without this call, all remote procedure calls on the binding handle will be unauthenticated. The chosen level of security and authentication applies only to that binding handle. Context handles derived from the binding handle will use the same security information, but subsequent modifications to the binding handle will not be reflected in the context handles. The security and authentication level stays in effect until another level of security is chosen, or until the process terminates. Most applications will not require a change in the security level.

The levels of security and authentication available for authenticated RPC are shown in the following table:

| Authentication-Level Constant | Description |
|---|---|
| RPC_C_AUTHN_LEVEL_DEFAULT | Uses the default authentication level for the specified authentication service. |
| RPC_C_AUTHN_LEVEL_NONE | Performs no authentication. |
| RPC_C_AUTHN_LEVEL_CONNECT | Authenticates only when the client establishes a relationship with the server. Datagram RPC does not support this level and instead uses the RPC_C_AUTHN_LEVEL_PKT level. |
| RPC_C_AUTHN_LEVEL_CALL | Authenticates only at the beginning of each remote procedure call when the server receives the request. Although defined by OSF, neither datagram nor connection-oriented RPC supports this level. RPC uses the RPC_C_AUTHN_LEVEL_PKT level instead. |
| RPC_C_AUTHN_LEVEL_PKT | Authenticates and verifies that all data received is from the expected client. |
| RPC_C_AUTHN_LEVEL_PKT_INTEGRITY | Authenticates and verifies that none of the data transferred between client and server has been modified. |
| RPC_C_AUTHN_LEVEL_PKT_PRIVACY | Authenticates all previous levels and encrypts the argument value of each remote procedure call. Note that this level is unavailable in France due to legal restrictions. |

**Note** The RPC_C_AUTHN_LEVEL_CALL, RPC_C_AUTHN_LEVEL_PKT, RPC_C_AUTHN_LEVEL_PKT_INTEGRITY, and RPC_C_AUTHN_LEVEL_PKT_PRIVACY are only supported for clients communicating with a Windows NT server. A Windows® 95 server can only accept incoming calls at the RPC_C_AUTHN_LEVEL_CONNECT level.

The level of security required depends entirely on the application. When choosing a security level for your application, remember that the higher the protection level, the greater the overhead required to create and maintain the levels. Additionally, there are performance trade-offs to consider. The checksum computation and encryption required by the RPC run-time library can make data protection a time-consuming operation. The more often credentials are checked, the more time it will take to get on with the business of the application. Use the authentication-level constant that offers the protection your application needs.

Note that authentication-level constants cannot be combined.

## Differences in Platforms

When developing applications for MS-DOS, you must feed in the password and credential information to **RpcBindingSetAuthInfo** manually. This is optional for a 16-bit or 32-bit Windows platform which, by default, uses the credentials for the currently logged-in user. If a Windows 95, Windows For Workgroups, or Windows 3.*x* workstation is not part of a domain, the user will be prompted for the password.

To manually pass credentials to **RpcBindingSetAuthInfo**, create a pointer to the SEC_WINNT_AUTH_IDENTITY structure and pass in the credential information under the *AuthIdentity* parameter. Note that this structure must remain valid for the lifetime of the binding handle.

# Windows 95 Considerations

For systems configured for NetWare clients, the server side of the application must obtain the server principal name, and then pass this value to **RpcServerRegisterAuthInfo**. Use the **RpcServerInqDefaultPrincName** routine to obtain the server principal name. In this situation, the client calls **RpcBindingSetAuthInfo** in the usual manner, but a value of NULL is specified for *PrincipalName*. Behind the scenes, the Windows 95 run-time library queries the server to obtain the value of *PrincipalName* specified to **RpcServerRegisterAuthInfo**. This is the name that is actually used. The binding handle will be authenticated on the NetWare server.

For Windows 95, if **RpcBindingSetAuthInfo** is called with a NULL server principal name (as described above), the binding handle must be fully bound. If it is a dynamic endpoint in which the server registers the endpoint with the endpoint mapper and, therefore, is not known by the client, you must use **RpcEpResolveBinding** to bind the handle. This is because in order to obtain the principal name from the server, the Windows 95 run-time library implicitly calls **RpcMgmtInqServerPrincName**; calls to management interfaces cannot be made to unbound handles. All RPC server processes have the same management interface. Registering the handle with the endpoint mapper is not sufficient to uniquely identify a server.

**Note** The **ncacn_np** and **ncalrpc** security descriptors are ignored by the Windows 95 run-time library, because Windows 95 does not support the Windows NT security model.

# Providing Client Credentials to the Server

Servers use the client's binding information to enforce security. Clients always pass a binding handle as the first parameter of a remote procedure call. However, servers cannot use the handle unless it is declared as the first parameter to remote procedures in either the IDL file or in the server's ACF. You can choose to list the binding handle in the IDL file, but this forces all clients to declare and manipulate the binding handle rather than using automatic or implicit binding if they choose.

Another method is to leave the binding handles out of the IDL file and to place the **explicit_handle** attribute into the server's ACF. In this way, the client can use whatever type of binding is best suited to the application, while the server uses the binding handle as though it were declared explicitly.

The processs of extracting the client credentials from the binding handle is shown below:

- RPC clients call **RpcBindingSetAuthInfo** and include their authentication information as part of the binding information passed to the server.
- Usually, the server calls **RpcImpersonateClient** in order to behave as though it were the client. If the binding handle is not authenticated, the call fails with RPC_S_NO_CONTEXT_AVAILABLE. To obtain the client's user name, call **GetUserName** while impersonating.
- The server will normally create objects with ACLs by using the Windows NT call **CreatePrivateObjectSecurity**. After this is accomplished, later security checks become automatic.

## Windows NT Transport Security

Although this is not the preferred method, you can add security features to your distributed application by using the security settings offered by the Windows NT named-pipe transport. These security settings are used with the Microsoft RPC functions that start with the prefixes "RpcServerUseProtseq" and "RpcServerUseAllProtseqs" and the functions **RpcImpersonateClient** and **RpcRevertToSelf**.

> **Note** If you are running an application that is a service and you are using NTLMSSP (Windows NT LAN Manager Security Support Provider) for security, you must add an explicit service dependency for your application. The NTLMSSP.DLL will call the Service Controller (SC) to begin the NTLMSSP service. However, an RPC application that is a service and is running as a system, must also contact the SC unless it is connecting to another service on the same computer.

## Impersonation

Impersonation is useful in a distributed computing environment when servers must pass client requests to other server processes or to the operating system. In this case, a server impersonates the client's security context. Other server processes can then handle the request as if it had been made by the original client.

For example, a client makes a request to Server A. If Server A must query Server B to complete the request, Server A impersonates client security context and makes the request to Server B on behalf of the client. Server B uses the original client's security context, instead of the security identity for Server A, to determine whether to complete the task.

The server calls **RpcImpersonateClient** to overwrite the security for the server thread with the client security context. After the task is completed, the server calls **RpcRevertToSelf** or **RpcRevertToSelfEx** to restore the security context defined for the server thread.

When binding, the client can specify quality-of-service information about security that specifies how the server can impersonate the client. For example, one of the settings lets the client specify that the server is not allowed to impersonate it.

## Using Transport-Level Security on the Server

When you use **ncacn_np** or **ncalrpc** as the protocol sequence, the server specifies a security descriptor for the endpoint at the time it selects the protocol sequence. The security descriptor is provided as an additional parameter (an extension to the standard OSF-DCE parameters) on all functions that start with the prefixes "RpcServerUseProtseq" and "RpcServerUseAllProtseqs." The security descriptor controls whether a client can connect to the endpoint.

Each Windows NT process and thread is associated with a security token. This token includes a default security descriptor that is used for any objects created by the process, such as the endpoint. If no security descriptor is specified when calling a function with the prefixes "RpcServerUseProtseq" and "RpcServerUseAllProtseqs," the default security descriptor from the process security token is applied to the endpoint.

To guarantee that the server application is accessible to all clients, the administrator should start the server application on a process that has a default security descriptor which can be used by all clients. In Windows NT, generally only system processes have a default security descriptor.

For more information about these functions and the functions **RpcImpersonateClient** and **RpcRevertToSelf**.

## Using Transport-Level Security on the Client

The client specifies how the server impersonates the client when the client establishes the string binding. This quality-of-service information is provided as an endpoint option in the string binding. The client can specify the level of impersonation, dynamic or static tracking, and the effective-only flag.

To supply quality-of-service information for the server, the client performs the following steps:

1. Imports a handle from the name-service database.
   The client specifies the name of the name-service database entry and obtains a binding handle.
2. Calls **RpcBindingToStringBinding** to obtain the protocol sequence, network address, and endpoint.
3. Calls **RpcStringBindingParse** to split the string binding into its component substrings.
4. Verifies that the protocol sequence is equal to **ncacn_np** or **ncalrpc**.
   Client quality-of-service information is supported only on named pipes and LRPC in Microsoft RPC.
5. Adds the security information to the endpoint string as an option.
   For more information about the syntax, see **String Binding**.
6. Calls **RpcStringBindingCompose** to reassemble the component strings, including the new endpoint options, in the correct string-binding syntax.
7. Calls **RpcBindingFromStringBinding** to obtain a new binding handle and to apply the quality-of-service information for the client.
8. Makes remote procedure calls using the handle.

Microsoft RPC supports Windows NT security features only on **ncacn_np** and **ncalrpc**. Windows NT security options for other transports are ignored.

> **Note** Because it does not support the Windows NT security model, the Windows 95 run-time library ignores the security descriptors **ncalrpc** and **ncacn_np**.

The following security parameters can be associated by the client with the binding for the named-pipe transport **ncacn_np** or **ncalrpc**:

- **Identification**, **Impersonation**, or **Anonymous**. Specifies the type of security used.
- **Dynamic** or **Static**. Determines whether security information associated with a thread is a copy of the security information created at the time the remote procedure call is made (static) or a pointer to the security information (dynamic).
  Static security information does not change. The dynamic setting reflects the current security settings, including changes made after the remote procedure call is made.
- **TRUE** or **FALSE**. Specifies the value of the effective-only flag. A value of TRUE indicates that only security settings set to "on" at the time of the call are effective. A value of FALSE indicates that all security settings are available and can be modified by the application.

Any combination of these settings can be assigned to the binding, as shown in the following example:

```
"Security=Identification Dynamic True"
"Security=Anonymous Static True"
"Security=Impersonation Static False"
```

Default security-parameter settings vary according to the transport protocol.

For more information about the security features of Windows NT, see your Microsoft Windows NT documentation. For information about the syntax of the endpoint options, see **endpoint**.

# Installing and Configuring RPC Applications

When Windows NT or Windows 95 is installed on a server or client, the RPC run-time files are automatically installed as well. No further RPC installation is required. You must ensure, however, that the version of Windows NT or Windows 95 is one that supports all the features used in your distributed application. See Targetting Stubs for Specific 32-Bit Platforms for more information.

To run MS-DOS or Microsoft Windows 16-bit clients on a 32-bit Windows operating system, your application's install program must install the proper Windows/MS-DOS executable files, as described below.

When you use an RPC application on a Windows 3.*x* or MS-DOS platform, the RPC run-time executable files must be copied to the Windows 3.*x* or MS-DOS computers that will be using the application. The directory \mstools\rpc_rt16 on the Win32 SDK CD contains a disk image with these files along with a setup program to install the files. Use this disk image to create an install disk for distribution with your RPC application.

When you build an RPC application for an Apple Macintosh client, the necessary files are linked to the application at build time and no additional RPC installation is needed.

For more details about redistributable files and licensing agreements, see \LICENSE\REDIST.TXT and \LICENSE\LICENSE.TXT on the Win32 SDK CD.

## See Also

Configuring the Name Service Provider, Configuring the Name Service for Windows 95, Developing 16-bit Windows and MS-DOS Client Applications, Developing Macintosh Client Applications.

# Configuring the Name Service Provider

If your distributed application registers its interface with a name-service, both the client and server must be using the same name-service. Microsoft RPC interoperates with the Microsoft Locator and any name-service provider that adheres to the Microsoft RPC name-service interface (NSI) – for example, the DCE Cell Directory Service accessed through Digital Equipment Corporation's name-service daemon (nsid). The Locator, which is designed for use in Windows environments, is the default name-service provider.

## Windows NT

When you install the Win32 SDK on Microsoft Windows NT, the Locator is automatically selected as the name-service provider. See Reconfiguring the Name Service for Windows NT for information on selecting a different name-service provider.

## Windows 95

Windows 95 does not use the Locator. If your Windows 95 application is to use a name service, it must be part of a workgroup or domain that includes a Windows NT machine to serve as a proxy name-service provider, or be connected to a host machine running the NSI daemon to serve as a gateway to DCE CDS. For more information, see Configuring the Name Service for Windows 95.

## MS-DOS and Windows 3.*x*

When you run SETUP.EXE to install the 16-bit RPC run-time library, you are prompted to select a name service provider.

When you choose Install Default Name Service Provider, the default name-service provider, the Microsoft Locator, is installed. The Microsoft Locator works in Microsoft Windows NT domains.

When you choose Install Custom Name Service Provider, complete the Define Network Address dialog box to install the DCE Cell Directory Service as your name-service provider. The DCE Cell Directory Service is the name-service provider used with DCE servers.

The network address is the name of the host computer that runs the NSI daemon (nsid). This machine acts as a gateway to the DCE Cell Directory Service, passing name-service interface function calls between computers that run Microsoft operating systems and DCE computers. The network address can be 80 characters or less – for example, 11.1.9.169 is a valid address.

## Configuring the Name Service for Windows 95

Windows 95 does not use the Microsoft Locator. In order to use a name service in a Windows 95 application, the Windows 95 machine must either:

- Be part of a workgroup or domain that includes a Windows NT machine to serve as a proxy name service provider, or
- Be connected to a host machine running the NSI daemon (nsid), which serves as a gateway to Digital Equipment Corporation's DCE Cell Directory Service.

Edit the Windows 95 registry to configure the name service provider.

### To designate a Microsoft Locator name-service provider for Windows 95

1. Use REGEDIT to edit the Windows 95 registry.
2. Select HKEY_LOCALMACHINE\SOFTWARE\Microsoft\Rpc.
3. Create a new key called NameService.
4. With the NameService key selected, create the new String Value names and modify them to contain the data as shown:

| Name | Data |
|---|---|
| 4DefaultSyntax | "3" |
| Protocol | "ncacn_np" |
| Endpoint | "\pipe\locator" |
| NetworkAddress | "myserver" (where myserver is the name of the NT machine) |
| ServerNetworkAddress | "myserver" |

### To designate a DCE CDS name-service provider for Windows 95

- Edit the Windows 95 registry as described above, using the following data:

| Name | Data |
|---|---|
| DefaultSyntax | "3" |
| Protocol | "ncacn_ip_tcp" |
| Endpoint | "" (an empty string) |
| NetworkAddress | "myserver" ( the name of the host machine running nsid) |
| ServerNetworkAddress | "myserver" ( the name of the host machine running nsid) |

**Note** You must have Digital Equipment Corporation's DCE Cell Directory Service product to configure the DCE CDS as your name-service provider. See the documentation provided by Digital Equipment Corporation for information about DCE CDS.

## Reconfiguring the Name Service for Windows NT

When you install the Win32 SDK on Microsoft Windows NT, the Microsoft Locator is automatically selected as the name-service provider. You can change the name-service provider through the Windows Control Panel.

### To reconfigure the name-service provider for Windows NT

1. In the Control Panel, choose the Network icon.

   The Network dialog box appears.
2. In the Network dialog box, choose Configure.
3. In the Network Software list, select RPC Configuration.

   The RPC Name Service Provider Configuration dialog box appears.
4. In the RPC Name Service Provider Configuration dialog box, select a name-service provider from the list.
   - When you choose the Microsoft Locator, choose OK. The configuration process is then complete.
   - When you choose the DCE Cell Directory Service, in the Network Address box type the name of the host computer that runs the NSI daemon (nsid), and then choose OK.

     The host computer that runs the nsid acts as a gateway to the DCE Cell Directory Service, passing name-service interface function calls between computers that run Microsoft operating systems and DCE computers. A network address can be up to 80 characters – for example, 11.1.9.169 is a valid address.

     **Note** You must have Digital Equipment Corporation's DCE CDS product to configure the DCE CDS as your name-service provider. See the documentation provided by Digital Equipment Corporation for information about DCE CDS.

## Reconfiguring the Name Service for Windows 3.*x*/MS-DOS

When you install the Win32 SDK for Windows 3.*x*/MS-DOS, you specify a name-service provider. You can change the name-service provider you specified by editing the RPCREG.DAT configuration file, which contains the name-service-provider parameters and RPC protocol settings. Use a text editor to change name-service provider entries.

### To reconfigure the Microsoft Locator name-service provider

1. Open the RPCREG.DAT file using a text editor.

   RPCREG.DAT is in the root directory unless you specified a different path during the Setup program.
2. Set the following values for the registry entries:

| Registry entry | Value |
|---|---|
| Software\Microsoft\RPC\NameService\Protocol | The protocol sequence for the protocol you are using. The default is **ncacn_np**. |
| Software\Microsoft\RPC\NameService\NetworkAddress | The name of the computer running the Locator that is used by clients during name-service lookup operations. The default is the primary domain controller. |
| Software\Microsoft\RPC\NameService\Endpoint | The name of the endpoint used by the name service. The default is \pipe\locator. |

3. Save and close the file.

### To configure the DCE CDS name-service provider

- You must have Digital Equipment Corporation's DCE Cell Directory Service product to configure the DCE CDS as your name-service provider. See the documentation provided by Digital Equipment Corporation for information about DCE CDS.

## Registry Information

When you install Microsoft Windows NT or Windows 95, or when you run the Windows 3.x/MS-DOS Setup programs, the RPC protocol information you specify is stored in the registry file. The 32-bit Windows registry entries are automatically configured and no further configuration is necessary. With MS-DOS and Windows 3.x, use a text editor to change entries in the RPCREG.DAT file:

| Registry entry | Description |
|---|---|
| SOFTWARE\Microsoft\Rpc\NameService\DefaultSyntax | Specifies the default syntax that is used by the RPC functions **RpcNsBindingImportBegin** and **RpcNsBindingExport**. This registry entry corresponds to the DCE environment variable RPC_DEFAULT_ENTRY_SYNTAX. |
| SOFTWARE\Microsoft\Rpc\NameService\NetworkAddress | Specifies the address of the computer running the Locator that is used by clients during name-service lookup operations. The default setting is the primary domain controller. |
| SOFTWARE\Microsoft\Rpc\NameService\ServerNetworkAddress | Specifies the address of the computer running the Locator that is used by servers during name-service export operations. Default is PDC (Windows NT only). |
| SOFTWARE\Microsoft\Rpc\NameService\Endpoint | Specifies the endpoint used by the name service. |
| SOFTWARE\Microsoft\Rpc\NameService\Protocol | Specifies the protocol used by the name service. |
| SOFTWARE\Microsoft\Rpc\ClientProtocols\ncacn_np | Specifies the name of the RPC client transport DLL for named pipes. |
| SOFTWARE\Microsoft\Rpc\ClientProtocols\ncacn_ip_tcp | Specifies the name of the RPC client transport DLL for TCP/IP. |
| SOFTWARE\Microsoft\Rpc\ClientProtocols\ncacn_nb_nb | Specifies the name of the RPC client transport DLL for NetBEUI over NetBIOS. |
| SOFTWARE\Microsoft\Rpc\ClientProtocols\ncalrpc | Specifies the name of the RPC client transport DLL for local RPC (Windows NT only). |
| SOFTWARE\Microsoft\Rpc\ServerProtocols\ncacn_np | Specifies the name of the RPC server transport DLL for named pipes. |
| SOFTWARE\Microsoft\Rpc\ServerProtocols\ncacn_ip_tcp | Specifies the name of the RPC server transport DLL for TCP/IP. |
| SOFTWARE\Microsoft\Rpc\ServerProtocols\ncacn_nb_nb | Specifies the name of the RPC server transport DLL for NetBEUI. |
| SOFTWARE\Microsoft\Rpc\ | Specifies the name of the RPC |

| | |
|---|---|
| ServerProtocols\ncalrpc | server transport DLL for local RPC (Windows NT only). |
| SOFTWARE\Microsoft\Rpc\NetBios | Consists of mapping strings that map protocols to NetBIOS lana numbers. For NetBIOS information, see the following section. |

The Microsoft RPC Setup program automatically maps protocol strings to NetBIOS lana numbers and writes these settings in the registry. These mappings work as long as you only have one network card and one network protocol. If you have more than one network card and network protocol, or if you change your network configuration after installing Microsoft RPC, you must update the registry to indicate the new correspondences between protocol strings and NetBIOS lana numbers.

For 32-bit Windows platforms, the mapping string appears in the registry tree under \SOFTWARE\Microsoft\Rpc\NetBios. For MS-DOS and Windows 3.x, the mapping string appears in the registry file RPCREG.DAT.

The mapping string uses the following syntax:

**ncacn_nb_***protocol digit***=***lana_number*

*protocol*

Specifies the protocol type. The valid *protocol* values are as follows:

| Protocol | Protocol type |
|---|---|
| nb | NetBEUI |
| tcp | TCP/IP |

*digit*

Specifies a unique number associated with each instance of a protocol. Use the value 0 for the first instance of a protocol and use the next consecutive number for each additional instance of that protocol. For example, assign the value ncacn_nb_nb0 to the first NetBEUI entry; assign the value ncacn_nb_nb1 to the second NetBEUI entry.

*lana_number*

Specifies the NetBIOS lana number.

A unique lana number is associated with each network adapter present in the computer. For LAN Manager networks, the lana numbers for each network card are available in the configuration files LANMAN.INI and PROTOCOL.INI. For more information about the lana number, see your network documentation.

For example, the following mapping string describes a configuration that uses the NetBEUI protocol over an adapter card that is assigned lana number 0:

```
ncacn_nb_nb0=0
```

When you install a second card that supports both XNS and NetBEUI protocols, the mapping strings appear as follows:

```
ncacn_nb_tcp0=0
ncacn_nb_nb1=1
```

## Configuring the Windows NT Registry for Port Allocations and Selective Binding

The following registry keys specify the system defaults for dynamic port allocation and for binding to NICs on multihomed machines. You must first create these keys, then specify the appropriate settings. If a key is missing or if it contains an invalid value, then the entire configuration is marked as invalid and all RpcServerUseProtseq*(ncacn_ip_tcp) calls will fail.

**Note** Ports allocated to a process remain allocated until that process dies. If all available ports are in use, then the API will return RPC_S_OUT_OF_RESOURCES.

| Port Key | Data Type | Description |
|---|---|---|
| HKEY_LOCAL_MACHINE\Software\Microsoft\Rpc\Internet\Ports | REG_MULTI_SZ | Specifies a set of IP port ranges consisting of either all the ports available from the Internet or all the ports not available from the Internet. Each string represents a single port or an inclusive set of ports (for example,"1000-1050" "1984"). If any entries are outside the range of zero to 65535, or if any string cannot be interpreted, the RPC run time will treat the entire configuration as invalid. |
| HKEY_LOCAL_MACHINE\Software\Microsoft\Rpc\Internet\PortsInternetAvailable | REG_SZ | Y or N (not case-sensitive). If Y, the ports listed in the Ports key are all the Internet-available ports on that machine. If N, the ports listed in the Ports key are all those ports that are not Internet-available. |
| HKEY_LOCAL_MACHINE\Software\Microsoft\Rpc\Internet\UseInternetPorts | REG_SZ | Y or N (not case-sensitive). Specifies the system default policy. If Y, the processes using the default will be assigned ports from the set of Internet-available ports, as defined above. If N, processes using the default will be assigned ports from the set of intranet-only ports. |
| HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\Rpc\Linkage\Bind | REG_MULTI_SZ | Lists the device names of all the NICs on which to bind by default (for example, \Device\AMDPCN1). If the key does not exist, the server will bind to all NICs. If the key does exist, the server will bind to the NICs specified in the key, unless the NICFlags field is set to RPC_C_BIND_TO_ALL_NICS. If the key has a null ("") value, the configuration will be marked as invalid and all calls to RpcServerUseProseq(ncacn_ip_tcp) will fail. |

## See Also

RPC_POLICY, **RpcServerUseAllProtseqsEx**, **RpcServerUseAllProtseqsIfEx**, **RpcServerUseProtseqEx**, **RpcServerUseProtseqEpEx**, **RpcServerUseProtseqIfEx**, **ncacn_ip_tcp**

## SPX/IPX Installation

When using the **ncacn_spx** and **ncadg_ipx** transports, the server name is exactly the same as the Windows NT or Windows 95 server name. However, since the names are distributed using Novell protocols, they must conform to the Novell naming conventions. If a server name is not a valid Novell name, servers will not be able to create endpoints with the **ncacn_spx** or **ncadg_ipx** transports.

A valid Novell server name contains only the characters between 0x20 and 0x7f. Lowercase characters are changed to uppercase. The following characters cannot be used:

"*,./:;<=>?[]\|]

To maintain compatibility with the first version of Windows NT, **ncacn_spx** and **ncadg_ipx** also allow you to use a special format of the server name known as the tilde name. The tilde name consists of a tilde, followed by the server's eight-digit network number, and then followed by its twelve-digit Ethernet address. Tilde names have an advantage in that they do not require any name service capabilities. Thus, if you are connected to a server, the tilde name will work.

The following tables contain two sample configurations used to illustrate the points previously described:

| Component | Configured As |
|---|---|
| Windows NT or Windows 95 Server | NWCS |
| Windows NT or Windows 95 Client | NWCS |
| Windows 3.*x*/MS-DOS Client | NetWare Redirector |

The configuration in the previous table requires that you have NetWare file servers or routers on your network. It will produce the best performance because the server names are stored in the NetWare Bindery.

| Component | Configured As |
|---|---|
| Windows NT or Windows 95 Server | SAP Agent |
| Windows NT or Windows 95 Client | IPX/SPX |
| Windows 3.*x*/MS-DOS Client | IPX/SPX |

The second configuration works in an environment that does not contain NetWare file servers or routers (for example, a network of two computers: a Windows NT server and an MS-DOS client). Name resolution, which is accomplished during the first call over a binding handle, will be slightly slower than the first configuration. In addition, the second configuration results in more traffic generated over the network.

To implement name resolution, when an RPC server uses an SPX or IPX endpoint, the server name and endpoint are registered as a SAP server of type 640 (hexadecimal). To resolve a server name, the RPC client sends a SAP request for all services of the same type, and then scans the list of responses for the name of the server. This process occurs during the first RPC call over each binding handle. For additional information on the SAP protocol for Novell, see your NetWare documentation.

**Note** The 16-bit Windows client applications that use the **ncacn_spx** or **ncadg_ipx** transports require that the file NWIPXSPX.DLL be installed in order to run under the Windows NT Windows on Windows (WOW) subsystem. Contact Novell to obtain this file.

## Configuring the Security Server

Use the following to configure the security server for RPC:

1. Start Windows NT and choose the Control Panel icon.
2. In the Control Panel, choose the Networks icon.
   The Network Settings dialog box appears.
3. In the Installed Network Software list, select RPC Configuration.
   The RPC Configuration dialog box appears.
4. In the Security Service Provider list, select from one or more security providers.
5. Select OK.

## Starting and Stopping the RPC Locator

On the Windows NT platform, the RPC run-time libraries automatically start the RPC Locator when it is needed.

If you need to clear the database, for example, during debugging, you can manually stop and start the Locator.

### To stop and start the RPC Locator

1. From the Control Panel, select Services.

   The Services dialog box appears.

1. In the Service box, select Remote Procedure Call (RPC) Locator and choose Start or Stop.

You can also start and stop the Locator from the command line by typing:

```
C:\> net [start/stop] rpclocator
```

**Note** Only an administrator can start the RPC Locator once it is stopped. If stopped, it will be restarted as necessary by the RPC run-time libraries.

# RPC Data Types and Structures

This section defines the following constants, data types, and data structures used by the Microsoft RPC run-time functions:

| Data type/structure | Description |
|---|---|
| RPC_C_AUTHN_LEVEL* | Authentication-level constants |
| RPC_C_AUTHN* | Authentication-service constants |
| RPC_C_AUTHZ* | Authorization-service constants |
| GUID | Globally unique identifier (UUID) |
| PROTSEQ | Protocol sequence string |
| RPC_AUTH_IDENTITY_HANDLE | Authorization-identity handle |
| RPC_AUTH_KEY_RETRIEVAL_FN | Authorization-key retrieval function |
| RPC_AUTHZ_HANDLE | Authorization handle |
| RPC_BINDING_HANDLE | Binding handle |
| RPC_BINDING_VECTOR | Count and array of binding handles |
| RPC_IF_HANDLE | Interface handle |
| RPC_IF_ID | Interface identifier |
| RPC_IF_ID_VECTOR | Count and array of interface identifiers |
| RPC_MGR_EPV | Manager entry-point vector |
| RPC_NS_HANDLE | Name-service handle |
| RPC_OBJECT_INQ_FN | Object-inquiry function |
| RPC_POLICY | Set port allocation and NIC binding policies |
| RPC_PROTSEQ_VECTOR | Count and array of protocol sequences |
| RPC_STATS_VECTOR | Statistics vector |
| RPC_STATUS | Status |
| SEC_WINNT_AUTH_IDENTITY | Authentication |
| String binding | String representation of a binding |
| String UUID | Unique identifier string |
| UUID | Universally unique identifier |
| UUID_VECTOR | Count and array of unique identifiers |

# Authentication-Level Constants

The *AuthnLevel* argument represents the authentication level supplied to the **RpcBindingInqAuthInfo** and **RpcBindingSetAuthInfo** run-time functions.

The levels are listed in order of increasing authentication. Each new level adds to the authentication provided by the previous level. If the RPC run-time library does not support the specified level, it automatically upgrades to the next higher supported level.

The following constants represent valid values for the *AuthnLevel* argument:

| Constant | Description |
|---|---|
| RPC_C_AUTHN_LEVEL_DEFAULT | Uses the default authentication level for the specified authentication service. |
| RPC_C_AUTHN_LEVEL_NONE | Performs no authentication. |
| RPC_C_AUTHN_LEVEL_CONNECT | Authenticates only when the client establishes a relationship with a server. |
| RPC_C_AUTHN_LEVEL_CALL | Authenticates only at the beginning of each remote procedure call when the server receives the request. Does not apply to remote procedure calls made using the connection-based protocol sequences (those that start with the prefix "ncacn"). If the protocol sequence in a binding handle is a connection-based protocol sequence and you specify this level, this routine instead uses the RPC_C_AUTHN_LEVEL_PKT constant. |
| RPC_C_AUTHN_LEVEL_PKT | Authenticates that all data received is from the expected client. |
| RPC_C_AUTHN_LEVEL_PKT_INTEGRITY | Authenticates and verifies that none of the data transferred between client and server has been modified. |
| RPC_C_AUTHN_LEVEL_PKT_PRIVACY | Authenticates all previous levels and encrypts the argument value of each remote procedure call. |

**Note** RPC_C_AUTHN_LEVEL_CALL, RPC_C_AUTHN_LEVEL_PKT, RPC_C_AUTHN_LEVEL_PKT_INTEGRITY, and RPC_C_AUTHN_LEVEL_PKT_PRIVACY are only supported for clients communicating with a Windows NT server. A Windows 95 server can only accept incoming calls at the RPC_C_AUTHN_LEVEL_CONNECT level.

## See Also

RpcBindingInqAuthInfo, RpcBindingSetAuthInfo

# Authentication-Service Constants

The *AuthnSvc* argument represents the authentication service supplied to the **RpcBindingInqAuthInfo** and **RpcBindingSetAuthInfo** run-time functions.

The following constants represent valid values for the *AuthnSvc* argument:

| Constant | Value | Service |
|---|---|---|
| RPC_C_AUTHN_DCE_PRIVATE | 1 | DCE private key authentication |
| RPC_C_AUTHN_DCE_PUBLIC | 2 | DCE public key authentication (reserved for future use) |
| RPC_C_AUTHN_DEC_PUBLIC | 4 | DEC public key authentication (reserved for future use) |
| RPC_C_AUTHN_DEFAULT | 0xffffffff | Default authentication service |
| RPC_C_AUTHN_NONE | 0 | No authentication |
| RPC_C_AUTHN_WINNT | 10 | NTLMSSP (NT LAN Manager Security Support Provider) |

Specify RPC_C_AUTHN_NONE to turn off authentication for remote procedure calls made using the binding handle.

When you specify RPC_C_AUTHN_DEFAULT, the RPC run-time library uses the RPC_C_AUTHN_DCE_PRIVATE authentication service for remote procedure calls made using the binding handle.

## See Also

**RpcBindingInqAuthInfo**, **RpcBindingSetAuthInfo**

# Authorization-Service Constants

The *AuthzSvc* argument represents the authorization service supplied to the **RpcBindingInqAuthInfo** and **RpcBindingSetAuthInfo** run-time functions.

The following constants represent valid values for the *AuthzSvc* argument:

| Constant | Value | Service |
|---|---|---|
| RPC_C_AUTHZ_NONE | 0 | Server performs no authorization. |
| RPC_C_AUTHZ_NAME | 1 | Server performs authorization based on the client's principal name. |
| RPC_C_AUTHZ_DCE | 2 | Server performs authorization checking using the client's DCE privilege attribute certificate (PAC) information, which is sent to the server with each remote procedure call made using the binding handle. Generally, access is checked against DCE access control lists (ACLs). |

## See Also

RpcBindingInqAuthInfo, RpcBindingSetAuthInfo

# GUID Quick Info

```
typedef struct _GUID {
    unsigned long Data1;
    unsigned short Data2;
    unsigned short Data3;
    unsigned char Data4[8];
} GUID;

typedef GUID UUID;
```

*Data1*

Specifies the first eight hexadecimal digits of the UUID.

*Data2*

Specifies the first group of four hexadecimal digits of the UUID.

*Data3*

Specifies the second group of four hexadecimal digits of the UUID.

*Data4*

Specifies an array of eight elements that contains the third and final group of eight hexadecimal digits of the UUID in elements 0 and 1, and the final 12 hexadecimal digits of the UUID in elements 2 through 7.

## Remarks

GUIDs are globally unique identifiers and are a Microsoft implementation of the DCE UUID.

UUIDs uniquely identify objects, such as interfaces, manager entry-point vectors, and client objects. The RPC run-time libraries use UUIDs to check for compatibility between clients and servers and to select among multiple implementations of an interface.

## See Also

**UUID**, **UUID_VECTOR**

# PROTSEQ

**unsigned char *** *Protseq[1]*;

*Protseq*

Points to a character string identifying the network protocol used to communicate between client and server.

## Remarks

The protocol sequence is a character string that represents a valid combination of an RPC protocol (such as "ncacn"), a transport protocol (such as "tcp"), and a network protocol (such as "ip"). Microsoft RPC supports the following protocol sequences:

| Protocol sequence | Description | Supporting Platforms |
|---|---|---|
| ncacn_nb_tcp | Connection-oriented NetBIOS over TCP | client only: MS-DOS, Windows 3.*x*<br>client and server: Windows NT |
| ncacn_nb_ipx | Connection-oriented NetBIOS over IPX | client only: MS-DOS, Windows 3.*x*<br>client and server: Windows NT |
| ncacn_nb_nb | Connection-oriented NetBEUI | client only: MS-DOS, Windows 3.*x*<br>client and server: Windows NT, Windows 95 |
| ncacn_ip_tcp | Connection-oriented TCP/IP | client only: MS-DOS,Windows 3.*x,* and Apple Macintosh<br>client and server: Windows 95 and Windows NT |
| ncacn_np | Connection-oriented named pipes | client only: MS-DOS, Windows 3.*x*, Windows 95<br>client and server: Windows NT |
| ncacn_spx | Connection-oriented SPX | client only: MS-DOS, Windows 3.*x*<br>client and server: Windows NT, Windows 95 |
| ncacn_dnet_nsp | Connection-oriented DECnet transport | client only: MS-DOS, Windows 3.*x* |
| ncacn_at_dsp | Connection-oriented AppleTalk DSP | client: Apple Macintosh<br>server: Windows NT |
| ncacn_vns_spp | Connection-oriented Vines SPP transport | client only: MS-DOS, Windows 3.*x*<br>client and server: Windows NT |
| ncadg_ip_udp | Datagram (connectionless) UDP/IP | client only: MS-DOS, Windows 3.*x*<br>client and server: Windows NT |
| ncadg_ipx | Datagram (connectionless) IPX | client only: MS-DOS, Windows 3.*x*<br>client and server: Windows NT |
| ncalrpc | Local procedure call | client and server: Windows NT |

and Windows 95

A server application can use a particular protocol sequence only when the RPC run-time library and operating-system software support that protocol. A server chooses to accept remote procedure calls over some or all of the supported protocol sequences.

Several server routines allow server applications to register protocol sequences with the run-time library. Microsoft RPC functions that require a protocol-sequence argument use the data type **unsigned char**.

A client can use the protocol-sequence strings to construct a string binding using the **RpcStringBindingCompose** routine.

**Note** The **ncalrpc** protocol sequence is supported only for 32-bit Windows applications.

The **ncacn_dnet_nsp** protocol sequence is supported only for MS-DOS, and 16-bit Windows client applications. This release of Microsoft RPC does not include support for the **ncacn_dnet_nsp** protocol sequence with 32-bit client or server applications.

16-bit Windows client applications that use the **ncacn_spx** or **ncadg_ipx** protocol sequences require that the file NWIPXSPX.DLL be installed in order to run under the Windows NT Windows on Windows (WOW) subsystem. Contact Novell to obtain this file.

The **ncacn_vns_spp** protocol sequence requires that Banyan's *Enterprise Client For Windows NT* be installed. Contact Banyan for more information.

## See Also

**RpcServerUseAllProtseqs**, **RpcServerUseAllProtseqsIf**, **RpcServerUseProtseq**, **RpcServerUseProtseqEp**, **RpcServerUseProtseqIf**, **RpcStringBindingCompose**

# RPC_AUTH_IDENTITY_HANDLE

**typedef void * RPC_AUTH_IDENTITY_HANDLE;**

## Remarks

An identity handle points to the data structure that contains the client's authentication and authorization credentials specified for remote procedure calls.

## See Also

**RpcBindingInqAuthInfo**, **RpcBindingSetAuthInfo**

# RPC_AUTH_KEY_RETRIEVAL_FN

```
typedef void ( * RPC_AUTH_KEY_RETRIEVAL_FN)
    (    void *    Arg,
         unsigned char *    ServerPrincName,
         unsigned long    KeyVer,
         void * *    Key,
         RPC_STATUS *    Status
    );
```

*Arg*

Points to a user-defined argument to the user-supplied encryption key acquisition function. The RPC run-time library uses the *Arg* argument supplied to **RpcServerRegisterAuthInfo**.

*ServerPrincName*

Points to the principal name to use for the server when authenticating remote procedure calls. The RPC run-time library uses the *ServerPrincName* argument supplied to **RpcServerRegisterAuthInfo**.

*KeyVer*

Specifies the value that the RPC run-time library automatically provides for the key-version argument. When the value is 0, the acquisition function must return the most recent key available.

*Key*

Points to a pointer to the authentication key returned by the user-supplied function.

*Status*

Points to the status returned by the acquisition function when it is called by the RPC run-time library to authenticate the client RPC request. If the status is other than RPC_S_OK, the request fails and the run-time library returns the error status to the client application.

## Remarks

An authorization key retrieval function specifies the address of a server-application-provided routine returning encryption keys.

## See Also

**RpcServerRegisterAuthInfo**

# RPC_AUTHZ_HANDLE

**typedef void *  RPC_AUTHZ_HANDLE;**

## Remarks

An authorization handle points to the privileges information for the client application that made the remote procedure call.

## See Also

**RpcBindingInqAuthClient**

# RPC_BINDING_HANDLE

**typedef RPC_BINDING_HANDLE handle_t;**

## Remarks

A binding handle is a pointer-sized opaque variable containing information that the RPC run-time library uses to access binding information. The run-time library uses binding information to establish a client-server relationship that allows the execution of remote procedure calls.

Based on the context in which a binding handle is created, the binding handle is considered a server binding handle or a client binding handle.

A server binding handle contains the information necessary for a client to establish a relationship with a specific server. Any number of RPC API run-time routines return a server binding handle that can be used for making a remote procedure call.

A client binding handle cannot be used to make a remote procedure call. The RPC run-time library creates and provides a client binding handle to a called server procedure (also called a server manager routine) as the RPC_BINDING_HANDLE parameter. The client binding handle contains information about the calling client.

The **RpcBinding*** and **RpcNsBinding*** routines return the status code RPC_S_WRONG_KIND_OF_BINDING when an application provides the incorrect binding-handle type.

An application can share a single binding handle across multiple threads of execution. The RPC run-time library manages concurrent remote procedure calls that use a single binding handle. However, the application is responsible for binding-handle concurrency control for operations that modify a binding handle. These operations include the following routines:

- **RpcBindingFree**
- **RpcBindingReset**
- **RpcBindingSetAuthInfo**
- **RpcBindingSetObject**

For example, if an application shares a binding handle across two threads of execution and resets the binding-handle endpoint in one of the threads by calling **RpcBindingReset**, the binding handle in the other thread is also reset. Similarly, freeing the binding handle in one thread by calling **RpcBindingFree** frees the binding handle in the other thread.

If you don't want concurrency, you can design an application to create a copy of a binding handle by calling **RpcBindingCopy**. In this case, an operation to the first binding handle has no effect on the second binding handle.

Routines requiring a binding handle as an argument show a data type of RPC_BINDING_HANDLE. Binding-handle arguments are passed by value.

# RPC_BINDING_VECTOR

```
#define rpc_binding_vector_t RPC_BINDING_VECTOR
typedef struct _RPC_BINDING_VECTOR {
    unsigned long       Count;
    RPC_BINDING_HANDLE  BindingH[1];
}   RPC_BINDING_VECTOR;
```

*Count*

Specifies the number of binding handles present in the binding-handle array *BindingH*.

*BindingH*

Specifies an array of binding handles that contains *Count* elements.

## Remarks

The binding-vector data structure contains a list of binding handles over which a server application can receive remote procedure calls.

The binding vector contains a count member (*Count*), followed by an array of binding-handle (*BindingH*) elements.

The RPC run-time library creates binding handles when a server application registers protocol sequences. To obtain a binding vector, a server application calls the **RpcServerInqBindings** routine.

A client application obtains a binding vector of compatible servers from the name-service database by calling the **RpcNsBindingLookupNext** routine.

In both routines, the RPC run-time library allocates memory for the binding vector. An application calls the **RpcBindingVectorFree** routine to free the binding vector.

To remove an individual binding handle from the vector, the application must set the value in the vector to NULL. When setting a vector element to NULL, the application must:

- Free the individual binding
- Not change the value of *Count*

Calling the **RpcBindingFree** routine allows an application to both free the unwanted binding handle and set the vector entry to a NULL value.

## See Also

**RpcBindingVectorFree**, **RpcEpRegister**, **RpcEpRegisterNoReplace**, **RpcEpUnregister**, **RpcNsBindingExport**, **RpcNsBindingLookupNext**, **RpcNsBindingSelect**, **RpcServerInqBindings**

# RPC_CLIENT_INTERFACE

## Remarks

The **RPC_CLIENT_INTERFACE** data structure is part of the private interface between the run-time libraries and the stubs. Most distributed applications that use Microsoft RPC do not need this data structure.

The data structure is defined in the header file RPCDCEP.H.

# RPC_DISPATCH_TABLE

## Remarks

The **RPC_DISPATCH_TABLE** data structure is part of the private interface between the run-time libraries and the stubs. Most distributed applications that use Microsoft RPC do not need this data structure.

The data structure is defined in the header file RPCDCEP.H.

# RPC_IF_HANDLE

**typedef void * RPC_IF_HANDLE;**

## Remarks

An interface handle is an opaque variable containing information the RPC run-time library uses to access the interface-specification data structure.

The MIDL compiler automatically creates an interface-specification data structure from each IDL file and creates a global variable of type RPC_IF_HANDLE for the interface specification.

The MIDL compiler includes an interface handle in each .H file generated for the interface.

Routines requiring the interface specification as an argument show a data type of RPC_IF_HANDLE.

The form of each interface handle name is as follows:

- *if-name***_ClientIfHandle** (for the client)
- *if-name***_ServerIfHandle** (for the server)

*if-name*

Specifies the interface identifier in the IDL file.

For example:

```
hello_ClientIfHandle
hello_ServerIfHandle
```

**Note** The maximum length of the interface handle name is 31 characters.

Because the **_ClientIfHandle** and **_ServerIfHandle** parts of the names require 15 characters, the *if-name* element can be no more than 16 characters long.

# RPC_IF_ID

```
typedef struct _RPC_IF_ID {
    UUID  Uuid;
    unsigned short  VersMajor;
    unsigned short  VersMinor;
}   RPC_IF_ID;
```

*Uuid*

Specifies the interface UUID.

*VersMajor*

Specifies the major version number, an integer from 0 to 65535, inclusive.

*VersMinor*

Specifies the minor version number, an integer from 0 to 65535, inclusive.

## Remarks

The interface-identification (ID) data structure contains the interface UUID and major and minor version numbers of an interface. The interface identification is a subset of the data contained in the interface-specification structure.

Routines that require an interface ID structure show a data type of **RPC_IF_ID**. In those routines, the application is responsible for providing memory for the structure.

## See Also

**RpcIfInqId**

# RPC_IF_ID_VECTOR

```
typedef struct _RPC_IF_ID_VECTOR {
    unsigned long  Count;
    RPC_IF_ID *  IfHandl[1];
}   RPC_IF_ID_VECTOR;
```

*Count*

Specifies the number of interface-identification data structures present in the array *IfHandl*.

*IfHandl*

Specifies an array of pointers to interface-identification data structures that contains *Count* elements.

## Remarks

The interface-identification (ID) vector data structure contains a list of interfaces offered by a server. The interface ID vector contains a count member (*Count*), followed by an array of pointers to interface IDs (**RPC_IF_ID**).

The interface ID vector is a read-only vector. To obtain a vector of the interface IDs registered by a server with the run-time library, an application calls the **RpcMgmtInqIfIds** routine. To obtain a vector of the interface IDs exported by a server, an application calls the **RpcNsMgmtEntryInqIfIds** routine.

The RPC run-time library allocates memory for the interface ID vector. The application calls the **RpcIfIdVectorFree** routine to free the interface ID vector.

## See Also

**RpcIfIdVectorFree**, **RpcMgmtInqIfIds**, **RpcNsMgmtEntryInqIfIds**

# RPC_MGR_EPV

**typedef void RPC_MGR_EPV;**

**typedef _***if-name***_SERVER_EPV {**
    *return-type* **(*** *Functionname*) (*param-list*);**
    **...** *//* **one entry for each function in IDL file**
**}** *if-name***_SERVER_EPV;**

*if-name*

Specifies the name of the interface indicated in the IDL file.

*return-type*

Specifies the type returned by the function *Functionname* indicated in the IDL file.

*Functionname*

Specifies the name of the function indicated in the IDL file.

*param-list*

Specifies the arguments indicated for the function *Functionname* in the IDL file.

## Remarks

The manager entry-point vector (EPV) is an array of function pointers. The array contains pointers to the implementations of the functions specified in the IDL file. The number of elements in the array is set to the number of functions specified in the IDL file. An application can also have multiple EPVs, representing multiple implementations of the functions specified in the interface.

The MIDL compiler generates a default EPV data type named *if-name***_SERVER_EPV**, where *if-name* specifies the interface identifier in the IDL file. The MIDL compiler initializes this default EPV to contain function pointers for each of the procedures specified in the IDL file.

When the server offers multiple implementations of the same interface, the server application must declare and initialize one variable of type *if-name***_SERVER_EPV** for each implementation of the interface. Each EPV must contain one entry point (function pointer) for each procedure defined in the IDL file.

## See Also

**RpcServerRegisterIfEx**

# RPC_NS_HANDLE

**typedef void * RPC_NS_HANDLE;**

## Remarks

A name-service handle is an opaque variable containing information the RPC run-time library uses to return the following RPC data from the name-service database:

- Server binding handles
- UUIDs of resources offered by server profile members
- Group members

The scope of a name-service handle is from a **Begin** routine through the corresponding **Done** routine.

Applications are responsible for concurrency control of name-service handles across threads.

## See Also

**RpcNsBindingImportBegin**, **RpcNsBindingImportDone**, **RpcNsBindingImportNext**, **RpcNsBindingLookupBegin**, **RpcNsBindingLookupDone**, **RpcNsBindingLookupNext**

# RPC_OBJECT_INQ_FN

```
typedef void RPC_OBJECT_INQ_FN(
        UUID *    ObjectUuid,
        UUID *    TypeUuid,
        RPC_STATUS *    Status
    );
```

*ObjectUuid*

Points to the variable that specifies the object UUID that is to be mapped to a type UUID.

*TypeUuid*

Points to the address of the variable that is to contain the type UUID derived from the object UUID. The type UUID is returned by the function.

*Status*

Points to a return value for the function.

## Remarks

The developer can replace the default mapping function that maps object UUIDs to type UUIDs by calling **RpcObjectSetInqFn** and supplying a pointer to a function of type RPC_OBJECT_INQ_FN. The supplied function must match the function prototype specified by the type definition: a function with three parameters and the function return value of void.

## See Also

**RpcObjectSetInqFn**

# RPC_POLICY

```
typedef struct _RPC_POLICY {
    unsigned int Length;
    unsigned long EndpointFlags;
    unsigned long NICFlags;
}   RPC_POLICY, __RPC_FAR  *  PRPC_POLICY;
```

*Length*

Specifies the size, in bytes, of the RPC_POLICY structure.

*EndpointFlags*

Specifies Internet or intranet (local network) ports for endpoint assignments.

*NICFlags*

Controls binding to network interface cards (NICs).

## Remarks

The RPC_Policy structure contains flags that allow you to restrict port allocation for dynamic ports and that allow multihomed machines to selectively bind to Network Interface Cards (NICs). The values in this structure are ignored if the client and server are using a protocol other than tcp/ip (protocol sequence **ncacn_ip_tcp**). See Configuring the Windows NT Registry for Port Allocations and Selective Binding for a description of the registry settings that are affected by the RPC_POLICY flags.

The *Length* member allows compatibility with future versions of this structure, which may contain additional fields. Set Length = sizeof(RPC_POLICY) when initializing the RPC_POLICY structure in your code.

The *EndpointFlags* member controls endpoint assignments. Allowable values for this field are:

| | |
|---|---|
| 0 | (Specifies the system default) |
| 1 | RPC_C_USE_INTERNET_PORT |
| 2 | RPC_C_USE_INTRANET_PORT |

**Note** If the registry does not contain any of the keys that specify the default policy for port allocation, then the *EndpointFlags* member will have no effect at run time. If a key is missing or contains an invalid value, then the entire configuration is marked as invalid and all RpcServerUseProtseq(ncacn_ip_tcp) calls will fail.

The *NICFlags* member tells RPC to either bind to NICs based on the settings in the registry, or to override the registry settings and bind to all NICs. Allowable values for this field are:

| | |
|---|---|
| 0 | Bind to NICs on the basis of the registry settings |
| 1 | RPC_C_BIND_TO_ALL_NICS |

**Note** If the Bind key is missing from the registry, then the *NICFlags* member will have no effect at run time. If the key contains an invalid value, then the entire configuration is marked as invalid and all RpcServerUseProtseq*(ncacn_ip_tcp) calls will fail.

## See Also

Configuring the Windows NT Registry for Port Allocations and Selective Binding, **RpcServerUseAllProtseqsEx**, **RpcServerUseAllProtseqsIfEx**, **RpcServerUseProtseqEx**, **RpcServerUseProtseqEpEx**, **RpcServerUseProtseqIfEx**

# RPC_PROTSEQ_VECTOR

```
typedef struct _RPC_PROTSEQ_VECTOR {
    unsigned long  Count;
    unsigned char *  Protseq[1];
}   RPC_PROTSEQ_VECTOR;
```

*Count*

Specifies the number of protocol-sequence strings present in the array *Protseq*.

*Protseq*

Specifies an array of pointers to protocol-sequence strings. The number of pointers present is specified by the *Count* field.

## Remarks

The protocol-sequence vector data structure contains a list of protocol sequences the RPC run-time library uses to send and receive remote procedure calls. The protocol-sequence vector contains a count member (*Count*), followed by an array of pointers to protocol-sequence strings (*Protseq*).

The protocol-sequence vector is a read-only vector. To obtain a protocol-sequence vector, a server application calls the **RpcNetworkInqProtseqs** routine. The RPC run-time library allocates memory for the protocol-sequence vector. The server application calls the **RpcProtseqVectorFree** routine to free the protocol-sequence vector.

## See Also

**RpcNetworkInqProtseqs**, **RpcProtseqVectorFree**

# RPC_STATS_VECTOR

```
typedef struct {
    unsigned int   Count;
    unsigned long  Stats[1];
}   RPC_STATS_VECTOR;
```

*Count*

Specifies the number of statistics values present in the array *Stats.*

*Stats*

Specifies an array of unsigned long integers representing server statistics that contains *Count* elements.

## Remarks

The statistics vector contains statistics from the RPC run-time library on a per-server basis. The statistics vector contains a count member (*Count*), followed by an array of statistics. Each array element contains an unsigned long value. The following list describes the statistics indexed by the specified constant:

| Constant | Statistics |
|---|---|
| RPC_C_STATS_CALLS_IN | The number of remote procedure calls received by the server |
| RPC_C_STATS_CALLS_OUT | The number of remote procedure calls initiated by the server |
| RPC_C_STATS_PKTS_IN | The number of network packets received by the server |
| RPC_C_STATS_PKTS_OUT | The number of network packets sent by the server |

To obtain run-time statistics, an application calls the **RpcMgmtInqStats** routine. The RPC run-time library allocates memory for the statistics vector. The application calls the **RpcMgmtStatsVectorFree** routine to free the statistics vector.

## See Also

**RpcMgmtInqStats**, **RpcMgmtStatsVectorFree**

# RPC_STATUS

**typedef long RPC_STATUS;** // Win32 definition
**typedef unsigned short RPC_STATUS;** // MS-DOS, Win16 definition

## Remarks

The type **RPC_STATUS** represents a platform-specific status code type. The **RPC_STATUS** type is returned by most RPC functions and is part of the **RPC_OBJECT_INQ_FN** function type definition.

## See Also

RPC_OBJECT_INQ_FN

# SEC_WINNT_AUTH_IDENTITY

For Windows 3.*x* and MS-DOS:

```
typedef struct _SEC_WINNT_AUTH_IDENTITY{
  char __RPC_FAR * User;
  char __RPC_FAR * Domain;
  char __RPC_FAR * Password;
} SEC_WINNT_AUTH_IDENTITY;
```

For Windows NT, Windows 95, and Macintosh:

```
typedef struct _SEC_WINNT_AUTH_IDENTITY {
  unsigned short __RPC_FAR * User;
  unsigned long UserLength;
  unsigned short __RPC_FAR * Domain;
  unsigned long DomainLength;
  unsigned short __RPC_FAR * Password;
  unsigned long PasswordLength;
  unsigned long Flags;                      //value may be 0x1 or 0x2
} SEC_WINNT_AUTH_IDENTITY, * PSEC_WINNT_AUTH_IDENTITY;
```

*User*

String containing the user name.

*Domain*

String containing the domain name or the workgroup name.

*Password*

String containing the user's password in the domain or workgroup.

*Flags*

Long value specifying that the strings are ANSI (0x1) or Unicode (0x2).

## Remarks

The SEC_WINNT_AUTH_IDENTITY structure allows you to pass a particular user name and password to the runtime library for the purpose of authentication. Note that this structure must remain valid for the lifetime of the binding handle.

For Windows 95, Windows 3.*x* and MS-DOS, the strings are ANSI; for Windows NT, the strings may be ANSI or Unicode, depending on the value you assign to the *Flags* field. For Windows NT, Windows 95, and Macintosh, the values for *UserLength*; *DomainLength*, and *PasswordLength* are the length of the corresponding string without the terminating 0X0000.

# String Binding

*ObjectUUID*@*ProtocolSequence*:*NetworkAddress*[*Endpoint*,*Option*]

*ObjectUUID*

Specifies the UUID of the object operated on by the remote procedure call. At the server, the RPC run-time library maps the object type to a manager entry-point vector (an array of function pointers) to invoke the correct manager routine. For a discussion of how to map object UUIDs to manager entry-point vectors, see **RpcServerRegisterIfEx**.

*Protocol* Sequence

Specifies a character string that represents a valid combination of an RPC protocol (such as "ncacn"), a transport protocol (such as "tcp"), and a network protocol (such as "ip"). Microsoft RPC supports the following protocol sequences:

| Protocol sequence | Description | Supporting Platforms |
|---|---|---|
| **ncacn_nb_tcp** | Connection-oriented NetBIOS over TCP | client only: MS-DOS, Windows 3.*x*<br>client and server: Windows NT |
| **ncacn_nb_ipx** | Connection-oriented NetBIOS over IPX | client only: MS-DOS, Windows 3.*x*<br>client and server: Windows NT |
| **ncacn_nb_nb** | Connection-oriented NetBEUI | client only: MS-DOS, Windows 3.*x*<br>client and server: Windows NT, Windows 95 |
| **ncacn_ip_tcp** | Connection-oriented TCP/IP | client only: MS-DOS,Windows 3.*x,* and Apple Macintosh<br>client and server: Windows 95 and Windows NT |
| **ncacn_np** | Connection-oriented named pipes | client only: MS-DOS, Windows 3.*x*, Windows 95<br>client and server: Windows NT |
| **ncacn_spx** | Connection-oriented SPX | client only: MS-DOS, Windows 3.*x*<br>client and server: Windows NT, Windows 95 |
| **ncacn_dnet_nsp** | Connection-oriented DECnet transport | client only: MS-DOS, Windows 3.*x* |
| **ncacn_at_dsp** | AppleTalk DSP | client: Apple Macintosh<br>server: Windows NT |
| **ncacn_vns_spp** | Connection-oriented Vines SPP transport | client only: MS-DOS, Windows 3.*x*<br>client and server: Windows NT |
| **ncadg_ip_udp** | Datagram (connectionless) UDP/IP | client only: MS-DOS, Windows 3.*x*<br>client and server: Windows NT |
| **ncadg_ipx** | Datagram (connectionless) IPX | client only: MS-DOS, Windows 3.*x*<br>client and server: Windows NT |
| **ncalrpc** | Local procedure call | client and server: Windows NT |

and Windows 95

*NetworkAddress*

Specifies the network address of the system to receive remote procedure calls. The format and content of the network address depend on the specified protocol sequence as follows:

| Protocol sequence | Network address | Examples |
|---|---|---|
| **ncacn_nb_tcp** | Windows NT machine name | myserver |
| **ncacn_nb_ipx** | Windows NT machine name | myserver |
| **ncacn_nb_nb** | Windows NT or Windows 95 machine name | myserver |
| **ncacn_ip_tcp** | four-octet internet address, or host name | 128.10.2.30<br>anynode.microsoft.com |
| **ncacn_np** | Windows NT server name (leading double backslashes are optional) | myserver \\myotherserver |
| **ncacn_spx** | IPX internet address, or Windows NT server name | ~0000000108002B30612C<br>myserver |
| **ncacn_dnet_nsp** | Area and node syntax | 4.120 |
| **ncacn_at_dsp** | Windows NT machine name, optionally followed by @ and the AppleTalk zone name. Defaults to @*, the client's zone, if no zone provided | servername@zonename<br>servername |
| **ncacn_vns_spp** | StreetTalk server name of the form item@group@organization | printserver@sdkdocs@microsoft |
| **ncadg_ip_udp** | four-octet internet address, or host name | 128.10.2.30<br>anynode.microsoft.com |
| **ncadg_ipx** | IPX internet address, or Windows NT server name | ~0000000108002B30612C<br>myserver |
| **ncalrpc** | Machine name | thismachine |

The network-address field is optional. When you do not specify a network address, the string binding refers to your local host. It is possible to specify the name of the local machine when you use the **ncalrpc** protocol sequence, however doing so is completely unnecessary.

*Endpoint*

Specifies the endpoint, or address, of the process to receive remote procedure calls. An endpoint can be preceded by the keyword **endpoint=**. Specifying the endpoint is optional if the server has registered its bindings with the endpoint mapper. See RpcEpRegister.

The format and content of an endpoint depend on the specified protocol sequence as shown in the Endpoint/Option Table, below.

*Option*

Specifies protocol-specific options.. The option field is not required. Each option is specified by a {name, value} pair that uses the syntax *option name=option value*. Options are defined for each protocol sequence as shown in the Endpoint/Option Table, below.

| Protocol sequence | Endpoint | Examples | Option name |
|---|---|---|---|
| ncacn_nb_tcp | Integer between 0 and 255. Many values between 0 and 32 are reserved by Microsoft. | 100 | None |
| ncacn_nb_ipx | (as above) | (as above) | None |
| ncacn_nb_nb | (as above) | (as above) | None |
| ncacn_ip_tcp | Internet port number | 1025 | None |
| ncacn_np | Windows NT named pipe. Name must start with "\\pipe". | \\pipe\\pipename | Security (NT only) |
| ncacn_spx | Integer between 1 and 65535. | 5000 | None |
| ncacn_dnet_nsp | DECnet phase IV object number (must be preceded by the # character), or object name | mailserver #17 | None |
| ncacn_at_dsp | A character string, up to 22 bytes long | myservicesendpoint | None |
| ncacn_vns_spp | Vines SPP port number between 250 and 511 | 500 | None |
| ncadg_ip_udp | Internet port number | 1025 | Security (32-bit only) |
| ncadg_ipx | Integer between 1 and 65535. | 5000 | Security (32-bit only) |
| ncalrpc | String specifying application or service name. The string cannot include any backslash characters. | my_printer | Security (NT only) |

The Security option name, supported for the **ncalrpc**, **ncacn_np**, **ncadg_ip_udp**, and **ncadg_ipx** protocol sequences, takes the following option values:

| Option name | Option value |
|---|---|
| Security | {identification \| anonymous \| impersonation} {dynamic \| static} {true \| false} |

If the Security option name is specified, one entry from each of the sets of Security option values must also be supplied. The option values must be separated by a single-space character. For example, the following *Option* fields are valid:

```
Security=identification dynamic true
Security=impersonation static true
```

The Security option values have the following meanings:

| Security option value | Description |
|---|---|
| Anonymous | The client is anonymous to the server. |

| | |
|---|---|
| Dynamic | A pointer to the security token is maintained. Security settings represent current settings and include changes made after the endpoint was created. |
| False | Effective = FALSE; all Windows NT security settings, including those set to OFF, are included in the token. |
| Identification | The server has information about client but cannot impersonate. |
| Impersonation | The server is the client on the client's behalf. |
| Static | Security settings associated with the endpoint represent a copy of the security information at the time the endpoint was created. The settings do not change. |
| True | Effective = TRUE; only Windows NT security settings set to ON are included in the token. |

For more information about Microsoft Windows NT security options, see your Microsoft Windows NT programming documentation.

## Remarks

The string binding is an unsigned character string composed of strings that represent the binding object UUID, the RPC protocol sequence, the network address, and the endpoint and endpoint options. White space is not allowed in string bindings except where required by the *Option* syntax.

Default settings for the *NetworkAddress, Endpoint*, and *Option* fields vary according to the value of the *ProtocolSequence* field.

For all string-binding fields, a single backslash character (\) is interpreted as an escape character. To specify a single literal backslash character, you must supply two backslash characters (\\).

The following are examples of valid string bindings. In these examples, *obj-uuid* is used for convenience to represent a valid UUID in string form. Instead of showing the UUID 308FB580-1EB2-11CA-923B-08002B1075A7, the examples show *obj-uuid*.

```
obj-uuid@ncacn_ip_tcp:16.20.16.27[2001]
obj-uuid@ncacn_ip_tcp:16.20.16.27[endpoint=2001]
obj-uuid@ncacn_nb_nb:
obj-uuid@ncacn_nb_nb:[100]
obj-uuid@ncacn_np:
obj-uuid@ncacn_np:[\\pipe\\p3,Security=impersonation static true]
obj-uuid@ncacn_np:\\\\marketing[\\pipe\\p2\\p3\\p4]
obj-uuid@ncacn_np:\\\\marketing[endpoint=\\pipe\\p2\\p3\\p4]
obj-uuid@ncacn_np:\\\\sales
obj-uuid@ncacn_np:\\\\sales[\\pipe\\p1,Security=identification dynamic true]
obj-uuid@ncalrpc:
obj-uuid@ncalrpc:[object1_name_demonstrating_that_these_can_be_lengthy]
obj-uuid@ncalrpc:[object2_name,Security=anonymous static true]
obj-uuid@ncacn_vns_spp:server@group@org[500]
obj-uuid@ncacn_dnet_nsp:took[elf_server]
obj-uuid@ncacn_dnet_nsp:took[endpoint=elf_server]
obj-uuid@ncadg_ip_udp:128.10.2.30
obj-uuid@ncadg_ip_udp:maryos.microsoft.com[1025]
```

```
obj-uuid@ncadg_ipx: ~0000000108002B30612C[5000]
obj-uuid@ncadg_ipx:printserver
obj-uuid@ncacn_spx:annaw[4390]
obj-uuid@ncacn_spx:~0000000108002B30612C
```

A string binding contains the character representation of a binding handle and sometimes portions of a binding handle. String bindings are convenient for representing portions of a binding handle, but they can't be used for making remote procedure calls. They must first be converted to a binding handle by calling the **RpcBindingFromStringBinding** routine.

Additionally, a string binding does not contain all of the information from a binding handle. For example, the authentication information, if any, associated with a binding handle is not translated into the string binding returned by calling the **RpcBindingToStringBinding** routine.

During the development of a distributed application, servers can communicate their binding information to clients using string bindings to establish a client-server relationship without using the endpoint-map database or name-service database. To establish such a relationship, use the function **RpcBindingToStringBinding** to convert one or more binding handles from a binding-handle vector to a string binding, and provide the string binding to the client.

## See Also

**RpcBindingFromStringBinding**, **RpcBindingToStringBinding**, **RpcEpRegister**

# String UUID

A string UUID contains the character-array representation of a UUID. A string UUID is composed of multiple fields of hexadecimal characters. Each field has a fixed length, and fields are separated by the hyphen character. For example:

```
989C6E5C-2CC1-11CA-A044-08002B1BB4F5
```

When providing a string UUID as an input argument to a RPC run-time routine, enter the alphabetic hexadecimal characters as either uppercase or lowercase characters. However, when a run-time routine returns a string UUID, the hexadecimal characters are always lowercase.

## See Also

**UUID**

# UUID

```
typedef struct _GUID {
    unsigned long Data1;
    unsigned short Data2;
    unsigned short Data3;
    unsigned char Data4[8];
} GUID;

typedef GUID UUID;

#define uuid_t UUID
```

*Data1*

Specifies the first eight hexadecimal digits of the UUID.

*Data2*

Specifies the first group of four hexadecimal digits of the UUID.

*Data3*

Specifies the second group of four hexadecimal digits of the UUID.

*Data4*

Specifies an array of eight elements that contains the third and final group of four hexadecimal digits of the UUID in elements 0 and 1, and the final 12 hexadecimal digits of the UUID in elements 2 through 7.

## Remarks

UUIDs uniquely identify objects such as interfaces, manager entry-point vectors, and client objects. The RPC run-time libraries use UUIDs to check for compatibility between clients and servers and to select among multiple implementations of an interface.

## See Also

**GUID**, **UUID_VECTOR**

# UUID_VECTOR

```
typedef struct _UUID_VECTOR {
    unsigned long  Count;
    UUID *  Uuid[1];
}   UUID_VECTOR;
```

*Count*

Specifies the number of UUIDs present in the array *Uuid.*

*Uuid*

Specifies an array of pointers to UUIDs that contains *Count* elements.

## Remarks

The UUID vector data structure contains a list of UUIDs. The UUID vector contains a count member followed by an array of pointers to UUIDs.

An application constructs a UUID vector to contain object UUIDs to be exported or unexported from the name service.

## See Also

**RpcEpRegister**, **RpcEpRegisterNoReplace**, **RpcEpUnregister**, **RpcNsBindingExport**, **RpcNsBindingUnexport**

# Function Reference

This section contains an alphabetical list of the functions supported in this version of Microsoft RPC. The documentation for each function includes a statement about the function's purpose, the syntax, a description of the function's input parameters, a description of its values, and additional remarks that can help you use the function in an application.

All pointers passed to RPC functions must include the **_ _RPC_FAR** attribute. For example, the pointer **RPC_BINDING_HANDLE *** becomes **RPC_BINDING_HANDLE _ _RPC_FAR *** and **char * * Ptr** becomes **char _ _RPC_FAR * _ _RPC_FAR * Ptr**.

# DceErrorInqText Quick Info

The **DceErrorInqText** function returns the message text for a status code.

This function is supported by both 32-bit platforms – Windows NT and Windows 95. Note that it is supported in ANSI only on Windows 95.

```
#include <rpc.h>

RPC_STATUS RPC_ENTRY DceErrorInqText(

  unsigned long   StatusToConvert,
  unsigned char *   ErrorText
);
```

## Parameters

*StatusToConvert*

Specifies the status code to convert to a text string.

*ErrorText*

Returns the text corresponding to the error code.

| Value | Meaning |
|---|---|
| RPC_S_OK | Operation completed successfully |
| RPC_S_INVALID_ARG | Unknown error code |

## Remarks

The **DceErrorInqText** routine returns a null-terminated character string message for a particular status code. If the call is not successful, **DceErrorInqText** returns a message as well as a failure code in Status.

# MesBufferHandleReset Quick Info

The **MesBufferHandleReset** function re-initializes the handle for buffer serialization.

```
#include <rpc.h>

#include <midles.h>

RPC_STATUS RPC_ENTRY MesBufferHandleReset(

  handle_t   Handle,
  unsigned long   HandleStyle,
  MIDL_ES_CODE   OpCode,
  char   *   *   ppBuffer,
  unsigned long   BufferSize,
  unsigned long *   pEncodedSize
);
```

## Parameters

*Handle*

The handle to be initialized.

*HandleStyle*

Specifies the style of handle. Valid styles are **MES_FIXED_BUFFER_HANDLE** or **MES_DYNAMIC_BUFFER_HANDLE**.

*OpCode*

Specifies the operation. Valid operations are **MES_ENCODE** or **MES_DECODE**.

*ppBuffer*

For **MES_DECODE**, points to a pointer to the buffer containing the data to be decoded.

For **MES_ENCODE**, points to a pointer to the buffer for fixed buffer style, and points to a pointer to return the buffer address for dynamic buffer style.

*BufferSize*

Specifies the number of bytes of data to be decoded in the buffer. Note that this is used only for the fixed buffer style of serialization.

*pEncodedSize*

Points to the size of the completed encoding. Note that this is used only when the operation is **MES_ENCODE**.

## Remarks

The **MesBufferHandleReset** routine is used by applications to re-initialize a buffer style handle and save memory allocations.

## Return Values

| Value | Meaning |
|---|---|
| RPC_S_OK | Success |
| RPC_S_INVALID_ARG | Invalid argument |

## See Also

MesEncodeFixedBufferHandleCreate, MesEncodeDynBufferHandleCreate

# MesDecodeBufferHandleCreate Quick Info

The **MesDecodeBufferHandleCreate** function creates a decoding handle and initializes it for a (fixed) buffer style of serialization.

```
#include <rpc.h>

#include <midles.h>

RPC_STATUS RPC_ENTRY MesDecodeBufferHandleCreate(

  char   *   Buffer,
  unsigned long   BufferSize,
  handle_t *   pHandle
);
```

## Parameters

*Buffer*

Points to the buffer containing the data to decode.

*BufferSize*

Specifies the number of bytes of data to decode in the buffer.

*pHandle*

Points to the address to which the handle will be written.

## Remarks

The **MesDecodeBufferHandleCreate** routine is used by applications to create a serialization handle and initialize the handle for the (fixed) buffer style of decoding. When using the fixed buffer style of decoding, the user supplies a single buffer containing all the encoded data. This buffer must have an address which is aligned at 8, and must be a multiple of 8 bytes in size. Further, it must be large enough to hold all of the data to decode.

## Return Values

| Value | Meaning |
|---|---|
| RPC_S_OK | Success |
| RPC_S_INVALID_ARG | Invalid argument |
| RPC_S_OUT_OF_MEMORY | Out of memory |
| RPC_X_INVALID_BUFFER | Invalid buffer |

## See Also

MesEncodeFixedBufferHandleCreate, MesHandleFree

# MesDecodeIncrementalHandleCreate

Quick Info

The **MesDecodeIncrementalHandleCreate** function creates a decoding handle for the incremental style of serialization.

```
#include <rpc.h>

#include <midles.h>

RPC_STATUS RPC_ENTRY MesDecodeIncrementalHandleCreate(

  void *   UserState,
  MIDL_ES_READ   ReadFn,
  handle_t   *   pHandle
);
```

## Parameters

*UserState*

Points to the user-supplied state object that coordinates the **Alloc**, **Write**, and **Read** routines.

*ReadFn*

Points to the **Read** routine.

*pHandle*

Pointer to the newly-created handle.

## Remarks

The **MesDecodeIncrementalHandleCreate** routine is used by applications to create the handle and initialize it for the incremental style of decoding. When using the incremental style of decoding, the user supplies a **Read** routine to provide a buffer containing the next part of the data to be decoded. The buffer must be aligned at eight, and the size of the buffer must be a multiple of eight. For additional information on the user-supplied **Alloc**, **Write** and **Read** routines, see Using Encoding Services.

## Return Values

| Value | Meaning |
|---|---|
| RPC_S_OK | Success |
| RPC_S_INVALID_ARG | Invalid argument |
| RPC_S_OUT_OF_MEMORY | Out of memory |

## See Also

**MesIncrementalHandleReset**, **MesHandleFree**

# MesEncodeDynBufferHandleCreate

Quick Info

The **MesEncodeDynBufferHandleCreate** function creates an encoding handle and then initializes it for a dynamic buffer style of serialization.

```
#include <rpc.h>

#include <midles.h>

RPC_STATUS RPC_ENTRY MesEncodeDynBufferHandleCreate(

  char   *   *   ppBuffer,
  unsigned long   *   pEncodedSize,
  handle_t *   pHandle
);
```

## Parameters

*ppBuffer*

Points to a pointer to the stub-supplied buffer containing the encoding after serialization is complete.

*pEncodedSize*

Specifies a pointer to the size of the completed encoding. The size will be written to the pointee by the subsequent encoding operation(s).

*pHandle*

Points to the address to which the handle will be written.

## Remarks

The **MesEncodeDynBufferHandleCreate** routine is used by applications to allocate the memory and initialize the handle for the dynamic buffer style of encoding. When using the dynamic buffer style of encoding, the buffer into which all the encoded data will be placed is supplied by the stub. This buffer will be allocated by the current client memory-management mechanism.

There can be performance implications when using this style for multiple encodings with the same handle. A single buffer is returned from an encoding and data is copied from intermediate buffers. The buffers are released when necessary.

## Return Values

| Value | Meaning |
|---|---|
| RPC_S_OK | Success |
| RPC_S_INVALID_ARG | Invalid argument |
| RPC_S_OUT_OF_MEMORY | Out of memory |

## See Also

MesBufferHandleReset, MesHandleFree

# MesEncodeFixedBufferHandleCreate

Quick Info

The **MesEncodeFixedBufferHandleCreate** function creates an encoding handle and then initializes it for a fixed buffer style of serialization.

```
#include <rpc.h>

#include <midles.h>

RPC_STATUS RPC_ENTRY MesEncodeFixedBufferHandleCreate(

  char *    Buffer,
  unsigned long    BufferSize,
  unsigned long *    pEncodedSize,
  handle_t *    pHandle
);
```

## Parameters

*Buffer*

Points to the user-supplied buffer.

*BufferSize*

Specifies the size of the user-supplied buffer.

*pEncodedSize*

Specifies a pointer to the size of the completed encoding. The size will be written to the pointee by the subsequent encoding operation(s).

*pHandle*

Points to the newly-created handle.

## Remarks

The **MesEncodeFixedBufferHandleCreate** routine is used by applications to create and initialize the handle for the fixed buffer style of encoding. When using the fixed buffer style of encoding, the user supplies a single buffer into which all the encoded data is placed. This buffer must have an address which is aligned at eight, and must be a multiple of eight bytes in size. Further, it must be large enough to hold an encoding of all the data, along with an encoding header for each routine being encoded.

When the handle is used for multiple encoding operations, the encoded size is cumulative.

## Return Values

| Value | Meaning |
|---|---|
| RPC_S_OK | Success |
| RPC_S_INVALID_ARG | Invalid argument |
| RPC_S_OUT_OF_MEMORY | Out of memory |

## See Also

MesDecodeBufferHandleCreate, MesHandleFree

# MesEncodeIncrementalHandleCreate

Quick Info

The **MesEncodeIncrementalHandleCreate** function creates an encoding and then initializes it for the incremental style of serialization.

```
#include <rpc.h>

#include <midles.h>

RPC_STATUS RPC_ENTRY MesEncodeIncrementalHandleCreate(

  void *   UserState,
  MIDL_ES_ALLOC   AllocFn,
  MIDL_ES_WRITE   WriteFn,
  handle_t   *   pHandle
);
```

## Parameters

*UserState*

Points to the user-supplied state object that coordinates the **Alloc**, **Write**, and **Read** routines.

*AllocFn*

Points to the **Alloc** routine.

*WriteFn*

Points to the **Write** routine.

*pHandle*

Points to the newly-created handle.

## Remarks

The **MesEncodeIncrementalHandleCreate** routine is used by applications to create and initialize the handle for the incremental style of encoding or decoding. When using the incremental style of encoding, the user supplies an Alloc routine to provide an empty buffer into which the encoded data is placed, and a Write routine to call when the buffer is full or the encoding is complete. For additional information on the user-supplied Alloc, Write and Read routines, see Using Encoding Services.

## Return Values

| Value | Meaning |
|---|---|
| RPC_S_OK | Success |
| RPC_S_INVALID_ARG | Invalid argument |
| RPC_S_OUT_OF_MEMORY | Out of memory |

## See Also

**MesIncrementalHandleReset**, **MesHandleFree**

# MesHandleFree Quick Info

The **MesHandleFree** function frees the memory allocated by the serialization handle.

```
#include <rpc.h>

#include <midles.h>

RPC_STATUS RPC_ENTRY MesHandleFree(

  handle_t  Handle
);
```

## Parameters

*Handle*

The handle to be freed.

## Remarks

The **MesHandleFree** routine is used by applications to free the resources of the handle after encoding or decoding data.

## Return Values

| Value | Meaning |
|---|---|
| RPC_S_OK | Success |

## See Also

**MesEncodeFixedBufferHandleCreate**, **MesDecodeBufferHandleCreate**, **MesEncodeDynBufferHandleCreate**, **MesEncodeIncrementalHandleCreate**

# MesIncrementalHandleReset Quick Info

The **MesIncrementalHandleReset** function re-initializes the handle for incremental serialization.

```
#include <rpc.h>

#include <midles.h>

RPC_STATUS RPC_ENTRY MesIncrementalHandleReset(

  handle_t   Handle,
  void   *   UserState,
  MIDL_ES_ALLOC   AllocFn,
  MIDL_ES_WRITE   WriteFn,
  MIDL_ES_READ   ReadFn,
  MIDL_ES_CODE   OpCode
);
```

## Parameters

*Handle*

The handle to be re-initialized.

*UserState*

Depending on the function, points to the user-supplied block that coordinates successive calls to the **Alloc**, **Write**, and **Read** routines.

*AllocFn*

Points to the **Alloc** routine. This argument can be NULL if the operation does not require it, or if the handle was previously initiated with the pointer.

*WriteFn*

Points to the **Write** routine. This argument can be NULL if the operation does not require it, or if the handle was previously initiated with the pointer.

*ReadFn*

Points to the **Read** routine. This argument can be NULL if the operation does not require it, or if the handle was previously initiated with the pointer.

*OpCode*

Specifies the operation. Valid operations are **MES_ENCODE** or **MES_DECODE**.

## Remarks

The **MesIncrementalHandleReset** routine is used by applications to re-initialize the handle for the incremental style of encoding or decoding. For additional information on the user-supplied **Alloc**, **Write** and **Read** routines, see Using Encoding Services.

## Return Values

| Value | Meaning |
|---|---|
| RPC_S_OK | Success |
| RPC_S_INVALID_ARG | Invalid argument |
| RPC_S_OUT_OF_MEMORY | Out of memory |

## See Also

MesEncodeIncrementalHandleCreate, MesHandleFree

# MesInqProcEncodingId Quick Info

The **MesInqProcEncodingId** function provides the identity of an encoding.

```
#include <rpc.h>

#include <midles.h>

RPC_STATUS RPC_ENTRY MesInqProcEncodingId(

  handle_t   Handle,
  PRPC_SYNTAX_IDENTIFIER   pInterfaceId,
  unsigned long   *   pProcNum
);
```

## Parameters

*Handle*

Specifies an encoding or decoding handle.

*pInterfaceId*

Points to the address in which the identity of the interface used to encode the data will be written. *pInterfaceId* consists of the interface UUID and the version number.

*pProcNum*

Specifies the number of the routine used to encode the data.

## Remarks

The **MesInqProcEncodingId** routine is used by applications to obtain the identity of the routine used to encode the data before calling a routine to decode it. When called with an encoding handle, it returns the identity of the last encoding operation. When called with a decoding handle, it returns the identity of the next decoding operation by pre-reading the buffer.

This routine can only be used to check the identity of a procedure encoding; it cannot be used to check the identity for a type encoding.

## Return Values

| Value | Meaning |
|---|---|
| RPC_S_OK | Success |
| RPC_S_INVALID_ARG | Invalid argument |
| RPC_S_OUT_OF_MEMORY | Out of memory |
| RPC_S_UNKNOWN_IF | Unknown interface |
| RPC_S_UNSUPPORTED_TRANS_SYN | Transfer syntax not supported by server |
| RPC_X_INVALID_ES_ACTION | Invalid operation for a given handle |
| RPC_X_WRONG_ES_VERSION | Incompatible version of the serializing package |
| RPC_X_SS_INVALID_BUFFER | Invalid buffer |

# RpcAbnormalTermination Quick Info

The **RpcAbnormalTermination** function determines whether termination statements are being executed due to an exception or not.

```
#include <rpc.h>

void RpcAbnormalTermination(VOID);
```

## Remarks

The **RpcAbnormalTermination** function should only be called from within the termination-statements section of an **RpcFinally** termination handler.

## Return Values

| Value | Meaning | Description |
|---|---|---|
| Zero | No exception | Termination statements are not being executed due to an exception |
| Nonzero | Exception | Termination statements are being executed due to an exception |

## See Also

RpcFinally

# RpcBindingCopy Quick Info

The **RpcBindingCopy** function copies binding information and creates a new binding handle.

```
#include <rpc.h>

RPC_STATUS RPC_ENTRY RpcBindingCopy(

  RPC_BINDING_HANDLE   SourceBinding,
  RPC_BINDING_HANDLE *   DestinationBinding
);
```

## Parameters

*SourceBinding*

Specifies the server binding handle whose referenced binding information is copied.

*DestinationBinding*

Returns a pointer to the server binding handle that refers to the copied binding information.

## Remarks

**Note** Microsoft RPC supports **RpcBindingCopy** only in client applications, not in server applications.

The **RpcBindingCopy** routine copies the server-binding information referenced by the *SourceBinding* argument. **RpcBindingCopy** uses the *DestinationBinding* argument to return a new server binding handle for the copied binding information. **RpcBindingCopy** also copies the authentication information from the *SourceBinding* argument to the *DestinationBinding* argument.

An application uses **RpcBindingCopy** when it wants to keep a change made to binding information by one thread from affecting the binding information used by other threads.

Once an application calls **RpcBindingCopy**, operations performed on the *SourceBinding* binding handle do not affect the binding information referenced by the *DestinationBinding* binding handle. Similarly, operations performed on the *DestinationBinding* binding handle do not affect the binding information referenced by the *SourceBinding* binding handle.

If an application wants one thread's changes to binding information to affect the binding information used by other threads, the application should share a single binding handle across the threads. In this case, the application is responsible for binding-handle concurrency control.

When an application is finished using the binding handle specified by the *DestinationBinding* argument, the application should call the **RpcBindingFree** routine to release the memory used by the *DestinationBinding* binding handle and its referenced binding information.

## Return Values

| Value | Meaning |
|---|---|
| RPC_S_OK | Success |
| RPC_S_INVALID_BINDING | Invalid binding handle |
| RPC_S_WRONG_KIND_OF_BINDING | Wrong kind of binding for operation |

## See Also

RpcBindingFree

# RpcBindingFree Quick Info

The **RpcBindingFree** function releases binding-handle resources.

```
#include <rpc.h>

RPC_STATUS RPC_ENTRY RpcBindingFree(

  RPC_BINDING_HANDLE *   Binding
);
```

## Parameters

*Binding*

Points to the server binding to free.

## Remarks

**Note** Microsoft RPC supports **RpcBindingFree** only in client applications, not in server applications.

The **RpcBindingFree** routine releases memory used by a server binding handle. Referenced binding information that was dynamically created during program execution is released as well. An application calls the **RpcBindingFree** routine when it is finished using the binding handle.

Binding handles are dynamically created by calling the following routines:

- **RpcBindingCopy**
- **RpcBindingFromStringBinding**
- **RpcServerInqBindings**
- **RpcNsBindingImportNext**
- **RpcNsBindingSelect**

If the operation successfully frees the binding, the *Binding* argument returns a value of NULL.

## Return Values

| Value | Meaning |
|---|---|
| RPC_S_OK | Success |
| RPC_S_INVALID_BINDING | Invalid binding handle |
| RPC_S_WRONG_KIND_OF_BINDING | Wrong kind of binding for operation |

## See Also

RpcBindingCopy, RpcBindingFromStringBinding, RpcBindingVectorFree, RpcNsBindingImportNext, RpcNsBindingLookupNext, RpcNsBindingSelect, RpcServerInqBindings

# RpcBindingFromStringBinding Quick Info

The **RpcBindingFromStringBinding** function returns a binding handle from a string representation of a binding handle.

```
#include <rpc.h>

RPC_STATUS RPC_ENTRY RpcBindingFromStringBinding(

  unsigned char *    StringBinding,
  RPC_BINDING_HANDLE *    Binding
);
```

## Parameters

*StringBinding*

Points to a string representation of a binding handle.

*Binding*

Returns a pointer to the server binding handle.

## Remarks

The **RpcBindingFromStringBinding** routine creates a server binding handle from a string representation of a binding handle.

The *StringBinding* argument does not have to contain an object UUID. In this case, the returned binding contains a nil UUID.

If the provided *StringBinding* argument does not contain an endpoint field, the returned *Binding* argument is a partially bound binding handle.

If the provided *StringBinding* argument contains an endpoint field, the endpoint is considered to be a well-known endpoint.

If the provided *StringBinding* argument does not contain a host address field, the returned *Binding* argument references the local host.

An application creates a string binding by calling the **RpcStringBindingCompose** routine or by providing a character-string constant.

When an application is finished using the *Binding* argument, the application should call the **RpcBindingFree** routine to release the memory used by the binding handle.

## Return Values

| Value | Meaning |
|---|---|
| RPC_S_OK | Success |
| RPC_S_INVALID_STRING_BINDING | Invalid string binding |
| RPC_S_PROTSEQ_NOT_SUPPORTED | Protocol sequence not supported on this host |
| RPC_S_INVALID_RPC_PROTSEQ | Invalid protocol sequence |
| RPC_S_INVALID_ENDPOINT_FORMAT | Invalid endpoint format |

| | |
|---|---|
| RPC_S_STRING_TOO_LONG | String too long |
| RPC_S_INVALID_NET_ADDR | Invalid network address |
| RPC_S_INVALID_ARG | Invalid argument |
| RPC_S_INVALID_NAF_ID | Invalid network-address-family ID |

## See Also

RpcBindingCopy, RpcBindingFree, RpcBindingToStringBinding, RpcStringBindingCompose

# RpcBindingInqAuthClient Quick Info

This function is supported by both 32-bit platforms – Windows NT and Windows 95.

```
#include <rpc.h>

RPC_STATUS RPC_ENTRY RpcBindingInqAuthClient(

  RPC_BINDING_HANDLE   ClientBinding,
  RPC_AUTHZ_HANDLE *   Privs,
  unsigned char * *    ServerPrincName,
  unsigned long *      AuthnLevel,
  unsigned long *      AuthnSvc,
  unsigned long *      AuthzSvc
);
```

## Parameters

*ClientBinding*

Specifies the client binding handle of the client that made the remote procedure call. This value can be zero (see Remarks).

*Privs*

Returns a pointer to a handle to the privileged information for the client application that made the remote procedure call on the *ClientBinding* binding handle.

The server application must cast the *ClientBinding* binding handle to the data type specified by the *AuthzSvc* argument. The data referenced by this argument is read-only and should not be modified by the server application. If the server wants to preserve any of the returned data, the server must copy the data into server-allocated memory. This parameter is not used by the RPC_C_AUTHN_WINNT authentication service. The returned pointer will always be NULL.

*ServerPrincName*

Returns a pointer to a pointer to the server principal name specified by the client application that made the remote procedure call on the *ClientBinding* binding handle. The content of the returned name and its syntax are defined by the authentication service in use.

Specify a null value to prevent **RpcBindingInqAuthClient** from returning the *ServerPrincName* argument. In this case, the application does not call the **RpcStringFree** routine.

*AuthnLevel*

Returns a pointer to the level of authentication requested by the client application that made the remote procedure call on the *ClientBinding* binding handle.

Specify a null value to prevent **RpcBindingInqAuthClient** from returning the *AuthnLevel* argument.

*AuthnSvc*

Returns a pointer to the authentication service requested by the client application that made the remote procedure call on the *ClientBinding* binding handle. For a list of the RPC-supported authentication levels, see Authentication-Level Constants.

Specify a null value to prevent **RpcBindingInqAuthClient** from returning the *AuthnSvc* argument. This parameter is not used by the RPC_C_AUTHN_WINNT authentication service. The returned value will always be RPC_S_AUTHZ_NONE.

*AuthzSvc*

Returns a pointer to the authorization service requested by the client application that made the

remote procedure call on the *Binding* binding handle. For a list of possible returns, see **RpcMgmtInqDefaultProtectLevel**.

Specify a null value to prevent **RpcBindingInqAuthClient** from returning the *AuthnSvc* argument.

## Remarks

A server application calls the **RpcBindingInqAuthClient** routine to obtain the principal name or privilege attributes of the authenticated client that made the remote procedure call. In addition, **RpcBindingInqAuthClient** returns the authentication service, authentication level, and server principal name specified by the client. The server can use the returned data for authorization purposes.

The RPC run-time library allocates memory for the returned *ServerPrincName* argument. The application is responsible for calling the **RpcStringFree** routine for the returned argument string.

For clients using the MIDL **auto_handle** or **implicit_handle** attribute, the server application should use zero as the value for the *ClientBinding* parameter. Using zero retrieves the authentication and authorization information from the currently executing remote procedure call.

## Return Values

| value | Meaning |
|---|---|
| RPC_S_OK | Success |
| RPC_S_INVALID_BINDING | Invalid binding handle |
| RPC_S_WRONG_KIND_OF_BINDING | Wrong kind of binding for operation |
| RPC_S_BINDING_HAS_NO_AUTH | Binding has no authentication information |

## See Also

**RpcBindingSetAuthInfo**, **RpcStringFree**

# RpcBindingInqAuthInfo Quick Info

The **RpcBindingInqAuthInfo** function returns authentication and authorization information from a binding handle.

```
#include <rpc.h>

RPC_STATUS RPC_ENTRY RpcBindingInqAuthInfo(

  RPC_BINDING_HANDLE   Binding,
  unsigned char * *    ServerPrincName,
  unsigned long *      AuthnLevel,
  unsigned long *      AuthnSvc,
  RPC_AUTH_IDENTITY_HANDLE *   AuthIdentity,
  unsigned long *      AuthzSvc
);
```

## Parameters

*Binding*

Specifies the server binding handle from which authentication and authorization information is returned.

*ServerPrincName*

Returns a pointer to a pointer to the expected principal name of the server referenced in the *Binding* argument. The content of the returned name and its syntax are defined by the authentication service in use.

Specify a null value to prevent **RpcBindingInqAuthInfo** from returning the *ServerPrincName* argument. In this case, the application does not call the **RpcStringFree** routine.

*AuthnLevel*

Returns a pointer to the level of authentication used for remote procedure calls made using the *Binding* binding handle. For a list of the RPC-supported authentication levels, see Authentication-Level Constants. Specify a null value to prevent the routine from returning the *AuthnLevel* argument.

The level returned in the *AuthnLevel* argument may be different from the level specified when the client called the **RpcBindingSetAuthInfo** routine. This discrepancy happens when the authentication level specified by the client was not supported by the RPC run-time library and the run-time library automatically upgraded to the next higher level.

*AuthnSvc*

Returns a pointer to the authentication service specified for remote procedure calls made using the *Binding* binding handle. For a list of the RPC-supported authentication services, see Authentication-Service Constants.

Specify a null value to prevent **RpcBindingInqAuthInfo** from returning the *AuthnSvc* argument.

*AuthIdentity*

Returns a pointer to a handle to the data structure that contains the client's authentication and authorization credentials specified for remote procedure calls made using the *Binding* binding handle.

Specify a null value to prevent **RpcBindingInqAuthInfo** from returning the *AuthIdentity* argument.

*AuthzSvc*

Returns a pointer to the authorization service requested by the client application that made the remote procedure call on the *Binding* binding handle. For a list of the RPC-supported authentication

services, see Authentication-Service Constants.

Specify a null value to prevent **RpcBindingInqAuthInfo** from returning the *AuthzSvc* argument.

## Remarks

A client application calls the **RpcBindingInqAuthInfo** routine to view the authentication and authorization information associated with a server binding handle. The client specifies the data by calling the **RpcBindingSetAuthInfo** routine.

The RPC run-time library allocates memory for the returned *ServerPrincName* argument. The application is responsible for calling the **RpcStringFree** routine for that returned argument string.

When a client application does not know a server's principal name, calling **RpcBindingInqAuthInfo** after making a remote procedure call provides the server's principal name. For example, clients that import from a group or profile may not know a server's principal name when calling the **RpcBindingSetAuthInfo** routine.

## Return Values

| Value | Meaning |
|---|---|
| RPC_S_OK | Success |
| RPC_S_INVALID_BINDING | Invalid binding handle |
| RPC_S_WRONG_KIND_OF_BINDING | Wrong kind of binding for operation |
| RPC_BINDING_HAS_NO_AUTH | Binding has no authentication information |

## See Also

**RpcBindingSetAuthInfo**, **RpcStringFree**

# RpcBindingInqObject Quick Info

The **RpcBindingInqObject** function returns the object UUID from a binding handle.

```
#include <rpc.h>

RPC_STATUS RPC_ENTRY RpcBindingInqObject(

  RPC_BINDING_HANDLE   Binding,
  UUID *   ObjectUuid
);
```

## Parameters

*Binding*

Specifies a client or server binding handle.

*ObjectUuid*

Returns a pointer to the object UUID found in the *Binding* argument. *ObjectUuid* is a unique identifier of an object to which a remote procedure call can be made.

## Remarks

An application calls the **RpcBindingInqObject** routine to see the object UUID associated with a client or server binding handle.

## Return Values

| Value | Meaning |
|---|---|
| RPC_S_OK | Success |
| RPC_S_INVALID_BINDING | Invalid binding handle |

## See Also

RpcBindingSetObject

# RpcBindingReset Quick Info

The **RpcBindingReset** function resets a binding handle so that the host is specified but the server on that host is unspecified.

```
#include <rpc.h>

RPC_STATUS RPC_ENTRY RpcBindingReset(

  RPC_BINDING_HANDLE Binding
);
```

## Parameters

*Binding*

Specifies the server binding handle to reset.

## Remarks

A client calls the **RpcBindingReset** routine to disassociate a particular server instance from the server binding handle specified in the *Binding* argument. The **RpcBindingReset** routine dissociates a server instance by removing the endpoint portion of the server address in the binding handle. The host remains unchanged in the binding handle. The result is a partially bound server binding handle.

**RpcBindingReset** does not affect the *Binding* argument's authentication information, if there is any.

If a client is willing to be serviced by any compatible server instance on the host specified in the binding handle, the client calls the **RpcBindingReset** routine before making a remote procedure call using the *Binding* binding handle.

When the client makes the next remote procedure call using the reset (partially bound) binding, the client's RPC run-time library uses a well-known endpoint from the client's interface specification, if any. Otherwise, the client's run-time library automatically communicates with the endpoint-mapping service on the specified remote host to obtain the endpoint of a compatible server from the endpoint-map database. If a compatible server is located, the RPC run-time library updates the binding with a new endpoint. If a compatible server is not found, the remote procedure call fails. For calls using a connection protocol (ncacn), the RPC_S_NO_ENDPOINT_FOUND status code is returned to the client. For calls using a datagram protocol (ncadg), the RPC_S_COMM_FAILURE status code is returned to the client.

Server applications should register all binding handles by calling **RpcEpRegister** and **RpcEpRegisterNoReplace** if the server wants to be available to clients that make a remote procedure call on a reset binding handle.

## Return Values

| Value | Meaning |
|---|---|
| RPC_S_OK | Success |
| RPC_S_INVALID_BINDING | Invalid binding handle |
| RPC_S_WRONG_KIND_OF_BINDING | Wrong kind of binding for operation |

## See Also

RpcEpRegister, RpcEpRegisterNoReplace

# RpcBindingServerFromClient Quick Info

The **RpcBindingServerFromClient** function converts a client binding handle to a server binding handle.

This function is supported by both 32-bit platforms – Windows NT and Windows 95.

```
#include <rpc.h>

RPC_STATUS RPC_ENTRY RpcBindingServerFromClient(

  RPC_BINDING_HANDLE   ClientBinding,
  RPC_BINDING_HANDLE *   ServerBinding
);
```

## Parameters

*ClientBinding*

Specifies the client binding handle to convert to a server binding handle.

*ServerBinding*

Returns a server binding handle.

## Remarks

An application calls the **RpcBindingServerFromClient** routine to convert a client binding handle into a partially-bound server binding handle.

The **RpcBindingServerFromClient** routine is supported for the following protocol sequences:

- ncadg_ip_udp
- ncadg_ipx
- ncacn_ip_tcp
- ncacn_spx.

An application gets a client binding handle from the RPC runtime. When the RPC arrives at a server, the runtime creates a client binding handle that contains information about the calling client. This handle is passed by the runtime to the server manager routine as the first argument.

The following information pertains to the server binding handle that is returned by **RpcBindingServerFromClient**:

- The returned handle is a partially bound handle. It contains a network address for the calling client, but lacks an endpoint.
- The returned handle contains the same object UUID used by the calling client. This can be the nil UUID. For more information on how a client specifies an object UUID for a call, see **RpcBindingsetObject**, **RpcNsBindingImportBegin**, **RpcNsBindingLookupBegin**, and **RpcBindingFromStringBinding**.
- The returned handle contains no authentication information.

## Return Values

| Value | Meaning |
|---|---|
| RPC_S_OK | Success |

| | |
|---|---|
| RPC_S_INVALID_BINDING | Invalid binding handle |
| RPC_S_WRONG_KIND_OF_BINDING | Wrong kind of binding for operation |
| RPC_S_CANNOT_SUPPORT | Cannot determine the client's host (not TCP or SPX) |

## See Also

**RpcBindingFree**, **RpcBindingSetObject**, **RpcEpRegister**, **RpcEpRegisterNoReplace**, **RpcNsBindingImportBegin**, **RpcNsBindingLookupBegin**, **RpcBindingFromStringBinding**

# RpcBindingSetAuthInfo Quick Info

The **RpcBindingSetAuthInfo** function sets authentication and authorization information into a binding handle.

```
#include <rpc.h>

RPC_STATUS RPC_ENTRY RpcBindingSetAuthInfo(

  RPC_BINDING_HANDLE   Binding,
  unsigned char *   ServerPrincName,
  unsigned long   AuthnLevel,
  unsigned long   AuthnSvc,
  RPC_AUTH_IDENTITY_HANDLE   AuthIdentity,
  unsigned long   AuthzSvc
);
```

## Parameters

*Binding*

Specifies the server binding handle into which authentication and authorization information is set.

*ServerPrincName*

Points to the expected principal name of the server referenced by the binding handle specified in the *Binding* argument. The content of the name and its syntax are defined by the authentication service in use.

*AuthnLevel*

Specifies the level of authentication to be performed on remote procedure calls made using the *Binding* binding handle. For a list of the RPC-supported authentication levels, see Authentication-Level Constants.

*AuthnSvc*

Specifies the authentication service to use. For a list of the RPC-supported authentication services, see Authentication-Service Constants.

Specify RPC_C_AUTHN_NONE to turn off authentication for remote procedure calls made using the *Binding* binding handle.

If RPC_C_AUTHN_DEFAULT is specified, the RPC run-time library uses the RPC_C_AUTHN_WINNT authentication service for remote procedure calls made using the *Binding* binding handle.

*AuthIdentity*

Specifies a handle for the data structure that contains the client's authentication and authorization credentials appropriate for the selected authentication and authorization service.

When using the RPC_C_AUTHN.WINNT authentication service *AuthIdentity* should be a pointer to a SEC_WINNT_AUTH_IDENTITY structure (defined in rpcdce.h).

Specify a null value to use the security login context for the current address space.

*AuthzSvc*

Specifies the authorization service implemented by the server for the interface of interest. The validity and trustworthiness of authorization data, like any application data, depends on the authentication service and authentication level selected. This parameter is ignored when using the RPC_C_AUTHN_WINNT authentication service.

For a list of the RPC-supported authentication services, see Authentication-Service Constants.

## Remarks

A client application calls the **RpcBindingSetAuthInfo** routine to set up a server binding handle for making authenticated remote procedure calls.

Unless a client calls **RpcBindingSetAuthInfo**, all remote procedure calls on the *Binding* binding handle are unathenticated. A client is not required to call this routine.

**Note** As long as the binding handle exists, RPC maintains a pointer to *AuthIdentity*. Be sure it is not on the stack and is not freed until the binding handle is freed. If the binding handle is copied, or if a context handle is created from the binding handle, then the *AuthIdentity* pointer will also be copied.

## Return Values

| Value | Meaning |
|---|---|
| RPC_S_OK | Success |
| RPC_S_INVALID_BINDING | Invalid binding handle |
| RPC_S_WRONG_KIND_OF_BINDING | Wrong kind of binding for operation |
| RPC_S_UNKNOWN_AUTHN_SERVICE | Unknown authentication service |

## See Also

**RpcBindingInqAuthInfo**, **RpcServerRegisterAuthInfo**

# RpcBindingSetObject Quick Info

The **RpcBindingSetObject** function sets the object UUID value in a binding handle.

```
#include <rpc.h>

RPC_STATUS RPC_ENTRY RpcBindingSetObject(

  RPC_BINDING_HANDLE   Binding,
  UUID *   ObjectUuid
);
```

## Parameters

*Binding*

Specifies the server binding into which the *ObjectUuid* is set.

*ObjectUuid*

Points to the UUID of the object serviced by the server specified in the *Binding* argument. *ObjectUuid* is a unique identifier of an object to which a remote procedure call can be made.

## Remarks

An application calls the **RpcBindingSetObject** routine to associate an object UUID with a server binding handle. The set-object operation replaces the previously associated object UUID with the UUID in the *ObjectUuid* argument.

To set the object UUID to the nil UUID, specify a null value or the nil UUID for the *ObjectUuid* argument.

## Return Values

| Value | Meaning |
|---|---|
| RPC_S_OK | Success |
| RPC_S_INVALID_BINDING | Invalid binding handle |
| RPC_S_WRONG_KIND_OF_BINDING | Wrong kind of binding for operation |

## See Also

RpcBindingFromStringBinding, RpcBindingInqObject

# RpcBindingToStringBinding Quick Info

The **RpcBindingToStringBinding** function returns a string representation of a binding handle.

```
#include <rpc.h>

RPC_STATUS RPC_ENTRY RpcBindingToStringBinding(

  RPC_BINDING_HANDLE   Binding,
  unsigned char * *    StringBinding
);
```

## Parameters

*Binding*

Specifies a client or server binding handle to convert to a string representation of a binding handle.

*StringBinding*

Returns a pointer to a pointer to the string representation of the binding handle specified in the *Binding* argument.

Specify a null value to prevent **RpcBindingToStringBinding** from returning the *StringBinding* argument. In this case, the application does not call the **RpcStringFree** routine.

## Remarks

The **RpcBindingToStringBinding** routine converts a client or server binding handle to its string representation.

The RPC run-time library allocates memory for the string returned in the *StringBinding* argument. The application is responsible for calling the **RpcStringFree** routine to deallocate that memory.

If the binding handle in the *Binding* argument contained a nil object UUID, the object UUID field is not included in the returned string.

To parse the returned *StringBinding* argument, call the **RpcStringBindingParse** routine.

## Return Values

| Value | Meaning |
|---|---|
| RPC_S_OK | Success |
| RPC_S_INVALID_BINDING | Invalid binding handle |

## See Also

RpcBindingFromStringBinding, RpcStringBindingParse, RpcStringFree

# RpcBindingVectorFree Quick Info

The **RpcBindingVectorFree** function frees the binding handles contained in the vector and the vector itself.

```
#include <rpc.h>

RPC_STATUS RPC_ENTRY RpcBindingVectorFree(

  RPC_BINDING_VECTOR * *   BindingVector
);
```

## Parameters

*BindingVector*

Points to a pointer to a vector of server binding handles. On return, the pointer is set to NULL.

## Remarks

An application calls the **RpcBindingVectorFree** routine to release the memory used to store a vector of server binding handles. The routine frees both the binding handles and the vector itself.

A server obtains a vector of binding handles by calling the **RpcServerInqBindings** routine. A client obtains a vector of binding handles by calling the **RpcNsBindingLookupNext** routine.

## Return Values

| Value | Meaning |
|---|---|
| RPC_S_OK | Success |
| RPC_S_INVALID_ARG | Invalid argument |
| RPC_S_INVALID_BINDING | Invalid binding handle |
| RPC_S_WRONG_KIND_OF_BINDING | Wrong kind of binding for operation |

## See Also

RpcNsBindingLookupNext, RpcServerInqBindings

# RpcCancelThread Quick Info

The **RpcCancelThread** function cancels a thread.

This function is supported only by Windows NT.

```
#include <rpc.h>

RPC_STATUS RPC_ENTRY RpcCancelThread(

  HANDLE   ThreadHandle
);
```

## Parameters

*ThreadHandle*

Specifies the handle of the thread to cancel.

## Remarks

The **RpcCancelThread** routine allows one client thread to cancel an RPC in progress on another client thread. When the routine is called, the server runtime is informed of the cancel operation. The server stub can determine if the call has been cancelled by calling **RpcTestCancel**. If the call has been cancelled, the server stub should clean up and return control to the client.

By default, the client waits forever for the server to return control after a cancel. To reduce this time, call **RpcMgmtSetCancelTimeout**, specifying the number of seconds to wait for a response. If the server does not return within this interval, the call fails at the client with an RPC_S_CALL_FAILED exception. The server stub continues to execute.

**Note** This routine is only supported for Windows NT clients.

## Return Values

| Value | Meaning |
|---|---|
| RPC_S_OK | Success |
| RPC_S_ACCESS_DENIED | Thread handle does not have privilege |
| RPC_S_CANNOT_SUPPORT | Called by an MS-DOS or Windows 3.*x* client |

# RpcEndExcept Quick Info

## See

RpcExcept

# RpcEndFinally Quick Info

## See

RpcFinally

# RpcEpRegister Quick Info

The **RpcEpRegister** function adds to or replaces server address information in the local endpoint-map database.

This function is supported by both 32-bit platforms – Windows NT and Windows 95.

```
#include <rpc.h>

RPC_STATUS RPC_ENTRY RpcEpRegister(

  RPC_IF_HANDLE   IfSpec,
  RPC_BINDING_VECTOR *   BindingVector,
  UUID_VECTOR *   UuidVector,
  unsigned char *   Annotation
);
```

## Parameters

*IfSpec*

Specifies an interface to register with the local endpoint-map database.

*BindingVector*

Points to a vector of binding handles over which the server can receive remote procedure calls.

*UuidVector*

Points to a vector of object UUIDs offered by the server. The server application constructs this vector.

A null argument value indicates there are no object UUIDs to register.

*Annotation*

Points to the character-string comment applied to each cross-product element added to the local endpoint-map database. The string can be up to 64 characters long, including the null terminating character. Specify a null value or a null-terminated string ("\0") if there is no annotation string.

The annotation string is used by applications for information only. RPC does not use this string to determine which server instance a client communicates with or for enumerating elements in the endpoint-map database.

## Remarks

The **RpcEpRegister** routine adds or replaces entries in the local host's endpoint-map database. For an existing database entry that matches the provided interface specification, binding handle, and object UUID, this routine replaces the entry's endpoint with the endpoint in the provided binding handle.

A server uses **RpcEpRegister** rather than **RpcEpRegisterNoReplace** when only a single instance of the server will run on the server's host. In other words, use this routine when no more than one server instance will offer the same interface UUID, object UUID, and protocol sequence at any one time.

When entries are not replaced, stale data accumulates each time a server instance stops running without calling **RpcEpUnregister**. Stale entries increase the likelihood that a client will receive endpoints to nonexistent servers. The client will spend time trying to communicate with a nonexistent server before obtaining another endpoint.

Using **RpcEpRegister** to replace existing endpoint-map database entries reduces the likelihood that a client will be given the endpoint of a nonexistent server instance. A server application calls this routine to

register endpoints specified by calling any of the following routines:

- **RpcServerUseAllProtseqs**
- **RpcServerUseProtseq**
- **RpcServerUseProtseqEp**

A server that calls only **RpcServerUseAllProtseqsIf** or **RpcServerUseProtseqIf** does not need to call **RpcEpRegister**. In this case, the client's run-time library uses an endpoint from the client's interface specification to fill in a partially bound binding handle.

If the server also exports to the name-service database, the server calls **RpcEpRegister** with the same *IfSpec*, *BindingVector*, and *UuidVector* that the server uses when calling the **RpcNsBindingExport** routine.

For automatically started servers running over one of the connection-based protocol sequences (**ncacn_np**, **ncacn_nb**, **ncacn_ip_tcp**, **ncacn_osi_dns**), the RPC run-time library automatically generates a dynamic endpoint. In this case, the server can call **RpcServerInqBindings** followed by **RpcEpRegister** to make itself available to multiple clients. Otherwise, the automatically started server is known only to the client for which the server was started.

Each element added to the endpoint-map database logically contains the following:

- Interface UUID
- Interface version (major and minor)
- Binding handle
- Object UUID (optional)
- Annotation (optional)

**RpcEpRegister** creates a cross-product from the *IfSpec*, *BindingVector*, and *UuidVector* arguments and adds each element in the cross-product as a separate registration in the endpoint-map database.

## Return Values

| Value | Meaning |
|---|---|
| RPC_S_OK | Success |
| RPC_S_NO_BINDINGS | No bindings |
| RPC_S_INVALID_BINDING | Invalid binding handle |
| RPC_S_WRONG_KIND_OF_BINDING | Wrong kind of binding for operation |

## See Also

**RpcBindingFromStringBinding**, **RpcEpRegisterNoReplace**, **RpcEpUnregister**, **RpcNsBindingExport**, **RpcServerInqBindings**, **RpcServerUseAllProtseqs**, **RpcServerUseAllProtseqsIf**, **RpcServerUseProtseq**, **RpcServerUseProtseqEp**, **RpcServerUseProtseqIf**

# RpcEpRegisterNoReplace Quick Info

The **RpcEpRegisterNoReplace** function adds server-address information to the local endpoint-map database.

```
#include <rpc.h>

RPC_STATUS RPC_ENTRY RpcEpRegisterNoReplace(

  RPC_IF_HANDLE   IfSpec,
  RPC_BINDING_VECTOR *   BindingVector,
  UUID_VECTOR *   UuidVector,
  unsigned char *   Annotation
);
```

## Parameters

*IfSpec*

Specifies an interface to register with the local endpoint-map database.

*BindingVector*

Points to a vector of binding handles over which the server can receive remote procedure calls.

*UuidVector*

Points to a vector of object UUIDs offered by the server. The server application constructs this vector.

A null argument value indicates there are no object UUIDs to register.

*Annotation*

Points to the character-string comment applied to each cross-product element added to the local endpoint-map database. The string can be up to 64 characters long, including the null terminating character. Specify a null value or a null-terminated string ("\0") if there is no annotation string.

The annotation string is used by applications for information only. RPC does not use this string to determine which server instance a client communicates with or to enumerate elements in the endpoint-map database.

## Remarks

The **RpcEpRegisterNoReplace** routine adds entries to the local host's endpoint-map database. This routine does not replace existing database entries.

A server uses **RpcEpRegisterNoReplace** rather than **RpcEpRegister** when multiple instances of the server will run on the same host. In other words, use this routine when more than one server instance will offer the same interface UUID, object UUID, and protocol sequence at any one time.

Because entries are not replaced when calling **RpcEpRegisterNoReplace**, servers must unregister themselves before they stop running. Otherwise, stale data accumulates each time a server instance stops running without calling **RpcEpUnregister**. Stale entries increase the likelihood that a client will receive endpoints to nonexistent servers. The client will spend time trying to communicate with a nonexistent server before obtaining another endpoint.

A server application calls **RpcEpRegisterNoReplace** to register endpoints specified by calling any of the following routines:

- **RpcServerUseAllProtseqs**

- **RpcServerUseProtseq**
- **RpcServerUseProtseqEp**

A server that calls only **RpcServerUseAllProtseqsIf** or **RpcServerUseProtseqIf** is not required to call **RpcEpRegisterNoReplace**. In this case, the client's run-time library uses an endpoint from the client's interface specification to fill in a partially bound binding handle.

If the server also exports to the name-service database, the server calls **RpcEpRegisterNoReplace** with the same *IfSpec*, *BindingVector*, and *UuidVector* arguments that the server uses when calling the **RpcNsBindingExport** routine.

For automatically started servers running over one of the connection-based protocol sequences (**ncacn_np**, **ncacn_nb**, **ncacn_ip_tcp**, **ncacn_osi_dns**), the RPC run-time library automatically generates a dynamic endpoint. In this case, the server can call **RpcServerInqBindings** followed by **RpcEpRegisterNoReplace** to make itself available to multiple clients. Otherwise, the automatically started server is known only to the client for which the server was started.

Each element added to the endpoint-map database logically contains the following:

- Interface UUID
- Interface version (major and minor)
- Binding handle
- Object UUID (optional)
- Annotation (optional)

**RpcEpRegisterNoReplace** creates a cross-product from the *IfSpec*, *BindingVector*, and *UuidVector* arguments and adds each element in the cross-product as a separate registration in the endpoint-map database.

## Return Values

| Value | Meaning |
|---|---|
| RPC_S_OK | Success |
| RPC_S_NO_BINDINGS | No bindings |
| RPC_S_INVALID_BINDING | Invalid binding handle |
| RPC_S_WRONG_KIND_OF_BINDING | Wrong kind of binding for operation |

## See Also

**RpcBindingFromStringBinding**, **RpcEpRegister**, **RpcEpUnregister**, **RpcNsBindingExport**, **RpcServerInqBindings**, **RpcServerUseAllProtseqs**, **RpcServerUseAllProtseqsIf**, **RpcServerUseProtseq**, **RpcServerUseProtseqEp**, **RpcServerUseProtseqIf**

# RpcEpResolveBinding Quick Info

The **RpcEpResolveBinding** function resolves a partially bound server binding handle into a fully bound server binding handle.

```
#include <rpc.h>

RPC_STATUS RPC_ENTRY RpcEpResolveBinding(

  RPC_BINDING_HANDLE   Binding,
  RPC_IF_HANDLE   IfSpec
);
```

## Parameters

*Binding*

Specifies a partially bound server binding handle to resolve to a fully bound server binding handle.

*IfSpec*

Specifies a stub-generated data structure specifying the interface of interest.

## Remarks

An application calls the **RpcEpResolveBinding** routine to resolve a partially bound server binding handle into a fully bound binding handle.

Resolving binding handles requires an interface UUID and an object UUID (which may be nil). The RPC run-time library asks the endpoint-mapping service on the host specified by the *Binding* argument to look up an endpoint for a compatible server instance. To find the endpoint, the endpoint-mapping service looks in the endpoint-map database for the interface UUID in the *IfSpec* argument and the object UUID in the *Binding* argument, if any.

How the resolve-binding operation functions depends on whether the specified binding handle is partially or fully bound. When the client specifies a partially bound handle, the resolve-binding operation has the following possible outcomes:

- If no compatible server instances are registered in the endpoint-map database, the resolve-binding operation returns the EPT_S_NOT_REGISTERED status code.
- If a compatible server instance is registered in the endpoint-map database, the resolve-binding operation returns a fully bound binding and the RPC_S_OK status code.

When the client specifies a fully bound binding handle, the resolve-binding operation returns the specified binding handle and the RPC_S_OK status code. The resolve-binding operation does not contact the endpoint-mapping service.

In neither the partially nor the fully bound binding case does the resolve-binding operation contact a compatible server instance.

## Return Values

| Value | Meaning |
|---|---|
| RPC_S_OK | Success |
| RPC_S_INVALID_BINDING | Invalid binding handle |
| RPC_S_WRONG_KIND_OF_BINDIN | Wrong kind of binding for |

G operation

## See Also

RpcBindingFromStringBinding, RpcBindingReset, RpcEpRegister, RpcEpRegisterNoReplace, RpcNsBindingImportBegin, RpcNsBindingImportDone, RpcNsBindingImportNext

# RpcEpUnregister Quick Info

The **RpcEpUnregister** function removes server-address information from the local endpoint-map database.

This function is supported by both 32-bit platforms – Windows NT and Windows 95.

```
#include <rpc.h>

RPC_STATUS RPC_ENTRY RpcEpUnregister(

  RPC_IF_HANDLE   IfSpec,
  RPC_BINDING_VECTOR *   BindingVector,
  UUID_VECTOR *   UuidVector
);
```

## Parameters

*IfSpec*

Specifies an interface to unregister from the local endpoint-map database.

*BindingVector*

Points to a vector of binding handles to unregister.

*UuidVector*

Points to an optional vector of object UUIDs to unregister. The server application constructs this vector. **RpcEpUnregister** unregisters all endpoint-map database elements that match the specified *IfSpec* and *BindingVector* arguments and the object UUID(s).

A null argument value indicates there are no object UUIDs to unregister.

## Remarks

The **RpcEpUnregister** routine removes elements from the local host's endpoint-map database. A server application calls this routine only when the server has previously registered endpoints and the server wants to remove that address information from the endpoint-map database.

Specifically, **RpcEpUnregister** allows a server application to remove its own endpoint-map database elements (server-address information) based on the interface specification or on both the interface specification and the object UUID(s) of the resource(s) offered.

The server calls the **RpcServerInqBindings** routine to obtain the required *BindingVector* argument. To unregister selected endpoints, the server can prune the binding vector prior to calling this routine.

**RpcEpUnregister** creates a cross-product from the *IfSpec*, *BindingVector*, and *UuidVector* arguments and removes each element in the cross-product from the endpoint-map database.

Use **RpcEpUnregister** cautiously: removing elements from the endpoint-map database may make servers unavailable to client applications that have not previously communicated with the server.

## Return Values

| Value | Meaning |
|---|---|
| RPC_S_OK | Success |
| RPC_S_NO_BINDINGS | No bindings |
| RPC_S_INVALID_BINDING | Invalid binding handle |

| | |
|---|---|
| RPC_S_WRONG_KIND_OF_BINDING | Wrong kind of binding for operation |

## See Also

RpcEpRegister, RpcEpRegisterNoReplace, RpcNsBindingUnexport, RpcServerInqBindings

# RpcExcept Quick Info

The **RpcExcept** function specifies exception handling.

**RpcTryExcept**
    **{**
    *guarded statements*
    **}**
  **RpcExcept(***expression***)**
    **{**
    *exception statements*
    **}**
  **RpcEndExcept;**

## Parameters

*guarded statements*

Specifies program statements that are guarded or monitored for exceptions during execution.

*expression*

Specifies an expression that is evaluated when an exception occurs. If *expression* evaluates to a non-zero value, the exception statements are executed. If *expression* evaluates to a zero value, unwinding continues to the next **RpcTryExcept** or **RpcTryFinally** routine.

*exception statements*

Specifies statements that are executed when the expression evaluates to a non-zero value.

## Remarks

If an exception does not occur, the *expression* and *exception statements* are skipped and execution continues at the statement following the **RpcEndExcept** keyword.

**RpcExceptionCode** can be used in both *expression* and *exception statements* to determine which exception occurred.

The following restrictions apply.

- Jumping (via a **goto**) into *guarded statements* is not allowed.
- Jumping (via a **goto**) into *exception statements* is not allowed.
- Returning or jumping (via a **goto**) from *guarded statements* is not allowed.
- Returning or jumping (via a **goto**) from *exception statements* is not allowed.

## See Also

RpcExceptionCode, RpcFinally, RpcRaiseException

# RpcExceptionCode Quick Info

The **RpcExceptionCode** function returns the exception code of an exception.

**unsigned long RpcExceptionCode(VOID);**

## Remarks

The **RpcExceptionCode** function can only be called from within the *expression* and *exception statements* of an **RpcTryExcept** exception handler.

## Return Values

No value is returned.

## See Also

**RpcExcept**, **RpcFinally**

# RpcFinally Quick Info

The **RpcFinally** function specifies termination handlers.

**RpcTryFinally**
**{**
*guarded statements*
**}**
**RpcFinally**
**{**
*termination statements*
**}**
**RpcEndFinally**;

## Parameters

*guarded statements*

Specifies statements that are executed while exceptions are being monitored. If an exception occurs during the execution of these statements, *termination statements* will be executed, then unwinding continues to the next **RpcTryExcept** or **RpcTryFinally** routine.

*termination statements*

Specifies statements that are executed when an exception occurs. After the termination statements are complete, the exception is raised again.

## Remarks

The **RpcAbnormalTermination** function can be used in *termination statements* to determine whether *termination statements* is being executed because an exception occurred. A non-zero return from **RpcAbnormalTermination** indicates that an exception occurred. A value of zero indicates that no exception occurred.

The following restrictions apply:

- Jumping (via a **goto**) into *guarded statements* is not allowed.
- Jumping (via a **goto**) into *termination statements* is not allowed.
- Returning or jumping (via a **goto**) from *guarded statements* is not allowed.
- Returning or jumping (via a **goto**) from *termination statements* is not allowed.

## See Also

**RpcAbnormalTermination**

# RpcIfIdVectorFree Quick Info

The **RpcIfIdVectorFree** function frees the vector and the interface-identification structures contained in the vector.

This function is supported by both 32-bit platforms – Windows NT and Windows 95.

```
#include <rpc.h>

RPC_STATUS RPC_ENTRY RpcIfIdVectorFree(

  RPC_IF_ID_VECTOR * *   IfIdVec
);
```

## Parameters

*IfIdVec*

Specifies the address of a pointer to a vector of interface information. On return, the pointer is set to NULL.

## Remarks

An application calls the **RpcIfIdVectorFree** routine to release the memory used to store a vector of interface identifications. **RpcIfIdVectorFree** frees memory containing the interface identifications and the vector itself. On return, this routine sets the *IfIdVec* argument to NULL.

An application obtains a vector of interface identifications by calling the **RpcNsMgmtEntryInqIfIds** and **RpcMgmtInqIfIds** routines.

## Return Values

| Value | Meaning |
|---|---|
| RPC_S_OK | Success |
| RPC_S_INVALID_ARG | Invalid argument |

## See Also

RpcIfInqId, RpcMgmtInqIfIds, RpcNsMgmtEntryInqIfIds

# RpcIfInqId Quick Info

The **RpcIfInqId** function returns the interface-identification part of an interface specification.

```
#include <rpc.h>

RPC_STATUS RPC_ENTRY RpcIfInqId(

  RPC_IF_HANDLE   RpcIfHandle,
  RPC_IF_ID *   RpcIfId
);
```

## Parameters

*RpcIfHandle*

Specifies a stub-generated data structure specifying the interface to inquire.

*RpcIfId*

Returns a pointer to the interface identification. The application provides memory for the returned data.

## Remarks

An application calls the **RpcIfInqId** routine to obtain a copy of the interface identification from the provided interface specification.

The returned interface identification consists of the interface UUID and interface version numbers (major and minor) specified in the *IfSpec* argument from the IDL file.

## Return Values

| Value | Meaning |
|---|---|
| RPC_S_OK | Success |

## See Also

RpcServerInqIf, RpcServerRegisterIf

# RpcImpersonateClient Quick Info

A server thread that is processing client remote procedure calls can call the **RpcImpersonateClient** function to impersonate the active client.

This function is supported only by Windows NT.

```
#include <rpc.h>

RPC_STATUS RPC_ENTRY RpcImpersonateClient(

  RPC_BINDING_HANDLE   CallHandle
);
```

## Parameters

*CallHandle*

Specifies a binding handle on the server that represents a binding to a client. The server impersonates the client indicated by this handle. If a value of zero is specified, the server impersonates the client that is being served by this server thread.

## Return Values

| Value | Meaning |
|---|---|
| RPC_S_OK | Success. |
| RPC_S_NO_CALL_ACTIVE | No client is active on this server thread. |
| RPC_S_CANNOT_SUPPORT | The function is not supported for either the operating system, the transport, or this security subsystem. |
| RPC_S_INVALID_BINDING | Invalid binding handle. |
| RPC_S_WRONG_KIND_OF_BINDING | Wrong kind of binding for operation. |
| RPC_S_NO_CONTEXT_AVAILABLE | The server does not have permission to impersonate the client. |

## Remarks

In a multithreaded application, if the call to **RpcImpersonateClient** is with a handle to another client thread, you must call **RpcRevertToSelfEx** with the handle to that thread to end impersonation.

## See Also

**RpcRevertToSelf**, Impersonation

# RpcMacSetYieldInfo Quick Info

**#include <rpc.h>**

**RPC_STATUS RPC_ENTRY RpcMacSetYieldInfo(**

**MACYIELDCALLBACK** *pfnCallback*
**);**

## Parameters

*pfnCallback*

Pointer to a callback function.
**typedef void (RPC_ENTRY *MACYIELDCALLBACK)(short *);**

## Remarks

The **RpcMacSetYieldInfo** function configures Macintosh client applications to yield to other applications during remote procedure calls.

If a yielding function is not registered, an RPC will not yield on the Mac. Register a yielding function by calling **RpcMacSetYieldInfo**.

The yielding function must yield until **pStatus* is not equal to 1. For example:

```
void RPC_ENTRY MacCallbackFunc (short *pStatus)
{
    MSG msg;
    while (*pStatus == 1)
    {
    if(PeekMessage(&msg, NULL, 0, 0, PM_REMOVE))
        {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
        }
    }
}
```

Note that rpc.h must be included before winerror.h (or any files that include it, such as winbase.h, windows.h, and so on).

## Return Values

| Value | Meaning |
|---|---|
| RPC_S_OK | The information was set successfully. |

# RpcMgmtEnableIdleCleanup Quick Info

The **RpcMgmtEnableIdleCleanup** function closes idle resources, such as network connections, on the client. Connection-oriented protocols set five minutes as the default waiting period to determine whether a resource is idle.

This function is supported by the Windows NT, Windows 95 and Windows 3.*x* platforms. It is not supported by MS-DOS.

**Note** **RpcMgmtEnableIdleCleanup** is a Microsoft extension to the DCE API set.

```
#include <rpc.h>

RPC_STATUS RPC_ENTRY RpcMgmtEnableIdleCleanup(VOID);
```

## Return Values

| Value | Meaning |
|---|---|
| RPC_S_OK | Success |
| RPC_S_OUT_OF_THREADS | Out of threads |
| RPC_S_OUT_OF_RESOURCES | Out of resources |
| RPC_S_OUT_OF_MEMORY | Out of memory |

## See Also

RpcServerUnregisterIf

# RpcMgmtEpEltInqBegin Quick Info

The **RpcMgmtEpEltInqBegin** function creates an inquiry context for viewing the elements in an endpoint map.

This function is supported by both 32-bit platforms – Windows NT and Windows 95.

```
#include <rpc.h>

RPC_STATUS RPC_ENTRY RpcMgmtEpEltInqBegin(

  RPC_BINDING_HANDLE   EpBinding,
  unsigned long   InquiryType,
  RPC_IF_ID *   IfId,
  unsigned long   VersOption,
  UUID *   ObjectUuid,
  RPC_EP_INQ_HANDLE *   InquiryContext
);
```

## Parameters

*EpBinding*

Specifies the host whose endpoint map elements will be viewed. Specify NULL to view elements from the local host.

*InquiryType*

Specifies an integer value that indicates the type of inquiry to perform on the endpoint map. The following are valid inquiry types:

| Value | Description |
|---|---|
| RPC_C_EP_ALL_ELTS | Returns every element from the endpoint map. The *IfId*, *VersOption*, and *ObjectUuid* parameters are ignored. |
| RPC_C_EP_MATCH_BY_IF | Searches the endpoint map for those elements that contain the interface identifier specified by the *IfId* and *VersOption* values. |
| RPC_C_EP_MATCH_BY_OBJ | Searches the endpoint map for those elements that contain the object UUID specified by *ObjectUuid*. |
| RPC_C_EP_MATCH_BY_BOTH | Searches the endpoint map for those elements that contain the interface identifier and object UUID specified by *IfId*, *VersOption*, and *ObjectUuid*. |

*IfId*

Specifies the interface identifier of the endpoint map elements to be returned by **RpcMgmtEpEltInqNext**. This parameter is only used when *InquiryType* is either RPC_C_EP_MATCH_BY_IF or RPC_C_EP_MATCH_BY_BOTH. Otherwise, it is ignored.

*VersOption*

Specifies how **RpcMgmtEpEltInqNext** uses the IfId parameter. This parameter is only used when *InquiryType* is either RPC_C_EP_MATCH_BY_IF or RPC_C_EP_MATCH_BY_BOTH. Otherwise, it is ignored. The following are valid values for this parameter:

| Value | Description |
|---|---|
| RPC_C_VERS_ALL | Returns endpoint map elements that offer the specified interface UUID, regardless of the version numbers. |
| RPC_C_VERS_COMPATIBLE | Returns endpoint map elements that offer the same major version of the specified interface UUID and a minor version greater than or equal to the minor version of the specified interface UUID. |
| RPC_C_VERS_EXACT | Returns endpoint map elements that offer the specified version of the specified interface UUID. |
| RPC_C_VERS_MAJOR_ONLY | Returns endpoint map elements that offer the same major version of the specified interface UUID and ignores the minor version. |
| RPC_C_VERS_UPTO | Returns endpoint map elements that offer a version of the specified interface UUID less than or equal to the specified major and minor version. |

*ObjectUuid*

Specifies the object UUID that **RpcMgmtEpEltInqNext** looks for in endpoint map elements. This parameter is used only when *InquiryType* is either RPC_C_EP_MATCH_BY_OBJ or RPC_C_EP_MATCH_BY_BOTH.

*InquiryContext*

Returns an inquiry context for use with **RpcMgmtEpEltInqNext** and **RpcMgmtEpEltInqDone**.

## Remarks

The **RpcMgmtEpEltInqBegin** routine creates an inquiry context for viewing server address information stored in the endpoint map. Using *InquiryType* and *VersOption*, an application specifies which of the following endpoint map elements are to be returned from calls to **RpcMgmtEpEltInqNext**:

- All elements.
- Those elements with the specified interface identifier.
- Those elements with the specified object UUID.
- Those elements with both the specified interface identifier and object UUID.

Before calling **RpcMgmtEpEltInqNext**, the application must first call this routine to create an inquiry context. After viewing the endpoint map elements, the application calls **RpcMgmtEpEltInqDone** to delete the inquiry context.

## Return Values

| Value | Meaning |
| --- | --- |
| RPC_S_OK | Success |

## See Also

RpcEpRegister

# RpcMgmtEpEltInqDone Quick Info

The **RpcMgmtEpEltInqDone** function deletes the inquiry context for viewing the elements in an endpoint map.

This function is supported by both 32-bit platforms – Windows NT and Windows 95.

```
#include <rpc.h>

RPC_STATUS RPC_ENTRY RpcMgmtEpEltInqDone(

  RPC_EP_INQ_HANDLE *   InquiryContext
);
```

## Parameters

*InquiryContext*

Specifies the inquiry context to delete and returns the value NULL.

## Remarks

The **RpcMgmtEpEltInqDone** routine deletes an inquiry context created by **RpcMgmtEpEltInqBegin**. An application calls this routine after viewing local endpoint map elements using **RpcMgmtEpEltInqNext**.

## Return Values

| Value | Meaning |
|---|---|
| RPC_S_OK | Success |

## See Also

RpcEpRegister

# RpcMgmtEpEltInqNext Quick Info

The **RpcMgmtEpEltInqNext** function returns one element from an endpoint map.

This function is supported by both 32-bit platforms – Windows NT and Windows 95.

```
#include <rpc.h>

RPC_STATUS RPC_ENTRY RpcMgmtEpEltInqNext(

  RPC_EP_INQ_HANDLE   InquiryContext,
  RPC_IF_ID * IfId,
  RPC_BINDING_HANDLE * Binding
  UUID *   ObjectUuid,
  unsigned char * *   Annotation
);
```

## Parameters

*InquiryContext*

Specifies an inquiry context. The inquiry context is returned from **RpcMgmtEpEltInqBegin**.

*IfId*

Returns the interface identifier of the endpoint map element.

*Binding*

Optional. Returns the binding handle from the endpoint map element.

*ObjectUuid*

Optional. Returns the object UUID from the endpoint map element.

*Annotation*

Optional. Returns the annotation string for the endpoint map element. When there is no annotation string in the endpoint map element, the empty string ("") is returned.

## Remarks

The **RpcMgmtEpEltInqNext** routine returns one element from the endpoint map. Elements selected depend on the inquiry context. The selection criteria are determined by *InquiryType* of the **RpcMgmtEpEltInqBegin** routine that returned *InquiryContext*.

An application can view all the selected endpoint map elements by repeatedly calling **RpcMgmtEpEltInqNext**. When all the elements have been viewed, this routine returns an RPC_X_NO_MORE_ENTRIES status. The returned elements are unordered.

When the respective arguments are non-NULL, the RPC run-time function library allocates memory for *Binding* and *Annotation* on each call to this routine. The application is responsible for calling **RpcBindingFree** for each returned *Binding* and **RpcStringFree** for each returned *Annotation*.

After viewing the endpoint map's elements, the application must call **RpcMgmtEpEltInqDone** to delete the inquiry context.

## Return Values

| Value | Meaning |
|---|---|

| | |
|---|---|
| RPC_S_OK | Success |

## See Also

RpcEpRegister

# RpcMgmtEpUnregister Quick Info

The **RpcMgmtEpUnregister** function removes server address information from an endpoint map.

This function is supported by both 32-bit platforms – Windows NT and Windows 95.

```
#include <rpc.h>

RPC_STATUS RPC_ENTRY RpcMgmtEpUnregister(

  RPC_BINDING_HANDLE   EpBinding,
  RPC_IF_ID *   IfId,
  RPC_BINDING_HANDLE  Binding,
  UUID *   ObjectUuid
);
```

## Parameters

*EpBinding*

Specifies the host whose endpoint map elements are to be unregistered. To remove elements from the same host as the calling application, the application specifies NULL. To remove elements from another host, the application specifies a server binding handle for any server residing on that host. Note that the application can specify the same binding handle it is using to make other remote procedure calls.

*IfId*

Specifies the interface identifier to remove from the endpoint map.

*Binding*

Specifies the binding handle to remove.

*ObjectUuid*

Specifies the optional object UUID to remove. The value NULL indicates there is no object UUID to remove.

## Remarks

The **RpcMgmtEpUnregister** routine unregisters an element from the endpoint map. A management program calls this routine to remove addresses of servers that are no longer available, or to remove addresses of servers that support objects that are no longered offered.

The *EpBinding* parameter must be a full binding. The object UUID associated with the *EpBinding* parameter must be a nil UUID. Specifying a non-nil UUID causes the routine to fail with the status code EPT_S_CANT_PERFORM_OP. Other than the host information and object UUID, all information in this argument is ignored.

An application calls **RpcMgmtEpEltInqNext** to view local endpoint map elements. The application can then remove the elements using **RpcMgmtEpUnregister**.

**Note** Use this routine with caution. Removing elements from the local endpoint map may make servers unavailable to client applications that do not already have a fully bound binding handle to the server.

## Return Values

| Value | Meaning |
| --- | --- |
| RPC_S_OK | Success |
| RPC_S_CANT_PERFORM_OP | Cannot perform the requested operation |

## See Also

RpcEpRegister, RpcEpUnregister

# RpcMgmtInqComTimeout Quick Info

The **RpcMgmtInqComTimeout** function returns the binding-communications timeout value in a binding handle.

```
#include <rpc.h>

RPC_STATUS RPC_ENTRY RpcMgmtInqComTimeout(

  RPC_BINDING_HANDLE   Binding,
  unsigned int *   Timeout
);
```

## Parameters

*Binding*

Specifies a binding.

*Timeout*

Returns a pointer to the timeout value from the *Binding* argument.

## Remarks

A client application calls **RpcMgmtInqComTimeout** to view the timeout value in a server binding handle. The timeout value specifies the relative amount of time that should be spent to establish a binding to the server before giving up. The table below shows the timeout values.

A client calls **RpcMgmtSetComTimeout** to change the timeout value.

## Return Values

| Value | Meaning |
|---|---|
| RPC_S_OK | Success |
| RPC_S_INVALID_BINDING | Invalid binding handle |
| RPC_S_WRONG_KIND_OF_BINDING | Wrong kind of binding for operation |

## See Also

RpcMgmtInqStats, RpcMgmtSetComTimeout

# RpcMgmtInqDefaultProtectLevel Quick Info

The **RpcMgmtInqDefaultProtectLevel** function returns the default authentication level for an authentication service.

```
#include <rpc.h>

RPC_STATUS RPC_ENTRY RpcMgmtInqDefaultProtectLevel(

  unsigned int   AuthnSvc,
  unsigned int   *   AuthnLevel
);
```

## Parameters

*AuthnSvc*

Specifies the authentication service for which to return the default authentication level. Possible values are as follows:

| Value | Description |
|---|---|
| RPC_C_AUTHN_NONE | No authentication |
| RPC_C_AUTHN_WINNT | 32-bit Windows authentication service |

*AuthnLevel*

Returns the default authentication level for the specified authentication service. The authentication level determines the degree to which authenticated communications between the client and server are protected. Possible values are as follows:

| Value | Description |
|---|---|
| RPC_C_AUTHN_LEVEL_DEFAULT | Uses the default authentication level for the specified authentication service. |
| RPC_C_AUTHN_LEVEL_NONE | Performs no authentication. |
| RPC_C_AUTHN_LEVEL_CONNECT | Authenticates only when the client establishes a relationship witha server. |
| RPC_C_AUTHN_LEVEL_CALL | Authenticates only at the beginning of each remote procedure call when the server receives the request. Does not apply to remote procedure calls made using the connection-based protocol sequences that start with the prefix "ncacn." If the protocol sequence in a binding is a connection-based protocol sequence and you specify this level, this routine instead uses the RPC_C_AUTHN_LEVEL_PKT constant. |
| RPC_C_AUTHN_LEVEL_PKT | Authenticates that all data received is from the expected client. |
| RPC_C_AUTHN_LEVEL_PKT | Authenticates and verifies that none |

| | |
|---|---|
| _INTEGRITY | of the data transferred between client and server has been modified. |
| RPC_C_AUTHN_LEVEL_PKT _PRIVACY | Authenticates all previous levels and encrypts the argument value of each remote procedure call. |

**Note** RPC_C_AUTHN_LEVEL_CALL, RPC_C_AUTHN_LEVEL_PKT, RPC_C_AUTHN_LEVEL_PKT_INTEGRITY, and RPC_C_AUTHN_LEVEL_PKT_PRIVACY are only supported for clients communicating with a Windows NT server. A Windows 95 server can only accept incoming calls at the RPC_C_AUTHN_LEVEL_CONNECT level.

## Remarks

An application calls the **RpcMgmtInqDefaultProtectLevel** routine to obtain the default authentication level for a specified authentication service.

## Return Values

| Value | Meaning |
|---|---|
| RPC_S_OK | Success |
| RPC_S_UNKNOWN_AUTH_SERVIC E | Unknown authentication service |

# RpcMgmtInqIfIds Quick Info

The **RpcMgmtInqIfIds** function returns a vector containing the identifiers of the interfaces offered by the server.

This function is supported by both 32-bit platforms – Windows NT and Windows 95.

```
#include <rpc.h>

RPC_STATUS RPC_ENTRY RpcMgmtInqIfIds(

  RPC_BINDING_HANDLE   Binding,
  RPC_IF_ID_VECTOR * *   IfIdVector
);
```

## Parameters

*Binding*

To receive interface identifiers about a remote application, specify a server binding handle for that application. To receive interface information about your own application, specify a value of NULL.

*IfIdVector*

Returns the address of an interface identifier vector.

## Remarks

An application calls the **RpcMgmtInqIfIds** routine to obtain a vector of interface identifiers about the specified server from the RPC run-time library.

The RPC run-time library allocates memory for the interface identifier vector. The application is responsible for calling the **RpcIfIdVectorFree** routine to release the memory used by this vector.

## Return Values

| Value | Meaning |
|---|---|
| RPC_S_OK | Success |
| RPC_S_INVALID_BINDING | Invalid binding handle |
| RPC_S_WRONG_KIND_OF_BINDING | Wrong kind of binding for operation |

# RpcMgmtInqServerPrincName Quick Info

The **RpcMgmtInqServerPrincName** function returns a server's principal name.

This function is supported by both 32-bit platforms − Windows NT and Windows 95. Note that it is supported only in ANSI on Windows 95.

```
#include <rpc.h>

RPC_STATUS RPC_ENTRY RpcMgmtInqServerPrincName(

  RPC_BINDING_HANDLE   Binding,
  unsigned int   AuthnSvc,
  unsigned char * *   ServerPrincName
);
```

## Parameters

*Binding*

To receive the principal name for a server, specify a server binding handle for that server. To receive the principal name for your own (local) application, specify a value of NULL.

*AuthnSvc*

Specifies the authentication service for which a principal name is returned. Possible values are as follows:

| Value | Description |
|---|---|
| RPC_C_AUTHN_NONE | No authentication |
| RPC_C_AUTHN_WINNT | Windows NT authentication service |

*ServerPrincName*

Returns a principal name that is registered for the authentication service in *AuthnSvc* by the server referenced in *Binding*. If multiple names are registered, only one name is returned.

## Remarks

An application calls the **RpcMgmtInqServerPrincName** routine to obtain the principal name of a server that is registered for a specified authentication service.

The RPC run-time library allocates memory for string returned in *ServerPrincName*. The application is responsible for calling the **RpcStringFree** routine to release the memory used by this routine.

## Return Values

| Value | Meaning |
|---|---|
| RPC_S_OK | Success |
| RPC_S_INVALID_BINDING | Invalid binding handle |
| RPC_S_WRONG_KIND_OF_BINDING | Wrong kind of binding for operation |

# RpcMgmtInqStats Quick Info

The **RpcMgmtInqStats** function returns RPC run-time statistics.

This function is supported by both 32-bit platforms – Windows NT and Windows 95.

```
#include <rpc.h>

RPC_STATUS RPC_ENTRY RpcMgmtInqStats(

  RPC_BINDING_HANDLE   Binding,
  RPC_STATS_VECTOR * *   Statistics
);
```

## Parameters

*Binding*

To receive statistics about a remote application, specify a server binding handle for that application. To receive statistics about your own (local) application, specify a value of NULL.

*Statistics*

Returns a pointer to a pointer to the statistics about the server specified by the *Binding* argument. Each statistic is an unsigned long value.

## Remarks

An application calls the **RpcMgmtInqStats** routine to obtain statistics about the specified server from the RPC run-time library.

Each array element in the returned statistics vector contains an unsigned long value. The following list describes the statistics indexed by the specified constant:

| Statistic | Description |
|---|---|
| RPC_C_STATS_CALLS_IN | The number of remote procedure calls received by the server |
| RPC_C_STATS_CALLS_OUT | The number of remote procedure calls initiated by the server |
| RPC_C_STATS_PKTS_IN | The number of network packets received by the server |
| RPC_C_STATS_PKTS_OUT | The number of network packets sent by the server |

The RPC run-time library allocates memory for the statistics vector. The application is responsible for calling the **RpcMgmtStatsVectorFree** routine to release the memory used by the statistics vector.

## Return Values

| Value | Meaning |
|---|---|
| RPC_S_OK | Success |
| RPC_S_INVALID_BINDING | Invalid binding handle |
| RPC_S_WRONG_KIND_OF_BINDIN | Wrong kind of binding for |

G operation

## See Also

RpcEpResolveBinding, RpcMgmtStatsVectorFree

# RpcMgmtIsServerListening Quick Info

The **RpcMgmtIsServerListening** function tells whether a server is listening for remote procedure calls.

This function is supported by both 32-bit platforms – Windows NT and Windows 95.

```
#include <rpc.h>

RPC_STATUS RPC_ENTRY RpcMgmtIsServerListening(

  RPC_BINDING_HANDLE   Binding
);
```

## Parameters

*Binding*

To determine whether a remote application is listening for remote procedure calls, specify a server binding handle for that application. To determine whether your own (local) application is listening for remote procedure calls, specify a value of NULL.

## Remarks

An application calls the **RpcMgmtIsServerListening** routine to determine whether the server specified in the *Binding* argument is listening for remote procedure calls.

**RpcMgmtIsServerListening** returns a true value if the server has called the **RpcServerListen** routine.

## Return Values

| Value | Meaning |
|---|---|
| RPC_S_OK | Server listening for remote procedure calls |
| RPC_S_SERVER_NOT_LISTENING | Server not listening for remote procedure calls |
| RPC_S_INVALID_BINDING | Invalid binding handle |
| RPC_S_WRONG_KIND_OF_BINDING | Wrong kind of binding for operation |

## See Also

RpcEpResolveBinding, RpcServerListen

# RpcMgmtSetAuthorizationFn Quick Info

The **RpcMgmtSetAuthorizationFn** function establishes an authorization function for processing remote calls to a server's management routines.

This function is supported only by Windows NT.

```
#include <rpc.h>

RPC_STATUS RPC_ENTRY RpcMgmtSetAuthorizationFn(

  RPC_MGMT_AUTHORIZATION_FN   AuthorizationFn
);
```

## Parameters

*AuthorizationFn*

Specifies an authorization function. The RPC server run-time library automatically calls this function whenever the server runtime receives a client request to execute one of the remote management routines. The server must implement this function. Applications specify NULL to unregister a previously registered authorization function. After such a call, default authorizations are used.

## Remarks

Server applications call the **RpcMgmtSetAuthorizationFn** routine to establish an authorization function that controls access to the server's remote management routines. When a server has not called **RpcMgmtSetAuthorizationFn**, or calls with a NULL value for *AuthorizationFn*, the server run-time library uses the following default authorizations:

| Remote routine | Default authorization |
|---|---|
| **RpcMgmtInqIfIds** | Enabled |
| **RpcMgmtInqServerPrincName** | Enabled |
| **RpcMgmtInqStats** | Enabled |
| **RpcMgmtIsServerListening** | Enabled |
| **RpcMgmtStopServerListening** | Disabled |

In the above table, "Enabled" indicates that all clients can execute the remote routine, and "Disabled" indicates that all clients are prevented from executing the remote routine.

The following example shows the prototype for authorization function that the server must implement:

```
typedef boolean32  (*RPC_MGMT_AUTHORIZATION_FN)
   (
   RPC_BINDING_HANDLE      ClientBinding              /* in */
   unsigned long           RequestedMgmtOperation     /* in */
   RPC_STATUS *            Status                     /* out */
   );
```

When a client requests one of the server's remote management functions, the server run-time library calls the authorization function with *ClientBinding* and *RequestedMgmtOperation*. The authorization function uses these parameters to determine whether the calling client can execute the requested management routine.

The value for *RequestedMgmtOperation* depends on the remote routine requested, as shown in the following:

| Called remote routine | ***RequestedMgmtOperation* value** |
|---|---|
| **RpcMgmtInqIfIds** | RPC_C_MGMT_INQ_IF_IDS |
| **RpcMgmtInqServerPrincName** | RPC_C_MGMT_INQ_PRINC_NAME |
| **RpcMgmtInqStats** | RPC_C_MGMT_INQ_STATS |
| **RpcMgmtIsServerListening** | RPC_C_MGMT_IS_SERVER_LISTEN |
| **RpcMgmtStopServerListening** | RPC_C_MGMT_STOP_SERVER_LISTEN |

The authorization function must handle all of these values.

The authorization function returns a Boolean value to indicate whether the calling client is allowed access to the requested management function. If the authorization function returns TRUE, the management routine can execute. If the authorization function returns FALSE, the management routine cannot execute. If this is the case, the routine returns a *Status* value to the client:

- If *Status* is either 0 (zero) or RPC_S_OK, the *Status* value RPC_S_ACCESS_DENIED is returned to the client by the remote management routine.
- If the authorization function returns any other value for *Status*, that *Status* value is returned to the client by the remote management routine.

## Return Values

| Value | Meaning |
|---|---|
| RPC_S_OK | Success |

## See Also

**RpcMgmtInqStats**, **RpcMgmtIsServerListening**, **RpcMgmtStopServerListening**, **RpcMgmtWaitServerListen**

# RpcMgmtSetCancelTimeout Quick Info

The **RpcMgmtSetCancelTimeout** function sets the lower bound on the time to wait before timing out after forwarding a cancel.

This function is supported only by Windows NT.

```
#include <rpc.h>

RPC_STATUS RPC_ENTRY RpcMgmtSetCancelTimeout(

  signed int   Seconds
);
```

## Parameters

*Seconds*

An integer specifying the number of seconds to wait for a server to acknowledge a cancel. To specify that a client waits an indefinite amount of time, supply the value RPC_C_CANCEL_INFINITE_TIMEOUT.

## Remarks

An application calls the **RpcMgmtSetCancelTimeout** routine to reset the amount of time the run-time library waits for a server to acknowledge a cancel. The application specifies either to wait forever or to wait a specified length of time in seconds. If the value of *Seconds* is 0 (zero), the call is immediately abandoned upon a cancel and control returns to the client application. The default value is RPC_C_CANCEL_INFINITE_TIMEOUT, which specifies waiting forever for the call to complete.

The value for the cancel time-out applies to all remote procedure calls made in the current thread. To change the time-out value, a multithreaded client must call this routine in each thread of execution.

**Note** This routine is only supported for Windows NT clients.

## Return Values

| Value | Meaning |
|---|---|
| RPC_S_OK | Success |
| RPC_S_CANNOT_SUPPORT | Called from an MS-DOS or Windows 3.*x* client |

# RpcMgmtSetComTimeout Quick Info

The **RpcMgmtSetComTimeout** function sets the binding-communications timeout value in a binding handle.

```
#include <rpc.h>

RPC_STATUS RPC_ENTRY RpcMgmtSetComTimeout(

  RPC_BINDING_HANDLE   Binding,
  unsigned int   Timeout
);
```

## Parameters

*Binding*

Specifies the server binding handle whose timeout value is set.

*Timeout*

Specifies the communications timeout value.

## Remarks

A client application calls **RpcMgmtSetComTimeout** to change the communications timeout value for a server binding handle. The timeout value specifies the relative amount of time that should be spent to establish a relationship to the server before giving up. Depending on the protocol sequence for the specified binding handle, the timeout value acts only as a hint to the RPC run-time library.

After the initial relationship is established, subsequent communications for the binding handle revert to not less than the default timeout for the protocol service. This means that after setting a short initial timeout establishing a connection, calls in progress will not be timed out any more aggressively than the default.

The timeout value can be any integer value from 0 to 10. For convenience, constants are provided for certain values in the timeout range. The following table contains the RPC-defined values that an application can use for the timeout argument:

| Manifest | Value | Description |
|---|---|---|
| RPC_C_BINDING_INFINITE_TIMEOUT | 10 | Keep trying to establish communications forever. |
| RPC_C_BINDING_MIN_TIMEOUT | 0 | Try the minimum amount of time for the network protocol being used. This value favors response time over correctness in determining whether the server is running. |
| RPC_C_BINDING_DEFAULT_TIMEOUT | 5 | Try an average amount of time for the network protocol being used. This value gives correctness in determining whether a server is running and gives response time equal weight. This is the default |

| | | |
|---|---|---|
| | | value. |
| RPC_C_BINDING_MAX_TIMEOUT | 9 | Try the longest amount of time for the network protocol being used. This value favors correctness in determining whether a server is running over response time. |

**Note** The values in the preceding table are not in seconds. These values represent a relative amount of time on a scale of zero to 10.

## Return Values

| Value | Meaning |
|---|---|
| RPC_S_OK | Success |
| RPC_S_INVALID_BINDING | Invalid binding handle |
| RPC_S_INVALID_TIMEOUT | Invalid timeout value |
| RPC_S_WRONG_KIND_OF_BINDING | Wrong kind of binding for operation |

## See Also

RpcMgmtInqComTimeout

# RpcMgmtSetServerStackSize Quick Info

The **RpcMgmtSetServerStackSize** function specifies the stack size for each server thread.

This function is supported by both 32-bit platforms – Windows NT and Windows 95.

```
#include <rpc.h>

RPC_STATUS RPC_ENTRY RpcMgmtSetServerStackSize(

  unsigned int   ThreadStackSize
);
```

## Parameters

*ThreadStackSize*

Specifies the stack size in bytes allocated for each thread created by **RpcServerListen**. This value is applied to all threads created for the server. Select this value based on the stack requirements of the remote procedures offered by the server.

## Remarks

A server application calls the **RpcMgmtSetServerStackSize** routine to specify the thread stack size to use when the RPC run-time library creates call threads for executing remote procedure calls. The *MaxCalls* argument in the **RpcServerListen** routine specifies the number of call threads created.

Servers that know the stack requirements of all the manager routines in the interfaces it offers can call the **RpcMgmtSetServerStackSize** routine to ensure that each call thread has the necessary stack size.

Calling **RpcMgmtSetServerStackSize** is optional. However, when used, it must be called before the server calls **RpcServerListen**. If a server does not call **RpcMgmtSetServerStackSize**, the default per thread stack size from the underlying threads package is used.

## Return Values

| Value | Meaning |
|---|---|
| RPC_S_OK | Success |
| RPC_S_INVALID_ARG | Invalid argument |

## See Also

RpcServerListen

# RpcMgmtStatsVectorFree Quick Info

The **RpcMgmtStatsVectorFree** function frees a statistics vector.

This function is supported by both 32-bit platforms – Windows NT and Windows 95.

```
#include <rpc.h>

RPC_STATUS RPC_ENTRY RpcMgmtStatsVectorFree(

  RPC_STATS_VECTOR * *   StatsVector
);
```

## Parameters

*StatsVector*

Points to a pointer to a statistics vector. On return, the pointer is set to NULL.

## Remarks

An application calls the **RpcMgmtStatsVectorFree** routine to release the memory used to store statistics.

An application obtains a vector of statistics by calling the **RpcMgmtInqStats** routine.

## Return Values

| Value | Meaning |
|---|---|
| RPC_S_OK | Success |

## See Also

RpcMgmtInqStats

# RpcMgmtStopServerListening Quick Info

The **RpcMgmtStopServerListening** function tells a server to stop listening for remote procedure calls. This function will not affect auto-listen interfaces. See **RpcServerRegisterIfEx** for more details.

This function is supported by both server platforms − Windows NT and Windows 95.

```
#include <rpc.h>

RPC_STATUS RPC_ENTRY RpcMgmtStopServerListening(

  RPC_BINDING_HANDLE   Binding
);
```

## Parameters

*Binding*

To direct a remote application to stop listening for remote procedure calls, specify a server binding handle for that application. To direct your own (local) application to stop listening for remote procedure calls, specify a value of NULL.

## Remarks

An application calls the **RpcMgmtStopServerListening** routine to direct a server to stop listening for remote procedure calls. If *DontWait* was true, the application should call **RpcMgmtWaitServerListen** to wait for all calls to complete.

When it receives a stop-listening request, the RPC run-time library stops accepting new remote procedure calls for all registered interfaces. Executing calls are allowed to complete, including callbacks.

After all calls complete, the **RpcServerListen** routine returns to the caller. If *DontWait* is true, the application calls **RpcMgmtServerListen** for all calls to complete.

## Return Values

| Value | Meaning |
|---|---|
| RPC_S_OK | Success |
| RPC_S_INVALID_BINDING | Invalid binding handle |
| RPC_S_WRONG_KIND_OF_BINDING | Wrong kind of binding for operation |

**Note** From the client-side, **RpcMgmtStopServerListening** is disabled by default. To enable this routine, create an authorization function in your server application that returns TRUE (to allow a remote shutdown) whenever **RpcMgmtStopServerListening** is called. Use **RpcMgmtSetAuthorizationFn** to give the client access to the management function.

## See Also

**RpcEpResolveBinding**, **RpcMgmtWaitServerListen**, **RpcServerListen**, **RpcServerRegisterIfEx**

# RpcMgmtWaitServerListen Quick Info

The **RpcMgmtWaitServerListen** function performs the wait operation usually associated with **RpcServerListen**.

```
#include <rpc.h>

RPC_STATUS RPC_ENTRY RpcMgmtWaitServerListen(VOID);
```

## Remarks

**Note** **RpcMgmtWaitServerListen** is a Microsoft extension to the DCE API set.

This function is supported by both 32-bit platforms – Windows NT and Windows 95.

When the **RpcServerListen** flag parameter *DontWait* has a nonzero value, the **RpcServerListen** function returns to the server application without performing the wait operation. In this case, the wait can be performed by **RpcMgmtWaitServerListen**.

Applications must call **RpcServerListen** with a nonzero value for the *DontWait* parameter before calling **RpcMgmtWaitServerListen**.

**RpcMgmtWaitServerListen** returns after the server application calls **RpcMgmtStopServerListening** and all active remote procedure calls complete, or after a fatal error occurs in the RPC run-time library.

## Return Values

| Value | Meaning |
|---|---|
| RPC_S_OK | All remote procedure calls are complete. |
| RPC_S_ALREADY_LISTENING | Another thread has called **RpcMgmtWaitServerListen** and has not yet returned. |
| RPC_S_NOT_LISTENING | The server application must call **RpcServerListen** before calling **RpcMgmtWaitServerListen**. |

## See Also

RpcMgmtStopServerListening, RpcServerListen

# RpcNetworkInqProtseqs Quick Info

The **RpcNetworkInqProtseqs** function returns all protocol sequences supported by both the RPC run-time library and the operating system.

This function is supported by both 32-bit platforms – Windows NT and Windows 95.

For a list of Microsoft RPC's supported protocol sequences, see the reference topic String Binding in RPC Data Types and Structures.

```
#include <rpc.h>

RPC_STATUS RPC_ENTRY RpcNetworkInqProtseqs(

  RPC_PROTSEQ_VECTOR * *   ProtSeqVector
);
```

## Parameters

*ProtSeqVector*

Returns a pointer to a pointer to a protocol sequence vector.

## Remarks

**Note** **RpcNetworkInqProtseqs** is available for server applications, not client applications, using Microsoft RPC. Use **RpcNetworkIsProtseqValid** in client applications.

A server application calls the **RpcNetworkInqProtseqs** routine to obtain a vector containing the protocol sequences supported by both the RPC run-time library and the operating system. If there are no supported protocol sequences, this routine returns the RPC_S_NO_PROTSEQS status code and a *ProtSeqVector* argument value of NULL.

The server is responsible for calling the **RpcProtseqVectorFree** routine to release the memory used by the vector.

## Return Values

| Value | Meaning |
|---|---|
| RPC_S_OK | Success |
| RPC_S_NO_PROTSEQS | No supported protocol sequences |

## See Also

**RpcProtseqVectorFree**

# RpcNetworkIsProtseqValid Quick Info

The **RpcNetworkIsProtseqValid** function tells whether the specified protocol sequence is supported by both the RPC run-time library and the operating system.

For a list of Microsoft RPC's supported protocol sequences, see the reference topic String Binding in RPC Data Types and Structures.

```
#include <rpc.h>

RPC_STATUS RPC_ENTRY RpcNetworkIsProtseqValid(

  unsigned char *   Protseq
);
```

## Parameters

*Protseq*

Points to a string identifier of the protocol sequence to be checked.

If the *Protseq* argument is not a valid protocol sequence string, **RpcNetworkIsProtseqValid** returns RPC_S_INVALID_RPC_PROTSEQ.

## Remarks

**Note** **RpcNetworkIsProtseqValid** is available for client applications, not for server applications. Use **RpcNetworkInqProtseqs** for server applications.

An application calls the **RpcNetworkIsProtseqValid** routine to determine whether an individual protocol sequence is available for making remote procedure calls.

A protocol sequence is valid if both the RPC run-time library and the operating system support the specified protocols. For a list of Microsoft RPC's supported protocol sequences, see the reference topic String Binding in RPC Data Types and Structures.

An application calls **RpcNetworkInqProtseqs** to see all of the supported protocol sequences.

## Return Values

| Value | Meaning |
|---|---|
| RPC_S_OK | Success; protocol sequence supported |
| RPC_S_PROTSEQ_NOT_SUPPORTED | Protocol sequence not supported on this host |
| RPC_S_INVALID_RPC_PROTSEQ | Invalid protocol sequence |

## See Also

**RpcNetworkInqProtseqs**

# RpcNsBindingExport Quick Info

The **RpcNsBindingExport** function establishes a name-service database entry with multiple binding handles and multiple objects for a server.

This function is supported by both 32-bit platforms – Windows NT and Windows 95.

```
#include <rpc.h>

RPC_STATUS RPC_ENTRY RpcNsBindingExport(

  unsigned long   EntryNameSyntax,
  unsigned char *   EntryName,
  RPC_IF_HANDLE   IfSpec,
  RPC_BINDING_VECTOR *   BindingVec,
  UUID_VECTOR *   ObjectUuidVec
);
```

## Parameters

*EntryNameSyntax*

Specifies an unsigned long value that indicates the syntax of the next argument, *EntryName*.

To use the syntax specified in the registry value HKEY_LOCAL_MACHINE\Software\Microsoft\Rpc\NameService\
DefaultSyntax, provide a value of RPC_C_NS_SYNTAX_DEFAULT.

*EntryName*

Points to the entry name to which binding handles and object UUIDs are exported. You may not provide a null or empty string.

To use the entry name specified in the registry value HKEY_LOCAL_MACHINE\Software\Microsoft\Rpc\NameService\
DefaultEntry, provide a null pointer or an empty string. In this case, the *EntryNameSyntax* parameter is ignored and the run-time library uses the default syntax *EntryName*.

*IfSpec*

Specifies a stub-generated data structure specifying the interface to export. A null argument value indicates there are no binding handles to export (only object UUIDs are to be exported) and the *BindingVec* argument is ignored.

*BindingVec*

Points to server bindings to export. A null argument value indicates there are no binding handles to export (only object UUIDs are to be exported).

*ObjectUuidVec*

Points to a vector of object UUIDs offered by the server. The server application constructs this vector. A null argument value indicates there are no object UUIDs to export (only binding handles are to be exported).

## Remarks

The **RpcNsBindingExport** routine allows a server application to publicly offer an interface in the name-service database for use by any client application.

To export an interface, the server application calls the **RpcNsBindingExport** routine with an interface and

the server binding handles a client can use to access the server.

A server application also calls the **RpcNsBindingExport** routine to publicly offer the object UUID(s) of resource(s) it offers, if any, in the name-service database.

A server can export interfaces and objects in a single call to **RpcNsBindingExport**, or it can export them separately.

If the name-service database entry specified by the *EntryName* argument does not exist, the **RpcNsBindingExport** routine tries to create it. In this case, the server application must have the privilege to create the entry.

In addition to calling **RpcNsBindingExport**, a server that called the **RpcServerUseAllProtseqs** or **RpcServerUseProtseq** routine must also register with the local endpoint-map database by calling either the **RpcEpRegister** or **RpcEpRegisterNoReplace** routine.

A server is not required to export its interface(s) to the name-service database. When a server does not export, only clients that privately know of that server's binding information can access its interface(s). For example, a client that has the information needed to construct a string binding can call the **RpcBindingFromStringBinding** to create a binding handle for making remote procedure calls to a server.

Before calling the **RpcNsBindingExport** routine, a server must do the following:

- Register one or more protocol sequences with the local RPC run-time library by calling one of the following routines:
  - **RpcServerUseAllProtseqs**
  - **RpcServerUseProtseq**
  - **RpcServerUseAllProtseqsIf**
  - **RpcServerUseProtseqIf**
  - **RpcServerUseProtseqEp**
- Obtain a list of server bindings by calling the **RpcServerInqBindings** routine.

The vector returned from the **RpcServerInqBindings** routine becomes the *Binding* argument for **RpcNsBindingExport**. To prevent a binding from being exported, set the selected vector element to a null value.

If a server exports to the same name-service database entry multiple times, the second and subsequent calls to **RpcNsBindingExport** add the binding information and object UUIDs when that data is different from the binding information already in the server entry. Existing data is not removed from the entry.

To remove binding handles and object UUIDs from the name-service database, a server application calls the **RpcNsBindingUnexport** routine.

A server entry must have at least one binding handle to exist. As a result, exporting only UUIDs to a non-existing entry has no effect, and unexporting all binding handles deletes the entry.

## Return Values

| Value | Meaning |
|---|---|
| RPC_S_OK | Success |
| RPC_S_NOTHING_TO_EXPORT | Nothing to export |
| RPC_S_INVALID_BINDING | Invalid binding handle |
| RPC_S_WRONG_KIND_OF_BINDING | Wrong kind of binding for operation |

| | |
|---|---|
| RPC_S_INVALID_NAME_SYNTAX | Invalid name syntax |
| RPC_S_UNSUPPORTED_NAME_SYNTAX | Unsupported name syntax |
| RPC_S_INCOMPLETE_NAME | Incomplete name |
| RPC_S_NO_NS_PRIVILEGE | No privilege for name-service operation |
| RPC_S_NAME_SERVICE_UNAVAILABLE | Name service unavailable |

## See Also

**RpcBindingFromStringBinding**, **RpcEpRegister**, **RpcEpRegisterNoReplace**, **RpcNsBindingUnexport**, **RpcServerInqBindings**, **RpcServerUseAllProtseqs**, **RpcServerUseAllProtseqsIf**, **RpcServerUseProtseq**, **RpcServerUseProtseqEp**, **RpcServerUseProtseqIf**

# RpcNsBindingImportBegin Quick Info

The **RpcNsBindingImportBegin** function creates an import context for an interface and an object.

```
#include <rpc.h>

RPC_STATUS RPC_ENTRY RpcNsBindingImportBegin(

  unsigned long   EntryNameSyntax,
  unsigned char *   EntryName,
  RPC_IF_HANDLE   IfSpec,
  UUID *   ObjUuid,
  RPC_NS_HANDLE *   ImportContext
);
```

## Parameters

*EntryNameSyntax*

Specifies an unsigned long value that indicates the syntax of the next argument, *EntryName*.

To use the syntax specified in the registry value HKEY_LOCAL_MACHINE\Software\Microsoft\Rpc\NameService\
DefaultSyntax, provide a value of RPC_C_NS_SYNTAX_DEFAULT.

*EntryName*

Points to an entry name at which the search for compatible binding handles begins.

To use the entry name specified in the registry value HKEY_LOCAL_MACHINE\Software\Microsoft\Rpc\NameService\
DefaultEntry, provide a null pointer or an empty string. In this case, the *EntryNameSyntax* parameter is ignored and the run-time library uses the default syntax *EntryName*.

*IfSpec*

Specifies a stub-generated data structure indicating the interface to import. If the interface specification has not been exported or is of no concern to the caller, specify a null value for this argument. In this case, the bindings returned are only guaranteed to be of a compatible and supported protocol sequence and to contain the specified object UUID. The desired interface may not be supported by the contacted server.

*ObjUuid*

Points to an optional object UUID.

For a non-nil UUID, compatible binding handles are returned from an entry only if the server has exported the specified object UUID.

When the *ObjUuid* argument has a null pointer value or a nil UUID, the returned binding handles contain one of the object UUIDs exported by the compatible server. If the server did not export any object UUIDs, the returned compatible binding handles contain a nil object UUID.

*ImportContext*

Specifies a returned name-service handle for use with the **RpcNsBindingImportNext** and **RpcNsBindingImportDone** routines.

## Remarks

The **RpcNsBindingImportBegin** routine creates an import context for importing client-compatible binding handles for servers that offer the specified interface and object.

Before calling the **RpcNsBindingImportNext** routine, the client application must first call **RpcNsBindingImportBegin** to create an import context. The arguments to this routine control the operation of the **RpcNsBindingImportNext** routine.

When finished importing binding handles, the client application calls the **RpcNsBindingImportDone** routine to delete the import context.

## Return Values

| Value | Meaning |
|---|---|
| RPC_S_OK | Success |
| RPC_S_INVALID_NAME_SYNTAX | Invalid name syntax |
| RPC_S_UNSUPPORTED_NAME_SYNTAX | Unsupported name syntax |
| RPC_S_INCOMPLETE_NAME | Incomplete name |
| RPC_S_ENTRY_NOT_FOUND | Name-service entry not found |
| RPC_S_NAME_SERVICE_UNAVAILABLE | Name service unavailable |
| RPC_S_INVALID_OBJECT | Invalid object |

## See Also

**RpcNsBindingImportDone**, **RpcNsBindingImportNext**

# RpcNsBindingImportDone Quick Info

The **RpcNsBindingImportDone** function signifies that a client has finished looking for a compatible server and deletes the import context.

```
#include <rpc.h>

RPC_STATUS RPC_ENTRY RpcNsBindingImportDone(

  RPC_NS_HANDLE *    ImportContext
);
```

## Parameters

*ImportContext*

Points to a name-service handle to free. The name-service handle *ImportContext* points to is created by calling the **RpcNsBindingImportBegin** routine.

An argument value of NULL is returned.

## Remarks

The **RpcNsBindingImportDone** routine frees an import context created by calling the **RpcNsBindingImportBegin** routine.

Typically, a client application calls **RpcNsBindingImportDone** after completing remote procedure calls to a server using a binding handle returned from the **RpcNsBindingImportNext** routine. However, a client application is responsible for calling **RpcNsBindingImportDone** for each created import context regardless of the status returned from the **RpcNsBindingImportNext** routine or the success in making remote procedure calls.

## Return Values

| Value | Meaning |
|---|---|
| RPC_S_OK | Success |

## See Also

RpcNsBindingImportBegin, RpcNsBindingImportNext

# RpcNsBindingImportNext Quick Info

The **RpcNsBindingImportNext** function looks up an interface, and optionally an object, from a name-service database and returns a binding handle of a compatible server (if found).

```
#include <rpc.h>

RPC_STATUS RPC_ENTRY RpcNsBindingImportNext(

  RPC_NS_HANDLE   ImportContext,
  RPC_BINDING_HANDLE *   Binding
);
```

## Parameters

*ImportContext*

Specifies a name-service handle returned from the **RpcNsBindingImportBegin** routine.

*Binding*

Returns a pointer to a client-compatible server binding handle for a server.

## Remarks

The **RpcNsBindingImportNext** routine returns one client-compatible server binding handle for a server offering the interface and object UUID specified by the *IfSpec* and *ObjUuid* arguments in the **RpcNsBindingImportBegin** routine. The **RpcNsBindingImportNext** routine communicates only with the name-service database, not directly with servers.

The returned compatible binding handle always contains an object UUID. Its value depends on the *ObjUuid* argument value specified in the **RpcNsBindingImportBegin** routine as follows:

- If a non-nil object UUID was specified, the returned binding handle contains that object UUID.
- If a nil object UUID or null value was specified, the object UUID returned in the binding handle depends on how the server exported object UUIDs:
  - If the server did not export any object UUIDs, the returned binding handle contains a nil object UUID.
  - If the server exported one object UUID, the returned binding handle contains that object UUID.
  - If the server exported multiple object UUIDs, the returned binding handle contains one of the object UUIDs. The import-next operation selects the returned object UUID in a non-deterministic fashion. As a result, a different object UUID can be returned for each compatible binding handle from a single server entry.

The **RpcNsBindingImportNext** routine selects and returns one server binding handle from the compatible binding handles found. The client application can use that binding handle to attempt to make a remote procedure call to the server. If the client fails to establish a relationship with the server, it can call the **RpcNsBindingImportNext** routine again.

Each time the client calls the **RpcNsBindingImportNext** routine, the routine returns another server binding handle. The returned binding handles are unordered.

A client application calls the **RpcNsBindingInqEntryName** routine to obtain the name-service database in the entry name from which the binding handle came.

When the search reaches the end of the name-service database, the routine returns a status of RPC_S_NO_MORE_BINDINGS and returns a binding argument value of NULL.

The **RpcNsBindingImportNext** routine allocates storage for the data referenced by the returned *Binding* argument. When a client application finishes with the binding handle, it must call the **RpcBindingFree** routine to deallocate the storage. Each call to the **RpcNsBindingImportNext** routine requires a corresponding call to the **RpcBindingFree** routine.

The client is responsible for calling the **RpcNsBindingImportDone** routine. **RpcNsBindingImportDone** deletes the import context. The client also calls the **RpcNsBindingImportDone** routine if the application wants to start a new search for compatible servers (by calling the **RpcNsBindingImportBegin** routine). The order of binding handles returned is different for each new search. This means the order in which binding handles are returned to an application can be different each time the application is run.

## Return Values

| Value | Meaning |
|---|---|
| RPC_S_OK | Success |
| RPC_S_NO_MORE_BINDINGS | No more bindings |
| RPC_S_NAME_SERVICE_UNAVAILABLE | Name service unavailable |

## See Also

**RpcBindingFree**, **RpcNsBindingImportBegin**, **RpcNsBindingImportDone**, **RpcNsBindingInqEntryName**, **RpcNsBindingLookupBegin**, **RpcNsBindingLookupDone**, **RpcNsBindingLookupNext**, **RpcNsBindingSelect**

# RpcNsBindingInqEntryName Quick Info

The **RpcNsBindingInqEntryName** function returns the entry name from which the binding handle came.

```
#include <rpc.h>

RPC_STATUS RPC_ENTRY RpcNsBindingInqEntryName(

  RPC_BINDING_HANDLE   Binding,
  unsigned long   EntryNameSyntax,
  unsigned char * *   EntryName
);
```

## Parameters

*Binding*

Specifies the binding handle whose name-service database entry name is returned.

*EntryNameSyntax*

Specifies an unsigned long value that indicates the syntax used in the returned argument, *EntryName*.

To use the syntax specified in the registry value HKEY_LOCAL_MACHINE\Software\Microsoft\Rpc\NameService\
DefaultSyntax, provide a value of RPC_C_NS_SYNTAX_DEFAULT.

*EntryName*

Returns a pointer to a pointer to the name of the name-service database entry in which *Binding* was found.

Specify a null value to prevent **RpcNsBindingInqEntryName** from returning the *EntryName* argument. In this case, the application does not call the **RpcStringFree** routine.

## Remarks

The **RpcNsBindingInqEntryName** routine returns the name of the name-service database entry from which a client-compatible binding handle came.

The RPC run-time library allocates memory for the string returned in the *EntryName* argument. The application is responsible for calling the **RpcStringFree** routine to deallocate that memory.

An entry name is associated only with binding handles returned from the **RpcNsBindingImportNext**, **RpcNsBindingLookupNext**, and **RpcNsBindingSelect** routines.

If the binding handle specified in the *Binding* argument was not returned from a name-service database entry (for example, if the binding handle was created by calling **RpcBindingFromStringBinding**), **RpcNsBindingInqEntryName** returns an empty string ("\0") and an RPC_S_NO_ENTRY_NAME status code.

## Return Values

| Value | Meaning |
|---|---|
| RPC_S_OK | Success |
| RPC_S_INVALID_BINDING | Invalid binding handle |
| RPC_S_NO_ENTRY_NAME | No entry name for binding |
| RPC_S_INVALID_NAME_SYNTAX | Invalid name syntax |

| | |
|---|---|
| RPC_S_UNSUPPORTED_NAME_SYNTAX | Unsupported name syntax |
| RPC_S_INCOMPLETE_NAME | Incomplete name |

## See Also

**RpcBindingFromStringBinding**, **RpcNsBindingImportNext**, **RpcNsBindingLookupNext**, **RpcNsBindingSelect**, **RpcStringFree**

# RpcNsBindingLookupBegin Quick Info

The **RpcNsBindingLookupBegin** function creates a lookup context for an interface and an object.

```
#include <rpc.h>

RPC_STATUS RPC_ENTRY RpcNsBindingLookupBegin(

  unsigned long   EntryNameSyntax,
  unsigned char *   EntryName,
  RPC_IF_HANDLE   IfSpec,
  UUID *   ObjUuid,
  unsigned long   BindingMaxCount,
  RPC_NS_HANDLE *   LookupContext
);
```

## Parameters

*EntryNameSyntax*

Specifies an unsigned long value that indicates the syntax of the next argument, *EntryName*.

To use the syntax specified in the registry value HKEY_LOCAL_MACHINE\Software\Microsoft\Rpc\NameService\
DefaultSyntax, provide a value of RPC_C_NS_SYNTAX_DEFAULT.

*EntryName*

Points to an entry name at which the search for compatible bindings begins.

To use the entry name specified in the registry value HKEY_LOCAL_MACHINE\Software\Microsoft\Rpc\NameService\
DefaultEntry, provide a null pointer or an empty string. In this case, the *EntryNameSyntax* parameter is ignored and the run-time library uses the default syntax *EntryName*.

*IfSpec*

Specifies a stub-generated data structure indicating the interface to look up. If the interface specification has not been exported or is of no concern to the caller, specify a null value for this argument. In this case, the bindings returned are only guaranteed to be of a compatible and supported protocol sequence and to contain the specified object UUID. The desired interface may not be supported by the contacted server.

*ObjUuid*

Points to an optional object UUID.

For a non-nil UUID, compatible binding handles are returned from an entry only if the server has exported the specified object UUID.

For a null pointer value or a nil UUID for this argument, the returned binding handles contain one of the object UUIDs exported by the compatible server. If the server did not export any object UUIDs, the returned compatible binding handles contain a nil object UUID.

*BindingMaxCount*

Specifies the maximum number of bindings to return in the *BindingVec* argument from the **RpcNsBindingLookupNext** routine.

Specify a value of zero to use the default count of RPC_C_BINDING_MAX_COUNT_DEFAULT.

*LookupContext*

Returns a pointer to a name-service handle for use with the **RpcNsBindingLookupNext** and

**RpcNsBindingLookupDone** routines.

## Remarks

The **RpcNsBindingLookupBegin** routine creates a lookup context for locating client-compatible binding handles to servers that offer the specified interface and object.

Before calling the **RpcNsBindingLookupNext** routine, the client application must first call **RpcNsBindingLookupBegin** to create a lookup context. The arguments to this routine control the operation of the **RpcNsBindingLookupNext** routine.

When finished locating binding handles, the client application calls the **RpcNsBindingLookupDone** routine to delete the lookup context.

## Return Values

| Value | Meaning |
|---|---|
| RPC_S_OK | Success |
| RPC_S_INVALID_NAME_SYNTAX | Invalid name syntax |
| RPC_S_UNSUPPORTED_NAME_SYNTAX | Unsupported name syntax |
| RPC_S_INCOMPLETE_NAME | Incomplete name |
| RPC_S_ENTRY_NOT_FOUND | Name-service entry not found |
| RPC_S_NAME_SERVICE_UNAVAILABLE | Name service unavailable |
| RPC_S_INVALID_OBJECT | Invalid object |

## See Also

RpcNsBindingLookupDone, RpcNsBindingLookupNext

# RpcNsBindingLookupDone Quick Info

The **RpcNsBindingLookupDone** function signifies that a client has finished looking for compatible servers and deletes the lookup context.

```
#include <rpc.h>

RPC_STATUS RPC_ENTRY RpcNsBindingLookupDone(

  RPC_NS_HANDLE *   LookupContext
);
```

## Parameters

*LookupContext*

Points to the name-service handle to free. The name-service handle *LookupContext* points to is created by calling the routine **RpcNsBindingLookupBegin**.

An argument value of NULL is returned.

## Remarks

The **RpcNsBindingLookupDone** routine frees a lookup context created by calling the **RpcNsBindingLookupBegin** routine.

Typically, a client application calls **RpcNsBindingLookupDone** after completing remote procedure calls to a server using a binding handle returned from the **RpcNsBindingLookupNext** routine. However, a client application is responsible for calling **RpcNsBindingLookupDone** for each created lookup context, regardless of the status returned from the **RpcNsBindingLookupNext** routine or the success in making remote procedure calls.

## Return Values

| Value | Meaning |
|---|---|
| RPC_S_OK | Success |

## See Also

RpcNsBindingLookupBegin, RpcNsBindingLookupNext

# RpcNsBindingLookupNext Quick Info

The **RpcNsBindingLookupNext** function returns a list of compatible binding handles for a specified interface and optionally an object.

```
#include <rpc.h>

RPC_STATUS RPC_ENTRY RpcNsBindingLookupNext(

  RPC_NS_HANDLE   LookupContext,
  RPC_BINDING_VECTOR * *   BindingVec
);
```

## Parameters

*LookupContext*

Specifies the name-service handle returned from the **RpcNsBindingLookupBegin** routine.

*BindingVec*

Returns a pointer to a pointer to a vector of client-compatible server binding handles.

## Remarks

The **RpcNsBindingLookupNext** routine returns a vector of client-compatible server binding handles for a server offering the interface and object UUID specified by the *IfSpec* and *ObjUuid* arguments in the **RpcNsBindingLookupBegin** routine.

The **RpcNsBindingLookupNext** routine communicates only with the name-service database, not directly with servers.

The **RpcNsBindingLookupNext** routine traverses name-service database entries collecting client-compatible server binding handles from each entry. If the entry at which the search begins (see the *EntryName* argument in **RpcNsBindingLookupBegin**) contains binding handles as well as an RPC group and/or a profile, **RpcNsBindingLookupNext** returns the binding handles from *EntryName* before searching the group or profile. This means that **RpcNsBindingLookupNext** can return a partially full vector before processing the members of the group or profile. Each binding handle in the returned vector always contains an object UUID. Its value depends on the *ObjUuid* argument value specified in the **RpcNsBindingLookupBegin** routine as follows:

- If a non-nil object UUID was specified, each returned binding handle contains that object UUID.
- If a nil object UUID or null value was specified, the object UUID returned in each binding handle depends on how the server exported object UUIDs:
  - If the server did not export any object UUIDs, each returned binding handle contains a nil object UUID.
  - If the server exported one object UUID, each returned binding handle contains that object UUID.
  - If the server exported multiple object UUIDs, each binding handle contains one of the object UUIDs. The lookup-next operation selects the returned object UUID in a non-deterministic fashion. For this reason, a different object UUID can be returned for each compatible binding handle from a single server entry.

From the returned vector of server binding handles, the client application can employ its own criteria for selecting individual binding handles, or the application can call the **RpcNsBindingSelect** routine to select a binding handle. The **RpcBindingToStringBinding** and **RpcStringBindingParse** routines will be helpful

for a client creating its own selection criteria.

The client application can use the selected binding handle to attempt to make a remote procedure call to the server. If the client fails to establish a relationship with the server, it can select another binding handle from the vector. When all of the binding handles in the vector have been used, the client application calls the **RpcNsBindingLookupNext** routine again.

Each time the client calls the **RpcNsBindingLookupNext** routine, the routine returns another vector of binding handles. The binding handles returned in each vector are unordered. The vectors returned from multiple calls to this routine are also unordered.

A client calls the **RpcNsBindingInqEntryName** routine to obtain the name-service database server entry name that the binding came from.

When the search reaches the end of the name-service database, **RpcNsBindingLookupNext** returns a status of RPC_S_NO_MORE_BINDINGS and returns a *BindingVec* argument value of NULL.

The **RpcNsBindingLookupNext** routine allocates storage for the data referenced by the returned *BindingVec* argument. When a client application finishes with the vector, it must call the **RpcBindingVectorFree** routine to deallocate the storage. Each call to the **RpcNsBindingLookupNext** routine requires a corresponding call to the **RpcBindingVectorFree** routine.

The client is responsible for calling the **RpcNsBindingLookupDone** routine. **RpcNsBindingLookupDone** deletes the lookup context. The client also calls the **RpcNsBindingLookupDone** routine if the application wants to start a new search for compatible servers (by calling the **RpcNsBindingLookupBegin** routine). The order of binding handles returned can be different for each new search.

## Return Values

| Value | Meaning |
|---|---|
| RPC_S_OK | Success |
| RPC_S_NO_MORE_BINDINGS | No more bindings |
| RPC_S_NAME_SERVICE_UNAVAILABLE | Name-service unavailable |

## See Also

**RpcBindingToStringBinding**, **RpcBindingVectorFree**, **RpcNsBindingInqEntryName**, **RpcNsBindingLookupBegin**, **RpcNsBindingLookupDone**, **RpcStringBindingParse**

# RpcNsBindingSelect Quick Info

The **RpcNsBindingSelect** function returns a binding handle from a list of compatible binding handles.

```
#include <rpc.h>

RPC_STATUS RPC_ENTRY RpcNsBindingSelect(

  RPC_BINDING_VECTOR *   BindingVec,
  RPC_BINDING_HANDLE *   Binding
);
```

## Parameters

*BindingVec*

Points to the vector of client-compatible server binding handles from which a binding handle is selected. The returned binding vector no longer references the selected binding handle, which is returned separately in the *Binding* argument.

*Binding*

Returns a pointer to a selected binding handle.

## Remarks

The **RpcNsBindingSelect** routine chooses and returns a client-compatible server binding handle from a vector of server binding handles.

Each time the client calls the **RpcNsBindingSelect** routine, the routine operation returns another binding handle from the vector.

When all of the binding handles have been returned from the vector, the routine returns a status of RPC_S_NO_MORE_BINDINGS and returns a *Binding* argument value of NULL.

The select operation allocates storage for the data referenced by the returned *Binding* argument. When a client finishes with the binding handle, it should call the **RpcBindingFree** routine to deallocate the storage. Each call to the **RpcNsBindingSelect** routine requires a corresponding call to the **RpcBindingFree** routine.

Clients can create their own select routines implementing application-specific selection criteria. In this case, the **RpcStringBindingParse** routine provides access to the fields of a binding.

## Return Values

| Value | Meaning |
|---|---|
| RPC_S_OK | Success |
| RPC_S_NO_MORE_BINDINGS | No more bindings |

## See Also

RpcBindingFree, RpcNsBindingLookupNext, RpcStringBindingParse

# RpcNsBindingUnexport Quick Info

The **RpcNsBindingUnexport** function removes the binding handles for an interface and objects from an entry in the name-service database.

This function is supported by both 32-bit platforms – Windows NT and Windows 95.

```
#include <rpc.h>

RPC_STATUS RPC_ENTRY RpcNsBindingUnexport(

  unsigned long   EntryNameSyntax,
  unsigned char *   EntryName,
  RPC_IF_HANDLE   IfSpec,
  UUID_VECTOR *   ObjectUuidVec
);
```

## Parameters

*EntryNameSyntax*

Specifies an unsigned long value that indicates the syntax of the next argument, *EntryName*.

To use the syntax specified in the registry value HKEY_LOCAL_MACHINE\Software\Microsoft\Rpc\NameService\
DefaultSyntax, provide a value of RPC_C_NS_SYNTAX_DEFAULT.

*EntryName*

Points to the entry name from which to remove binding handles and object UUIDs.

*IfSpec*

Specifies an interface. A null argument value indicates not to unexport any binding handles (only object UUIDs are to be unexported).

*ObjectUuidVec*

Points to a vector of object UUIDs that the server no longer wants to offer. The application constructs this vector. A null argument value indicates there are no object UUIDs to unexport (only binding handles are to be unexported).

## Remarks

The **RpcNsBindingUnexport** routine allows a server application to remove the following from a name-service database entry:

- All the binding handles for a specific interface
- One or more object UUIDs of resources
- Both the binding handles and object UUIDs of resources

The **RpcNsBindingUnexport** routine unexports only the binding handles that match the interface UUID and the major and minor interface version numbers found in the *IfSpec* argument.

A server application can unexport the specified interface and objects in a single call to **RpcNsBindingUnexport**, or it can unexport them separately.

If **RpcNsBindingUnexport** does not find any binding handles for the specified interface, the routine

returns an RPC_S_INTERFACE_NOT_FOUND status code and does not unexport the object UUIDs, if any were specified.

If one or more binding handles for the specified interface are found and unexported without error, **RpcNsBindingUnexport** unexports the specified object UUIDs, if any.

If any of the specified object UUIDs were not found, **RpcNsBindingUnexport** returns the RPC_S_NOT_ALL_OBJS_UNEXPORTED status code.

In addition to calling **RpcNsBindingUnexport**, a server should also call the **RpcEpUnregister** routine to unregister the endpoints the server previously registered with the local endpoint-map database.

A server entry must have at least one binding handle to exist. As a result, exporting only UUIDs to a non-existing entry has no effect, and unexporting all binding handles deletes the entry.

Use **RpcNsBindingUnexport** judiciously. To keep an automatically activated server available, you must leave its binding handles in the name-service database between the times when server processes are activated. Therefore, reserve this routine for when you expect a server to be unavailable for an extended time – for example, when it is being permanently removed from service.

**Note** Name-service databases are designed to be relatively stable. In replicated name-service databases, frequent use of the **RpcNsBindingExport** and **RpcNsBindingUnexport** routines causes the name-service database to repeatedly remove and replace the same entry and can cause performance problems.

## Return Values

| Value | Meaning |
|---|---|
| RPC_S_OK | Success |
| RPC_S_INVALID_VERS_OPTION | Invalid version option |
| RPC_S_NOTHING_TO_UNEXPORT | Nothing to unexport |
| RPC_S_INVALID_NAME_SYNTAX | Invalid name syntax |
| RPC_S_UNSUPPORTED_NAME_SYNTAX | Unsupported name syntax |
| RPC_S_INCOMPLETE_NAME | Incomplete name |
| RPC_S_ENTRY_NOT_FOUND | Name-service entry not found |
| RPC_S_NAME_SERVICE_UNAVAILABLE | Name service unavailable |
| RPC_S_INTERFACE_NOT_FOUND | Interface not found |
| RPC_S_NOT_ALL_OBJS_UNEXPORTED | Not all objects unexported |

## See Also

RpcEpUnregister, RpcNsBindingExport

# RpcNsEntryExpandName Quick Info

The **RpcNsEntryExpandName** function expands a name-service entry name.

```
#include <rpc.h>

RPC_STATUS RPC_ENTRY RpcNsEntryExpandName(

  unsigned long   EntryNameSyntax,
  unsigned char *   EntryName,
  unsigned char * *   ExpandedName
);
```

## Parameters

*EntryNameSyntax*

Specifies an integer value that indicates the syntax of the next argument, *EntryName*.

To use the syntax specified in the registry value HKEY_LOCAL_MACHINE\Software\Microsoft\Rpc\NameService\
DefaultSyntax, provide a value of RPC_C_NS_SYNTAX_DEFAULT.

*EntryName*

Points to the entry name to expand.

*ExpandedName*

Returns a pointer to a pointer to the expanded version of *EntryName*.

## Remarks

**Note** This DCE function is not supported by the Microsoft Locator version 1.0.

An application calls the **RpcNsEntryExpandName** routine to obtain a fully expanded entry name.

The RPC run-time library allocates memory for the returned *ExpandedName* argument. The application is responsible for calling the **RpcStringFree** routine for that returned argument string.

The returned expanded entry name accounts for local name translations and for differences in locally defined naming schemas.

## Return Values

| Value | Meaning |
|---|---|
| RPC_S_OK | Success |
| RPC_S_INCOMPLETE_NAME | Incomplete name |

## See Also

RpcStringFree

# RpcNsEntryObjectInqBegin Quick Info

The **RpcNsEntryObjectInqBegin** function creates an inquiry context for a name-service database entry's objects.

```
#include <rpc.h>

RPC_STATUS RPC_ENTRY RpcNsEntryObjectInqBegin(

  unsigned long   EntryNameSyntax,
  unsigned char *   EntryName,
  RPC_NS_HANDLE *   InquiryContext
);
```

## Parameters

*EntryNameSyntax*

Specifies an integer value that indicates the syntax to use in the next argument, *EntryName*.

To use the syntax specified in the registry value HKEY_LOCAL_MACHINE\Software\Microsoft\Rpc\NameService\
DefaultSyntax, provide a value of RPC_C_NS_SYNTAX_DEFAULT.

*EntryName*

Points to the name-service database entry name for which object UUIDs are to be viewed.

*InquiryContext*

Returns a pointer to a name-service handle for use with the **RpcNsEntryObjectInqNext** and **RpcNsEntryObjectInqDone** routines.

## Remarks

The **RpcNsEntryObjectInqBegin** routine creates an inquiry context for viewing the object UUIDs exported to *EntryName*.

Before calling the **RpcNsEntryObjectInqNext** routine, the application must first call **RpcNsEntryObjectInqBegin** to create an inquiry context.

When finished viewing the object UUIDs, the application calls the **RpcNsEntryObjectInqDone** routine to delete the inquiry context.

## Return Values

| Value | Meaning |
|---|---|
| RPC_S_OK | Success |
| RPC_S_INVALID_NAME_SYNTAX | Invalid name syntax |
| RPC_S_UNSUPPORTED_NAME_SYNTAX | Unsupported name syntax |
| RPC_S_INCOMPLETE_NAME | Incomplete name |
| RPC_S_ENTRY_NOT_FOUND | Name-service entry not found |
| RPC_S_NAME_SERVICE_UNAVAILABLE | Name service unavailable |

## See Also

RpcNsBindingExport, RpcNsEntryObjectInqDone, RpcNsEntryObjectInqNext

# RpcNsEntryObjectInqDone Quick Info

The **RpcNsEntryObjectInqDone** function deletes the inquiry context for a name-service database entry's objects.

```
#include <rpc.h>

RPC_STATUS RPC_ENTRY RpcNsEntryObjectInqDone(

  RPC_NS_HANDLE *   InquiryContext
    );
```

## Parameters

*InquiryContext*

Points to a name-service handle specifying the object UUIDs exported to the *EntryName* argument specified in the **RpcNsEntryObjectInqBegin** routine.

An argument value of NULL is returned.

## Remarks

The **RpcNsEntryObjectInqDone** routine frees an inquiry context created by calling the **RpcNsEntryObjectInqBegin** routine.

An application calls **RpcNsEntryObjectInqDone** after viewing exported object UUIDs using the **RpcNsEntryObjectInqNext** routine.

## Return Values

| Value | Meaning |
|---|---|
| RPC_S_OK | Success |

## See Also

RpcNsEntryObjectInqBegin, RpcNsEntryObjectInqNext

# RpcNsEntryObjectInqNext Quick Info

The **RpcNsEntryObjectInqNext** function returns one object at a time from a name-service database entry.

```
#include <rpc.h>

RPC_STATUS RPC_ENTRY RpcNsEntryObjectInqNext(

  RPC_NS_HANDLE   InquiryContext,
  UUID *   ObjUuid
);
```

## Parameters

*InquiryContext*

Specifies a name-service handle that indicates the object UUIDs for a name-service database entry.

*ObjUuid*

Returns a pointer to an exported object UUID.

## Remarks

The **RpcNsEntryObjectInqNext** routine returns one of the object UUIDs exported to the name-service database entry specified by the *EntryName* argument in the **RpcNsEntryObjectInqBegin** routine.

An application can view all of the exported object UUIDs by repeatedly calling the **RpcNsEntryObjectInqNext** routine. When all the object UUIDs have been viewed, this routine returns an RPC_S_NO_MORE_MEMBERS status code. The returned object UUIDs are unordered.

The application supplies the memory for the object UUID returned in the *ObjUuid* argument.

After viewing the object UUIDs, the application must call the **RpcNsEntryObjectInqDone** routine to release the inquiry context.

The order in which object UUIDs are returned can be different for each viewing of an entry. This means that the order in which object UUIDs are returned to an application can be different each time the application is run.

## Return Values

| Value | Meaning |
|---|---|
| RPC_S_OK | Success |
| RPC_S_NO_MORE_MEMBERS | No more members |
| RPC_S_INCOMPLETE_NAME | Incomplete name |
| RPC_S_ENTRY_NOT_FOUND | Name-service entry not found |
| RPC_S_NAME_SERVICE_UNAVAILABLE | Name-service unavailable |

## See Also

RpcNsBindingExport, RpcNsEntryObjectInqBegin, RpcNsEntryObjectInqDone

# RpcNsGroupDelete Quick Info

The **RpcNsGroupDelete** function deletes a group attribute.

```
#include <rpc.h>

RPC_STATUS RPC_ENTRY RpcNsGroupDelete(

  unsigned long   GroupNameSyntax,
  unsigned char *   GroupName
);
```

## Parameters

*GroupNameSyntax*

Specifies an integer value that indicates the syntax of the next parameter, *GroupName*.

To use the syntax specified in the registry value HKEY_LOCAL_MACHINE\Software\Microsoft\Rpc\NameService\
DefaultSyntax, provide a value of RPC_C_NS_SYNTAX_DEFAULT.

*GroupName*

Points to the name of the RPC group to delete.

## Remarks

**Note** This DCE function is not supported by the Microsoft Locator.

The **RpcNsGroupDelete** routine deletes the group attribute from the specified name-service database entry.

Neither the specified name-service database entry nor the group members are deleted.

## Return Values

| Value | Meaning |
|---|---|
| RPC_S_OK | Success |
| RPC_S_INVALID_NAME_SYNTAX | Invalid name syntax |
| RPC_S_UNSUPPORTED_NAME_SYNTAX | Unsupported name syntax |
| RPC_S_INCOMPLETE_NAME | Incomplete name |
| RPC_S_ENTRY_NOT_FOUND | Name-service entry not found |
| RPC_S_NAME_SERVICE_UNAVAILABLE | Name service unavailable |

## See Also

RpcNsGroupMbrAdd, RpcNsGroupMbrRemove

# RpcNsGroupMbrAdd Quick Info

The **RpcNsGroupMbrAdd** function adds an entry name to a group. If necessary, it creates the entry.

```
#include <rpc.h>

RPC_STATUS RPC_ENTRY RpcNsGroupMbrAdd(

  unsigned long   GroupNameSyntax,
  unsigned char *   GroupName,
  unsigned long   MemberNameSyntax,
  unsigned char *   MemberName
);
```

## Parameters

*GroupNameSyntax*

Specifies an integer value that indicates the syntax of the next argument, *GroupName*.

To use the syntax specified in the registry value HKEY_LOCAL_MACHINE\Software\Microsoft\Rpc\NameService\
DefaultSyntax, provide a value of RPC_C_NS_SYNTAX_DEFAULT.

*GroupName*

Points to the name of the RPC group to receive a new member.

*MemberNameSyntax*

Specifies an integer value that indicates the syntax to use in the *MemberName* argument.

To use the syntax specified in the registry value HKEY_LOCAL_MACHINE\Software\Microsoft\Rpc\NameService\
DefaultSyntax, provide a value of RPC_C_NS_SYNTAX_DEFAULT.

*MemberName*

Points to the name of the new RPC group member.

## Remarks

**Note** This DCE function is not supported by the Microsoft Locator.

The **RpcNsGroupMbrAdd** adds a name-service database entry name as a member to the RPC group attribute.

If the *GroupName* entry does not exist, **RpcNsGroupMbrAdd** tries to create the entry with a group attribute and adds the group member specified by the *MemberName* argument. In this case, the application must have the privilege to create the entry. Otherwise, a management application with the necessary privilege should create the entry by calling the **RpcNsMgmtEntryCreate** routine before the application is run.

## Return Values

| Value | Meaning |
|---|---|
| RPC_S_OK | Success |
| RPC_S_INVALID_NAME_SYNTAX | Invalid name syntax |

| | |
|---|---|
| RPC_S_UNSUPPORTED_NAME_SYNTAX | Unsupported name syntax |
| RPC_S_INCOMPLETE_NAME | Incomplete name |
| RPC_S_NAME_SERVICE_UNAVAILABLE | Name service unavailable |

## See Also

RpcNsGroupMbrRemove, RpcNsMgmtEntryCreate

# RpcNsGroupMbrInqBegin Quick Info

The **RpcNsGroupMbrInqBegin** function creates an inquiry context for viewing group members.

```
#include <rpc.h>

RPC_STATUS RPC_ENTRY RpcNsGroupMbrInqBegin(

  unsigned long   GroupNameSyntax,
  unsigned char *   GroupName,
  unsigned long   MemberNameSyntax,
  RPC_NS_HANDLE *   InquiryContext
);
```

## Parameters

*GroupNameSyntax*

Specifies an integer value that indicates the syntax of the next argument, *GroupName*.

To use the syntax specified in the registry value HKEY_LOCAL_MACHINE\Software\Microsoft\Rpc\NameService\
DefaultSyntax, provide a value of RPC_C_NS_SYNTAX_DEFAULT.

*GroupName*

Points to the name of the RPC group to view.

*MemberNameSyntax*

Specifies an integer value that indicates the syntax of the return argument, *MemberName*, in the **RpcNsGroupMbrInqNext** routine.

To use the syntax specified in the registry value HKEY_LOCAL_MACHINE\Software\Microsoft\Rpc\NameService\
DefaultSyntax, provide a value of RPC_C_NS_SYNTAX_DEFAULT.

*InquiryContext*

Returns a pointer to a name-service handle for use with the **RpcNsGroupMbrInqNext** and **RpcNsGroupMbrInqDone** routines.

## Remarks

**Note** This DCE function is not supported by the Microsoft Locator.

The **RpcNsGroupMbrInqBegin** routine creates an inquiry context for viewing the members of an RPC group.

Before calling the **RpcNsGroupMbrInqNext** routine, the application must first call **RpcNsGroupMbrInqBegin** to create an inquiry context.

When finished viewing the RPC group members, the application calls the **RpcNsGroupMbrInqDone** routine to delete the inquiry context.

## Return Values

| Value | Meaning |
|---|---|

| | |
|---|---|
| RPC_S_OK | Success |
| RPC_S_INVALID_NAME_SYNTAX | Invalid name syntax |
| RPC_S_UNSUPPORTED_NAME_SYNTAX | Unsupported name syntax |
| RPC_S_INCOMPLETE_NAME | Incomplete name |
| RPC_S_ENTRY_NOT_FOUND | Name-service entry not found |
| RPC_S_NAME_SERVICE_UNAVAILABLE | Name service unavailable |

## See Also

RpcNsGroupMbrAdd, RpcNsGroupMbrInqDone, RpcNsGroupMbrInqNext

# RpcNsGroupMbrInqDone Quick Info

The **RpcNsGroupMbrInqDone** function deletes the inquiry context for a group.

```
#include <rpc.h>

RPC_STATUS RPC_ENTRY RpcNsGroupMbrInqDone(

  RPC_NS_HANDLE *   InquiryContext
  );
```

## Parameters

*InquiryContext*

Points to a name-service handle to free. An argument value of NULL is returned.

## Remarks

**Note** This DCE function is not supported by the Microsoft Locator.

The **RpcNsGroupMbrInqDone** routine frees an inquiry context created by calling the **RpcNsGroupMbrInqBegin** routine.

An application calls **RpcNsGroupMbrInqDone** after viewing RPC group members using the **RpcNsGroupMbrInqNext** routine.

## Return Values

| Value | Meaning |
|---|---|
| RPC_S_OK | Success |
| RPC_S_INVALID_NS_HANDLE | Invalid name-service handle |

## See Also

RpcNsGroupMbrInqBegin, RpcNsGroupMbrInqNext

# RpcNsGroupMbrInqNext Quick Info

The **RpcNsGroupMbrInqNext** function returns one entry name from a group at a time.

```
#include <rpc.h>

RPC_STATUS RPC_ENTRY RpcNsGroupMbrInqNext(

  RPC_NS_HANDLE   InquiryContext,
  unsigned char * *   MemberName
);
```

## Parameters

*InquiryContext*

Specifies a name-service handle.

*MemberName*

Returns a pointer to a pointer to an RPC group member name.

The syntax of the returned name was specified by the *MemberNameSyntax* argument in the **RpcNsGroupMbrInqBegin** routine.

Specify a null value to prevent **RpcNsGroupMbrInqNext** from returning the *MemberName* argument. In this case, the application does not call the **RpcStringFree** routine.

## Remarks

**Note** This DCE function is not supported by the Microsoft Locator.

The **RpcNsGroupMbrInqNext** routine returns one member of the RPC group specified by the *GroupName* argument in the **RpcNsGroupMbrInqBegin** routine.

An application can view all the members of an RPC group set by repeatedly calling the **RpcNsGroupMbrInqNext** routine. When all the group members have been viewed, this routine returns an RPC_S_NO_MORE_MEMBERS status code. The returned group members are unordered.

On each call to **RpcNsGroupMbrInqNext** that returns a member name, the RPC run-time library allocates memory for the returned *MemberName*. The application is responsible for calling the **RpcStringFree** routine for each returned *MemberName* string.

After viewing the RPC group's members, the application must call the **RpcNsGroupMbrInqDone** routine to release the inquiry context.

The order in which group members are returned can be different for each viewing of a group. This means that the order in which group members are returned to an application can be different each time the application is run.

## Return Values

| Value | Meaning |
|---|---|
| RPC_S_OK | Success |
| RPC_S_INVALID_NS_HANDLE | Invalid name-service handle |

| | |
|---|---|
| RPC_S_NO_MORE_MEMBERS | No more members |
| RPC_S_NAME_SERVICE_UNAVAILABLE | Name service unavailable |

## See Also

RpcNsGroupMbrInqBegin, RpcNsGroupMbrInqDone, RpcStringFree

# RpcNsGroupMbrRemove Quick Info

The **RpcNsGroupMbrRemove** function removes an entry name from a group.

```
#include <rpc.h>

RPC_STATUS RPC_ENTRY RpcNsGroupMbrRemove(

  unsigned long   GroupNameSyntax,
  unsigned char *   GroupName,
  unsigned long   MemberNameSyntax,
  unsigned char *   MemberName
);
```

## Parameters

*GroupNameSyntax*

Specifies an integer value that indicates the syntax of the next argument, *GroupName*.

To use the syntax specified in the registry value HKEY_LOCAL_MACHINE\Software\Microsoft\Rpc\NameService\
DefaultSyntax, provide a value of RPC_C_NS_SYNTAX_DEFAULT.

*GroupName*

Points to the name of the RPC group from which to remove the member name.

*MemberNameSyntax*

Specifies an integer value that indicates the syntax to use in the *MemberName* argument.

To use the syntax specified in the registry value HKEY_LOCAL_MACHINE\Software\Microsoft\Rpc\NameService\
DefaultSyntax, provide a value of RPC_C_NS_SYNTAX_DEFAULT.

*MemberName*

Points to the name of the member to remove from the RPC group attribute in the entry *GroupName*.

## Remarks

**Note** This DCE function is not supported by the Microsoft Locator.

The **RpcNsGroupMbrRemove** routine removes a member from the RPC group attribute in the *GroupName* argument.

## Return Values

| Value | Meaning |
|---|---|
| RPC_S_OK | Success |
| RPC_S_INVALID_NAME_SYNTAX | Invalid name syntax |
| RPC_S_UNSUPPORTED_NAME_SYNTAX | Unsupported name syntax |
| RPC_S_INCOMPLETE_NAME | Incomplete name |
| RPC_S_ENTRY_NOT_FOUND | Name-service entry not found |

| | |
|---|---|
| RPC_S_NAME_SERVICE_UNAVAILABLE | Name service unavailable |
| RPC_S_GROUP_MEMBER_NOT_FOUND | Group member not found |

## See Also

RpcNsGroupMbrAdd

# RpcNsMgmtBindingUnexport Quick Info

The **RpcNsMgmtBindingUnexport** function removes multiple binding handles and objects from an entry in the name-service database.

```
#include <rpc.h>

RPC_STATUS RPC_ENTRY RpcNsMgmtBindingUnexport(

  unsigned long   EntryNameSyntax,
  unsigned char *   EntryName,
  RPC_IF_ID *   IfId,
  unsigned long   VersOption,
  UUID_VECTOR *   ObjectUuidVec
);
```

## Parameters

*EntryNameSyntax*

Specifies an integer value that indicates the syntax of the next argument, *EntryName*.

To use the syntax specified in the registry value HKEY_LOCAL_MACHINE\Software\Microsoft\Rpc\NameService\
DefaultSyntax, provide a value of RPC_C_NS_SYNTAX_DEFAULT.

*EntryName*

Points to the name of the entry from which to remove binding handles and object UUIDs.

*IfId*

Points to an interface identification. A null argument value indicates not to unexport any binding handles (only object UUIDs are to be unexported).

*VersOption*

Specifies how the **RpcNsMgmtBindingUnexport** routine uses the *VersMajor* and *VersMinor* fields of the *IfId* argument.

The following table describes valid values for the *VersOption* argument:

| ***VersOption* values** | **Description** |
|---|---|
| RPC_C_VERS_ALL | Unexports all bindings for the interface UUID in *IfId*, regardless of the version numbers. For this value, specify 0 for both the major and minor versions in *IfId*. |
| RPC_C_VERS_IF_ID | Unexports the bindings for the compatible interface UUID in *IfId* with the same major version and with a minor version greater than or equal to the minor version in *IfId*. |
| RPC_C_VERS_EXACT | Unexports the bindings for the interface UUID in *IfId* with the same major and minor versions as in *IfId*. |
| RPC_C_VERS_MAJOR_ONLY | Unexports the bindings for the interface UUID in *IfId* with the |

| | |
|---|---|
| | same major version as in *IfId* (ignores the minor version). For this value, specify 0 for the minor version in *IfId*. |
| RPC_C_VERS_UPTO | Unexports the bindings that offer a version of the specified interface UUID less than or equal to the specified major and minor version. (For example, if the *IfId* contained V2.0 and the name-service database entry contained binding handles with the versions V1.3, V2.0, and V2.1, the **RpcNsMgmtBindingUnexport** routine unexports the binding handles with V1.3 and V2.0.) |

*ObjectUuidVec*

Points to a vector of object UUIDs that the server no longer wants to offer. The application constructs this vector. A null argument value indicates there are no object UUIDs to unexport (only binding handles are to be unexported).

## Remarks

The **RpcNsMgmtBindingUnexport** routine allows a management application to remove one of the following from a name-service database entry:

- All the binding handles for a specified interface UUID, qualified by the interface version numbers (major and minor)
- One or more object UUIDs of resources
- Both binding handles and object UUIDs of resources

A management application can unexport interfaces and objects in a single call to **RpcNsMgmtBindingUnexport**, or it can unexport them separately.

If **RpcNsMgmtBindingUnexport** does not find any binding handles for the specified interface, the routine returns an RPC_S_INTERFACE_NOT_FOUND status code and does not unexport the object UUIDs, if any were specified.

If one or more binding handles for the specified interface are found and unexported without error, **RpcNsMgmtBindingUnexport** unexports the specified object UUIDs, if any.

If any of the specified object UUIDs were not found, **RpcNsMgmtBindingUnexport** returns the RPC_S_NOT_ALL_OBJS_UNEXPORTED status code.

In addition to calling **RpcNsMgmtBindingUnexport**, a management application should also call the **RpcMgmtEpUnregister** routine to unregister the servers that have registered with the endpoint-map database.

**Note** Name-service databases are designed to be relatively stable. In replicated name services, frequent use of the **RpcNsBindingExport** and **RpcNsBindingUnexport** routines causes the name service to repeatedly remove and replace the same entry and can cause performance problems.

## Return Values

| Value | Meaning |
|---|---|
| RPC_S_OK | Success |
| RPC_S_INVALID_VERS_OPTION | Invalid version option |
| RPC_S_NOTHING_TO_UNEXPORT | Nothing to unexport |
| RPC_S_INVALID_NAME_SYNTAX | Invalid name syntax |
| RPC_S_UNSUPPORTED_NAME_SYNTAX | Unsupported name syntax |
| RPC_S_INCOMPLETE_NAME | Incomplete name |
| RPC_S_ENTRY_NOT_FOUND | Name-service entry not found |
| RPC_S_NAME_SERVICE_UNAVAILABLE | Name service unavailable |
| RPC_S_INTERFACE_NOT_FOUND | Interface not found |
| RPC_S_NOT_ALL_OBJS_UNEXPORTED | Not all objects unexported |

## See Also

**RpcMgmtEpUnregister**, **RpcNsBindingExport**, **RpcNsBindingUnexport**

# RpcNsMgmtEntryCreate Quick Info

The **RpcNsMgmtEntryCreate** function creates a name-service database entry.

```
#include <rpc.h>

RPC_STATUS RPC_ENTRY RpcNsMgmtEntryCreate(

  unsigned long   EntryNameSyntax,
  unsigned char *   EntryName
);
```

## Parameters

*EntryNameSyntax*

Specifies an integer value that indicates the syntax of the next argument, *EntryName*.

To use the syntax specified in the registry value HKEY_LOCAL_MACHINE\Software\Microsoft\Rpc\NameService\
DefaultSyntax, provide a value of RPC_C_NS_SYNTAX_DEFAULT.

*EntryName*

Points to the name of the entry to create.

## Remarks

**Note** This DCE function is not supported by the Microsoft Locator.

The **RpcNsMgmtEntryCreate** routine creates an entry in the name-service database.

A management application can call **RpcNsMgmtEntryCreate** to create a name-service database entry for use by another application that does not itself have the necessary name-service database privileges to create an entry.

## Return Values

| Value | Meaning |
|---|---|
| RPC_S_OK | Success |
| RPC_S_INVALID_NAME_SYNTAX | Invalid name syntax |
| RPC_S_UNSUPPORTED_NAME_SYNTAX | Unsupported name syntax |
| RPC_S_INCOMPLETE_NAME | Incomplete name |
| RPC_S_ENTRY_ALREADY_EXISTS | Name-service entry already exists |
| RPC_S_NAME_SERVICE_UNAVAILABLE | Name service unavailable |

## See Also

RpcNsMgmtEntryDelete

# RpcNsMgmtEntryDelete Quick Info

The **RpcNsMgmtEntryDelete** function deletes a name-service database entry.

```
#include <rpc.h>

RPC_STATUS RPC_ENTRY RpcNsMgmtEntryDelete(

  unsigned long   EntryNameSyntax,
  unsigned char *   EntryName
);
```

## Parameters

*EntryNameSyntax*

Specifies an integer value that indicates the syntax of the next argument, *EntryName*.

To use the syntax specified in the registry value HKEY_LOCAL_MACHINE\Software\Microsoft\Rpc\NameService\
DefaultSyntax, provide a value of RPC_C_NS_SYNTAX_DEFAULT.

*EntryName*

Points to the name of the entry to delete.

## Remarks

The **RpcNsMgmtEntryDelete** routine removes an entry from the name-service database.

Management applications use this routine only when an entry is no longer needed – for example, when a server is being permanently removed from service.

Because name-service databases are designed to be relatively stable, the frequent use of the **RpcNsMgmtEntryDelete** routine in client or server applications can result in performance problems. Creating and deleting entries in client or server applications causes the name-service database to repeatedly remove and replace the same entry. This can lead to performance problems in replicated name-service databases.

## Return Values

| Value | Meaning |
|---|---|
| RPC_S_OK | Success |
| RPC_S_INVALID_NAME_SYNTAX | Invalid name syntax |
| RPC_S_UNSUPPORTED_NAME_SYNTAX | Unsupported name syntax |
| RPC_S_INCOMPLETE_NAME | Incomplete name |
| RPC_S_ENTRY_NOT_FOUND | Name-service entry not found |
| RPC_S_NAME_SERVICE_UNAVAILABLE | Name service unavailable |
| RPC_S_NOT_RPC_ENTRY | Not an RPC entry |

## See Also

RpcNsMgmtEntryCreate

# RpcNsMgmtEntryInqIfIds Quick Info

The **RpcNsMgmtEntryInqIfIds** function returns the list of interfaces exported to a name-service database entry.

```
#include <rpc.h>

RPC_STATUS RPC_ENTRY RpcNsMgmtEntryInqIfIds(

  unsigned long   EntryNameSyntax,
  unsigned char *   EntryName,
  RPC_IF_ID_VECTOR * *   IfIdVec
);
```

## Parameters

*EntryNameSyntax*

Specifies an integer value that indicates the syntax of the next argument, *EntryName*.

To use the syntax specified in the registry value HKEY_LOCAL_MACHINE\Software\Microsoft\Rpc\NameService\
DefaultSyntax, provide a value of RPC_C_NS_SYNTAX_DEFAULT.

*EntryName*

Points to the name-service database entry name for which an interface identification vector is returned.

*IfIdVec*

Returns a pointer to a pointer to the interface-identification vector.

## Remarks

The **RpcNsMgmtEntryInqIfIds** routine returns an interface-identification vector containing the interfaces of binding handles exported by a server to *EntryName*.

**RpcNsMgmtEntryInqIfIds** uses an expiration age of 0, causing an immediate update of the local copy of name-service data.

The calling application is responsible for calling the **RpcIfIdVectorFree** routine to release memory used by the vector.

## Return Values

| Value | Meaning |
|---|---|
| RPC_S_OK | Success |
| RPC_S_INVALID_NAME_SYNTAX | Invalid name syntax |
| RPC_S_UNSUPPORTED_NAME_SYNTAX | Unsupported name syntax |
| RPC_S_INCOMPLETE_NAME | Incomplete name |
| RPC_S_ENTRY_NOT_FOUND | Name-service entry not found |
| RPC_S_NAME_SERVICE_UNAVAILABLE | Name service unavailable |

## See Also

RpcIfIdVectorFree, RpcIfInqId, RpcNsBindingExport

# RpcNsMgmtHandleSetExpAge Quick Info

The **RpcNsMgmtHandleSetExpAge** function sets the expiration age of a name-service handle for local copies of name-service data.

```
#include <rpc.h>

RPC_STATUS RPC_ENTRY RpcNsMgmtHandleSetExpAge(

  RPC_NS_HANDLE   NsHandle,
  unsigned long   ExpirationAge
);
```

## Parameters

*NsHandle*

Specifies a name-service handle that an expiration age is set for. A name-service handle is returned from a name-service "begin" operation.

*ExpirationAge*

Specifies an integer value in seconds that sets the expiration age of local name-service data read by all "next" routines using the specified *NsHandle* argument.

An expiration age of 0 causes an immediate update of the local name-service data.

## Remarks

The **RpcNsMgmtHandleSetExpAge** routine sets a handle-expiration age for a specified name-service handle (*NsHandle*). The expiration age is the amount of time that a local copy of data from a name-service attribute can exist before a request from the application for the attribute requires updating the local copy. When an application begins running, the RPC run-time library specifies a random value of two hours as the default expiration age. The default is global to the application. A handle-expiration age applies only to a specific name-service handle and temporarily overrides the current global expiration age.

Normally, you should avoid using **RpcNsMgmtHandleSetExpAge**; instead, you should rely on the application's global expiration age.

A handle-expiration age is used exclusively by name-service "next" operations (which read data from name-service attributes). A "next" operation normally starts by looking for a local copy of the attribute data being requested by an application. In the absence of a local copy, the "next" operation creates one with fresh attribute data from the name-service database. If a local copy already exists, the operation compares its actual age to the expiration age being used by the application (which in this case is the expiration age set for the name-service handle). If the actual age exceeds the handle-expiration age, the operation automatically tries to update the local copy with fresh attribute data. If updating is impossible, the old local data remains in place and the "next" operation fails, returning the RPC_S_NAME_SERVICE_UNAVAILABLE status code.

The scope of a handle-expiration age is a single series of "next" operations. The **RpcNsMgmtHandleSetExpAge** routine operates within the following context:

- A "begin" operation creates a name-service handle.
- A call to the **RpcNsMgmtHandleSetExpAge** routine creates an expiration age for the handle.
- A series of "next" operations for the name-service handle uses the handle expiration age.
- A "done" operation for the name-service handle deletes both the handle and its expiration age.

Setting the handle-expiration age to a small value causes the name-service "next" operations to frequently update local data for any name-service attribute requested by your application. For example, setting the expiration age to 0 forces the "next" operation to update local data for the name-service attribute requested by your application. Therefore, setting a small handle-expiration age can create performance problems for your application. Furthermore, if your application is using a remote name-service server, a small expiration age can adversely affect network performance for all applications.

Limit the use of **RpcNsMgmtHandleSetExpAge** to the following situations:

- When you must always get accurate name-service data.

  For example, during management operations to update a profile, you may need to always see the profile's current contents. In this case, before beginning to inquire about a profile, your application should call the **RpcNsMgmtHandleSetExpAge** routine and specify 0 for the *ExpirationAge* argument.
- When a request using the default expiration age has failed, and your application needs to retry the operation.

  For example, a client application using name-service "import" operations should first try to obtain bindings using the application's default expiration age. However, sometimes the "import-next" operation returns either no binding handles or an insufficient number of them. In this case, the client could retry the "import" operation and, after the **RpcNsBindingImportBegin** call, include a **RpcNsMgmtHandleSetExpAge** call and specify 0 for the *ExpirationAge* argument. When the client calls the "import-next" routine again, the small handle-expiration age causes the "import-next" operation to update the local attribute data.

## Return Values

| Value | Meaning |
|---|---|
| RPC_S_OK | Success |
| RPC_S_NAME_SERVICE_UNAVAILABLE | Name service unavailable |

## See Also

RpcNsBindingImportBegin, RpcNsMgmtInqExpAge, RpcNsMgmtSetExpAge

# RpcNsMgmtInqExpAge Quick Info

The **RpcNsMgmtInqExpAge** function returns the global expiration age for local copies of name-service data.

```
#include <rpc.h>

RPC_STATUS RPC_ENTRY RpcNsMgmtInqExpAge(

  unsigned long *   ExpirationAge
);
```

## Parameters

*ExpirationAge*

Returns a pointer to the default expiration age, in seconds. This value is used by all name-service "read" operations (that is, all "next" operations).

## Remarks

The **RpcNsMgmtInqExpAge** routine returns the expiration age that the application is using. The expiration age is the amount of time in seconds that a local copy of data from a name-service attribute can exist before a request from the application for the attribute requires updating the local copy. When an application begins running, the RPC run-time library specifies a random value of two hours as the default expiration age. The default is global to the application.

An expiration age is used by name-service "next" operations (which read data from name-service attributes). A "next" operation normally starts by looking for a local copy of the attribute data being requested by an application. In the absence of a local copy, the "next" operation creates one with fresh attribute data from the name-service database. If a local copy already exists, the operation compares its actual age to the expiration age being used by the application. If the actual age exceeds the expiration age, the operation automatically tries to update the local copy with fresh attribute data. If updating is impossible, the old local data remains in place and the "next" operation fails.

Applications normally should use only the default expiration age. For special cases, however, an application can substitute a user-supplied global expiration age for the default by calling the **RpcNsMgmtSetExpAge** routine. The **RpcNsMgmtInqExpAge** routine returns the current global expiration age, whether a default or a user-supplied value.

An application can also override the global expiration age temporarily by calling the **RpcNsMgmtHandleSetExpAge** routine.

## Return Values

| Value | Meaning |
|---|---|
| RPC_S_OK | Success |

## See Also

RpcNsMgmtHandleSetExpAge, RpcNsMgmtSetExpAge

# RpcNsMgmtSetExpAge Quick Info

The **RpcNsMgmtSetExpAge** function modifies the application's global expiration age for local copies of name-service data.

```
#include <rpc.h>

RPC_STATUS RPC_ENTRY RpcNsMgmtSetExpAge(

  unsigned long   ExpirationAge
);
```

## Parameters

*ExpirationAge*

Specifies an integer value in seconds that indicates the default expiration age for local name-service data. This expiration age is applied to all name-service "read" operations (that is, all "next" operations).

An expiration age of 0 causes an immediate update of the local name-service data.

To reset the expiration age to an RPC-assigned random value of two hours, specify a value of RPC_C_NS_DEFAULT_EXP_AGE.

## Remarks

The **RpcNsMgmtSetExpAge** routine modifies the global expiration age of an application. The expiration age is the amount of time that a local copy of data from a name-service attribute can exist before a request from the application for the attribute requires updating the local copy. When an application begins running, the RPC run-time library specifies a random value of between 8 and 12 hours as the default expiration age. The default is global to the application.

Normally, you should avoid using **RpcNsMgmtSetExpAge**; instead, you should rely on the default expiration age.

An expiration age is used by name-service "next" operations (which read data from name-service attributes). A "next" operation normally starts by looking for a local copy of the attribute data being requested by an application. In the absence of a local copy, the "next" operation creates one with fresh attribute data from the name-service database. If a local copy already exists, the operation compares its actual age to the expiration age being used by the application. If the actual age exceeds the expiration age, the operation automatically tries to update the local copy with fresh attribute data. If updating is impossible, the old local data remains in place and the "next" operation fails, returning the RPC_S_NAME_SERVICE_UNAVAILABLE status code.

Setting the expiration age to a small value causes the name-service "next" operations to frequently update local data for any name-service attribute requested by your application. For example, setting the expiration age to 0 forces all "next" operations to update local data for the name-service attribute requested by your application. Therefore, setting small expiration ages can create performance problems for your application and increase network traffic. Furthermore, if your application is using a remote name-service server, a small expiration age can adversely affect network performance for all applications.

## Return Values

| Value | Meaning |
|---|---|
| RPC_S_OK | Success |
| RPC_S_NAME_SERVICE_UNAVAILABL | Name service unavailable |

E

## See Also

RpcNsMgmtHandleSetExpAge

# RpcNsProfileDelete Quick Info

The **RpcNsProfileDelete** function deletes a profile attribute.

```
#include <rpc.h>

RPC_STATUS RPC_ENTRY RpcNsProfileDelete(

  unsigned long   ProfileNameSyntax,
  unsigned char *   ProfileName
);
```

## Parameters

*ProfileNameSyntax*

Specifies an integer value that indicates the syntax of the next argument, *ProfileName*.

To use the syntax specified in the registry value HKEY_LOCAL_MACHINE\Software\Microsoft\Rpc\NameService\
DefaultSyntax, provide a value of RPC_C_NS_SYNTAX_DEFAULT.

*ProfileName*

Points to the name of the profile to delete.

## Remarks

**Note** This DCE function is not supported by the Microsoft Locator.

The **RpcNsProfileDelete** routine deletes the profile attribute from the specified name-service entry (*ProfileName*).

Neither *ProfileName* nor the entry names included as members in each profile element are deleted.

Use **RpcNsProfileDelete** cautiously; deleting a profile can have the unwanted effect of breaking a hierarchy of profiles.

## Return Values

| Value | Meaning |
|---|---|
| RPC_S_OK | Success |
| RPC_S_INVALID_NAME_SYNTAX | Invalid name syntax |
| RPC_S_UNSUPPORTED_NAME_SYNTAX | Unsupported name syntax |
| RPC_S_INCOMPLETE_NAME | Incomplete name |
| RPC_S_ENTRY_NOT_FOUND | Name-service entry not found |
| RPC_S_NAME_SERVICE_UNAVAILABLE | Name service unavailable |

## See Also

RpcNsProfileEltAdd, RpcNsProfileEltRemove

# RpcNsProfileEltAdd Quick Info

The **RpcNsProfileEltAdd** function adds an element to a profile. If necessary, it creates the entry.

```
#include <rpc.h>

RPC_STATUS RPC_ENTRY RpcNsProfileEltAdd(

  unsigned long   ProfileNameSyntax,
  unsigned char *   ProfileName,
  RPC_IF_ID *   IfId,
  unsigned long   MemberNameSyntax,
  unsigned char *   MemberName,
  unsigned long   Priority,
  unsigned char *   Annotation
);
```

## Parameters

*ProfileNameSyntax*

Specifies an integer value that indicates the syntax of the next argument, *ProfileName*.

To use the syntax specified in the registry value HKEY_LOCAL_MACHINE\Software\Microsoft\Rpc\NameService\
DefaultSyntax, provide a value of RPC_C_NS_SYNTAX_DEFAULT.

*ProfileName*

Points to the name of the profile to receive a new element.

*IfId*

Points to the interface identification of the new profile element. To add or replace the default profile element, specify a null value.

*MemberNameSyntax*

Specifies an integer value that indicates the syntax of the next argument, *MemberName*.

To use the syntax specified in the registry value HKEY_LOCAL_MACHINE\Software\Microsoft\Rpc\NameService\
DefaultSyntax, provide a value of RPC_C_NS_SYNTAX_DEFAULT.

*MemberName*

Points to a name-service entry name to include in the new profile element.

*Priority*

Specifies an integer value (0 through 7) that indicates the relative priority for using the new profile element during the "import" and "lookup" operations. A value of 0 is the highest priority; a value of 7 is the lowest priority.

When adding a default profile member, use a value of 0.

*Annotation*

Points to an annotation string stored as part of the new profile element. The string can be up to 17 characters long. Specify a null value or a null-terminated string if there is no annotation string.

The string is used by applications for informational purposes only. For example, an application can use this string to store the interface-name string specified in the IDL file.

RPC does not use the annotation string during "lookup" or "import" operations or for enumerating

profile elements.

## Remarks

**Note** This DCE function is not supported by the Microsoft Locator.

The **RpcNsProfileEltAdd** routine adds an element to the profile attribute of the name-service entry specified by the *ProfileName* argument.

If the *ProfileName* entry does not exist, **RpcNsProfileEltAdd** tries to create the entry with a profile attribute and adds the profile element specified by the *IfId*, *MemberName*, *Priority*, and *Annotation* arguments. In this case, the application must have the privilege to create the entry. Otherwise, a management application with the necessary privileges should create the entry by calling the **RpcNsMgmtEntryCreate** routine before the application is run.

If an element with the specified member name and interface identification is already in the profile, **RpcNsProfileEltAdd** updates the element's priority and annotation string using the values provided in the *Priority* and *Annotation* arguments.

## Return Values

| Value | Meaning |
|---|---|
| RPC_S_OK | Success |
| RPC_S_INVALID_NAME_SYNTAX | Invalid name syntax |
| RPC_S_UNSUPPORTED_NAME_SYNTAX | Unsupported name syntax |
| RPC_S_INCOMPLETE_NAME | Incomplete name |
| RPC_S_NAME_SERVICE_UNAVAILABLE | Name service unavailable |

## See Also

RpcIfInqId, RpcNsMgmtEntryCreate, RpcNsProfileEltRemove

# RpcNsProfileEltInqBegin Quick Info

The **RpcNsProfileEltInqBegin** function creates an inquiry context for viewing the elements in a profile.

```
#include <rpc.h>

RPC_STATUS RPC_ENTRY RpcNsProfileEltInqBegin(

  unsigned long   ProfileNameSyntax,
  unsigned char *   ProfileName,
  unsigned long   InquiryType,
  RPC_IF_ID *   IfId,
  unsigned long   VersOption,
  unsigned long   MemberNameSyntax,
  unsigned char *   MemberName,
  RPC_NS_HANDLE *   InquiryContext
);
```

## Parameters

*ProfileNameSyntax*

Specifies an integer value that indicates the syntax of the next argument, *ProfileName*.

To use the syntax specified in the registry value HKEY_LOCAL_MACHINE\Software\Microsoft\Rpc\NameService\
DefaultSyntax, provide a value of RPC_C_NS_SYNTAX_DEFAULT.

*ProfileName*

Points to the name of the profile to view.

*InquiryType*

Specifies an integer value indicating the type of inquiry to perform on the profile. The following table lists valid inquiry types:

| Inquiry type | Description |
|---|---|
| RPC_C_PROFILE_DEFAULT_ELT | Searches the profile for the default profile element, if any. The *IfId*, *VersOption*, and *MemberName* arguments are ignored. |
| RPC_C_PROFILE_ALL_ELTS | Returns every element from the profile. The *IfId*, *VersOption*, and *MemberName* arguments are ignored. |
| RPC_C_PROFILE_MATCH_BY_IF | Searches the profile for the elements that contain the interface identification specified by the *IfId* and *VersOption* values. The *MemberName* argument is ignored. |
| RPC_C_PROFILE_MATCH_BY_MBR | Searches the profile for the elements that contain the member name specified by the *MemberName* argument. The *IfId* and *VersOption* arguments are ignored. |
| RPC_C_PROFILE_MATCH_BY_BOTH | Searches the profile for the elements that contain the interface identification |

and member identified by the *IfId*, *VersOption*, and *MemberName* arguments.

*IfId*

Points to the interface identification of the profile elements to be returned by the **RpcNsProfileEltInqNext** routine.

The *IfId* argument is used only when specifying a value of RPC_C_PROFILE_MATCH_BY_IF or RPC_C_PROFILE_MATCH_BY_BOTH for the *InquiryType* argument. Otherwise, *IfId* is ignored and a null value can be specified.

*VersOption*

Specifies how the **RpcNsProfileEltInqNext** routine uses the *IfId* argument.

The *VersOption* argument is used only when specifying a value of RPC_C_PROFILE_MATCH_BY_IF or RPC_C_PROFILE_MATCH_BY_BOTH for the *InquiryType* argument. Otherwise, this argument is ignored and a 0 value can be specified.

The following table describes valid values for the *VersOption* argument.

| Values | Description |
|---|---|
| RPC_C_VERS_ALL | Returns profile elements that offer the specified interface UUID, regardless of the version numbers. For this value, specify 0 for both the major and minor versions in *IfId*. |
| RPC_C_VERS_COMPATIBLE | Returns profile elements that offer the same major version of the specified interface UUID and a minor version greater than or equal to the minor version of the specified interface UUID. |
| RPC_C_VERS_EXACT | Returns profile elements that offer the specified version of the specified interface UUID. |
| RPC_C_VERS_MAJOR_ONLY | Returns profile elements that offer the same major version of the specified interface UUID (ignores the minor version). For this value, specify 0 for the minor version in *IfId*. |
| RPC_C_VERS_UPTO | Returns profile elements that offer a version of the specified interface UUID less than or equal to the specified major and minor version. (For example, if the *IfId* contained V2.0 and the profile contained elements with V1.3, V2.0, and V2.1, the **RpcNsProfileEltInqNext** routine returns the elements with V1.3 and V2.0.) |

*MemberNameSyntax*

Specifies an integer value that indicates the syntax of the next argument, *MemberName*, and of the

return argument *MemberName* in the **RpcNsProfileEltInqNext** routine.

To use the syntax specified in the registry value HKEY_LOCAL_MACHINE\Software\Microsoft\Rpc\NameService\
DefaultSyntax, provide a value of RPC_C_NS_SYNTAX_DEFAULT.

*MemberName*

Points to the member name that the **RpcNsProfileEltInqNext** routine looks for in profile elements.

The *MemberName* argument is used only when specifying a value of RPC_C_PROFILE_MATCH_BY_MBR or RPC_C_PROFILE_MATCH_BY_BOTH for the *InquiryType* argument. Otherwise, *MemberName* is ignored and a null value can be specified.

*InquiryContext*

Returns a pointer to a name-service handle for use with the **RpcNsProfileEltInqNext** and **RpcNsProfileEltInqDone** routines.

## Remarks

**Note** This DCE function is not supported by the Microsoft Locator.

The **RpcNsProfileEltInqBegin** routine creates an inquiry context for viewing the elements in a profile.

Using the *InquiryType* argument, an application specifies which of the following profile elements are to be returned from calls to the **RpcNsProfileEltInqNext** routine:

- The default element
- All elements
- Elements with the specified interface identification
- Elements with the specified member name
- Elements with both the specified interface identification and member name

Before calling the **RpcNsProfileEltInqNext** routine, the application must first call **RpcNsProfileEltInqBegin** to create an inquiry context.

When finished viewing the profile elements, the application calls the **RpcNsProfileEltInqDone** routine to delete the inquiry context.

## Return Values

| Value | Meaning |
|---|---|
| RPC_S_OK | Success |
| RPC_S_INVALID_VERS_OPTION | Invalid version option |
| RPC_S_INVALID_NAME_SYNTAX | Invalid name syntax |
| RPC_S_UNSUPPORTED_NAME_SYNTAX | Unsupported name syntax |
| RPC_S_INCOMPLETE_NAME | Incomplete name |
| RPC_S_ENTRY_NOT_FOUND | Name-service entry not found |
| RPC_S_NAME_SERVICE_UNAVAILABLE | Name service unavailable |

## See Also

RpcIfInqId, RpcNsProfileEltInqDone, RpcNsProfileEltInqNext

# RpcNsProfileEltInqDone Quick Info

The **RpcNsProfileEltInqDone** function deletes the inquiry context for viewing the elements in a profile.

```
#include <rpc.h>

RPC_STATUS RPC_ENTRY RpcNsProfileEltInqDone(

  RPC_NS_HANDLE *   InquiryContext
  );
```

## Parameters

*InquiryContext*

Points to a name-service handle to free. The name-service handle *InquiryContext* points to is created by calling the **RpcNsProfileEltInqBegin** routine.

An argument value of NULL is returned.

## Remarks

**Note** This DCE function is not supported by the Microsoft Locator.

The **RpcNsProfileEltInqDone** routine frees an inquiry context created by calling the **RpcNsProfileEltInqBegin** routine.

An application calls **RpcNsProfileEltInqDone** after viewing profile elements using the **RpcNsProfileEltInqNext** routine.

## Return Values

| Value | Meaning |
|---|---|
| RPC_S_OK | Success |

## See Also

RpcNsProfileEltInqBegin, RpcNsProfileEltInqNext

# RpcNsProfileEltInqNext Quick Info

The **RpcNsProfileEltInqNext** function returns one element at a time from a profile.

```
#include <rpc.h>

RPC_STATUS RPC_ENTRY RpcNsProfileEltInqNext(

  RPC_NS_HANDLE   InquiryContext,
  RPC_IF_ID *   IfId,
  unsigned char * *   MemberName,
  unsigned long *   Priority,
  unsigned char * *   Annotation
);
```

## Parameters

*InquiryContext*

Specifies a name-service handle returned from the **RpcNsProfileEltInqBegin** routine.

*IfId*

Returns a pointer to the interface identification of the profile element.

*MemberName*

Returns a pointer to a pointer to the profile element's member name.

The syntax of the returned name was specified by the *MemberNameSyntax* argument in the **RpcNsProfileEltInqBegin** routine.

Specify a null value to prevent **RpcNsProfileEltInqNext** from returning the *MemberName* argument. In this case, the application does not call the **RpcStringFree** routine.

*Priority*

Returns a pointer to the profile-element priority.

*Annotation*

Returns a pointer to a pointer to the annotation string for the profile element. If there is no annotation string in the profile element, the string "\0" is returned.

Specify a null value to prevent **RpcNsProfileEltInqNext** from returning the *Annotation* argument. In this case, the application does not need to call the **RpcStringFree** routine.

## Remarks

**Note** This DCE function is not supported by the Microsoft Locator.

The **RpcNsProfileEltInqNext** routine returns one element from the profile specified by the *ProfileName* argument in the **RpcNsProfileEltInqBegin** routine. Regardless of the value specified for the *InquiryType* argument in **RpcNsProfileEltInqBegin**, **RpcNsProfileEltInqNext** returns all the components (interface identification, member name, priority, annotation string) of a profile element.

An application can view all the selected profile entries by repeatedly calling the **RpcNsProfileEltInqNext** routine. When all the elements have been viewed, this routine returns a RPC_S_NO_MORE_ELEMENTS status code. The returned elements are unordered.

On each call to **RpcNsProfileEltInqNext** that returns a profile element, the RPC run-time library allocates memory for the returned member name and annotation string. The application is responsible for calling the **RpcStringFree** routine for each returned member name and annotation string.

After viewing the profile's elements, the application must call the **RpcNsProfileEltInqDone** routine to release the inquiry context.

## Return Values

| Value | Meaning |
|---|---|
| RPC_S_OK | Success |
| RPC_S_INCOMPLETE_NAME | Incomplete name |
| RPC_S_NAME_SERVICE_UNAVAILABLE | Name service unavailable |
| RPC_S_NO_MORE_ELEMENTS | No more elements |

## See Also

**RpcNsProfileEltInqBegin**, **RpcNsProfileEltInqDone**, **RpcStringFree**

# RpcNsProfileEltRemove Quick Info

The **RpcNsProfileEltRemove** function removes an element from a profile.

```
#include <rpc.h>

RPC_STATUS RPC_ENTRY RpcNsProfileEltRemove(

  unsigned long   ProfileNameSyntax,
  unsigned char *   ProfileName,
  RPC_IF_ID *   IfId,
  unsigned long   MemberNameSyntax,
  unsigned char *   MemberName
);
```

## Parameters

*ProfileNameSyntax*

Specifies an integer value that indicates the syntax of the next argument, *ProfileName*.

To use the syntax specified in the registry value HKEY_LOCAL_MACHINE\Software\Microsoft\Rpc\NameService\
DefaultSyntax, provide a value of RPC_C_NS_SYNTAX_DEFAULT.

*ProfileName*

Points to the name of the profile from which to remove an element.

*IfId*

Points to the interface identification of the profile element to be removed.

Specify a null value to remove the default profile member.

*MemberNameSyntax*

Specifies an integer value that indicates the syntax of the next argument, *MemberName*.

To use the syntax specified in the registry value HKEY_LOCAL_MACHINE\Software\Microsoft\Rpc\NameService\
DefaultSyntax, provide a value of RPC_C_NS_SYNTAX_DEFAULT.

*MemberName*

Points to the name-service entry name in the profile element to remove.

## Remarks

**Note** This DCE function is not supported by the Microsoft Locator.

The **RpcNsProfileEltRemove** routine removes a profile element from the profile attribute in the *ProfileName* entry. The **RpcNsProfileEltRemove** routine requires an exact match of the *MemberName* and *IfId* arguments in order to remove a profile element.

The entry (*MemberName*) included as a member in the profile element is not deleted.

Use **RpcNsProfileEltRemove** cautiously: removing elements from a profile can have the unwanted effect of breaking a hierarchy of profiles.

## Return Values

| Value | Meaning |
| --- | --- |
| RPC_S_OK | Success |
| RPC_S_INVALID_NAME_SYNTAX | Invalid name syntax |
| RPC_S_UNSUPPORTED_NAME_SYNTAX | Unsupported name syntax |
| RPC_S_INCOMPLETE_NAME | Incomplete name |
| RPC_S_ENTRY_NOT_FOUND | Name-service entry not found |
| RPC_S_NAME_SERVICE_UNAVAILABLE | Name service unavailable |

## See Also

RpcNsProfileDelete, RpcNsProfileEltAdd

# RpcObjectInqType Quick Info

The **RpcObjectInqType** function returns the type of an object.

This function is supported by both 32-bit platforms – Windows NT and Windows 95.

```
#include <rpc.h>

RPC_STATUS RPC_ENTRY RpcObjectInqType(

  UUID *   ObjUuid,
  UUID *   TypeUuid
);
```

## Parameters

*ObjUuid*

Points to the object UUID whose associated type UUID is returned.

*TypeUuid*

Returns a pointer to the type UUID of the *ObjUuid* argument.

Specify an argument value of NULL to prevent the return of a type UUID. In this way, an application can determine (from the returned status) whether *ObjUuid* is registered without specifying an output type UUID variable.

## Remarks

A server application calls the **RpcObjectInqType** routine to obtain the type UUID of an object.

If the object was registered with the RPC run-time library using the **RpcObjectSetType** routine, the registered type is returned.

Optionally, an application can privately maintain an object/type registration. In this case, if the application has provided an object inquiry function (see **RpcObjectSetInqFn**), the RPC run-time library uses that function to determine an object's type.

The **RpcObjectInqType** routine obtains the returned type UUID as described in the following table:

| Object UUID registered | Inquiry function registered | Return value |
|---|---|---|
| Yes (**RpcObjectSetType**) | Ignored | The object's registered type UUID |
| No | Yes (**RpcObjectSetInqFn**) | The type UUID returned from the inquiry function |
| No | No | The nil UUID |

## Return Values

| Value | Meaning |
|---|---|
| RPC_S_OK | Success |
| RPC_S_OBJECT_NOT_FOUND | Object not found |

## See Also

RpcObjectSetInqFn, RpcObjectSetType

# RpcObjectSetInqFn Quick Info

The **RpcObjectSetInqFn** function registers an object-inquiry function. A null value turns off a previously registered object-inquiry function.

This function is supported by both 32-bit platforms – Windows NT and Windows 95.

```
#include <rpc.h>

RPC_STATUS RPC_ENTRY RpcObjectSetInqFn(

  RPC_OBJECT_INQ_FN   InquiryFn
);
```

## Parameters

*InquiryFn*

Specifies an object-type inquiry function. When an application calls the **RpcObjectInqType** routine and the RPC run-time library finds that the specified object is not registered, the run-time library automatically calls **RpcObjectSetInqFn** to determine the object's type.

The following C-language definition for RPC_OBJECT_INQ_FN illustrates the prototype for the object-inquiry function:

```
typedef void ( *   RPC_OBJECT_INQ_FN) (
     UUID  *       ObjectUuid,
     UUID  *       TypeUuid,
     RPC_STATUS *  Status);
```

The *TypeUuid* and *Status* values are returned as the output from the **RpcObjectInqType** routine.

## Remarks

A server application calls the **RpcObjectSetInqFn** routine to specify a function to determine an object's type. If an application privately maintains an object/type registration, the specified inquiry function returns the type UUID of an object.

The RPC run-time library automatically calls the inquiry function when the application calls **RpcObjectInqType** and the object of interest was not previously registered with the **RpcObjectSetType** routine.

## Return Values

| Value | Meaning |
|---|---|
| RPC_S_OK | Success |

## See Also

RpcObjectInqType, RpcObjectSetType

# RpcObjectSetType Quick Info

The **RpcObjectSetType** function assigns the type of an object.

This function is supported by both 32-bit platforms – Windows NT and Windows 95.

```
#include <rpc.h>

RPC_STATUS RPC_ENTRY RpcObjectSetType(

  UUID *   ObjUuid,
  UUID *   TypeUuid
);
```

## Parameters

*ObjUuid*

Points to an object UUID to associate with the type UUID in the *TypeUuid* argument.

*TypeUuid*

Points to the type UUID of the *ObjUuid* argument.

Specify an argument value of NULL or a nil UUID to reset the object type to the default association of object UUID/nil type UUID.

## Remarks

A server application calls the **RpcObjectSetType** routine to assign a type UUID to an object UUID.

By default, the RPC run-time library automatically assigns all object UUIDs with the nil type UUID. A server application that contains one implementation of an interface (one manager entry-point vector [EPV]) does not need to call **RpcObjectSetType** provided the server registered the interface with the nil type UUID (see **RpcServerRegisterIf**).

A server application that contains multiple implementations of an interface (multiple manager EPVs – that is, multiple type UUIDs) calls **RpcObjectSetType** once for each different object UUID/non-nil type UUID association the server supports. Associating each object with a type UUID tells the RPC run-time library which manager EPV (interface implementation) to use when the server receives a remote procecure call for a non-nil object UUID.

The RPC run-time library allows an application to set the type for an unlimited number of objects.

To remove the association between an object UUID and its type UUID (established by calling **RpcObjectSetType**), a server calls **RpcObjectSetType** again specifying a null value or a nil UUID for the *TypeUuid* argument. This resets the object UUID/type UUID association to the default association of object UUID/nil type UUID.

A server cannot assign a type to the nil object UUID. The RPC run-time library automatically assigns the nil object UUID a nil type UUID.

## Return Values

| Value | Meaning |
|---|---|
| RPC_S_OK | Success |
| RPC_S_INVALID_OBJECT | Invalid object |
| RPC_S_ALREADY_REGISTERED | Object already registered |

## See Also

RpcServerRegisterIf

# RpcProtseqVectorFree Quick Info

The **RpcProtseqVectorFree** function frees the protocol sequences contained in the vector and the vector itself.

This function is supported by both 32-bit platforms – Windows NT and Windows 95.

For a list of Microsoft RPC's supported protocol sequences, see the reference topic String Binding in RPC Data Types and Structures.

```
#include <rpc.h>

RPC_STATUS RPC_ENTRY RpcProtseqVectorFree(

  RPC_PROTSEQ_VECTOR * *   ProtSeqVector
);
```

## Parameters

*ProtSeqVector*

Points to a pointer to a vector of protocol sequences. On return, the pointer is set to NULL.

**Note** **RpcProtseqVectorFree** is available for server applications, not client applications, using Microsoft RPC.

## Remarks

A server calls the **RpcProtseqVectorFree** routine to release the memory used to store a vector of protocol sequences and the individual protocol sequences. **RpcProtseqVectorFree** sets the *ProtSeqVector* argument to a null value.

A server obtains a vector of protocol sequences by calling the **RpcNetworkInqProtseqs** routine.

## Return Values

| Value | Meaning |
|---|---|
| RPC_S_OK | Success |

## See Also

**RpcNetworkInqProtseqs**

# RpcRaiseException Quick Info

Use the **RpcRaiseException** function to raise an exception. The **RpcRaiseException** function does not return to the caller.

```
void RPC_ENTRY RpcRaiseException (

  RPC_STATUS   Exception
);
```

## Parameters

*Exception*

Specifies the exception code for the exception. The following exception codes are defined:

| Exception code | Description |
|---|---|
| RPC_S_ACCESS_DENIED | Access denied |
| RPC_S_ADDRESS_ERROR | An addressing error occurred in the RPC server |
| RPC_S_ALREADY_LISTENING | Server already listening |
| RPC_S_ALREADY_REGISTERED | Object already registered |
| RPC_S_BINDING_HAS_NO_AUTH | Binding has no authentication |
| RPC_S_BINDING_INCOMPLETE | The binding handle is a required parameter. |
| RPC_S_BUFFER_TOO_SMALL | Insufficient buffer |
| RPC_S_CALL_CANCELLED | The remote procedure call exceeded the cancel timeout and was canceled. |
| RPC_S_CALL_FAILED | Call failed |
| RPC_S_CALL_FAILED_DNE | Call failed and did not execute |
| RPC_S_CALL_IN_PROGRESS | Call already in progress for this thread |
| RPC_S_CANNOT_SUPPORT | Operation is not supported |
| RPC_S_CANT_CREATE_ENDPOINT | Cannot create endpoint |
| RPC_S_COMM_FAILURE | Unable to communicate with the server |
| RPC_S_DUPLICATE_ENDPOINT | Endpoint already exists |
| RPC_S_ENTRY_ALREADY_EXISTS | Name-service entry already exists |
| RPC_S_ENTRY_NOT_FOUND | Name-service entry not found |
| RPC_S_FP_DIV_ZERO | A floating-point operation in the server caused a division by zero |
| RPC_S_FP_OVERFLOW | Floating-point overflow has occurred in the RPC server |
| RPC_S_FP_UNDERFLOW | Floating-point underflow has occurred in the server |
| RPC_S_GROUP_MEMBER_NOT_FOUND | Group member not found |
| RPC_S_INCOMPLETE_NAME | Incomplete name |
| RPC_S_INTERFACE_NOT_FOUND | Interface not found |
| RPC_S_INTERNAL_ERROR | Internal error |

| | |
|---|---|
| RPC_S_INVALID_ARG | Invalid argument |
| RPC_S_INVALID_AUTH_IDENTITY | Invalid authentication |
| RPC_S_INVALID_BINDING | Invalid binding handle |
| RPC_S_INVALID_BOUND | Invalid bound |
| RPC_S_INVALID_ENDPOINT_FORMAT | Invalid endpoint format |
| RPC_S_INVALID_INQUIRY_CONTEXT | Invalid inquiry context |
| RPC_S_INVALID_INQUIRY_TYPE | Invalid inquiry type |
| RPC_S_INVALID_LEVEL | Invalid parameter |
| RPC_S_INVALID_NAF_IF | Invalid network-address family ID |
| RPC_S_INVALID_NAME_SYNTAX | Invalid name syntax |
| RPC_S_INVALID_NET_ADDR | Invalid network address |
| RPC_S_INVALID_NETWORK_OPTIONS | Invalid network options |
| RPC_S_INVALID_OBJECT | Invalid object |
| RPC_S_INVALID_RPC_PROTSEQ | Invalid protocol sequence |
| RPC_S_INVALID_SECURIT_DESC | Invalid security descriptor |
| RPC_S_INVALID_STRING_BINDING | Invalid string binding |
| RPC_S_INVALID_STRING_UUID | Invalid string UUID |
| RPC_S_INVALID_TAG | Invalid tag |
| RPC_S_INVALID_TIMEOUT | Invalid timeout value |
| RPC_S_INVALID_VERS_OPTION | Invalid version option |
| RPC_S_MAX_CALLS_TOO_SMALL | Maximum-calls value too small |
| RPC_S_NAME_SERVICE_UNAVAILABLE | Name service unavailable |
| RPC_S_NO_BINDINGS | No bindings |
| RPC_S_NO_CALL_ACTIVE | No remote procedure active in this thread |
| RPC_S_NO_CONTEXT_AVAILABLE | No security context is available to perform impersonation |
| RPC_S_NO_ENDPOINT_FOUND | No endpoint found |
| RPC_S_NO_ENTRY_NAME | No entry name for binding |
| RPC_S_NO_ENV_SETUP | No environment variable is set up |
| RPC_S_NO_INTERFACES | No interfaces are registered |
| RPC_S_NO_INTERFACES_EXPORTED | No interfaces have been exported |
| RPC_S_NO_MORE_BINDINGS | No more bindings |
| RPC_NO_MORE_ELEMENTS | There are no more elements. |
| RPC_S_NO_MORE_MEMBERS | No more members |
| RPC_S_NO_NS_PRIVILEGE | No privilege for name-service operation |
| RPC_S_NO_PRINC_NAME | No principal name is registered |
| RPC_S_NO_PROTSEQS | No supported protocol sequences |
| RPC_S_NO_PROTSEQS_REGISTERED | No protocol sequences registered |

| | |
|---|---|
| RPC_S_NOT_ALL_OBJS_UNEXPORTED | Not all objects unexported |
| RPC_S_NOT_CANCELLED | The thread is not canceled |
| RPC_S_NOT_LISTENING | Server not listening |
| RPC_S_NOT_RPC_ERROR | The status code requested is not valid |
| RPC_S_NOTHING_TO_EXPORT | Nothing to export |
| RPC_S_NOTHING_TO_UNEXPORT | Nothing to unexport |
| RPC_S_OBJECT_NOT_FOUND | Object not found |
| RPC_S_OK | Success |
| RPC_S_OUT_OF_MEMORY | Out of memory |
| RPC_S_OUT_OF_RESOURCES | Out of resources |
| RPC_S_OUT_OF_THREADS | Out of threads |
| RPC_S_PROCNUM_OUT_OF_RANGE | Procedure number is out of range |
| RPC_S_PROTOCOL_ERROR | An RPC protocol error occurred |
| RPC_S_PROTSEQ_NOT_FOUND | Protocol sequence not found |
| RPC_S_PROTSEQ_NOT_SUPPORTED | Protocol sequence not supported |
| RPC_S_SERVER_OUT_OF_MEMORY | Server out of memory |
| RPC_S_SERVER_TOO_BUSY | Server too busy |
| RPC_S_SERVER_UNAVAILABLE | The server is unavailable |
| RPC_S_STRING_TOO_LONG | String too long |
| RPC_S_TYPE_ALREADY_REGISTERED | Type UUID already registered |
| RPC_S_UNKNOWN_AUTHN_LEVEL | Unknown authentication level |
| RPC_S_UNKNOWN_AUTHN_SERVICE | Unknown authentication service |
| RPC_S_UNKNOWN_AUTHN_TYPE | Unknown authentication type |
| RPC_S_UNKNOWN_IF | Unknown interface |
| RPC_S_UNKNOWN_MGR_TYPE | Unknown manager type |
| RPC_S_UNSUPPORTED-TRANS_SYN | Transfer syntax is not supported by the server |
| RPC_S_UNSUPPORTED_NAME_SYNTAX | Unsupported name syntax |
| RPC_S_UNSUPPORTED_TYPE | Unsupported UUID type |
| RPC_S_UUID_LOCAL_ONLY | The UUID that is only valid for this computer has been allocated |
| RPC_S_UUID_NO_ADDRESS | No network address is available to use to construct a UUID |
| RPC_S_WRONG_KIND_OF_BINDING | Wrong kind of binding for operation |
| RPC_S_ZERO_DIVIDE | Attempt to divide an integer by zero |
| RPC_X_BAD_STUB_DATA | The stub received bad data |
| RPC_X_BYTE_COUNT_TOO_SMALL | Byte count is too small |
| RPC_X_ENUM_VALUE_OUT_OF RANGE | The enumeration value is out of range |
| RPC_X_ENUM_VALUE_TOO_LARGE | The enumeration value is out of range |

| | |
|---|---|
| RPC_X_INVALID_BOUND | Specified bounds of an array inconsistent |
| RPC_X_INVALID_TAG | Discriminant value does not match any case values; no default case |
| RPC_X_NO_MEMORY | Insufficient memory available to set up necessary data structures |
| RPC_X_NO_MORE_ENTRIES | List of servers available for *AutoHandle* binding has been exhausted |
| RPC_X_NULL_REF_POINTER | A null reference pointer was passed to the stub |
| RPC_X_SS_BAD_ES_VERSION | The operation for the serializing handle is not valid |
| RPC_X_SS_CANNOT_GET_CALL_HANDLE | The stub is unable to get the remote procedure call handle |
| RPC_X_SS_CHAR_TRANS_OPEN_FAIL | File designated by DCERPCCHARTRANS cannot be opened |
| RPC_X_SS_CHAR_TRANS_SHORT_FILE | File containing character-translation table has fewer than 512 bytes |
| RPC_X_SS_CONTEXT_DAMAGED | Only raised on caller side; UUID in **in**, **out** context handle changed during call |
| RPC_X_SS_CONTEXT_MISMATCH | Only raised on callee side; UUID in **in** handle does not correspond to any known context |
| RPC_X_SS_HANDLES_MISMATCH | The binding handles passed to a remote procedure call don't match |
| RPC_X_SS_IN_NULL_CONTEXT | Null context handle passed in **in** parameter position |
| RPC_X_SS_INVALID_BUFFER | The buffer is not valid for the operation. |
| RPC_X_SS_WRONG_ES_VERSION | The software version is incorrect |
| RPC_X_SS_WRONG_STUB_VERSION | The stub version is incorrect |

## Remarks

The **RpcRaiseException** routine raises an exception; this exception can then be handled by the exception handler.

## Return Values

No value is returned.

## See Also

**RpcAbnormalTermination**, **RpcExcept**, **RpcFinally**

# RpcRevertToSelf Quick Info

After calling **RpcImpersonateClient** and completing any tasks that require client impersonation, the server calls **RpcRevertToSelf** to end impersonation and to reestablish its own security identity.

This function is supported only by Windows NT.

```
#include <rpc.h>

RPC_STATUS RPC_ENTRY RpcRevertToSelf (VOID);
```

## Return Values

| Value | Meaning |
|---|---|
| RPC_S_OK | Success. |
| RPC_S_NO_CALL_ACTIVE | Server does not have a client to impersonate. |
| RPC_S_INVALID_BINDING | Invalid binding handle. |
| RPC_S_WRONG_KIND_OF_BINDING | Wrong kind of binding for operation. |
| RPC_S_CANNOT_SUPPORT | Not supported for this operating system, this transport, or this security subsystem. |

## Remarks

In a multithreaded application, if the call to **RpcImpersonateClient** is with a handle to another client thread, you must call **RpcRevertToSelfEx** with the handle to that thread to end impersonation.

## See Also

**RpcImpersonateClient**, Impersonation

# RpcRevertToSelfEx Quick Info

After calling **RpcImpersonateClient** and completing any tasks that require client impersonation, the server calls **RpcRevertToSelfEx** to end impersonation and to reestablish its own security identity.

This function is supported only by Windows NT.

```
#include <rpc.h>

RPC_STATUS RPC_ENTRY RpcRevertToSelf Ex(

  RPC_BINDING_HANDLE   CallHandle
);
```

## Parameters

*CallHandle*

Specifies a binding handle on the server that represents a binding to the client that the server impersonated. A value of zero specifies the client handle of the current thread; in this case the functionality of **RpcRevertToSelfEx** is identical to that of **RpcRevertToSelf**.

## Return Values

| Value | Meaning |
|---|---|
| RPC_S_OK | Success. |
| RPC_S_NO_CALL_ACTIVE | Server does not have a client to impersonate. |
| RPC_S_INVALID_BINDING | Invalid binding handle. |
| RPC_S_WRONG_KIND_OF_BINDING | Wrong kind of binding for operation. |
| RPC_S_CANNOT_SUPPORT | Not supported for this operating system, this transport, or this security subsystem. |

## Remarks

**RpcRevertToSelfEx** allows a server to impersonate a client and then revert in a multithreaded operation where the call to impersonate a client can come from a thread other than the thread originally dispatched from the RPC. For example, consider a primary thread, called thread1, which is dispatched from a remote client and which wakes up a worker thread, called thread2. If thread2 requires that the server impersonate the client, the server calls RpcImpersonateClient(THREAD1_CALL_HANDLE), performs the required task, calls RpcRevertToSelfEx(THREAD1_CALL_HANDLE) to end the impersonation, and then wakes up thread1.

## See Also

**RpcImpersonateClient**, **RpcRevertToSelf**, Impersonation

# RpcServerInqBindings Quick Info

The **RpcServerInqBindings** function returns the binding handles over which remote procedure calls can be received.

This function is supported by both 32-bit platforms – Windows NT and Windows 95.

```
#include <rpc.h>

RPC_STATUS RPC_ENTRY RpcServerInqBindings(

  RPC_BINDING_VECTOR * *   BindingVector
);
```

## Parameters

*BindingVector*

Returns a pointer to a pointer to a vector of server binding handles.

## Remarks

A server application calls the **RpcServerInqBindings** routine to obtain a vector of server binding handles. Binding handles are created by the RPC run-time library when a server application calls the following routines to register protocol sequences:

- **RpcServerUseAllProtseqs**
- **RpcServerUseProtseq**
- **RpcServerUseAllProtseqsIf**
- **RpcServerUseProtseqIf**
- **RpcServerUseProtseqEp**

The returned binding vector can contain binding handles with dynamic endpoints or binding handles with well-known endpoints, depending on which of the above routines the server application called.

A server uses the vector of binding handles for exporting to the name service, for registering with the local endpoint-map database, or for conversion to string bindings.

If there are no binding handles (no registered protocol sequences), this routine returns the RPC_S_NO_BINDINGS status code and a *BindingVector* argument value of NULL.

The server is responsible for calling the **RpcBindingVectorFree** routine to release the memory used by the vector.

## Return Values

| Value | Meaning |
|---|---|
| RPC_S_OK | Success |
| RPC_S_NO_BINDINGS | No bindings |

## See Also

RpcBindingVectorFree, RpcEpRegister, RpcEpRegisterNoReplace, RpcNsBindingExport, RpcServerUseAllProtseqs, RpcServerUseAllProtseqsIf, RpcServerUseProtseq,

RpcServerUseProtseqEp, RpcServerUseProtseqIf

# RpcServerInqDefaultPrincName

The **RpcServerInqDefaultPrincName** function obtains the default principal name from the server.

This function is supported only by Windows 95.

```
#include <rpc.h>

RPC_STATUS RPC_ENTRY RpcServerInqDefaultPrincName(

  unsigned long   AuthnSvc,
    RPC_CHAR * *   PrincName,
);
```

## Parameters

*AuthnSvc*

Specifies an authentication service to use when the server receives a request for a remote procedure call.

*PrincName*

Points to the principal name to use for the server when authenticating remote procedure calls using the service specified by the *AuthnSvc* argument. The content of the name and its syntax are defined by the authentication service in use.

## Remarks

In a NetWare-only environment, server application calls the **RpcServerInqDefaultPrincName** routine to obtain the name of the NetWare server when authenticated RPC is required. The value obtained from this routine is then passed to **RpcServerRegisterAuthInfo**.

## Return Values

| Value | Meaning |
|---|---|
| RPC_S_OK | Success |
| RPC_S_OUT_OF_MEMORY | Insufficient memory to complete the operation |

## See Also

**RpcBindingSetAuthInfo**, **RpcServerRegisterAuthInfo**

# RpcServerInqIf Quick Info

The **RpcServerInqIf** function returns the manager entry-point vector (EPV) registered for an interface.

This function is supported by both 32-bit platforms – Windows NT and Windows 95.

```
#include <rpc.h>

RPC_STATUS RPC_ENTRY RpcServerInqIf(

  RPC_IF_HANDLE   IfSpec,
  UUID *   MgrTypeUuid,
  RPC_MGR_EPV * *   MgrEpv
);
```

## Parameters

*IfSpec*

Specifies the interface whose manager EPV is returned.

*MgrTypeUuid*

Points to the manager type UUID whose manager EPV is returned.

Specifying an argument value of NULL (or a nil UUID) signifies to return the manager EPV registered with *IfSpec* and the nil manager type UUID.

*MgrEpv*

Returns a pointer to the manager EPV for the requested interface.

## Remarks

A server application calls the **RpcServerInqIf** routine to determine the manager EPV for a registered interface and manager type UUID.

## Return Values

| Value | Meaning |
|---|---|
| RPC_S_OK | Success |
| RPC_S_UNKNOWN_IF | Unknown interface |
| RPC_S_UNKNOWN_MGR_TYPE | Unknown manager type |

## See Also

RpcServerRegisterIf

# RpcServerListen Quick Info

The **RpcServerListen** function tells the RPC run-time library to listen for remote procedure calls. This function will not affect auto-listen interfaces. See **RpcServerRegisterIfEx** for more details.

This function is supported by both server platforms − Windows NT and Windows 95.

```
#include <rpc.h>

RPC_STATUS RPC_ENTRY RpcServerListen(

  unsigned int   MinimumCallThreads,
  unsigned int   MaxCalls,
  unsigned int   DontWait
);
```

## Parameters

*MinimumCallThreads*

Specifies the minimum number of call threads.

*MaxCalls*

Specifies the recommended maximum number of concurrent remote procedure calls the server can execute. To allow efficient performance, the RPC run-time libraries interpret the *MaxCalls* parameter as a suggested limit rather than as an absolute upper bound.

Use RPC_C_LISTEN_MAX_CALLS_DEFAULT to specify the default value.

*DontWait*

Specifies a flag controlling the return from **RpcServerListen**. A value of non-zero indicates that **RpcServerListen** should return immediately after completing function processing. A value of zero indicates that **RpcServerListen** should not return until **RpcMgmtStopServerListening** has been called and all remote calls have completed.

## Remarks

**Note** The Microsoft RPC implementation of **RpcServerListen** includes two new, additional parameters that do not appear in the DCE specification: *DontWait* and *MinimumCallThreads*.

A server calls the **RpcServerListen** routine when the server is ready to process remote procedure calls. RPC allows a server to simultaneously process multiple calls. The *MaxCalls* argument recommends the maximum number of concurrent remote procedure calls the server should execute.

The *MaxCalls* value should be equal to or greater than the largest *MaxCalls* value specified to the routines **RpcServerUseProtseq**, **RpcServerUseProtseqEp**, **RpcServerUseProtseqIf**, **RpcServerUseAllProtseqs**, and **RpcServerUseAllProtseqsIf**.

A server application is responsible for concurrency control between the server manager routines because each routine executes in a separate thread.

When the *DontWait* parameter has a value of zero, the RPC run-time library continues listening for remote procedure calls (that is, the routine does not return to the server application) until one of the following events occurs:

- One of the server application's manager routines calls the **RpcMgmtStopServerListening** routine.
- A client calls a remote procedure provided by the server that directs the server to call **RpcMgmtStopServerListening**.
- A client calls **RpcMgmtStopServerListening** with a binding handle to the server.

Once it receives a stop-listening request, the RPC run-time library stops accepting new remote procedure calls for all registered interfaces. Executing calls are allowed to complete, including callbacks.

After all calls complete, the **RpcServerListen** routine returns to the caller.

When the *DontWait* parameter has a non-zero value, **RpcServerListen** returns to the server immediately after processing all the instructions associated with the function. You can use the **RpcMgmtWaitServerListen** routine to perform the "wait" operation usually associated with **RpcServerListen**.

## Return Values

| Value | Meaning |
|---|---|
| RPC_S_OK | Success |
| RPC_S_ALREADY_LISTENING | Server already listening |
| RPC_S_NO_PROTSEQS_REGISTERED | No protocol sequences registered |
| RPC_S_MAX_CALLS_TOO_SMALL | Maximum calls value too small |

## See Also

**RpcMgmtStopServerListening**, **RpcMgmtWaitServerListen**, **RpcServerRegisterIf**, **RpcServerRegisterIfEx**, **RpcServerUseAllProtseqs**, **RpcServerUseAllProtseqsIf**, **RpcServerUseProtseq**, **RpcServerUseProtseqEp**, **RpcServerUseProtseqIf**

# RpcServerRegisterAuthInfo Quick Info

The **RpcServerRegisterAuthInfo** function registers authentication information with the RPC run-time library.

This function is supported by both 32-bit platforms – Windows NT and Windows 95.

```
#include <rpc.h>

RPC_STATUS RPC_ENTRY RpcServerRegisterAuthInfo(

  unsigned char *   ServerPrincName,
  unsigned long   AuthnSvc,
  RPC_AUTH_KEY_RETRIEVAL_FN   GetKeyFn,
  void *   Arg
);
```

## Parameters

*ServerPrincName*

Points to the principal name to use for the server when authenticating remote procedure calls using the service specified by the *AuthnSvc* argument. The content of the name and its syntax are defined by the authentication service in use.

*AuthnSvc*

Specifies an authentication service to use when the server receives a request for a remote procedure call.

*GetKeyFn*

Specifies the address of a server-application-provided routine that returns encryption keys.

Specify a NULL argument value to use the default method of encryption-key acquisition. In this case, the authentication service specifies the default behavior. Set this parameter to NULL when using the RPC_C_AUTHN_WINNT authentication service.

| Authentication service | GetKeyFn | Arg | Run-time behavior |
|---|---|---|---|
| RPC_C_AUTHN_DCE_PRIVATE | NULL | Non-null | Uses default method of encryption-key acquisition from specified key table; specified argument is passed to default acquisition function |
| RPC_C_AUTHN_DCE_PRIVATE | Non-null | NULL | Uses specified encryption-key acquisition function to obtain keys from default key table |
| RPC_C_AUTHN_DCE_PRIVATE | Non-null | Non-null | Uses specified encryption-key acquisition function to obtain keys from specified key table; specified argument is passed to acquisition function |
| RPC_C_AUTHN_DEC_PUBLIC | Ignored | Ignored | Reserved for future use |
| RPC_C_AUTHN_WINNT | Ignored | Ignored | Does not support |

The following C-language definition for RPC_AUTH_KEY_RETRIEVAL_FN illustrates the prototype for **RpcServerRegisterAuthInfo**:

```
typedef void (* RPC_AUTH_KEY_RETRIEVAL_FN)(
    void * arg, /* in */
    unsigned char * ServerPrincName, /* in */
    unsigned int key_ver, /* in */
    void * * key, /* out */
    unsigned int * status /* out */);
```

The RPC run-time library passes the *ServerPrincName* argument value from **RpcServerRegisterAuthInfo** as the *ServerPrincName* argument value to the *GetKeyFn* acquisition function.

The RPC run-time library automatically provides a value for the key version (*KeyVer*) argument. For a *KeyVer* argument value of zero, the acquisition function must return the most recent key available.

The retrieval function returns the authentication key in the *Key* argument.

If the acquisition function called from **RpcServerRegisterAuthInfo** returns a status other than RPC_S_OK, **RpcServerRegisterAuthInfo** fails and returns an error code to the server application.

If the acquisition function called by the RPC run-time library while authenticating a client's remote procedure call request returns a status other than RPC_S_OK, the request fails and the RPC run-time library returns an error code to the client application.

*Arg*

Points to an argument to pass to the *GetKeyFn* routine, if specified.

## Remarks

A server application calls the **RpcServerRegisterAuthInfo** routine to register an authentication service to use for authenticating remote procedure calls. A server calls this routine once for each authentication service and/or principal name the server wants to register.

The authentication service specified by a client application (using **RpcBindingSetAuthInfo** or **RpcServerRegisterAuthInfo**) must be one of the authentication services specified by the server application. Otherwise, the client's remote procedure call fails and an RPC_S_UNKNOWN_AUTHN_SERVICE status code is returned.

## Return Values

| Value | Meaning |
|---|---|
| RPC_S_OK | Success |
| RPC_S_UNKNOWN_AUTHN_SERVICE | Unknown authentication service |

## See Also

**RpcBindingSetAuthInfo**

# RpcServerRegisterIf Quick Info

The **RpcServerRegisterIf** function registers an interface with the RPC run-time library.

This function is supported by both 32-bit platforms – Windows NT and Windows 95.

```
#include <rpc.h>

RPC_STATUS RPC_ENTRY RpcServerRegisterIf(

  RPC_IF_HANDLE   IfSpec,
  UUID *   MgrTypeUuid,
  RPC_MGR_EPV *   MgrEpv
);
```

## Parameters

*IfSpec*

Specifies a MIDL-generated data structure indicating the interface to register.

*MgrTypeUuid*

Points to a type UUID to associate with the *MgrEpv* argument. Specifying a null argument value (or a nil UUID) registers *IfSpec* with a nil type UUID.

*MgrEpv*

Specifies the manager routines' entry-point vector (EPV). To use the MIDL-generated default EPV, specify a null value.

## Remarks

A server can register an unlimited number of interfaces with the RPC run-time library. Registration makes an interface available to clients using a binding handle to the server.

To register an interface, the server application code calls the **RpcServerRegisterIf** routine. For each implementation of an interface offered by a server, it must register a separate manager EPV.

To register an interface, the server provides the following information:

- Interface specification

  The interface specification is a data structure that the MIDL compiler generates. The server specifies the interface using the *IfSpec* argument.
- Manager type UUID and manager EPV

  The manager type UUID and the manager EPV determine which manager routine executes when a server receives a remote procedure call request from a client.

  The server specifies the manager type UUID and EPV using the *MgrTypeUuid* and *MgrEpv* arguments. Note that when specifying a non-nil manager type UUID, the server must also call the **RpcObjectSetType** routine to register objects of this non-nil type.

If your server application needs to register an auto-listen interface or use a callback function for authentication purposes, use **RpcServerRegisterIfEx**.

## See Also

Registering the Interface, **RpcBindingFromStringBinding**, **RpcBindingSetObject**,

**RpcNsBindingExport**, **RpcNsBindingImportBegin**, **RpcNsBindingLookupBegin**, **RpcObjectSetType**, **RpcServerUnregisterIf**

# RpcServerRegisterIfEx Quick Info

The **RpcServerRegisterIfEx** function registers an interface with the RPC run-time library.

This function is supported by both server platforms − Windows NT and Windows 95.

```
#include <rpc.h>

RPC_STATUS RPC_ENTRY RpcServerRegisterIfEx(

    RPC_IF_HANDLE   IfSpec,
    UUID *   MgrTypeUuid,
    RPC_MGR_EPV *   MgrEpv,
    unsigned int Flags,
    unsigned int MaxCalls,
    RPC_IF_CALLBACK_FN* IfCallback
  );
```

## Parameters

*IfSpec*

Specifies a MIDL-generated data structure indicating the interface to register.

*MgrTypeUuid*

Points to a type UUID to associate with the *MgrEpv* argument. Specifying a null argument value (or a nil UUID) registers *IfSpec* with a nil type UUID.

*MgrEpv*

Specifies the manager routines' entry-point vector (EPV). To use the MIDL-generated default EPV, specify a null value.

*Flags*

The following flags are defined:

| Value | Meaning |
|---|---|
| 0 | standard interface semantics |
| RPC_IF_AUTOLISTEN | This is an *auto-listen* interface. See **Remarks** for more details. |
| RPC_IF_OLE | Reserved for OLE. Do not use this flag. |

*MaxCalls*

Specifies the maximum number of concurrent remote procedure call requests the server can accept. The RPC run-time library guarantees that the server can accept at least this number of concurrent call requests. The actual number can be greater and can vary for each protocol sequence. Use RPC_C_PROTSEQ_MAX_REQS_DEFAULT to specify the default value.

*IfCallback*

Specifies a security callback function, or NULL for no callback. Each registered interface can have a different callback function. See **Remarks** for more details.

## Remarks

The parameters and effects of **RpcServerRegisterIfEx** subsume those of **RpcServerRegisterIf**. The

difference is the ability to register an auto-listen interface and to specify a security callback function.

The server application code calls the **RpcServerRegisterIfEx** routine to register an interface. To register an interface, the server provides the following information:

- Interface specification.

  The interface specification is a data structure that the MIDL compiler generates.
- Manager type UUID and manager EPV

  The manager type UUID and the manager EPV determine which manager routine executes when a server receives a remote procedure call request from a client. For each implementation of an interface offered by a server, it must register a separate manager EPV.

  Note that when specifying a non-nil manager type UUID, the server must also call the **RpcObjectSetType** routine to register objects of this non-nil type.

Specifying the RPC_IF_AUTOLISTEN flags marks the interface as an *auto-listen* interface. The runtime begins listening for calls as soon as the interface is registered, and stops listening when the interface is unregistered. A call to **RpcServerUnregisterIf** for this interface will wait for the completion of all pending calls on this interface. Calls to **RpcServerListen** and **RpcServerStopServerListening** will not affect the interface, nor will a call to **RpcServerUnregisterIf** with *IfSpec* == NULL. This allows a DLL to register and unregister RPC interfaces without changing the main application's RPC state.

Specifying a security callback function allows the server application to restrict access to its interfaces on a per-client basis. Remember that, by default, security is optional; the server runtime will dispatch unsecured calls even if the server has called **RpcServerRegisterAuthInfo**. If the server wants to accept only authenticated clients, each server stub must call **RpcServerInqAuthClient** to retrieve the security level, or attempt to impersonate the client with **RpcImpersonateClient**.

When a server app specifies a security callback function for its interface(s), the RPC runtime automatically rejects unauthenticated calls to that interface. In addition, the runtime records the interfaces that each client has used. When a client makes an RPC to a heretofore unused interface, the RPC runtime will call the interface's security callback function.

The signature for the callback function is as follows:

```
typedef RPC_STATUS
RPC_IF_CALLBACK_FN (
    IN RPC_IF_ID * Interface,
    IN void * Context
    ) ;
```

*Interface* contains the UUID and version of the interface in question.

*Context* is a server binding handle representing the client. The callback function may pass this handle to **RpcImpersonateClient** or **RpcBindingServerToClient** to gain information about the client.

The callback function should return RPC_S_OK if the client is allowed to call methods in this interface. Any other return code will cause the client to receive the exception RPC_S_ACCESS_DENIED.

In some cases the RPC runtime may call the security callback function more than once per client per interface. Be sure your callback function can handle this possibility.

## See Also

Registering the Interface, **RpcBindingFromStringBinding**, **RpcBindingSetObject**, **RpcNsBindingExport**, **RpcNsBindingImportBegin**, **RpcNsBindingLookupBegin**, **RpcObjectSetType**, **RpcServerRegisterIf**, **RpcServerUnregisterIf**

# RpcServerUnregisterIf Quick Info

The **RpcServerUnregisterIf** function unregisters an interface from the RPC run-time library.

This function is supported by both 32-bit platforms − Windows NT and Windows 95.

```
#include <rpc.h>

RPC_STATUS RPC_ENTRY RpcServerUnregisterIf(

  RPC_IF_HANDLE   IfSpec,
  UUID *   MgrTypeUuid,
  unsigned int   WaitForCallsToComplete
);
```

## Parameters

*IfSpec*

Specifies the interface to unregister.

Specify a null value to unregister all interfaces previously registered with the type UUID value specified in the *MgrTypeUuid* argument.

*MgrTypeUuid*

Points to the type UUID of the manager entry-point vector (EPV) to unregister. The value of *MgrTypeUuid* should be the same value as was provided in a call to the **RpcServerRegisterIf** or **RpcServerRegisterIfEx** routine.

Specify a null value to unregister the interface specified in the *IfSpec* argument for all previously registered type UUIDs.

Specify a nil UUID to unregister the MIDL-generated default manager EPV. In this case, all manager EPVs registered with a non-nil type UUID remain registered.

*WaitForCallsToComplete*

Specifies a flag that indicates whether to unregister immediately or to wait until all current calls are complete.

Specify a value of zero to disregard calls in progress and unregister immediately. Specify any non-zero value to wait until all active calls complete.

## Remarks

A server calls the **RpcServerUnregisterIf** routine to remove the association between an interface and a manager EPV.

Specify the manager EPV to remove in the *MgrTypeUuid* argument by providing the type UUID value that was specified in a call to the **RpcServerRegisterIf** routine. Once unregistered, an interface is no longer available to client applications.

When an interface is unregistered, the RPC run-time library stops accepting new calls for that interface. Executing calls on the interface are allowed to complete, including callbacks.

The following table summarizes the behavior of **RpcServerUnregisterIf**:

| IfSpec | MgrTypeUuid | Behavior |
|---|---|---|
| Non-null | Non-null | Unregisters the manager EPV associated with the specified |

| | | |
|---|---|---|
| | | arguments. |
| Non-null | NULL | Unregisters all manager EPVs associated with the *IfSpec* argument. |
| NULL | Non-null | Unregisters all manager EPVs associated with the *MgrTypeUuid* argument. |
| NULL | NULL | Unregisters all manager EPVs. This call has the effect of preventing the server from receiving any new remote procedure calls because all the manager EPVs for all interfaces have been unregistered. |

**Note** If *IfSpec* is NULL, this function will leave auto-listen interfaces registered. Auto-listen interfaces must be individually unregistered. See **RpcServerRegisterIfEx** for more details.

## Return Values

| Value | Meaning |
|---|---|
| RPC_S_OK | Success |
| RPC_S_UNKNOWN_MGR_TYPE | Unknown manager type |
| RPC_S_UNKNOWN_IF | Unknown interface |

## See Also

RpcServerRegisterIf, RpcServerRegisterIfEx

# RpcServerUseAllProtseqs Quick Info

The **RpcServerUseAllProtseqs** function tells the RPC run-time library to use all supported protocol sequences for receiving remote procedure calls.

This function is supported by both 32-bit platforms – Windows NT and Windows 95.

For a list of Microsoft RPC's supported protocol sequences, see the reference topic String Binding in RPC Data Types and Structures.

```
#include <rpc.h>

RPC_STATUS RPC_ENTRY RpcServerUseAllProtseqs(

  unsigned int   MaxCalls,
  void *   SecurityDescriptor
);
```

## Parameters

*MaxCalls*

Specifies the maximum number of concurrent remote procedure call requests the server can accept.

The RPC run-time library guarantees that the server can accept at least this number of concurrent call requests. The actual number can be greater and can vary for each protocol sequence.

Use RPC_C_PROTSEQ_MAX_REQS_DEFAULT to specify the default value.

*SecurityDescriptor*

Points to an optional parameter provided for the Microsoft Windows NT security subsystem.Note that this parameter does not appear in the DCE specification for this API.

## Remarks

A server application calls the **RpcServerUseAllProtseqs** routine to register all of the supported protocol sequences with the RPC run-time library. To receive remote procedure calls, a server must register at least one protocol sequence with the RPC run-time library.

For each protocol sequence registered by a server, the RPC run-time library creates one or more binding handles through which the server receives remote procedure call requests. The RPC run-time library creates different binding handles for each protocol sequence. Each binding handle contains an endpoint dynamically generated by the RPC run-time library or the operating system.

The *MaxCalls* argument allows the server to specify the maximum number of concurrent remote procedure call requests the server wants to be able to handle.

See Server Application RPC API Calls for a description of the routines that a server will typically call after registering protocol sequences.

To selectively register protocol sequences, a server calls the **RpcServerUseProtseq**, **RpcServerUseProtseqIf**, or **RpcServerUseProtseqEp** routine.

## Return Values

| Value | Meaning |
|---|---|
| RPC_S_OK | Success |
| RPC_S_NO_PROTSEQS | No supported protocol |

| | |
|---|---|
| | sequences |
| RPC_S_OUT_OF_MEMORY | Insufficient memory available |
| RPC_S_INVALID_SECURITY_DESC | Security descriptor invalid |

## See Also

RpcBindingToStringBinding, RpcBindingVectorFree, RpcEpRegister, RpcEpRegisterNoReplace, RpcNsBindingExport, RpcServerInqBindings, RpcServerListen, RpcServerRegisterIf, RpcServerUseAllProtseqsIf, RpcServerUseProtseq, RpcServerUseProtseqEp, RpcServerUseProtseqIf

# RpcServerUseAllProtseqsEx Quick Info

The **RpcServerUseAllProtseqsEx** function tells the RPC run-time library to use all supported protocol sequences for receiving remote procedure calls. For a list of Microsoft RPC's supported protocol sequences, see the reference topic String Binding in RPC Data Types and Structures.

This function is supported only on Windows NT 4.0.

```
#include <rpc.h>

RPC_STATUS RPC_ENTRY RpcServerUseAllProtseqsEx(

  unsigned int   MaxCalls,
  void *   SecurityDescriptor,
  PRPC_POLICY Policy
);
```

## Parameters

*MaxCalls*

Specifies the maximum number of concurrent remote procedure call requests the server can accept.

The RPC run-time library guarantees that the server can accept at least this number of concurrent call requests. The actual number can be greater and can vary for each protocol sequence. Use RPC_C_PROTSEQ_MAX_REQS_DEFAULT to specify the default value.

*SecurityDescriptor*

Points to an optional parameter provided for the Microsoft Windows NT security subsystem.

*Policy*

Points to the RPC_POLICY structure, which allows you to override the default policies for dynamic port allocation and binding to network interface cards (NICs) on multihomed machines.

## Remarks

The parameters and effects of **RpcServerUseAllProtseqsEx** subsume those of **RpcServerUseAllProtseqs**. The difference is the *Policy* parameter, which allows you to restrict port allocation for dynamic ports and allows multihomed machines to selectively bind to specified NICs.

Setting the *NICFlags* field of the RPC_POLICY structure to zero makes this extended API functionally equivalent to the original **RpcServerUseAllProtseqs,** and the server will bind to NICs based on the settings in the system registry. For information on how the registry settings define the available Internet and intranet ports, see Configuring the Windows NT Registry for Port Allocations and Selective Binding.

**Note** The flag settings in the *Policy* field are effective only when the **ncacn_ip_tcp** protocol sequence is in use. For all other protocol sequences, the RPC run time ignores these values.

A server application calls the **RpcServerUseAllProtseqsEx** routine to register all of the supported protocol sequences with the RPC run-time library. To receive remote procedure calls, a server must register at least one protocol sequence with the RPC run-time library.

For each protocol sequence registered by a server, the RPC run-time library creates one or more binding handles through which the server receives remote procedure call requests. The RPC run-time library creates different binding handles for each protocol sequence. Each binding handle contains an endpoint

dynamically generated by the RPC run-time library or the operating system.

The *MaxCalls* argument allows the server to specify the maximum number of concurrent remote procedure call requests the server wants to handle.

To selectively register protocol sequences, a server calls the **RpcServerUseProtseqEx**, **RpcServerUseProtseqIfEx**, or **RpcServerUseProtseqEpEx** routine.

## Return Values

| Value | Meaning |
|---|---|
| RPC_S_OK | Success |
| RPC_S_NO_PROTSEQS | No supported protocol sequences |
| RPC_S_OUT_OF_MEMORY | Insufficient memory available |
| RPC_S_INVALID_SECURITY_DESC | Security descriptor is invalid |

See Server Application RPC API Calls for a description of the routines that a server will typically call after registering protocol sequences.

## See Also

Configuring the Windows NT Registry for Port Allocations and Selective Binding, **RpcServerUseAllProtseqsIfEx**, **RpcServerUseProtseqEx**, **RpcServerUseProtseqEpEx**, **RpcServerUseProtseqIfEx**

# RpcServerUseAllProtseqsIf Quick Info

The **RpcServerUseAllProtseqsIf** function tells the RPC run-time library to use all the specified protocol sequences and endpoints in the interface specification for receiving remote procedure calls.

This function is supported by both 32-bit platforms – Windows NT and Windows 95.

For a list of Microsoft RPC's supported protocol sequences, see the reference topic String Binding in RPC Data Types and Structures.

```
#include <rpc.h>

RPC_STATUS RPC_ENTRY RpcServerUseAllProtseqsIf(

  unsigned int   MaxCalls,
  RPC_IF_HANDLE   IfSpec,
  void *   SecurityDescriptor
);
```

## Parameters

*MaxCalls*

Specifies the maximum number of concurrent remote procedure call requests the server can accept.

The RPC run-time library guarantees that the server can accept at least this number of concurrent call requests. The actual number can be greater and can vary for each protocol sequence.

Use RPC_C_PROTSEQ_MAX_REQS_DEFAULT to specify the default value.

*IfSpec*

Specifies the interface containing the protocol sequences and corresponding endpoint information to use in creating binding handles.

*SecurityDescriptor*

Points to an optional parameter provided for the Microsoft Windows NT security subsystem. Note that this parameter does not appear in the DCE specification for this API.

## Remarks

A server application calls the **RpcServerUseAllProtseqsIf** routine to register with the RPC run-time library all the protocol sequences and associated endpoint-address information provided in the IDL file.

To receive remote procedure call requests, a server must register at least one protocol sequence with the RPC run-time library.

For each protocol sequence registered by a server, the RPC run-time library creates one or more binding handles through which the server receives remote procedure call requests. The RPC run-time library creates different binding handles for each protocol sequence.

The *MaxCalls* argument allows the server to specify the maximum number of concurrent remote procedure call requests the server wants to handle.

See Server Application RPC API Calls for a description of the routines that a server will typically call after registering protocol sequences.

To register selected protocol sequences specified in the IDL file, a server calls the **RpcServerUseProtseqIf** routine.

## Return Values

| Value | Meaning |
| --- | --- |
| RPC_S_OK | Success |
| RPC_S_NO_PROTSEQS | No supported protocol sequences |
| RPC_S_INVALID_ENDPOINT_FORMAT | Invalid endpoint format |
| RPC_S_OUT_OF_MEMORY | Out of memory |
| RPC_S_DUPLICATE_ENDPOINT | Endpoint is duplicate |
| RPC_S_INVALID_SECURITY_DESC | Security descriptor invalid |
| RPC_S_INVALID_RPC_PROTSEQ | RPC protocol sequence invalid |

## See Also

**RpcBindingVectorFree**, **RpcEpRegister**, **RpcEpRegisterNoReplace**, **RpcNsBindingExport**, **RpcServerInqBindings**, **RpcServerListen**, **RpcServerRegisterIfEx**, **RpcServerRegisterIf**, **RpcServerUseAllProtseqs**, **RpcServerUseProtseq**, **RpcServerUseProtseqEp**, **RpcServerUseProtseqIf**

# RpcServerUseAllProtseqsIfEx Quick Info

The **RpcServerUseAllProtseqsIfEx** function tells the RPC run-time library to use all the specified protocol sequences and endpoints in the interface specification for receiving remote procedure calls. For a list of Microsoft RPC's supported protocol sequences, see the reference topic String Binding in RPC Data Types and Structures.

This function is supported only on Windows NT 4.0.

```
#include <rpc.h>

RPC_STATUS RPC_ENTRY RpcServerUseAllProtseqsIfEx(

  unsigned int   MaxCalls,
  RPC_IF_HANDLE   IfSpec,
  void *   SecurityDescriptor,
  PRPC_POLICY Policy
);
```

## Parameters

*MaxCalls*

Specifies the maximum number of concurrent remote procedure call requests the server can accept.

The RPC run-time library guarantees that the server can accept at least this number of concurrent call requests. The actual number can be greater and can vary for each protocol sequence. Use RPC_C_PROTSEQ_MAX_REQS_DEFAULT to specify the default value.

*IfSpec*

Specifies the interface containing the protocol sequences and corresponding endpoint information to use in creating binding handles.

*SecurityDescriptor*

Points to an optional parameter provided for the Microsoft Windows NT security subsystem.

*Policy*

Points to the RPC_POLICY structure, which contains flags to restrict port allocation for dynamic ports and which allow multihomed machines to selectively bind to network interface cards.

## Remarks

The parameters and effects of **RpcServerUseAllProtseqsIfEx** subsume those of **RpcServerUseAllProtseqsIf**. The difference is the *Policy* field, which allows you to restrict port allocation for dynamic ports and allows multihomed machines to selectively bind to network interface cards.

Setting the *NICFlags* field of the RPC_POLICY structure to zero makes this extended function functionally equivalent to the original **RpcServerUseAllProtseqsIf,** and the server will bind to NICs based on the settings in the system registry. For information on how the registry settings define the available Internet and intranet ports, see Configuring the Windows NT Registry for Port Allocations and Selective Binding.

**Note** The flag settings in the *Policy* field are effective only when the **ncacn_ip_tcp** protocol sequence is in use. For all other protocol sequences, the RPC run time ignores these values.

A server application calls the **RpcServerUseAllProtseqsIfEx** routine to register with the RPC run-time library all the protocol sequences and associated endpoint-address information provided in the IDL file.

To receive remote procedure call requests, a server must register at least one protocol sequence with the RPC run-time library.

For each protocol sequence registered by a server, the RPC run-time library creates one or more binding handles through which the server receives remote procedure call requests. The RPC run-time library creates different binding handles for each protocol sequence.

The *MaxCalls* argument allows the server to specify the maximum number of concurrent remote procedure call requests the server wants to handle.

To register selected protocol sequences specified in the IDL file, a server calls the **RpcServerUseProtseqIfEx** routine.

## Return Values

| Value | Meaning |
|---|---|
| RPC_S_OK | Success |
| RPC_S_NO_PROTSEQS | No supported protocol sequences |
| RPC_S_INVALID_ENDPOINT_FORMAT | Invalid endpoint format |
| RPC_S_OUT_OF_MEMORY | Out of memory |
| RPC_S_DUPLICATE_ENDPOINT | Endpoint is a duplicate |
| RPC_S_INVALID_SECURITY_DESC | Security descriptor is invalid |
| RPC_S_INVALID_RPC_PROTSEQ | RPC protocol sequence is invalid |

See Server Application RPC API Calls for a description of the routines that a server will typically call after registering protocol sequences.

## See Also

Configuring the Windows NT Registry for Port Allocations and Selective Binding, **RpcServerUseAllProtseqsEx**, **RpcServerUseProtseqEx**, **RpcServerUseProtseqEpEx**, **RpcServerUseProtseqIfEx**

# RpcServerUseProtseq Quick Info

The **RpcServerUseProtseq** function tells the RPC run-time library to use the specified protocol sequence for receiving remote procedure calls.

This function is supported by both 32-bit platforms – Windows NT and Windows 95.

For a list of Microsoft RPC's supported protocol sequences, see the reference topic String Binding in RPC Data Types and Structures.

```
#include <rpc.h>

RPC_STATUS RPC_ENTRY RpcServerUseProtseq(

  unsigned char *   ProtSeq,
    unsigned int    MaxCalls,
  void *    SecurityDescriptor
);
```

## Parameters

*ProtSeq*

Points to a string identifier of the protocol sequence to register with the RPC run-time library.

*MaxCalls*

Specifies the maximum number of concurrent remote procedure call requests the server wants to handle.

The RPC run-time library guarantees that the server can accept at least this number of concurrent call requests. The actual number can be greater, depending on the selected protocol sequence.

Use RPC_C_PROTSEQ_MAX_REQS_DEFAULT to specify the default value.

*SecurityDescriptor*

Points to an optional parameter provided for the Microsoft Windows NT security subsystem. Note that this parameter does not appear in the DCE specification for this API.

## Remarks

A server application calls the **RpcServerUseProtseq** routine to register one protocol sequence with the RPC run-time library. To receive remote procedure call requests, a server must register at least one protocol sequence with the RPC run-time library. A server application can call **RpcServerUseProtseq** multiple times to register additional protocol sequences.

For each protocol sequence registered by a server, the RPC run-time library creates one or more binding handles through which the server receives remote procedure call requests. The RPC run-time library creates different binding handles for each protocol sequence. Each binding handle contains an endpoint dynamically generated by the RPC run-time library.

The *MaxCalls* argument allows the server to specify the maximum number of concurrent remote procedure call requests the server wants to handle.

See Server Application RPC API Calls for a description of the routines that a server will typically call after registering protocol sequences.

To register all protocol sequences, a server calls the **RpcServerUseAllProtseqs** routine.

## Return Values

| Value | Meaning |
| --- | --- |
| RPC_S_OK | Success |
| RPC_S_PROTSEQ_NOT_SUPPORTED | Protocol sequence not supported on this host |
| RPC_S_INVALID_RPC_PROTSEQ | Invalid protocol sequence |
| RPC_S_OUT_OF_MEMORY | Out of memory |
| RPC_S_INVALID_SECURITY_DESC | Security descriptor invalid |

## See Also

**RpcBindingVectorFree**, **RpcEpRegister**, **RpcEpRegisterNoReplace**, **RpcNetworkIsProtseqValid**, **RpcNsBindingExport**, **RpcServerInqBindings**, **RpcServerListen**, **RpcServerRegisterIfEx**, **RpcServerRegisterIf**, **RpcServerUseAllProtseqs**, **RpcServerUseAllProtseqsIf**, **RpcServerUseProtseqEp**, **RpcServerUseProtseqIf**

# RpcServerUseProtseqEx Quick Info

The **RpcServerUseProtseqEx** function tells the RPC run-time library to use the specified protocol sequence for receiving remote procedure calls. For a list of Microsoft RPC's supported protocol sequences, see the reference topic String Binding in RPC Data Types and Structures.

This function is supported only on Windows NT 4.0.

```
#include <rpc.h>

RPC_STATUS RPC_ENTRY RpcServerUseProtseq(

  unsigned char *   ProtSeq,
    unsigned int    MaxCalls,
  void *    SecurityDescriptor,
  PRPC_POLICY Policy
);
```

## Parameters

*ProtSeq*

Points to a string identifier of the protocol sequence to register with the RPC run-time library.

*MaxCalls*

Specifies the maximum number of concurrent remote procedure call requests the server wants to handle.

The RPC run-time library guarantees that the server can accept at least this number of concurrent call requests. The actual number can be greater, depending on the selected protocol sequence. Use RPC_C_PROTSEQ_MAX_REQS_DEFAULT to specify the default value.

*SecurityDescriptor*

Points to an optional parameter provided for the Microsoft Windows NT security subsystem.

*Policy*

Points to the RPC_POLICY structure, which contains flags to restrict port allocation for dynamic ports and which allow multihomed machines to selectively bind to network interface cards.

## Remarks

The parameters and effects of **RpcServerUseProtseqEx** subsume those of **RpcServerUseProtseq**. The difference is the *Policy* field, which allows you to restrict port allocation for dynamic ports and allows multihomed machines to selectively bind to network interface cards.

Setting the *NICFlags* field of the RPC_POLICY structure to zero makes this extended function functionally equivalent to the original **RpcServerUseProtseq,** and the server will bind to NICs based on the settings in the system registry. For information on how the registry settings define the available Internet and intranet ports, see Configuring the Windows NT Registry for Port Allocations and Selective Binding.

**Note** The flag settings in the *Policy* field are effective only when the **ncacn_ip_tcp** protocol sequence is in use. For all other protocol sequences, the RPC run time ignores these values.

A server application calls the **RpcServerUseProtseqEx** routine to register one protocol sequence with

the RPC run-time library. To receive remote procedure call requests, a server must register at least one protocol sequence with the RPC run-time library. A server application can call **RpcServerUseProtseqEx** multiple times to register additional protocol sequences.

For each protocol sequence registered by a server, the RPC run-time library creates one or more binding handles through which the server receives remote procedure call requests. The RPC run-time library creates different binding handles for each protocol sequence. Each binding handle contains an endpoint dynamically generated by the RPC run-time library.

The *MaxCalls* argument allows the server to specify the maximum number of concurrent remote procedure call requests the server wants to handle.

To register all protocol sequences, a server calls the **RpcServerUseAllProtseqsEx** routine.

## Return Values

| Value | Meaning |
|---|---|
| RPC_S_OK | Success |
| RPC_S_PROTSEQ_NOT_SUPPORTED | Protocol sequence is not supported on this host |
| RPC_S_INVALID_RPC_PROTSEQ | Invalid protocol sequence |
| RPC_S_OUT_OF_MEMORY | Out of memory |
| RPC_S_INVALID_SECURITY_DESC | Security descriptor is invalid |

See Server Application RPC API Calls for a description of the routines that a server will typically call after registering protocol sequences.

## See Also

Configuring the Windows NT Registry for Port Allocations and Selective Binding, **RpcServerUseAllProtseqsEx**, **RpcServerUseAllProtseqsIfEx**, **RpcServerUseProtseqEpEx**, **RpcServerUseProtseqIfEx**

# RpcServerUseProtseqEp Quick Info

The **RpcServerUseProtseqEp** function tells the RPC run-time library to use the specified protocol sequence combined with the specified endpoint for receiving remote procedure calls.

This function is supported by both 32-bit platforms – Windows NT and Windows 95.

For a list of Microsoft RPC's supported protocol sequences, see the reference topic String Binding in RPC Data Types and Structures.

```
#include <rpc.h>

RPC_STATUS RPC_ENTRY RpcServerUseProtseqEp(

  unsigned char *   Protseq,
    unsigned int    MaxCalls,
  unsigned char *   Endpoint,
  void *    SecurityDescriptor
);
```

## Parameters

*Protseq*

Points to a string identifier of the protocol sequence to register with the RPC run-time library.

*MaxCalls*

Specifies the maximum number of concurrent remote procedure call requests the server wants to handle.

The RPC run-time library guarantees that the server can accept at least this number of concurrent call requests. The actual number can be greater, depending on the selected protocol sequence.

Use RPC_C_PROTSEQ_MAX_REQS_DEFAULT to specify the default value.

*Endpoint*

Points to the endpoint-address information to use in creating a binding for the protocol sequence specified in the *Protseq* argument.

*SecurityDescriptor*

Points to an optional parameter provided for the Microsoft Windows NT security subsystem. Note that this parameter does not appear in the DCE specification for this API.

## Remarks

A server application calls the **RpcServerUseProtseqEp** routine to register one protocol sequence with the RPC run-time library. With each protocol sequence registration, **RpcServerUseProtseqEp** includes the specified endpoint-address information.

To receive remote procedure call requests, a server must register at least one protocol sequence with the RPC run-time library. A server application can call this routine multiple times to register additional protocol sequences and endpoints.

For each protocol sequence registered by a server, the RPC run-time library creates one or more binding handles through which the server receives remote procedure call requests.

The *MaxCalls* argument allows the server to specify the maximum number of concurrent remote

procedure call requests the server wants to be able to handle.

See Server Application RPC API Calls for a description of the routines that a server will typically call after registering protocol sequences.

## Return Values

| Value | Meaning |
|---|---|
| RPC_S_OK | Success |
| RPC_S_PROTSEQ_NOT_SUPPORTED | Protocol sequence not supported on this host |
| RPC_S_INVALID_RPC_PROTSEQ | Invalid protocol sequence |
| RPC_S_INVALID_ENDPOINT_FORMAT | Invalid endpoint format |
| RPC_S_OUT_OF_MEMORY | Out of memory |
| RPC_S_DUPLICATE_ENDPOINT | Endpoint is duplicate |
| RPC_S_INVALID_SECURITY_DESC | Security descriptor invalid |

## See Also

**RpcBindingVectorFree**, **RpcEpRegister**, **RpcEpRegisterNoReplace**, **RpcNsBindingExport**, **RpcServerInqBindings**, **RpcServerListen**, **RpcServerRegisterIf**, **RpcServerUseAllProtseqs**, **RpcServerUseAllProtseqsIf**, **RpcServerUseProtseq**, **RpcServerUseProtseqIf**

# RpcServerUseProtseqEpEx Quick Info

The **RpcServerUseProtseqEpEx** function tells the RPC run-time library to use the specified protocol sequence combined with the specified endpoint for receiving remote procedure calls. For a list of Microsoft RPC's supported protocol sequences, see the reference topic String Binding in RPC Data Types and Structures.

This function is supported only on Windows NT 4.0.

```
#include <rpc.h>

RPC_STATUS RPC_ENTRY RpcServerUseProtseqEpEx(

  unsigned char *   Protseq,
    unsigned int    MaxCalls,
  unsigned char *   Endpoint,
  void *    SecurityDescriptor,
  PRPC_POLICY Policy
 );
```

## Parameters

*Protseq*

Points to a string identifier of the protocol sequence to register with the RPC run-time library.

*MaxCalls*

Specifies the maximum number of concurrent remote procedure call requests the server wants to handle.

The RPC run-time library guarantees that the server can accept at least this number of concurrent call requests. The actual number can be greater, depending on the selected protocol sequence. Use RPC_C_PROTSEQ_MAX_REQS_DEFAULT to specify the default value.

*Endpoint*

Points to the endpoint-address information to use in creating a binding for the protocol sequence specified in the *Protseq* argument.

*SecurityDescriptor*

Points to an optional parameter provided for the Microsoft Windows NT security subsystem.

*Policy*

Points to the RPC_POLICY structure, which contains flags to restrict port allocation for dynamic ports and which allow multihomed machines to selectively bind to network interface cards.

## Remarks

The parameters and effects of **RpcServerUseProtseqEpEx** subsume those of **RpcServerUseProtseqEp**. The difference is the *Policy* field, which allows you to restrict port allocation for dynamic ports and allows multihomed machines to selectively bind to network interface cards.

Setting the *NICFlags* field of the RPC_POLICY structure to zero makes this extended function functionally equivalent to the original **RpcServerUseProtseqEp**, and the server will bind to NICs based on the settings in the system registry. For information on how the registry settings define the available Internet and intranet ports, see Configuring the Windows NT Registry for Port Allocations and Selective Binding.

**Note** The flag settings in the *Policy* field are effective only when the **ncacn_ip_tcp** protocol sequence is in use. For all other protocol sequences, the RPC run time ignores these values.

A server application calls the **RpcServerUseProtseqEpEx** routine to register one protocol sequence with the RPC run-time library. With each protocol sequence registration, **RpcServerUseProtseqEpEx** includes the specified endpoint-address information.

To receive remote procedure call requests, a server must register at least one protocol sequence with the RPC run-time library. A server application can call this routine multiple times to register additional protocol sequences and endpoints.

For each protocol sequence registered by a server, the RPC run-time library creates one or more binding handles through which the server receives remote procedure call requests.

The *MaxCalls* argument allows the server to specify the maximum number of concurrent remote procedure call requests the server wants to handle.

## Return Values

| Value | Meaning |
|---|---|
| RPC_S_OK | Success |
| RPC_S_PROTSEQ_NOT_SUPPORTED | Protocol sequence is not supported on this host |
| RPC_S_INVALID_RPC_PROTSEQ | Invalid protocol sequence |
| RPC_S_INVALID_ENDPOINT_FORMAT | Invalid endpoint format |
| RPC_S_OUT_OF_MEMORY | Out of memory |
| RPC_S_DUPLICATE_ENDPOINT | Endpoint is a duplicate |
| RPC_S_INVALID_SECURITY_DESC | Security descriptor is invalid |

See Server Application RPC API Calls for a description of the routines that a server will typically call after registering protocol sequences.

## See Also

Configuring the Windows NT Registry for Port Allocations and Selective Binding, **RpcServerUseAllProtseqsEx**, **RpcServerUseAllProtseqsIfEx**, **RpcServerUseProtseqEx**, **RpcServerUseProtseqIfEx**

# RpcServerUseProtseqIf Quick Info

The **RpcServerUseProtseqIf** function tells the RPC run-time library to use the specified protocol sequence combined with the endpoints in the interface specification for receiving remote procedure calls.

This function is supported by both 32-bit platforms – Windows NT and Windows 95.

For a list of Microsoft RPC's supported protocol sequences, see the reference topic String Binding in RPC Data Types and Structures.

```
#include <rpc.h>

RPC_STATUS RPC_ENTRY RpcServerUseProtseqIf(

  unsigned char *   Protseq,
    unsigned int    MaxCalls,
  RPC_IF_HANDLE     IfSpec,
  void *            SecurityDescriptor
);
```

## Parameters

*Protseq*

Points to a string identifier of the protocol sequence to register with the RPC run-time library.

*MaxCalls*

Specifies the maximum number of concurrent remote procedure call requests the server wants to be able to handle.

The RPC run-time library guarantees that the server can accept at least this number of concurrent call requests. The actual number can be greater, depending on the selected protocol sequence.

Use RPC_C_PROTSEQ_MAX_REQS_DEFAULT to specify the default value.

*IfSpec*

Specifies the interface containing endpoint information to use in creating a binding for the protocol sequence specified in the *Protseq* argument.

*SecurityDescriptor*

Points to an optional parameter provided for the Microsoft Windows NT security subsystem.

## Remarks

**Note** The Microsoft RPC implementation of **RpcServerUseProtseqIf** includes a new, additional parameter, *SecurityDescriptor*, that does not appear in the DCE specification.

A server application calls the **RpcServerUseProtseqIf** routine to register one protocol sequence with the RPC run-time library. With each protocol-sequence registration, the routine includes the endpoint-address information provided in the IDL file.

To receive remote procedure call requests, a server must register at least one protocol sequence with the RPC run-time library. A server application can call this routine multiple times to register additional protocol sequences.

For each protocol sequence registered by a server, the RPC run-time library creates one or more binding handles through which the server receives remote procedure call requests.

The *MaxCalls* argument allows the server to specify the maximum number of concurrent remote procedure call requests the server wants to be able to handle.

See Server Application RPC API Calls for a description of the routines that a server will typically call after registering protocol sequences.

To register all protocol sequences from the IDL file, a server calls the **RpcServerUseAllProtseqsIf** routine.

## Return Values

| Value | Meaning |
|---|---|
| RPC_S_OK | Success |
| RPC_S_PROTSEQ_NOT_FOUND | The endpoint for this protocol sequence not specified in the IDL file |
| RPC_S_PROTSEQ_NOT_SUPPORTED | Protocol sequence not supported on this host |
| RPC_S_INVALID_RPC_PROTSEQ | Invalid protocol sequence |
| RPC_S_INVALID_ENDPOINT_FORMAT | Invalid endpoint format |
| RPC_S_OUT_OF_MEMORY | Out of memory |
| RPC_S_INVALID_SECURITY_DESC | Security descriptor invalid |

## See Also

**RpcBindingVectorFree**, **RpcEpRegister**, **RpcEpRegisterNoReplace**, **RpcNsBindingExport**, **RpcServerInqBindings**, **RpcServerListen**, **RpcServerRegisterIf**, **RpcServerUseAllProtseqs**, **RpcServerUseAllProtseqsIf**, **RpcServerUseProtseq**, **RpcServerUseProtseqEp**

# RpcServerUseProtseqIfEx Quick Info

The **RpcServerUseProtseqIfEx** function tells the RPC run-time library to use the specified protocol sequence combined with the endpoints in the interface specification for receiving remote procedure calls. For a list of Microsoft RPC's supported protocol sequences, see the reference topic String Binding in RPC Data Types and Structures.

This function is supported only on Windows NT 4.0.

```
#include <rpc.h>

RPC_STATUS RPC_ENTRY RpcServerUseProtseqIfEx(

  unsigned char *   Protseq,
    unsigned int    MaxCalls,
  RPC_IF_HANDLE     IfSpec,
  void *            SecurityDescriptor,
  PRPC_POLICY       Policy
);
```

## Parameters

*Protseq*

Points to a string identifier of the protocol sequence to register with the RPC run-time library.

*MaxCalls*

Specifies the maximum number of concurrent remote procedure call requests the server wants to handle.

The RPC run-time library guarantees that the server can accept at least this number of concurrent call requests. The actual number can be greater, depending on the selected protocol sequence.

Use RPC_C_PROTSEQ_MAX_REQS_DEFAULT to specify the default value.

*IfSpec*

Specifies the interface containing endpoint information to use in creating a binding for the protocol sequence specified in the *Protseq* argument.

*SecurityDescriptor*

Points to an optional parameter provided for the Microsoft Windows NT security subsystem.

*Policy*

Points to the RPC_POLICY structure, which contains flags to restrict port allocation for dynamic ports and that allow multi-homed machines to selectively bind to network interface cards.

## Remarks

The parameters and effects of **RpcServerUseProtseqsIfEx** subsume those of **RpcServerUseProtseqsIf**. The difference is the *Policy* field, which allows you to restrict port allocation for dynamic ports and allows multi-homed machines to selectively bind to network interface cards.

Setting the *NICFlags* field of the RPC_POLICY structure to 0 makes this extended API functionally equivalent to the original **RpcServerUseProtseqIf,** and the server will bind to NICs based on the settings in the system registry. For information on how the registry settings define the available Internet and intranet ports, see Configuring the Windows NT Registry for Port Allocations and Selective Binding.

**Note** The flag settings in the *Policy* field are effective only when the **ncacn_ip_tcp** protocol sequence is in use; for all other protocol sequences, the RPC runtime ignores these values.

A server application calls the **RpcServerUseProtseqIfEx** routine to register one protocol sequence with the RPC run-time library. With each protocol-sequence registration, the routine includes the endpoint-address information provided in the IDL file.

To receive remote procedure call requests, a server must register at least one protocol sequence with the RPC run-time library. A server application can call this routine multiple times to register additional protocol sequences.

For each protocol sequence registered by a server, the RPC run-time library creates one or more binding handles through which the server receives remote procedure call requests.

The *MaxCalls* argument allows the server to specify the maximum number of concurrent remote procedure call requests the server wants to handle.

To register all protocol sequences from the IDL file, a server calls the **RpcServerUseAllProtseqsIfEx** routine.

## Return Values

| Value | Meaning |
|---|---|
| RPC_S_OK | Success |
| RPC_S_PROTSEQ_NOT_FOUND | The endpoint for this protocol sequence is not specified in the IDL file |
| RPC_S_PROTSEQ_NOT_SUPPORTED | Protocol sequence is not supported on this host |
| RPC_S_INVALID_RPC_PROTSEQ | Invalid protocol sequence |
| RPC_S_INVALID_ENDPOINT_FORMAT | Invalid endpoint format |
| RPC_S_OUT_OF_MEMORY | Out of memory |
| RPC_S_INVALID_SECURITY_DESC | Security descriptor is invalid |

See Server Application RPC API Calls for a description of the routines that a server will typically call after registering protocol sequences.

## See Also

Configuring the Windows NT Registry for Port Allocations and Selective Binding
**RpcServerUseAllProtseqsEx**, **RpcServerUseAllProtseqsIfEx**, **RpcServerUseProtseqEx**, **RpcServerUseProtseqEpEx**

# RpcSmAllocate Quick Info

The **RpcSmAllocate** function allocates memory within the RPC stub memory management function and returns a pointer to the allocated memory or NULL.

```
#include <rpc.h>

void * RPC_ENTRY RpcSmAllocate(

  size_t Size,
  RPC_STATUS* pStatus
);
```

## Parameters

*Size*

Specifies the size of memory to allocate (in bytes).

*pStatus*

Specifies a pointer to the returned status.

## Remarks

The **RpcSmAllocate** routine allows an application to allocate memory within the RPC stub memory management environment. Prior to calling **RpcSmAllocate**, the memory management environment must already be established. For memory management called within the stub, the server stub itself may establish the necessary environment. See **RpcSmEnableAllocate** for more information. When using **RpcSmAllocate** to allocate memory not called from the stub, the application must call **RpcSmEnableAllocate** to establish the required memory management environment.

The **RpcSmAllocate** routine returns a pointer to the allocated memory if the call is successful. Otherwise, a NULL is returned.

When the stub establishes the memory management, it frees any memory allocated by **RpcSmAllocate**. The application can free such memory before returning to the calling stub by calling **RpcSmFree**.

By contrast, when the application establishes the memory management, it must free any memory allocated. It does so by calling either **RpcSmFree** or **RpcSmDisableAllocate**.

To manage the same memory within the stub memory management environment, multiple threads can call **RpcSmAllocate** and **RpcSmFree**. In this case, the threads must share the same stub memory management thread handle. Applications pass thread handles from thread to thread by calling **RpcSmGetThreadHandle** and **RpcSmSetThreadHandle**.

See **Memory Management** for a complete discussion of the various memory management conditions supported by RPC.

## Return Values

| Value | Meaning |
|---|---|
| RPC_S_OK | Success |
| RPC_S_OUT_OF_MEMORY | Out of Memory |

## See Also

**RpcSmEnableAllocate**, **RpcSmDisableAllocate**, **RpcSmFree**, **RpcSmGetThreadHandle**, **RpcSmSetThreadHandle**

# RpcSmClientFree Quick Info

The **RpcSmClientFree** function frees memory returned from a client stub.

```
#include <rpc.h>

RPC_STATUS RPC_ENTRY RpcSmClientFree(

  void  *  NodeToFree
);
```

## Parameters

*NodeToFree*

Specifies a pointer to memory returned from a client stub.

## Remarks

The **RpcSmClientFree** routine releases memory allocated and returned from a client stub. The memory management handle of the thread calling this routine must match the handle of the thread that made the RPC call. Use **RpcSmGetThreadHandle** and **RpcSmSetThreadHandle** to pass handles from thread to thread.

Note that using **RpcSmClientFree** allows a routine to free dynamically-allocated memory returned by an RPC call without knowing the memory management environment from which it was called.

## Return Values

| Value | Meaning |
|---|---|
| RPC_S_OK | Success |

## See Also

**RpcSmFree**, **RpcSmGetThreadHandle**, **RpcSmSetClientAllocFree**, **RpcSmSetThreadHandle**, **RpcSmSwapClientAllocFree**

# RpcSmDestroyClientContext Quick Info

The **RpcSmDestroyClientContext** function reclaims the client memory resources for a context handle and makes the context handle NULL.

```
#include <rpc.h>

RPC_STATUS RPC_ENTRY RpcSmDestroyClientContext(

  void* *   ContextHandle
);
```

## Parameters

*ContextHandle*

Specifies the context handle that can no longer be used.

## Remarks

The **RpcSmDestroyClientContext** routine is used by client applications to reclaim resources used for an inactive context handle. Applications can call **RpcSmDestroyClientContext** after a communications error makes the context handle unusable.

Note that when this routine reclaims the memory resources, it also makes the context handle NULL.

## Return Values

| Value | Meaning |
|---|---|
| RPC_S_OK | Success |
| RPC_X_SS_CONTEXT_MISMATCH | Invalid handle |

## See Also

RpcSmFree, RpcSmGetThreadHandle, RpcSmSetClientAllocFree, RpcSmSetThreadHandle, RpcSmSwapClientAllocFree

# RpcSmDisableAllocate Quick Info

The **RpcSmDisableAllocate** function frees resources and memory within the stub memory management environment.

```
#include <rpc.h>

RPC_STATUS RPC_ENTRY RpcSmDisableAllocate (VOID);
```

## Remarks

The **RpcSmDisableAllocate** routine frees all the resources used by a call to **RpcSmEnableAllocate**. It also releases memory that was allocated by a call to **RpcSmAllocate** after the call to **RpcSmEnableAllocate**.

Note that **RpcSmEnableAllocate** and **RpcSmDisableAllocate** must be used together as matching pairs.

## Return Values

| Value | Meaning |
|---|---|
| RPC_S_OK | Success |

## See Also

RpcSmAllocate, RpcSmEnableAllocate

# RpcSmEnableAllocate Quick Info

The **RpcSmEnableAllocate** function establishes the stub memory management environment.

```
#include <rpc.h>

RPC_STATUS RPC_ENTRY RpcSmEnableAllocate (VOID);
```

## Remarks

In cases where the stub memory management is not enabled by the server stub itself, applications call the **RpcSmEnableAllocate** routine to establish the stub memory management environment. This environment must be established prior to making a call to **RpcSmAllocate**. In OSF-compatibility (/**osf**) mode, for server manager code called from the stub, the memory management environment may be established by the server stub itself by using pointer manipulation or the **enable_allocate** attribute. In default (Microsoft-extended) mode, the environment is established only upon request by using the **enable_allocate** attribute (see The **MIDL Reference**.) Otherwise, call **RpcSmEnableAllocate** before calling **RpcSmAllocate**. See **Memory Management** for a complete discussion of the memory management conditions used by RPC.

**RpcSmGetThreadHandle** and **RpcSmSetThreadHandle**

## Return Values

| Value | Meaning |
|---|---|
| RPC_S_OK | Success |
| RPC_S_OUT_OF_MEMORY | Out of memory |

## See Also

**RpcSmAllocate**, **RpcSmDisableAllocate**

# RpcSmFree Quick Info

The **RpcSmFree** function releases memory allocated by **RpcSmAllocate**.

```
#include <rpc.h>

RPC_STATUS RPC_ENTRY RpcSmFree(

  void  *  NodeToFree
);
```

## Parameters

*NodeToFree*

Specifies a pointer to memory allocated by **RpcSmAllocate** or **RpcSsAllocate**.

## Remarks

The **RpcSmFree** routine is used by applications to free memory allocated by **RpcSmAllocate**. In cases where the stub allocates the memory for the application, the **RpcSmFree** routine can also be used to release memory. See **Memory Management** for a complete discussion of memory management conditions supported by RPC.

Note that the handle of the thread calling **RpcSmFree** must match the handle of the thread that allocated the memory by calling **RpcSmAllocate**. Use **RpcSmGetThreadHandle** and **RpcSmSetThreadHandle** to pass handles from thread to thread.

## Return Values

| Value | Meaning |
|---|---|
| RPC_S_OK | Success |

## See Also

RpcSmAllocate, RpcSmGetThreadHandle, RpcSmSetThreadHandle

# RpcSmGetThreadHandle Quick Info

The **RpcSmGetThreadHandle** function returns a thread handle, or NULL, for the stub memory management environment.

```
#include <rpc.h>

RPC_SS_THREAD_HANDLE RPC_ENTRY RpcSmGetThreadHandle (

  RPC_STATUS *   pStatus
);
```

## Parameters

*pStatus*

Specifies a pointer to the returned status.

## Remarks

The **RpcSmGetThreadHandle** routine is called by applications to obtain a thread handle for the stub memory management environment. A thread used to manage memory for the stub memory management environment uses **RpcSmGetThreadHandle** to receive a handle for its memory environment. In this way, another thread that calls **RpcSmSetThreadHandle** by using this handle can then use the same memory management environment.

The same memory management thread handle must be used by multiple threads calling **RpcSmAllocate** and **RpcSmFree** in order to manage the same memory. Before spawning new threads to manage the same memory, the thread that established the memory management environment (parent thread) calls **RpcSmGetThreadHandle** to obtain a thread handle for this environment. Then, the spawned threads call **RpcSmSetThreadHandle** with the new manager handle provided by the parent thread.

Note that the **RpcSmGetThreadHandle** routine is usually called by a server manager procedure before additional threads are spawned. The stub sets up the memory management environment for the manager procedure, and the manager calls **RpcSmGetThreadHandle** to make this environment available to the other threads.

A thread can also call **RpcSmGetThreadHandle** and **RpcSmSetThreadHandle** to save and restore its memory management environment.

## Return Values

| Value | Meaning |
|---|---|
| RPC_S_OK | Success |

## See Also

RpcSmAllocate, RpcSmFree, RpcSmSetThreadHandle

# RpcSmSetClientAllocFree Quick Info

The **RpcSmSetClientAllocFree** function enables the memory allocation and release mechanisms used by the client stubs.

```
#include <rpc.h>

RPC_STATUS RPC_ENTRY RpcSmSetClientAllocFree(

  RPC_CLIENT_ALLOC   *   pfnAllocate,
  RPC_CLIENT_FREE    *   pfnFree
);
```

## Parameters

*pfnAllocate*

Specifies the routine used to allocate memory.

*pfnFree*

Specifies the routine used to release memory and used with the routine specified by *pfnAllocate*.

## Remarks

By overriding the default routines used by the client stub to manage memory, the **RpcSmSetClientAllocFree** routine establishes the memory allocation and memory freeing mechanisms. Note that the default routines are **free** and **malloc**, unless the remote call occurs within manager code. In this case, the default memory management routines are **RpcSmFree** and **RpcSmAllocate**.

## Return Values

| Value | Meaning |
|---|---|
| RPC_S_OK | Success |
| RPC_S_OUT_OF_MEMORY | Out of memory |

## See Also

RpcSmAllocate, RpcSmFree

# RpcSmSetThreadHandle Quick Info

The **RpcSmSetThreadHandle** function sets a thread handle for the stub memory management environment.

```
#include <rpc.h>

RPC_STATUS RPC_ENTRY RpcSmSetThreadHandle (

  RPC_SS_THREAD_HANDLE   Handle
);
```

## Parameters

*Handle*

Specifies a thread handle returned by a call to **RpcSmGetThreadHandle**.

## Remarks

The **RpcSmSetThreadHandle** routine is called by an application to set a thread handle for the stub memory management environment. A thread used to manage memory for the stub memory management environment calls **RpcSmGetThreadHandle** to obtain a handle for its memory environment. In this way, another thread that calls **RpcSmSetThreadHandle** by using this handle can then use the same memory management environment.

The same memory management thread handle must be used by multiple threads calling **RpcSmAllocate** and **RpcSmFree** in order to manage the same memory. Before spawning new threads to manage the same memory, the thread that established the memory management environment (parent thread) calls **RpcSmGetThreadHandle** to obtain a thread handle for this environment. Then, the spawned threads call **RpcSmSetThreadHandle** with the new manager handle provided by the parent thread.

Note that the **RpcSmSetThreadHandle** routine is usually called by a thread spawned by a server manager procedure. The stub sets up the memory management environment for the manager procedure, and the manager calls **RpcSmGetThreadHandle** to obtain a thread handle. Then, each spawned thread calls **RpcSmGetThreadHandle** to get access to the manager's memory management environment.

A thread can also call **RpcSmGetThreadHandle** and **RpcSmSetThreadHandle** to save and restore its memory management environment.

## Return Values

| Value | Meaning |
|---|---|
| RPC_S_OK | Success |

## See Also

RpcSmAllocate, RpcSmGetThreadHandle, RpcSmFree

# RpcSmSwapClientAllocFree Quick Info

The **RpcSmSwapClientAllocFree** function exchanges the memory allocation and release mechanisms used by the client stubs with one supplied by the client.

```
#include <rpc.h>

RPC_STATUS RPC_ENTRY RpcSmSwapClientAllocFree(

  RPC_CLIENT_ALLOC   *   pfnAllocate,
  RPC_CLIENT_FREE    *   pfnFree,
  RPC_CLIENT_ALLOC **    pfnOldAllocate,
  RPC_CLIENT_FREE  **    pfnOldFree
);
```

## Parameters

*pfnAllocate*

Specifies a new routine to allocate memory.

*pfnFree*

Specifies a new routine to release memory.

*pfnOldAllocate*

Returns the previous routine to allocate memory before the call to this routine.

*pfnOldFree*

Returns the previous routine to release memory before the call to this routine.

## Remarks

The **RpcSmSwapClientAllocFree** routine exchanges the current memory allocation and memory freeing mechanisms with those supplied by the client.

## Return Values

| Value | Meaning |
|---|---|
| RPC_S_OK | Success |
| RPC_S_INVALID_ARG | Invalid argument(s) |

## See Also

RpcSmAllocate, RpcSmFree, RpcSmSetClientAllocFree

# RpcSsAllocate Quick Info

The **RpcSsAllocate** function allocates memory within the RPC stub memory management function, and returns a pointer to the allocated memory or NULL.

```
#include <rpc.h>

void __RPC_FAR * RPC_ENTRY RpcSsAllocate(

  size_t   Size
);
```

## Parameters

*Size*

Specifies the size of memory to allocate (in bytes).

## Remarks

The **RpcSsAllocate** routine allows an application to allocate memory within the RPC stub memory management function. Prior to calling **RpcSsAllocate**, the memory management environment must already be established. For memory management called within the stub, the stub itself usually establishes the necessary environment. See Chapter 8, "Memory Management," for a complete discussion of the various memory management conditions supported by RPC. When using **RpcSsAllocate** to allocate memory not called from the stub, the application must call **RpcSsEnableAllocate** to establish the required memory management environment.

The **RpcSsAllocate** routine returns a pointer to the allocated memory, if the call was successful. Otherwise, it raises an exception.

When the stub establishes the memory management, it frees any memory allocated by **RpcSsAllocate**. The application can free such memory before returning to the calling stub by calling **RpcSsFree**.

By contrast, when the application establishes the memory management, it must free any memory allocated. It does so by calling either **RpcSsFree** or **RpcSsDisableAllocate**.

To manage the same memory within the stub memory management environment, multiple threads can call **RpcSsAllocate** and **RpcSsFree**. In this case, the threads must share the same stub memory management thread handle. Applications pass thread handles from thread to thread by calling **RpcSsGetThreadHandle** and **RPCSsSetThreadHandle**.

**Note** The **RpcSsAllocate** routine raises exceptions, while the **RpcSmAllocate** routine returns the error code.

## Return Values

| Value | Meaning |
|---|---|
| RPC_S_OUT_OF_MEMORY | Out of memory |

## See Also

RpcSmAllocate, RpcSsDisableAllocate, RpcSsEnableAllocate, RpcSsFree, RpcSsGetThreadHandle, RpcSsSetThreadHandle

# RpcSsDestroyClientContext Quick Info

The **RpcSsDestroyClientContext** function destroys a context handle no longer needed by the client without contacting the server.

```
#include <rpc.h>

void RPC_ENTRY RpcSsDestroyClientContext(

  void   * * ContextHandle
);
```

## Parameters

*ContextHandle*

Specifies the context handle to be destroyed. The handle is set to NULL before **RpcSsDestroyClientContext** returns.

## Remarks

**RpcSsDestroyClientContext** is used by the client application to reclaim the memory resources used to maintain a context handle on the client. This function is used when *ContextHandle* is no longer valid, such as when a communication failure has occurred and the server is no longer available. The context handle is set to NULL.

Do not use **RpcSsDestroyClientContext** to replace a server function that closes the context handle.

## Return Values

| Value | Meaning |
|---|---|
| RPC_S_OK | Success |
| RPC_X_SS_CONTEXT_MISMATCH | Invalid context handle |

## See Also

RpcBindingReset

# RpcSsDisableAllocate Quick Info

The **RpcSsDisableAllocate** function frees resources and memory within the stub memory management environment.

```
#include <rpc.h>

void RPC_ENTRY RpcSsDisableAllocate (VOID);
```

## Remarks

The **RpcSsDisableAllocate** routine frees all the resources used by a call to **RpcSsEnableAllocate**. It also releases memory that was allocated by a call to **RpcSsAllocate** after the call to **RpcSsEnableAllocate**.

**RpcSsEnableAllocate** and **RpcSsDisableAllocate** must be used together as matching pairs.

## See Also

RpcSmDisableAllocate, RpcSsAllocate, RpcSsEnableAllocate

# RpcSsDontSerializeContext Quick Info

The **RpcSsDontSerializeContext** function disables runtime serialization of multiple calls dispatched to server manager routines on the same context handle.

```
#include <rpc.h>

void RPC_ENTRY RpcSsDontSerializeContext(void);
```

## Remarks

Normally RPC runtime serializes calls on the same context handle dispatched to server manager routines. Context handles are maintained on a per client basis and typically represent the server-side state. This means that your server manager does not have to guard against another thread from the same client changing the context or from the context running down while a call is dispatched.

Calling the **RpcSsDontSerializeContext** routine prevents RPC runtime from performing this serialization service.

**Note** A call to **RpcSsDontSerializeContext** affects the entire process and is unrevertable. If you use this routine you should expect, and provide routines to handle, asynchronous context rundowns.

# RpcSsEnableAllocate Quick Info

The **RpcSsEnableAllocate** function establishes the stub memory management environment.

```
#include <rpc.h>

void RPC_ENTRY RpcSsEnableAllocate (VOID);
```

## Remarks

In cases where the stub memory management is not enabled by the stub itself, the **RpcSsEnableAllocate** routine is called by applications to establish the stub memory management environment. This environment must be established prior to making a call to **RpcSsAllocate**. For server manager code called from the stub, the memory management environment is usually established by the stub itself. Otherwise, call **RpcSsEnableAllocate** before calling **RpcSsAllocate**. See Memory Management for a complete discussion of the memory management conditions used by RPC. To learn how spawned threads use a stub memory management environment, see **RpcSsGetThreadHandle** and **RpcSsSetThreadHandle** later in this section.

**Note** The **RpcSsEnableAllocate** routine raises exceptions, while the **RpcSmEnableAllocate** routine returns the error code.

## Return Values

| Value | Meaning |
|---|---|
| RPC_S_OUT_OF_MEMORY | Out of memory |

## See Also

**RpcSmEnableAllocate**, **RpcSsAllocate**, **RpcSsDisableAllocate**

# RpcSsFree Quick Info

The **RpcSsFree** function releases memory allocated by **RpcSsAllocate**.

```
#include <rpc.h>

void RPC_ENTRY RpcSsFree(

  void *   NodeToFree
);
```

## Parameters

*NodeToFree*

Specifies a pointer to memory allocated by **RpcSsAllocate** or **RpcSmAllocate**.

## Remarks

The **RpcSsFree** routine is used by applications to free memory allocated by **RpcSsAllocate**. In cases where the stub allocates the memory for the environment, the **RpcSsFree** routine can also be used to release memory. See **Memory Management** for a complete discussion of memory management conditions supported by RPC.

Note that the handle of the thread calling **RpcSsFree** must match the handle of the thread that allocated the memory by calling **RpcSsAllocate**. Use **RpcSsGetThreadHandle** and **RpcSsSetThreadHandle** to pass handles from thread to thread.

## See Also

**RpcSmFree**, **RpcSsAllocate**, **RpcSsGetThreadHandle**, **RpcSsSetThreadHandle**

# RpcSsGetThreadHandle Quick Info

The **RpcSsGetThreadHandle** function returns a thread handle for the stub memory management environment.

```
#include <rpc.h>

RPC_SS_THREAD_HANDLE RPC_ENTRY RpcSsGetThreadHandle (VOID);
```

## Remarks

The **RpcSsGetThreadHandle** routine is called by applications to obtain a thread handle for the stub memory management environment. A thread used to manage memory for the stub memory management environment uses **RpcSsGetThreadHandle** to receive a handle for its memory environment. In this way, another thread that calls **RpcSsSetThreadHandle** by using this handle can then use the same memory management environment.

The same thread handle must be used by multiple threads calling **RpcSsAllocate** and **RpcSsFree** in order to manage the same memory. Before spawning new threads to manage the same memory, the thread that established the memory management environment (parent thread) calls **RpcSsGetThreadHandle** to obtain a thread handle for this environment. Then, the spawned threads call **RpcSsSetThreadHandle** with the handle provided by the parent thread.

The **RpcSsGetThreadHandle** routine is usually called by a server manager procedure before additional threads are spawned. The stub sets up the memory management environment for the manager procedure, and the manager calls **RpcSsGetThreadHandle** to make this environment available to the other threads.

A thread can also call **RpcSsGetThreadHandle** and **RpcSsSetThreadHandle** to save and restore its memory management environment.

**Note** The **RpcSsGetThreadHandle** routine raises exceptions, while the **RpcSmGetThreadHandle** routine returns the error code.

## Return Values

| Value | Meaning |
|---|---|
| RPC_S_OK | Success |

## See Also

RpcSmGetThreadHandle, RpcSsAllocate, RpcSsFree, RpcSsSetThreadHandle

# RpcSsSetClientAllocFree Quick Info

The **RpcSsSetClientAllocFree** function enables the memory allocation and release mechanisms used by the client stubs.

```
#include <rpc.h>

void RPC_ENTRY RpcSsSetClientAllocFree(

  RPC_CLIENT_ALLOC *   pfnAllocate,
  RPC_CLIENT_FREE   *   pfnFree
);
```

## Parameters

*pfnAllocate*

Specifies the routine used to allocate memory.

*pfnFree*

Specifies the routine used to release memory and used with the routine specified by *pfnAllocate*.

## Remarks

By overriding the default routines used by the client stub to manage memory, the **RpcSsSetClientAllocFree** routine establishes the memory allocation and memory freeing mechanisms. Note that the default routines are **free** and **malloc**, unless the remote call occurs within manager code. In this case, the default memory management routines are **RpcSsFree** and **RpcSsAllocate**.

Note that when this routine reclaims the memory resources, it also makes the context handle NULL.

**Note** The **RpcSsSetClientAllocFree** routine raises exceptions, while the **RpcSmSetClientAllocFree** routine returns the error code.

## Return Values

| Value | Meaning |
|---|---|
| RPC_S_OUT_OF_MEMORY | Out of memory |

## See Also

RpcSmSetClientAllocFree, RpcSsAllocate, RpcSsFree

# RpcSsSetThreadHandle Quick Info

The **RpcSsSetThreadHandle** function sets a thread handle for the stub memory management environment.

```
#include <rpc.h>

void RPC_ENTRY RpcSsSetThreadHandle(

  RPC_SM_THREAD_HANDLE   Handle
);
```

## Parameters

*Handle*

Specifies a thread handle returned by a call to **RpcSsGetThreadHandle**.

## Remarks

The **RpcSsSetThreadHandle** routine is called by an application to set a thread handle for the stub memory management environment. A thread used to manage memory for the stub memory management environment calls **RpcSsGetThreadHandle** to obtain a handle for its memory environment. In this way, another thread that calls **RpcSsSetThreadHandle** by using this handle can then use the same memory management environment.

The same thread handle must be used by multiple threads calling **RpcSsAllocate** and **RpcSsFree** in order to manage the same memory. Before spawning new threads to manage the same memory, the thread that established the memory management environment (parent thread) calls **RpcSsGetThreadHandle** to obtain a thread handle for this environment. Then, the spawned threads call **RpcSsSetThreadHandle** with the handle provided by the parent thread.

The **RpcSsSetThreadHandle** routine is usually called by a thread spawned by a server manager procedure. The stub sets up the memory management environment for the manager procedure, and the manager calls **RpcSsGetThreadHandle** to obtain a thread handle. Then, each spawned thread calls **RpcSsGetThreadHandle** to get access to the manager's memory management environment.

A thread can also call **RpcSsGetThreadHandle** and **RpcSsSetThreadHandle** to save and restore its memory management environment.

**Note** The **RpcSsSetThreadHandle** routine raises exceptions, while the **RpcSmSetThreadHandle** routine returns the error code.

## See Also

**RpcSmSetThreadHandle**, **RpcSsAllocate**, **RpcSsFree**, **RpcSsGetThreadHandle**

# RpcSsSwapClientAllocFree Quick Info

The **RpcSsSwapClientAllocFree** function exchanges the memory allocation and release mechanisms used by the client stubs with one supplied by the client.

```
#include <rpc.h>

void RPC_ENTRY RpcSsSwapClientAllocFree(

  RPC_CLIENT_ALLOC*   pfnAllocate,
  RPC_CLIENT_FREE*    pfnFree,
  RPC_CLIENT_ALLOC**  pfnOldAllocate,
  RPC_CLIENT_FREE**   pfnOldFree
);
```

## Parameters

*pfnAllocate*

Specifies a new routine to allocate memory.

*pfnFree*

Specifies a new routine to release memory.

*pfnOldAllocate*

Returns the previous routine to allocate memory before the call to this routine.

*pfnOldFree*

Returns the previous routine to release memory before the call to this routine.

## Remarks

The **RpcSsSwapClientAllocFree** routine exchanges the current memory allocation and memory freeing mechanisms with those supplied by the client.

**Note** The **RpcSsSwapClientAllocFree** routine raises exceptions, while the **RpcSmSwapClientAllocFree** routine returns the error code.

## Return Values

| Value | Meaning |
|---|---|
| RPC_S_OK | Success |
| RPC_S_OUT_OF_MEMORY | Out of memory |

## See Also

RpcSmSwapClientAllocFree, RpcSsAllocate, RpcSsFree, RpcSsSetClientAllocFree

# RpcStringBindingCompose Quick Info

The **RpcStringBindingCompose** function combines the components of a string binding into a string binding.

```
#include <rpc.h>

RPC_STATUS RPC_ENTRY RpcStringBindingCompose(

  unsigned char *    ObjUuid,
    unsigned char *    ProtSeq,
  unsigned char *    NetworkAddr,
  unsigned char *    EndPoint,
  unsigned char *    Options,
  unsigned char * *    StringBinding
);
```

## Parameters

*ObjUuid*

Points to a NULL-terminated string representation of an object UUID. For example, the string "6B29FC40-CA47-1067-B31D-00DD010662DA" represents a valid UUID.

*ProtSeq*

Points to a NULL-terminated string representation of a protocol sequence.

For a list of Microsoft RPC's supported protocol sequences, see the reference topic String Binding in RPC Data Types and Structures.

*NetworkAddr*

Points to a NULL-terminated string representation of a network address. The network-address format is associated with the protocol sequence. For more information, see the RPC data types and structures reference entry for string binding.

*EndPoint*

Points to a NULL-terminated string representation of an endpoint. The endpoint format and content are associated with the protocol sequence. For example, the endpoint associated with the protocol sequence **ncacn_np** is a pipe name in the format "\pipe\pipename". For more information, see the RPC data types and structures reference entry for string binding.

*Options*

Points to a NULL-terminated string representation of network options. The option string is associated with the protocol sequence. For more information, see the RPC data types and structures reference entry for string binding.

*StringBinding*

Returns a pointer to a pointer to a NULL-terminated string representation of a binding handle.

Specify a null value to prevent **RpcStringBindingCompose** from returning the *StringBinding* argument. In this case, the application does not call the **RpcStringFree** routine. For more information, see the RPC data types and structures reference entry for string binding.

## Remarks

An application calls the **RpcStringBindingCompose** routine to combine the components of a string-

binding handle into a string-binding handle.

The RPC run-time library allocates memory for the string returned in the *StringBinding* argument. The application is responsible for calling the **RpcStringFree** routine to deallocate that memory.

Specify a null argument value or provide an empty string ("\0") for each input string that has no data.

Literal backslash characters within C-language strings must be quoted. The actual C string for the server name appears as "*servername*", and the actual C string for a pipe name appears as "*pipe**pipename*".

## Return Values

| Value | Meaning |
|---|---|
| RPC_S_OK | Success |
| RPC_S_INVALID_STRING_UUID | String representation of the UUID not valid. |

## See Also

**RpcBindingFromStringBinding**, **RpcBindingToStringBinding**, **RpcStringBindingParse**, **RpcStringFree**

# RpcStringBindingParse Quick Info

The **RpcStringBindingParse** function returns the object UUID part and the address parts of a string binding as separate strings.

```
#include <rpc.h>

RPC_STATUS RPC_ENTRY RpcStringBindingParse(

  unsigned char *   StringBinding,
  unsigned char * * ObjectUuid,
  unsigned char * * ProtSeq,
  unsigned char * * NetworkAddr,
  unsigned char * * EndPoint,
  unsigned char * * NetworkOptions
);
```

## Parameters

*StringBinding*

Points to a NULL-terminated string representation of a binding.

*ObjectUuid*

Returns a pointer to a pointer to a NULL-terminated string representation of an object UUID.

Specify a null value to prevent **RpcStringBindingParse** from returning the *ObjectUuid* argument. In this case, the application does not call the **RpcStringFree** routine.

*ProtSeq*

Returns a pointer to a pointer to a NULL-terminated string representation of a protocol sequence.

For a list of Microsoft RPC's supported protocol sequences, see the reference topic String Binding in RPC Data Types and Structures.

Specify a null value to prevent **RpcStringBindingParse** from returning the *ProtSeq* argument. In this case, the application does not call the **RpcStringFree** routine.

*NetworkAddr*

Returns a pointer to a pointer to a NULL-terminated string representation of a network address.

Specify a null value to prevent **RpcStringBindingParse** from returning the *NetworkAddr* argument. In this case, the application does not call the **RpcStringFree** routine.

*EndPoint*

Returns a pointer to a pointer to a NULL-terminated string representation of an endpoint.

Specify a null value to prevent **RpcStringBindingParse** from returning the *EndPoint* argument. In this case, the application does not call the **RpcStringFree** routine.

*NetworkOptions*

Returns a pointer to a pointer to a NULL-terminated string representation of network options.

Specify a null value to prevent **RpcStringBindingParse** from returning the *NetworkOptions* argument. In this case, the application does not call the **RpcStringFree** routine.

## Remarks

An application calls the **RpcStringBindingParse** routine to parse a string representation of a binding handle into its component fields.

The RPC run-time library allocates memory for each component string returned. The application is responsible for calling the **RpcStringFree** routine once for each returned string to deallocate the memory for that string.

If any field of the *StringBinding* argument is empty, the **RpcStringBindingParse** routine returns an empty string ("\0") in the corresponding output argument.

## Return Values

| Value | Meaning |
| --- | --- |
| RPC_S_OK | Success |
| RPC_S_INVALID_STRING_BINDING | Invalid string binding |

## See Also

**RpcBindingFromStringBinding**, **RpcBindingToStringBinding**, **RpcStringBindingCompose**, **RpcStringFree**

# RpcStringFree Quick Info

The **RpcStringFree** function frees a character string allocated by the RPC run-time library.

```
#include <rpc.h>

RPC_STATUS RPC_ENTRY RpcStringFree(

  unsigned char * *   String
);
```

## Parameters

*String*

Points to a pointer to the character string to free.

## Remarks

The **RpcStringFree** routine deallocates the memory containing a NULL-terminated character string returned by the RPC run-time library.

An application is responsible for calling **RpcStringFree** once for each character string allocated and returned by calls to other RPC run-time library routines.

## Return Values

| Value | Meaning |
|---|---|
| RPC_S_OK | Success |

## See Also

RpcBindingToStringBinding, RpcNsBindingInqEntryName, RpcStringBindingParse

# RpcTestCancel Quick Info

The **RpcTestCancel** function checks for a cancel indication.

This function is supported only by Windows NT.

```
#include <rpc.h>

RPC_STATUS RPC_ENTRY RpcTestCancel(VOID);
```

## Remarks

An application server stub calls the **RpcTestCancel** routine to determine whether the call has been cancelled. If the call has been cancelled, RPC_S_OK is returned; otherwise, another value is returned.

This routine should be called periodically by the server stub so that it can respond to cancels in a timely fashion. If the routine returns RPC_S_OK, the stub should clean up its data structures and return to the client.

## Return Values

| Value | Meaning |
|---|---|
| RPC_S_OK | Call has been cancelled |
| Other values | Call has not been cancelled |

# RpcTryExcept Quick Info

## See

RpcExcept

# RpcTryFinally Quick Info

## See

RpcFinally

# RpcWinSetYieldInfo Quick Info

The **RpcWinSetYieldInfo** function configures Microsoft Windows 3.x client applications to yield to other applications during remote procedure calls.

**Note** This function is only available for 16-bit Windows client applications. A 32-bit application should take advantage of the preemptive multitasking and multithreading support that the 32-bit Windows operating systems provide.

```
#include <rpc.h>

RPC_STATUS

RpcWinSetYieldInfo(

  HWND   hWnd,
  BOOL   fCustomYield,
  WORD   wMsg,
  DWORD  dwOtherInfo
);
```

## Parameters

*hWnd*

Identifies the application window that receives messages relating to yielding. Applications should usually specify the parent window of the dialog box.

Standard yield applications receive messages for both the start and end of the yield period. Custom yield applications receive messages that indicate when the RPC operation has completed.

*fCustomYield*

Specifies the yielding method. The following values are defined:

| Value | Yield method |
|---|---|
| TRUE | Custom yield |
| FALSE | Standard yield |

*wMsg*

Specifies the message that is posted by the RPC run-time library to notify the application of RPC events. The message value should be in the range beginning with WM_USER. If a zero value is specified, no message is posted.

For standard-yield applications, the message indicates the beginning or end of the yield period. This allows the application to refrain from performing operations that are illegal during an RPC operation. Standard-yield applications use the following values of *wParam* and *lParam* with this message:

| Parameter | Value | Description |
|---|---|---|
| *wParam* | 1 | Yield period beginning |
| *wParam* | 0 | Yield period ending |
| *lParam* | - | Unused |

For a custom-yield application, the *wMsg* message notifies the application that the RPC operation is complete. When the application receives this message, it should immediately return control to the RPC run-time library by having the callback function return. The values of *wParam* and *lParam* are set to zero and are not used.

*dwOtherInfo*

Specifies additional information about the yielding behavior.

For standard-yield applications, *dwOtherInfo* contains an optional **HANDLE** to an application-supplied dialog-box resource. This handle is passed as the second parameter to the **DialogBoxIndirect** function. If the handle specified by *dwOtherInfo* is zero, the default dialog box supplied by the RPC run-time library is used. For more informatin about **DialogBoxIndirect**, see your Windows API reference documentation.

For custom-yield applications, *dwOtherInfo* contains the procedure-instance address of the application-supplied callback function.

## Remarks

The **RpcWinSetYieldInfo** function supports two yielding methods:

- Standard yield method. The RPC run-time library provides a standard modal dialog box that includes a single push-button control with an IDCANCEL ID. The dialog box prevents direct user input, such as mouse and keyboard events, from being sent to the application. The application continues to receive messages while the dialog box is present. The IDCANCEL message indicates that the application user wants to end the remote procedure.
- Custom yield method. The application provides a callback function that the RPC run-time library calls while a remote operation is in progress. The callback function must retrieve messages from the message queue (including mouse and keyboard messages) and must process messages (both queued and non-queued). The RPC run-time library posts a message to the application's queue when the RPC operation is complete. The callback function returns a boolean value to the RPC run-time library.

When a conventional RPC client application makes a remote procedure call, the MIDL-generated stub calls the RPC run-time library and the library calls the appropriate transport. These calls are synchronous and block until the server side sends back a response. In the cooperatively multitasked Windows 3.x environment, an active, blocked application prevents Windows and other Windows applications from running. The **RpcWinSetYieldInfo** function allows you to direct the application to yield to Windows and other Windows applications while waiting for an RPC operation to finish.

Windows RPC client applications can be organized into three classes that correspond to levels of yielding support: no yielding, standard yielding, and custom yielding.

- Some applications do not yield. RPC calls block until completion.
- Standard-yield applications are RPC-aware applications that yield but do not need to perform special handling.
- Custom-yield applications are those that are RPC aware and want to perform special handling while an RPC operation is in progress.

You can replace the provided dialog-box resource with an application-specified dialog-box resource. The resource must use the same style as the default and must contain a single push-button control with an IDCANCEL ID. The dialog-box function is part of the RPC run-time library and cannot be replaced.

To yield in a well-behaved manner from within the context of a pending RPC operation, applications must observe the following rules:

- Do not make another RPC call. If the RPC run-time library detects that a new call is being made during the yielding period, it returns an error to the caller. This is particularly important if the application makes RPC calls in response to common messages, such as WM_PAINT.
- Do not exit the application. Do not close the window specified by the *hWnd* handle parameter. Your application can process WM_CLOSE messages in the window procedure and not call **DefWindowProc** during the yielding period. For more information about **DefWindowProc**, see your Windows API reference documentation.
- Return FALSE in response to WM_QUERYENDSESSION messages. Alternatively, a custom-yield application can use this message as a signal to cause *YieldFunctionName* to return FALSE to the RPC run-time library and end the yielding period.

There is no guarantee that any code that supports yielding will be invoked. Whether or not an application yields depends on the specific call, the current state of the underlying system, and the implementation of the underlying RPC transport. Applications should not rely on this code to do anything other than manage yielding.

The **RpcWinSetYieldInfo** function can be called more than once by an application. Each call simply replaces the information stored in the previous calls.

## Return Values

| Value | Meaning |
|---|---|
| RPC_S_OK | The information was set successfully. |
| RPC_S_OUT_OF_MEMORY | Memory could not be allocated to store the information for this task. |

## See Also

DefWindowProc, DialogBoxIndirect, MakeProcInstance, YieldFunctionName

# RpcWinSetYieldTimeout Quick Info

The **RpcWinSetYieldTimeout** function configures the amount of time an RPC call will wait for the server to respond before invoking the application's RPC yielding mechanism. This function is only available for Windows 3.*x* applications.

```
#include <rpc.h>

RPC_STATUS RPC_ENTRY RpcWinSetYieldTimeout (

  unsigned int   Timeout
);
```

## Parameters

*Timeout*

Specifies the timeout value in milliseconds. If this function is not called, the default is 500 milliseconds.

## Remarks

Depending on the type of yielding specified in **RpcWinSetYieldInfo**, this can either produce a dialog box or signal the application.

If the *Timeout* value is small, the yielding mechanism can be invoked too often. This results in loss of performance. Conversely, if the value specified for *Timeout* is too large, the application and system will be frozen for the timeout period. To avoid this, use timeouts in the range of 500 to 2000 milliseconds.

The **RpcWinSetYieldTimeout** function can be called more than once by an application. Each call simply replaces the information stored in the previous calls.

## Return Values

| Value | Meaning |
|---|---|
| RPC_S_OK | Success |
| RPC_S_CANNOT_SUPPORT | **RpcWinSetYieldInfo** must be called prior to **RpcWinSetYieldTimeout.** |

## See Also

**RpcWinSetYieldInfo**

# UuidCompare Quick Info

The **UuidCompare** function compares two UUIDs.

```
#include <rpc.h>

signed int RPC_ENTRY UuidCompare(

  UUID *    Uuid1,
  UUID *    Uuid2,
  RPC_STATUS *    Status
);
```

## Parameters

*Uuid1*

Specifies a pointer to a UUID. This UUID is compared with the UUID specified in the *Uuid2* argument.

*Uuid2*

Specifies a pointer to a UUID. This UUID is compared with the UUID specified in the *Uuid1* argument.

*Status*

Returns any errors that may occur, and will normally be set by the function to RPC_S_OK upon return.

## Remarks

An application calls the **UuidCompare** routine to compare two UUIDs and determine their order. To determine order, one of the following is returned:

| Returned Value | Meaning |
|---|---|
| -1 | The *Uuid1* argument is less than the *Uuid2* argument. |
| 0 | The *Uuid1* argument is equal to the *Uuid2* argument. |
| 1 | The *Uuid1* argument is greater than the *Uuid2* argument. |

## See Also

**UuidCreate**

# UuidCreate Quick Info

The **UuidCreate** function creates a new UUID.

```
#include <rpc.h>

RPC_STATUS RPC_ENTRY UuidCreate(

  UUID *    Uuid
);
```

## Parameters

*Uuid*

Returns a pointer to the created UUID.

## Remarks

An application calls the **UuidCreate** routine to create a new UUID.

## Return Values

| Value | Meaning |
|---|---|
| RPC_S_OK | Success |
| RPC_S_UUID_NO_ADDRESS | Cannot get Ethernet or token-ring hardware address for this computer |

## See Also

**UuidFromString**, **UuidToString**

# UuidCreateNil Quick Info

The **UuidCreateNil** function creates a nil-valued UUID.

```
#include <rpc.h>

RPC_ENTRY UuidCreateNil(

  UUID *    Nil_Uuid,
  RPC_STATUS *    Status
);
```

## Parameters

*Nil_Uuid*

Returns a nil-valued UUID.

*Status*

Returns any errors that may occur. The parameter is typically set by the function to RPC_S_OK upon return.

## Remarks

An application calls the **UuidCreateNil** routine to create a nil-valued UUID.

# UuidEqual Quick Info

The **UuidEqual** function determines if two UUIDs are equal.

```
#include <rpc.h>

int RPC_ENTRY UuidEqual(

  UUID *    Uuid1,
  UUID *    Uuid2,
  RPC_STATUS *    Status
);
```

## Parameters

*Uuid1*

Specifies a pointer to a UUID. This UUID is compared with the UUID specified in the *Uuid2* argument.

*Uuid2*

Specifies a pointer to a UUID. This UUID is compared with the UUID specified in the *Uuid1* argument.

*Status*

Returns any errors that may occur, and will normally be set by the function to RPC_S_OK upon return.

## Remarks

An application calls the **UuidEqual** routine to compare two UUIDs and determine whether they are equal. Upon completion, one of the following is returned:

| Returned Value | Meaning |
|---|---|
| TRUE | The *Uuid1* argument is equal to the *Uuid2* argument. |
| FALSE | The *Uuid1* argument is not equal to the *Uuid2* argument. |

## See Also

**UuidCreate**

# UuidFromString Quick Info

The **UuidFromString** function converts a string to a UUID.

```
#include <rpc.h>

RPC_STATUS RPC_ENTRY UuidFromString(

  unsigned char *    StringUuid,
  UUID *    Uuid
);
```

## Parameters

*StringUuid*

Points to a string representation of a UUID.

*Uuid*

Returns a pointer to a UUID in binary form.

## Remarks

An application calls the **UuidFromString** routine to convert a string UUID to a binary UUID.

## Return Values

| Value | Meaning |
|---|---|
| RPC_S_OK | Success |
| RPC_S_INVALID_STRING_UUID | The string UUID is invalid |

## See Also

UuidToString

# UuidHash Quick Info

The **UuidHash** function creates a hash value for a UUID.

```
#include <rpc.h>

unsigned short RPC_ENTRY UuidHash(

  UUID *    Uuid,
  RPC_STATUS *    Status
);
```

## Parameters

*Uuid*

Specifies the UUID for which a hash value is created.

*Status*

Returns any errors that may occur, and will normally be set by the function to RPC_S_OK upon return.

## Remarks

An application calls the **UuidHash** routine to generate a hash value for a specified UUID. The hash value returned is implementation dependent and may vary from implementation to implementation.

## See Also

**UuidCreate**

# UuidIsNil Quick Info

The **UuidIsNil** function determines if a UUID is a nil-valued UUID.

```
#include <rpc.h>

int RPC_ENTRY UuidIsNil(

  UUID *    Uuid,
  RPC_STATUS *    Status
);
```

## Parameters

*Uuid*

Specifies a UUID to test for nil value.

*Status*

Returns any errors that may occur, and will typically be set by the function to RPC_S_OK upon return.

## Remarks

An application calls the **UuidIsNil** routine to determine whether the specified UUID is a nil-valued UUID. This routine acts as though the application called the **UuidCreateNil** routine, and then called the **UuidEqual** routine to compare the returned nil-value UUID to the UUID specified in the *Uuid* argument.

Upon completion, one of the following is returned:

| Returned Value | Meaning |
|---|---|
| TRUE | The *Uuid* argument is a nil-valued UUID. |
| FALSE | The *Uuid* argument is not a nil-valued UUID. |

## See Also

UuidCreate

# UuidToString Quick Info

The **UuidToString** function converts a UUID to a string.

```
#include <rpc.h>

RPC_STATUS RPC_ENTRY UuidToString(

  UUID *    Uuid,
  unsigned char * *    StringUuid
);
```

## Parameters

*Uuid*

Points to a binary UUID.

*StringUuid*

Returns a pointer to a pointer to the string representation of the UUID specified in the *Uuid* argument.

Specify a null value to prevent **UuidToString** from returning the *StringUuid* argument. In this case, the application does not call the **RpcStringFree** routine.

## Remarks

An application calls **UuidToString** to convert a binary UUID to a string UUID. The RPC run-time library allocates memory for the string returned in the *StringUuid* argument. The application is responsible for calling the **RpcStringFree** routine to deallocate that memory.

## Return Values

| Value | Meaning |
|---|---|
| RPC_S_OK | Success |
| RPC_S_OUT_OF_MEMORY | No memory |

## See Also

RpcStringFree, UuidFromString

# YieldFunctionName

*YieldFunctionName* is a placeholder name for the application-supplied function name provided as a parameter to the **RpcWinSetYieldInfo** routine.

**BOOL FAR PASCAL** *YieldFunctionName***(VOID);**

## Remarks

The callback function must retrieve messages from the message queue (including mouse and keyboard messages) and must process messages, both queued and non-queued.

*YieldFunctionName* should return TRUE when the application is notified that the RPC operation has completed (by receiving the *wMsg* message). It is an error for *YieldFunctionName* to return TRUE if it has not been notified that the RPC operation has completed.

*YieldFunctionName* should return FALSE if the user wants to cancel the RPC operation in progress. The RPC run-time library then attempts to abort the current operation, which is likely to result in the RPC call returning an error to the application. Note that due to race conditions, the operation can complete successfully even if *YieldFunctionName* returns FALSE.

## See Also

**RpcWinSetYieldInfo**

## Error Codes

RPC functions can return the following Win32 error codes:

| Manifest | Description |
|---|---|
| EPT_S_CANT_CREATE | The endpoint-map database cannot be created. |
| EPT_S_CANT_PERFORM_OP | The operation cannot be performed. |
| EPT_S_INVALID_ENTRY | The entry is invalid. |
| EPT_S_NOT_REGISTERED | There are no more endpoints available from the endpoint-map database. |
| RPC_S_ACCESS_DENIED | The user does not have sufficient privilege to complete the operation. |
| RPC_S_ADDRESS_ERROR | An addressing error has occurred on the server. |
| RPC_S_ALREADY_LISTENING | The server is already listening. |
| RPC_S_ALREADY_REGISTERED | The object UUID has already been registered. |
| RPC_S_BINDING_HAS_NO_AUTH | The binding does not contain any authentication information. |
| RPC_S_BINDING_INCOMPLETE | The binding handle is a required parameter. |
| RPC_S_BUFFER_TOO_SMALL | The buffer used to transmit data is too small. |
| RPC_S_CALL_CANCELLED | The remote procedure call exceeded the cancel timeout and was cancelled. |
| RPC_S_CALL_FAILED | The remote procedure call failed. |
| RPC_S_CALL_FAILED_DNE | The remote procedure call failed and did not execute. |
| RPC_S_CALL_IN_PROGRESS | A remote procedure call is already in progress for this thread. |
| RPC_S_CANNOT_SUPPORT | The requested operation is not supported. |
| RPC_S_CANT_CREATE_ENDPOINT | The endpoint cannot be created. |
| RPC_S_COMM_FAILURE | Unable to communicate with the server. |
| RPC_S_DUPLICATE_ENDPOINT | The endpoint is a duplicate. |
| RPC_S_ENTRY_ALREADY_EXISTS | The entry already exists. |
| RPC_S_ENTRY_NOT_FOUND | The entry is not found. |

| | |
|---|---|
| RPC_S_FP_DIV_ZERO | A floating-point operation at the server has caused a divide by zero. |
| RPC_S_FP_OVERFLOW | A floating-point overflow has occurred at the server. |
| RPC_S_FP_UNDERFLOW | A floating-point underflow occurred at the server. |
| RPC_S_GROUP_MEMBER_NOT_FOUND | The group member has not been found. |
| RPC_S_INCOMPLETE_NAME | The entry name is incomplete. |
| RPC_S_INTERFACE_NOT_FOUND | The interface has not been found. |
| RPC_S_INTERNAL_ERROR | An internal error has occurred in a remote procedure call. |
| RPC_S_INVALID_ARG | The specified argument is not valid. |
| RPC_S_INVALID_AUTH_IDENTITY | The security context is invalid. |
| RPC_S_INVALID_BINDING | The binding handle is invalid. |
| RPC_S_INVALID_BOUND | The array bounds are invalid. |
| RPC_S_INVALID_ENDPOINT_FORMAT | The endpoint format is invalid. |
| RPC_S_INVALID_LEVEL | The level parameter is invalid. |
| RPC_S_INVALID_NAF_ID | The network-address family is invalid. |
| RPC_S_INVALID_NAME_SYNTAX | The name syntax is invalid. |
| RPC_S_INVALID_NET_ADDR | The network address is invalid. |
| RPC_S_INVALID_NETWORK_OPTIONS | The network options are invalid. |
| RPC_S_INVALID_OBJECT | The object is invalid. |
| RPC_S_INVALID_RPC_PROTSEQ | The RPC protocol sequence is invalid. |
| RPC_S_INVALID_SECURITY_DESC | The security descriptor is not in the valid format. |
| RPC_S_INVALID_STRING_BINDING | The string binding is invalid. |
| RPC_S_INVALID_STRING_UUID | The string UUID is invalid. |
| RPC_S_INVALID_TAG | The discriminant value does not match any of the case values. There is no default case. |
| RPC_S_INVALID_TIMEOUT | The timeout value is invalid. |
| RPC_S_INVALID_VERS_OPTION | The version option is |

| | |
|---|---|
| | invalid. |
| RPC_S_MAX_CALLS_TOO_SMALL | The maximum number of calls is too small. |
| RPC_S_NAME_SERVICE_UNAVAILABLE | The name service is unavailable. |
| RPC_S_NO_BINDINGS | There are no bindings. |
| RPC_S_NO_CALL_ACTIVE | There is no remote procedure call active in this thread. |
| RPC_S_NO_CONTEXT_AVAILABLE | No security context is available to allow impersonation. |
| RPC_S_NO_ENDPOINT_FOUND | No endpoint has been found. |
| RPC_S_NO_ENTRY_NAME | The binding does not contain an entry name. |
| RPC_S_NO_ENV_SETUP | No environment variable is set up. |
| RPC_S_NO_INTERFACES | No interfaces are registered. |
| RPC_S_NO_INTERFACES_EXPORTED | No interfaces have been exported. |
| RPC_S_NO_MORE_BINDINGS | There are no more bindings. |
| RPC_S_NO_MORE_ELEMENTS | There are no more elements. |
| RPC_S_NO_MORE_MEMBERS | There are no more members. |
| RPC_S_NO_NS_PRIVILEGE | There is no privilege for a name-service operation. |
| RPC_S_NO_PRINC_NAME | No principal name is registered. |
| RPC_S_NO_PROTSEQS | There are no protocol sequences. |
| RPC_S_NO_PROTSEQS_REGISTERED | No protocol sequences have been registered. |
| RPC_S_NOT_ALL_OBJS_UNEXPORTED | Not all objects are unexported. |
| RPC_S_NOT_CANCELLED | The thread is not cancelled. |
| RPC_S_NOT_LISTENING | The server is not listening. |
| RPC_S_NOT_RPC_ERROR | The status code requested is not valid. |
| RPC_S_NOTHING_TO_EXPORT | There is nothing to export. |
| RPC_S_OBJECT_NOT_FOUND | The object UUID has not been found. |
| RPC_S_OK | The call has completed successfully. |
| RPC_S_OUT_OF_MEMORY | The needed memory is not |

| | |
|---|---|
| | available. |
| RPC_S_OUT_OF_RESOURCES | Not enough resources are available to complete this operation. |
| RPC_S_OUT_OF_THREADS | The RPC run-time library was not able to create another thread. |
| RPC_S_PROCNUM_OUT_OF_RANGE | The procedure number is out of range. |
| RPC_S_PROTOCOL_ERROR | An RPC protocol error has occurred. |
| RPC_S_PROTSEQ_NOT_FOUND | The RPC protocol sequence has not been found. |
| RPC_S_PROTSEQ_NOT_SUPPORTED | The RPC protocol sequence is not supported. |
| RPC_S_SEC_PKG_ERROR | There is an error with the security package. |
| RPC_S_SERVER_NOT_LISTENING | The server is not listening for remote procedure calls. |
| RPC_S_SERVER_OUT_OF_MEMORY | The server has insufficient memory to complete this operation. |
| RPC_S_SERVER_TOO_BUSY | The server is too busy to complete this operation. |
| RPC_S_SERVER_UNAVAILABLE | The server is unavailable. |
| RPC_S_STRING_TOO_LONG | The string is too long. |
| RPC_S_TYPE_ALREADY_REGISTERED | The type UUID has already been registered. |
| RPC_S_UNKNOWN_AUTHN_LEVEL | The authentication level is unknown. |
| RPC_S_UNKNOWN_AUTHN_SERVICE | The authentication service is unknown. |
| RPC_S_UNKNOWN_AUTHN_TYPE | The authentication type is unknown. |
| RPC_S_UNKNOWN_AUTHZ_SERVICE | The authorization service is unknown. |
| RPC_S_UNKNOWN_IF | The interface is unknown. |
| RPC_S_UNKNOWN_MGR_TYPE | The manager type is unknown. |
| RPC_S_UNSUPPORTED_AUTHN_LEVEL | The authentication level is not supported. |
| RPC_S_UNSUPPORTED_NAME_SYNTAX | The name syntax is not supported. |
| RPC_S_UNSUPPORTED_TRANS_SYN | The transfer syntax is not supported by the server. |
| RPC_S_UNSUPPORTED_TYPE | The type UUID is not supported. |
| RPC_S_UUID_LOCAL_ONLY | The UUID that is only valid for this computer has been |

| | |
|---|---|
| | allocated. |
| RPC_S_UUID_NO_ADDRESS | No network address is available for constructing a UUID. |
| RPC_S_WRONG_KIND_OF_BINDING | The binding handle is not the correct type. |
| RPC_S_ZERO_DIVIDE | The server has attempted an integer divide by zero. |
| RPC_X_BAD_STUB_DATA | The stub has received bad data. |
| RPC_X_BYTE_COUNT_TOO_SMAL | The byte count is too small. |
| RPC_X_ENUM_VALUE_OUT_OF_RANGE | The enumeration value is out of range. |
| RPC_X_ENUM_VALUE_TOO_LARGE | The enumeration constant must be less than 65535. |
| RPC_X_INVALID_BOUND | The specified bounds of an array are inconsistent. |
| RPC_X_INVALID_TAG | The discriminant value does not match any of the case values. There is no default case. |
| RPC_X_NO_MEMORY | Insufficient memory is available. |
| RPC_X_NO_MORE_ENTRIES | The list of servers available for the **auto_handle** binding has been exhausted. |
| RPC_X_NULL_REF_POINTER | A null reference pointer has been passed to the stub. |
| RPC_X_SS_BAD_ES_VERSION | The operation for the serializing handle is not valid. |
| RPC_X_SS_CANNOT_GET_CALL_HANDLE | The stub is unable to get the call handle. |
| RPC_X_SS_CHAR_TRANS_OPEN_FAIL | The file designated by DCERPCCHARTRANS cannot be opened. |
| RPC_X_SS_CHAR_TRANS_SHORT_FILE | The file containing the character-translation table has fewer than 512 bytes. |
| RPC_X_SS_CONTEXT_DAMAGED | The context handle changed during a call. Only raised on the client side. |
| RPC_X_SS_CONTEXT_MISMATCH | The context handle does not match any known context handles. |
| RPC_X_SS_HANDLES_MISMATCH | The binding handles passed to a remote procedure call do not match. |
| RPC_X_SS_IN_NULL_CONTEXT | A null context handle is |

| | |
|---|---|
| | passed in an **in** parameter position. |
| RPC_X_SS_INVALID_BUFFER | The buffer is not valid for the operation. |
| RPC_X_SS_WRONG_ES_VERSION | The software version is incorrect. |
| RPC_X_SS_WRONG_STUB_VERSION | The stub version is incorrect. |

## Samples

The Win32 SDK includes sample programs that demonstrate a variety of RPC concepts, as follows:

- CALLBACK demonstrates use of the **callback** attribute.
- CLUUID demonstrates use of the client object UUID to enable a client to select among multiple implementations of a remote procedure.
- DATA directory contains four programs: DUNION illustrates discriminated (nonencapsulated) unions; INOUT demonstrates **in**, **out** parameters; REPAS demonstrates the **represent_as** attribute; XMIT demonstrates the **transmit_as** attribute.
- DICT is a remote splay tree-based dictionary program that uses the **implicit_handle**, **context_handle**, **in**, and **out** attributes.
- DOCTOR is an RPC psychotherapy application that demonstrates arrays, strings, and the **size_is** attribute.
- DYNEPT demonstrates a client application managing its connection to the server through dynamic endpoints.
- DYNOUT demonstrates how to allocate memory at a server for an n-byte object and pass it back to the client as an [out]-only parameter. The client then frees the memory. This technique allows the stub to call the server without knowing in advance how much data will be returned.
- HANDLES directory contains three programs, AUTO, CXHNDL, USRDEF, which demonstrate **auto_handle**, **context_handle**, and generic (user-defined) handles, respectively.
- HELLO is a client/server implementation of "Hello, world."
- INTEROP demonstrates portability between OSF DCE and Microsoft® RPC.
- MANDEL is a distributed fractal drawing program. It uses **ref** pointers, **implicit_handle** attribute, and **handle_t** primitive types.
- NS directory contains the NHELLO program, which demonstrates name service usage. The CDS directory contains the files that describe the gateway protocol to the DCE Cell Directory Service (CDS).
- OBJECT directory contains two programs that demonstrate OLE custom interfaces. CALLAS uses the **call_as** attribute to transmit a nonremotable interface. OHELLO demonstrates correct reference counting and shutdown behavior for a multiple-use local server. OHELLO also demostrates how to use the Win32 registry functions to install a local server and a proxy DLL in the registry.
- PICKLE directory contains two programs: PICKLP demonstrates data procedure serialization; PICKLT demonstrates data type serialization; both programs use the **encode** and **decode** attributes.
- PIPES demonstrates the use of the **pipe** type constructor.
- RPCSSM demonstrates the rpcss memory management model.
- RPCSVC demonstrates the implementation of a Windows NT™ service with RPC.
- STROUT demonstrates how to allocate memory at a server for a two-dimensional object (an array of pointers) and pass it back to the client as an [out]-only parameter. The client then frees the memory. This technique allows the stub to call the server without knowing in advance how much data will be returned.
  This program also allows the user to compile either for UNICODE or ANSI.
- WHELLO is a client/server "Hello, World" for 16-bit Windows. Client and server applications can also run on the same Windows NT workstation.
- YIELD demonstrates use of the **RpcWinSetYieldInfo** API function to prevent a 16-bit Windows distributed application from blocking during lengthy remote procedure calls.

You can browse through most of these files in the Win32 SDK on-line help book Code Samples | RPC. All of the source files and makefiles for these programs are located in the Win32 SDK directory mstools\samples\rpc.

# A

**aliasing**

In RPC, having two or more pointers to the same data object.

**AppleTalk Data Stream Protocol (ADSP)**

A network protocol for interprocess communication between Apple Macintosh computers and other platforms.

**attribute**

Any keyword of the Interface Definition Language that describes a property of a data type or remote procedure call.

## B

**bind**

In RPC, the process through which a logical connection is established between a client and a server in order to direct a remote procedure call to that server.

**binding**

A relationship between a client and a server that is established over a specific protocol sequence to a specific host system and endpoint. Also used as a short form of binding handle.

**binding handle**

A data structure that represents the logical connection between a client and a server.

**binding vector**

An RPC data structure that contains a list of binding handles over which a server application can receive remote procedure calls.

## C

**Cell Directory Service (CDS)**
The name-service provider for the Open Software Foundation's Distributed Computing Environment.

**client stub**
MIDL-generated C-language source code that contains all the functions necessary for the client application to make remote procedure calls using the model of a traditional function call in a standalone application. The client stub is responsible for marshaling the input parameters and unmarshaling the output parameters. See also server stub, proxy stub.

**conformant array**
In RPC, an array whose size is determined at run time by another parameter, structure field, or expression.

**connection-oriented**
Describes a communications protocol or transport that provides a virtual circuit through which data packets are received in the same order as they were transmitted. If the connection between machines fails, the application is notified. TCP and SPX are examples of connection-oriented protocols. See also datagram.

**connectionless**
See datagram.

**context rundown**
A server notification that results from an unexpected termination of the binding between client and server applications.

# D

**datagram**

Describes a communications protocol or transport in which data packets are routed independently of each other and may follow different routes and arrive in a different order from which they were sent. UDP and IPX are examples of transport-layer datagram protocols. See also connection-oriented.

**discriminated union**

(or **variant record**) A union that includes a discriminator as part of the data structure so that the currently valid data type is transmitted along with the union. See also encapsulated union, non-encapsulated union.

**Distributed Computing Environment (DCE)**

The Open Software Foundation's specification for a set of integrated services, including remote procedure calls, distributed file systems, and security services. The OSF-DCE RPC standard is the basis for Microsoft RPC.

**dynamic endpoint**

An endpoint (network-specific server address) that is requested and assigned at run time. See also well-known endpoint.

# E

**encoding services**

MIDL-generated stub routines that provide support for data encoding and decoding (also known as "pickling" or "serialization"). These services allow you to control the buffers containing the data to be marshaled and unmarshaled. See also type serialization, procedure serialization.

**endpoint**

A network-specific address of a server process for remote procedure calls. The actual name of the endpoint depends on the protocol sequence being used. See also dynamic endpoint and well-known endpoint.

**endpoint mapper**

(or endpoint-mapping service) Part of the RPC subsystem (RPCSS) that allows the run-time library to dynamically assign and resolve endpoints. See also endpoint.

**encapsulated union**

A MIDL construct that allows unions to be passed as part of a remote procedure call by embedding the union in a structure in which the discriminant is the first field of the structure, and the union is the second (and final) field of the structure. The IDL keyword **switch** specifies that a union is encapsulated. See also non-encapsulated union.

**entry point vector (EPV)**

An array of pointers to functions that implement the operations specified in the interface. Each element in the array corresponds to a function defined in the IDL file. Entry-point vectors allow distributed applications to support more than one implementation of the functions defined in the IDL file.

# F

**firewall**

A control system that prevents unauthorized users from gaining access to a local network that is connected to the Internet.

**full pointer**

In RPC, a pointer that has all the capabilities normally associated with pointers in C/C++. Specifically, a full pointer can be null and can be aliased to another pointer parameter. The **ptr** attribute designates a full pointer. See also unique pointer and reference pointer.

**fully bound handle**

A binding handle that includes endpoint information. See also partially bound handle.

# I

**idempotent**
In RPC, describes a remote procedure call that does not change a state and returns the same information each time it is called with the same input parameters.

**input parameter**
In a remote procedure call, a parameter containing data that the client application is transmitting to the server application. The **in** attribute designates an input parameter.

**Interface Definition Language (IDL)**
The OSF-DCE standard language for specifying the interface for remote procedure calls. See also MIDL.

**Internet Protocol (IP)**
A connectionless network-layer communications protocol. See also datagram.

**Internetwork Packet Exchange (IPX)**
A Novell NetWare communication protocol that uses datagram sockets to route information packets over local-area and wide-area networks.

**intranet**
A private network that uses Internet products and technologies (for example, web servers), but is not available to external Internet users.

# L

**local remote procedure call (LRPC)**

In RPC, describes a remote procedure call to another process on the same machine as the calling process.

**Locator**

The Microsoft® Windows NT® name-service provider. See also name service.

# M

**major version number**

See version number.

**manager**

In RPC, a set of server routines that implements the interface operations.

**marshaling**

The process through which operation parameters are packaged into NDR format so that they may be transmitted across process boundaries.

**Microsoft Interface Definition Language (MIDL)**

Microsoft's implementation and extension of OSF-DCE Interface Definition Language.

**minor version number**

See version number.

# N

**name service**
A service that maps names to objects and stores the name/object pairs in a database. For example, the RPC name service maps a logical name to a binding handle so client applications can refer to that logical name, rather than a protocol sequence and network address. See also nsid, CDS, Locator.

**Network Computing Architecture (NCA)**
A collection of guidelines for distributed computing. The RPC communication protocols follow these guidelines.

**Name Service Independent (NSI)**
A standard for API functions that allows a distributed application to access RPC name-service database elements through various name-service providers, such as OSF-DCE Cell Directory Service or the Microsoft Locator. See also name-service interface daemon (nsid).

**name-service interface daemon (nsid)**
A service that provides an interface between the Microsoft Locator and the OSF-DCE Cell Directory Service name service databases for RPC name-service functions.

**named pipe**
A connection-oriented protocol, based on Server Message Blocks (SMBs) and NetBIOS, used for communication between a server process and one or more client processes.

**NetBIOS Extended User Interface (NetBEUI)**
LAN Manager's native transport protocol and network device driver. See also NetBIOS.

**Network Basic Input/Output System (NetBIOS)**
A software interface between the Microsoft® MS-DOS® operating system, the I/O bus, and a local area network.

**Network Data Representation (NDR)**
A standard format used during network transmission that is independent of the data-type format on any particular computer architecture. Transmitted data includes information that specifies its NDR format.

**network address**
An address that identifies a server on a network.

**non-encapsulated union**
A discriminated union that is less restrictive than an encapsulated union in that the discriminant and the union are not tightly bound. If the union is a parameter, the discriminant is another parameter; if the union is a structure field, the discriminant is another structure field. The IDL keywords **switch_is** and **switch_type** identify the discriminant and its type. See also encapsulated union.

**non-idempotent**

In RPC, indicates that a remote procedure call cannot be executed more than once because it will return a different value or change a state.

## O

**open array**

In RPC, an array that is both conformant and varying; that is, both its size and the range of transmitted elements are determined at run time by other parameters, structure fields, or expressions.

**Open Software Foundation (OSF)**

A consortium of companies, formed to define the distributed computing environment (DCE).

**Object Description Language (ODL)**

A subset of MIDL attributes, keywords, statements, and directives used to define type libraries for OLE Automation applications.

**output parameter**

In a remote procedure call, a parameter containing data that the server application is transmitting to the client application. The **out** attribute designates an output parameter.

# P

**partially bound handle**
A binding handle that does not include endpoint information. See also fully bound handle.

**pickling**
See serialization.

**pipe**
An IDL type constructor that supports transmission of an open-ended stream of data between client and server applications.

**procedure serialization**
Data serialization that uses a MIDL-generated serialization stub to accomplish encoding and decoding of one or more types with a single procedure call. Procedure serialization is accomplished by applying the **encode** and **decode** attributes to a function prototype in the ACF file. See also type serialization.

**protocol sequence**
A character string that represents a valid combination of an RPC protocol, a network layer protocol, and a transport layer protocol. For example, the protocol sequence ncacn_ip_tcp describes an NCA connection over an Internet Protocol with a Transmission Control Protocol as transport.

**proxy stub**
MIDL-generated C or C++ language source code that contains all the functions necessary for a custom OLE interface.

# R

**RPC object**

Server instances or other resources, such a devices, database, and queues, that are operated on and managed by RPC server applications. Each object is uniquely identified by one or more object UUIDs.

**RPC Subsystem (RPCSS)**

A Windows NT subsystem that includes a variety of RPC and OLE services, including the endpoint mapper, OLE Service Control Manager (SCM), and the DCOM Object Resolver. Do not confuse this with the RPC-specific memory allocator package, RpcSs.

**reference pointer**

In RPC, the simplest pointer type. A reference pointer always points to valid storage and that storage does not change (although the contents may change). A reference pointer cannot be aliased. The **ref** attribute designates a reference pointer. See also unique pointer and full pointer.

## S

**Sequenced Packet Protocol (SPP)**
Banyan Vines connection-oriented communication protocol for routing information packets over local area networks.

**Sequenced Packet Exchange (SPX)**
A Novell NetWare connection-oriented communication protocol for routing information packets over local area and wide area networks.

**serialization**
(or **pickling**) In RPC, the process of marshaling data to (encoding) and unmarshaling data from (decoding) buffers that you control. This is in contrast to traditional RPC usage, where the stubs and the RPC runtime control the marshaling buffers. See also procedure serialization, type serialization.

**server stub**
MIDL-generated C-language source code that contains all the functions necessary for the server application to handle remote requests using local procedure calls. See also client stub.

**session**
In RPC, an established relationship between a client application and a server application. See also bind, binding handle.

**static callback function**
A remote procedure that is part of the client side of a distributed application and that a server can call to obtain information from the client. The **callback** attribute designates a static callback function.

**string binding**
A character string that consists of the object UUID, protocol sequence, network address, endpoint, and endpoint options, all of which can be used to create a binding handle to the specified server.

# T

**Transmission Control Protocol (TCP)**
A connection-oriented network transport layered on top of IP.

**type serialization**
Data serialization that uses MIDL-generated routines to size, encode, and decode objects of a specified type. The client application calls these routines to serialize the data. Type serialization is accomplished by applying the **encode** and **decode** attributes to a single data type, or to an interface, in the ACF file. See also procedure serialization.

# U

**User Datagram Protocol (UDP)**
A network transport that uses connectionless datagram sockets and is layered on top of IP.

**unbind**
In RPC, to terminate the logical connection between a client and server.

**unique pointer**
In RPC, a pointer that can be null or point to existing data and whose value can change during a remote procedure call. A unique pointer cannot be aliased. The **unique** attribute designates a unique pointer. See also full pointer, reference pointer.

**Universal Unique Identifier (UUID)**
(or **GUID**) A 128-bit value used in cross-process communication to identify entities such as client and server interfaces, manager entry-point vectors, and RPC objects. See also uuidgen.

**uuidgen**
A utility program, provided with the Win32 SDK, that uses a time value and your machine's network card ID to generate UUIDs that are guaranteed to be unique.

# V

**varying array**

In RPC, an array whose range of transmitted elements is determined at run time by another parameter, structure field, or expression. See also conformant array and open array.

**version number**

In RPC, two numbers, separated by a decimal point, that identify the version of an interface. To be compatible, the major version number (the number to the left of the decimal point) must be the same for both client and server, and the minor version number of the server must be greater than or equal to the minor version number of the client.

# W

**well-known endpoint**

An endpoint that does not change. Well-known endpoint information is stored as part of the binding handle, or within the name-service database server entry. See also dynamic endpoint.