Introduction to RPC

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 MS-DOS®, Microsoft Windows®, 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 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 programming matured as a discipline, this monolithic code was organized into procedures, and commonly used procedures were 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, group10519 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. The main procedure cannot find out how procedures A, B, and X, for example, do their work. The main procedure only calls another procedure; it has no information about how that procedure is implemented.

{ewc msdncd, EWGraphic, group10519 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 perform 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 that allows procedure libraries written in C to run on other computers. In fact, an application can link with libraries implemented using RPC without indicating to the user that it 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, 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, group10519 2 /a "SDK_a04.bmp"}

Network systems support the development of client-server applications through an interprocess communication (IPC) facility that allows the client and server to 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, then calls the appropriate procedure. The server performs all the computations needed to satisfy the request, then 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.
- When the network includes different kinds of computers, you must translate data to different internal formats.
- You must 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, and 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 nearly eliminates the need to write network-specific code, 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 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.

The 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 promulgates them. The components of the OSF distributed computing environment (DCE) are shown in the following figure.

{ewc msdncd, EWGraphic, group10519 3 /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 can't 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, 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 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. The MIDL 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 that 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.

How RPC Works

The RPC tools make it look 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, group10519 4 /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 runs, perhaps generating output parameters and a return value. When the remote procedure is complete, 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 and Microsoft 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 that 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.

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 of the CPU horsepower available on the network.

Microsoft RPC allows a process running in one address space to make a procedure call that will be 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 simplifies the development and test cycles.
- RPC hides many details of the network interface from the developer. You don't 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.

About This Guide

This guide explains the Microsoft RPC programming model, standards, and tools in detail.

A Tutorial Introduction 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. Data and Language Features demonstrates the use of standard data types. Arrays and Pointers explains what happens when pointers are used 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. Using Encoding Services describes the methods for encoding or decoding data.

The IDL and ACF Files and Run-time RPC Functions describe the Microsoft RPC tools: the Microsoft Interface Definition Language and compiler and the run-time libraries. "The 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. "Run-time RPC Functions" describes the functions in the run-time library that applications use to manage their own client and server binding.

Building RPC Applications provides examples of how to build distributed applications on several operating system platforms.

The following topics present the material related to RPC in alphabetical order. The MIDL Reference contains a reference for each keyword in MIDL and the ACF, as well as for important language productions and concepts. The MIDL Command-Line Reference alphabetizes information on each command-line switch and switch option recognized by the MIDL compiler. RPC Data Types and Structures defines the constants, data types, and data structures used by the RPC functions. The Function Reference lists those functions.

Installing RPC discusses installing RPC in the MS-DOS, Microsoft Windows 3.*x*, and Microsoft Windows NT environments.

Also included are Error Codes, and MIDL Compiler Errors and Warnings.

This guide offers general information about how to use the Microsoft RPC tools to develop a distributed application. For information about Microsoft Windows, see the Microsoft Windows software development kit. For information about the Microsoft Windows NT operating system, see the Microsoft Windows NT operating system software development kit. For information about Microsoft C/C++ version 7.0, see your C programming documentation.

A Tutorial Introduction

This introduction to RPC begins with a very simple application and focuses on the differences between stand-alone C programs and distributed applications that use RPC. This sample application is not meant to be an exhaustive demonstration of the capabilities of RPC. Succeeding topics explore the rich features of the Microsoft Interface Definition Language (MIDL) and offer more detailed examples. This introductory program provides a working example that quickly demonstrates distributed applications.

The application described in this tutorial prints the words "Hello, world." The client computer makes a remote procedure call to the server and the server prints the words to its standard output.

This distributed application requires two distinct executable programs – one for the client and one for the server. Like other C programs, these client and server executables will be based on C-language source files written by the developer. Unlike most C programs, however, some of the C-language source files are automatically generated by an RPC tool: the MIDL compiler.

To make this a distributed application, you will create a file that includes a function prototype for the remote procedure. The prototype is associated with attributes that describe how the data associated with the remote procedure is to be transmitted over the network. Attributes, data, and function prototypes describe the interface between the client and server. The interface is associated with a unique identifier that distinguishes this interface from all others.

You will create a file that declares a variable, the binding handle, that the client and server use to represent their logical connection through this interface.

You will also write client and server main programs that call RPC run-time functions to establish the interface.

Don't worry if you don't understand every attribute, file, and concept mentioned in this section. Successive sections discuss all the RPC components and concepts in greater detail. The purpose of this section is to provide a quick overview of the entire territory. As you build the application, you will see all the files and all the procedural steps for even the most complex distributed application.

The Client Application

A stand-alone application that can be executed on a single computer consists of a call to a single function, called **HelloProc** in the following example:

```
/* file: helloc.c (stand-alone) */

void HelloProc(unsigned char * pszString);

void main(void)
{
     unsigned char * pszString = "Hello, world";

    HelloProc(pszString);
}
```

The **HelloProc** function calls the C library function **printf** to display the text "Hello, world":

```
/* file: hellop.c */

#include <stdio.h>

void HelloProc(unsigned char * pszString)
{
    printf("%s\n", pszString);
}
```

HelloProc is defined in its own source file, HELLOP.C, so it can be compiled and linked with either a stand-alone application or the distributed application.

The Interface Definition Language File

The first step in creating a distributed application is to provide a way for the client and the server to find each other and communicate over the network by defining a formal interface using the Microsoft Interface Definition Language (MIDL). The interface consists of data types, function prototypes, attributes, and interface information.

Interface information is formally defined in its own file, which takes the extension IDL. For convenience, this example uses the same name, HELLO, for both the IDL file and the C-language file. You can also use separate names for the two files:

```
/* file: hello.idl */

[ uuid (6B29FC40-CA47-1067-B31D-00DD010662DA),
  version(1.0)
]
interface hello
{
void HelloProc([in, string] unsigned char * pszString);
}
```

The IDL file's constructs differ from constructs in C-language source files, but they are easy to use once you become familiar with them. The information provided in the IDL file replaces an enormous amount of network programming and is virtually all you need to coordinate the client and server applications.

The IDL file has two parts: the interface header and the interface body. The interface header includes information about the interface as a whole, such as its identifier and version number. It consists of the material enclosed in square brackets and ends with the keyword **interface** and the interface name. The interface header in this example includes the keywords **uuid**, **version,** and **interface**. The interface body contains data and function prototypes. The interface body is enclosed in braces.

The UUID is the universally unique identifier, a string of five groups of hexadecimal digits separated by hyphens. The five groups contain eight digits, four digits, four digits, four digits, and 12 digits, respectively. For example, "6B29FC40-CA47-1067-B31D-00DD010662DA" is a valid UUID. In the Microsoft Windows NT environment the UUID is also known as a *GUID*, or globally unique identifier. The interface UUID is generated from a utility program, **uuidgen**, that generates unique identifiers in the required format.

The interface body contains C-like data-type definitions and function prototypes that are augmented with attributes. Attributes appear in square brackets. The attributes describe how the data is to be transmitted over the network.

The interface body in this sample application contains the function prototype **HelloProc**. The single argument, *pszString*, is designated as an **in** parameter, which means it is passed from the client to the server. Parameters can also be designated as **out**, which are passed from the server to the client, or **in, out,** which are passed in both directions. These attributes tell the run-time libraries how to pass data between the client and server. The **string** attribute indicates that the parameter is a special case of character array.

Compile the IDL file using the MIDL compiler. The MIDL compiler generates C-language client and server stub files and a header file. The header file produced from the interface definition file HELLO.IDL is named, by default, HELLO.H. The generated header file includes the header file RPC.H and function prototypes from the IDL file. The header file RPC.H defines data and functions used by the generated header file:

```
/* file: hello.h (fragment) */
```

```
#include <rpc.h>

void HelloProc(unsigned char * pszString);
```

Rather than duplicate these function prototypes, the client source should include the header file that is generated from the IDL file:

```
/* file: helloc.c (distributed version) */

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

void main(void)
{
    char * pszString = "Hello, world";
    ...
    HelloProc(pszString);
    ...
}
```

The IDL file defines the network contract between the client and server. The network contract is a firm agreement about the sequence, types, and sizes of data that are to be passed over the network.

The Application Configuration File

As specified by the distributed computing environment (DCE) standard, you must also define an application configuration file, or ACF, that is processed by the MIDL compiler with the IDL file. The ACF contains RPC data and attributes that do not relate to transmitted data.

For example, a data object called the binding handle represents the connection between the client application and the server application. The client calls run-time functions to establish the valid binding handle; the handle can then be used by the run-time functions whenever the client application calls a remote procedure. The binding handle is not part of the function prototype and is not transmitted over the network, so it is defined in the ACF.

The ACF for the "Hello, world" program appears as follows:

```
/* file: hello.acf */

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

The format of the ACF is similar to that of the IDL file. Attributes appear in square brackets, followed by the keyword **interface** and the interface name. The interface name specified in the ACF must match the interface name specified in the IDL file.

The ACF contains the attribute **implicit_handle** to indicate that the handle is a global variable that is accessed by the functions in the run-time library. The **implicit_handle** keyword is associated with the handle type and the handle name; in this example, the handle is of the MIDL data type **handle_t**. The handle name *hello_IfHandle* specified in the ACF is defined in the generated header file HELLO.H. This handle will be used in calls to the client run-time library functions.

Adding Functions RPC Requires

The RPC run-time libraries often call user-supplied functions. This approach allows the MIDL compiler to generate C code automatically while still allowing you to control how operations themselves are performed.

The user-supplied functions include functions with fixed names and names that are based on IDL-file attributes or data types. For example, whenever the run-time libraries allocate or free memory, they call the user-supplied functions **midl_user_allocate** and **midl_user_free**.

The C run-time libraries in the "Hello, world" example call the memory-allocation function **midl_user_allocate** and the memory-free function **midl_user_free**. This sample application doesn't have complex memory-management requirements, so the functions are simply implemented in terms of the C-library functions **malloc** 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);
}
```

Calling the Client Functions

RPC supports two types of binding: automatic binding and application-managed binding. When you use automatic binding, you don't have to define a binding handle or make any calls to client run-time functions to validate the binding handle. When your application manages the handle, it must make these definitions and calls. Binding-Handle Types, explains these types in more detail.

In this tutorial, the client manages its connection to the server. The client must call run-time functions to bind to the server before calling the remote procedure and must unbind after all remote procedure calls are complete. The structure of a client application that manages its own connection to the server is described in the following code fragment. The remote procedure calls are sandwiched between calls to these client run-time library functions:

```
/* file: helloc.c (fragment) */

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

void main(void)
{
    RpcStringBindingCompose(...);
    RpcBindingFromStringBinding(...);

    HelloProc(pszString);    // make remote calls

    RpcStringFree(...);
    RpcBindingFree(...);
}
```

The first two run-time API calls establish the valid handle to the server. This handle is then used to make the remote procedure call. The final two run-time API calls clean up.

The Microsoft RPC functions use data structures that represent the binding, interface, protocol sequence, and endpoint. The binding is the connection between the client and server; the interface is the collection of data types and procedures made available by the server; the protocol sequence is the underlying network transport to be used for network data transfer, and the endpoint is a network name that is specific to the protocol sequence. This example uses named pipes as the protocol sequence, and the endpoint is named *\pipe\hello*.

In the following code illustration, the **RpcStringBindingCompose** function combines the protocol sequence, the network address (server name), the endpoint (pipe name), and other string elements into the form required by the next function, **RpcBindingFromStringBinding**. **RpcStringBindingCompose** also allocates memory for a character string that is large enough to hold the data. **RpcBindingFromStringBinding** uses the string to generate a handle that represents the binding between the client and the server.

After the remote procedure calls are complete, **RpcStringFree** frees memory that was allocated by **RpcStringBindingCompose** for the string-binding data structure. **RpcBindingFree** releases the handle.

The complete client application, with some code added to handle command-line input, appears as follows:

```
#include <stdlib.h>
#include <stdio.h>
#include <ctype.h>
#include "hello.h"    // header file generated by the MIDL compiler

void Usage(char * pszProgramName)
```

```
{
    fprintf(stderr, "Usage:  %s\n", pszProgramName);
    fprintf(stderr, " -p protocol_sequence\n");
    fprintf(stderr, " -n network_address\n");
    fprintf(stderr, " -e endpoint\n");
    fprintf(stderr, " -o options\n");
    fprintf(stderr, " -s string\n");
    exit(1);
}

void main(int argc, char **argv)
{
    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;
    int i;

    /* Allow the user to override settings with command line switches */
    for (i = 1; i < argc; i++) {
        if ((*argv[i] == '-') || (*argv[i] == '/')) {
              switch (tolower(*(argv[i]+1))) {
                case 'p':  // protocol sequence
                    pszProtocolSequence = argv[++i];
                    break;
                case 'n':  // network address
                    pszNetworkAddress = argv[++i];
                    break;
                case 'e':  // endpoint
                    pszEndpoint = argv[++i];
                    break;
                case 'o':  // options
                    pszOptions = argv[++i];
                    break;
                case 's':  // string
                    pszString = argv[++i];
                    break;
                case 'h':
                case '?':
                default:
                    Usage(argv[0]);
            }
         }
         else
           Usage(argv[0]);
}

    /* Use a convenience function to concatenate the elements of */
    /* the string binding into the proper sequence               */
```

```
    status = RpcStringBindingCompose(
                            pszUuid,
                            pszProtocolSequence,
                            pszNetworkAddress,
                            pszEndpoint,
                            pszOptions,
                            &pszStringBinding);
    printf("RpcStringBindingCompose returned 0x%x\n", status);
    printf("pszStringBinding = %s\n", pszStringBinding);
    if (status)
        exit(status);

    /* Set the binding handle that will */
    /* be used to bind to the server  */
    status = RpcBindingFromStringBinding(
                              pszStringBinding,
                              &hello_IfHandle);
    printf("RpcBindingFromStringBinding returned 0x%x\n", status);
    if (status)
           exit(status);

    printf("Calling the remote procedure 'HelloProc'\n");
    printf("Print the string '%s' on the server\n", pszString);

    RpcTryExcept {
        HelloProc(pszString);  // make call with user message
        printf("Calling the remote procedure 'Shutdown'\n");
        Shutdown();  // shut down the server side
    }
    RpcExcept(1) {
        ulCode = RpcExceptionCode();
        printf("Runtime reported exception 0x%lx \n", ulCode);
    }
    RpcEndExcept

    /*  The calls to the remote procedures are complete. */
    /*  Free the string and the binding handle           */
    status = RpcStringFree(&pszStringBinding);
    printf("RpcStringFree returned 0x%x\n", status);
    if (status) {
        exit(status);
    }

    status = RpcBindingFree(&hello_IfHandle);
    printf("RpcBindingFree returned 0x%x\n", status);
    if (status)
        exit(status);

    exit(0);

}
```

Building the Client Application

A distributed application requires that you take an extra preliminary step before compiling and linking the C source code: compiling the IDL and ACF files using the MIDL compiler.

You must take care to identify the operating system that will build the application, the operating system(s) that will run the client and server applications, and the network protocol sequence that will be used. These choices determine the versions of the MIDL and C compilers to use, the versions of header files to include in your applications, and the versions of the RPC run-time libraries to link with your applications.

For simplicity, we will assume that this first example will use the same operating system – Microsoft Windows NT – for the build, client, and server platforms, and that the example will use named pipes as the protocol sequence.

MIDL Compilation

The IDL file HELLO.IDL and the application configuration file HELLO.ACF are compiled using the MIDL compiler:

```
# makefile fragment
midl hello.idl
```

{ewc msdncd, EWGraphic, group10520 0 /a "SDK_a13.bmp"}

The MIDL compiler generates the header file HELLO.H and the C-language client stub file HELLO_C.C. (The MIDL compiler also produces the server stub file HELLO_S.C, but ignore this file for now.)

C Compilation

The rest of the development process is familiar: you compile the C-language sources and link them with the RPC run-time libraries for the target platform and any other libraries required by the application. The following commands compile the sample client application:

```
# makefile fragment
# CC refers to the C compiler
# CFLAGS refers to C compiler switches
# CVARS refers to variables that control #ifdef directives
#
$(CC) $(CFLAGS) $(CVARS) helloc.c
$(CC) $(CFLAGS) $(CVARS) hello_c.c
```

Note The compiler commands provided in this topic and in the sample applications are used with a specific software configuration that consists of the **nmake** utility, the Microsoft C compiler, and the Microsoft Windows NT operating system. See your compiler documentation for specific information.

Linking

The client sources are then linked with the client run-time library, the network data representation library, and the standard C run-time libraries for this platform. For a list of libraries for all platforms, see Building RPC Applications.

```
# makefile fragment
# LINK refers to the linker
# CONFLAGS refers to flags for console apps
# CONLIBS refers to libraries for console apps
#
client.exe : helloc.obj hello_c.obj
$(LINK) $(CONFLAGS) -out:client.exe helloc.obj hello_c.obj \
     $(CONLIBS) rpcrt4.lib
```

Note The linker commands and arguments may vary for your computer configuration. See your compiler documentation for more information.

Client Build Summary

The following fragment from the **nmake** utility MAKEFILE shows dependencies among the files that are used to build the client application.

```
# makefile fragment

client.exe : helloc.obj hello_c.obj
 ...
hello.h hello_c.c : hello.idl hello.acf
 ...
helloc.obj : helloc.c hello.h
 ...
hello_c.obj : hello_c.c hello.h
 ...
```

The preceding example used the default filenames that are produced by the MIDL compiler. The default name for the client stub file is formed from the name of the IDL file (without extension) and the characters _C.C. If the name (without extension) is longer than six characters, some file systems may not accept the stub file name.

The stub files do not have to use the default names. You can customize the names of the client stub using the MIDL compiler switch **/cstub**:

```
midl hello.idl -cstub mystub.c
```

If you specify your own filenames on the MIDL compiler command line, you should use these names in subsequent compile and link commands.

The Server Application

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

Depending on the size of your application and your coding standards, you can choose to implement the remote procedures in one or more separate files. In this example, the main routine appears in the source file HELLOS.C, and the remote procedure is provided in a separate file named HELLOP.C.

The benefit to organizing the remote procedures in separate filesis 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, the same source files can be compiled and linked with the server application.

Like the client-application source file, the server-application source file should include the HELLO.H header file to obtain definitions for the RPC data and functions and for the interface-specific data and functions.

Calling the Server Functions

In the following example, the server calls the functions **RpcServerUseProtseqEp** and **RpcServerRegisterIf** to make binding information available to the client. The server then calls the **RpcServerListen** function to indicate that it is waiting for client requests:

```
/* file: hellos.c (fragment) */

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

void main(void)
{
    RpcServerUseProtseqEp(...);
    RpcServerRegisterIf(...);
    RpcServerListen(...);
}
```

RpcServerUseProtseqEp identifies the server endpoint and the network protocol sequence. **RpcServerRegisterIf** registers the interface, and **RpcServerListen** instructs the server to start listening for client requests. The endpoint, interface, and other data structures related to binding are described in more detail in Binding and Handles.

The server application must also include two functions called by the server stubs, **midl_user_allocate** and **midl_user_free**. These functions allocate and free memory on the server when a remote procedure must pass parameters to the server. In the following example, **midl_user_allocate** and **midl_user_free** are implemented using the C-library functions **malloc** and **free**:

```
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);
}
```

The complete code for HELLOS.C appears as follows:

```
#include <stdlib.h>
#include <stdio.h>
#include <ctype.h>
#include "hello.h"    // header file generated by the MIDL compiler

#define PURPOSE \
"This Microsoft RPC sample program demonstrates\n\
the use of the [string] attribute. For more information\n\
about the attributes and the RPC Functions, see the\n\
RPC Programmer's Guide and Reference.\n\n"

void Usage(char * pszProgramName)
{
    fprintf(stderr, "%s", PURPOSE);
    fprintf(stderr, "Usage:  %s\n", pszProgramName);
```

```
    fprintf(stderr, " -p protocol_sequence\n");
    fprintf(stderr, " -e endpoint\n");
    fprintf(stderr, " -m maxcalls\n");
    fprintf(stderr, " -n mincalls\n");
    fprintf(stderr, " -f flag_wait_op\n");
    exit(1);
}

void main(int argc, char * argv[])
{
    RPC_STATUS status;
    unsigned char * pszProtocolSequence = "ncacn_np";
    unsigned char * pszSecurity         = NULL;
    unsigned char * pszEndpoint         = "\\pipe\\hello";
    unsigned int    cMinCalls           = 1;
    unsigned int    cMaxCalls           = 20;
    unsigned int    fDontWait           = FALSE;
    int i;

/* Allow the user to override settings with command line switches */
for (i = 1; i < argc; i++) {
    if ((*argv[i] == '-') || (*argv[i] == '/')) {
        switch (tolower(*(argv[i]+1))) {
            case 'p':  // protocol sequence
                pszProtocolSequence = argv[++i];
                break;
            case 'e': // endpoint
                pszEndpoint = argv[++i];
                break;
            case 'm': // max concurrent calls
                cMaxCalls = (unsigned int) atoi(argv[++i]);
                break;
            case 'n': // min concurrent calls
                cMinCalls = (unsigned int) atoi(argv[++i]);
                break;
            case 'f': // flag
    fDontWait = (unsigned int) atoi(argv[++i]);
    break;
            case 'h':
            case '?':
            default:
    Usage(argv[0]);
    }
}
else
    Usage(argv[0]);
}

    status = RpcServerUseProtseqEp(
                         pszProtocolSequence,
                         cMaxCalls,
                         pszEndpoint,
                         pszSecurity);  // Security descriptor
    printf("RpcServerUseProtseqEp returned 0x%x\n", status);
    if (status)
```

```
            exit(status);

    status = RpcServerRegisterIf(
                        hello_ServerIfHandle,
                        NULL,   // MgrTypeUuid
                        NULL);  // MgrEpv; null means use default
    printf("RpcServerRegisterIf returned 0x%x\n", status);
    if (status)
            exit(status);

    printf("Calling RpcServerListen\n");
    status = RpcServerListen(
                      cMinCalls,
                      cMaxCalls,
                      fDontWait);
    printf("RpcServerListen returned: 0x%x\n", status);
    if (status)
            exit(status);

    if (fDontWait) {
            printf("Calling RpcMgmtWaitServerListen\n");
            status = RpcMgmtWaitServerListen();  // wait operation
            printf("RpcMgmtWaitServerListen returned: 0x%x\n", status);
                  if (status)
                     exit(status);

    }

}

/* MIDL allocate and free */

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);
}
```

Building the Server Application

Building the server application is very similar to building the client application.

MIDL Compilation

Assume that the server stub source file was generated with the client stub file and the header file. The MIDL compiler produces all three of these files at the same time. In the following example, it's not necessary to call the MIDL compiler twice. The MIDL compiler command line appears as follows:

```
# makefile fragment
midl hello.idl
```

C Compilation

The C source files that contain the server RPC function calls and the remote procedures are compiled with the stub source file generated by the MIDL compiler:

```
$(CC) $(CFLAGS) $(CVARS) hellos.c
$(CC) $(CFLAGS) $(CVARS) hellop.c
$(CC) $(CFLAGS) $(CVARS) hello_s.c
```

Note The compiler commands provided in this topic and in the sample applications are used with a specific software configuration that consists of the **nmake** utility, the Microsoft C compiler, and the Microsoft Windows NT operating system. See your compiler documentation for specific information.

Linking

Once the C source files are compiled, the server sources are linked with the server run-time libraries and the standard C run-time libraries for the specified platform, protocol sequence (named pipes), and memory model:

```
# makefile fragment
# LINK refers to the linker
# CONFLAGS refers to flags for console apps
# CONLIBS refers to libraries for console apps
#
server.exe : hellos.obj hellop.obj hello_s.obj
$(LINK) $(CONFLAGS) -out:server.exe hellos.obj hellop.obj
 hello_s.obj $(CONLIBS) rpcrt4.lib
```

Note The linker commands and arguments may vary for your computer configuration. See your compiler documentation for more information.

Server Build Summary

The following fragment from the **nmake** utility MAKEFILE shows dependencies among the files used to build the server application. The executable relies on the server source and the server stub files. All server sources rely on the header file HELLO.H:

```
# makefile fragment

server.exe : hellos.obj hellop.obj hello_s.obj
    ...
hello.h hello_s.c : hello.idl hello.acf
    ...
hellos.obj : hellos.c hello.h
    ...
hellop.obj : hellop.c hello.h
    ...
hello_s.obj : hello_s.c hello.h
    ...
```

Stopping the Server Application

A robust server application should stop listening for clients and remove the interface from the registry shut down gracefully. The two core server functions that accomplish these goals are **RpcMgmtStopServerListening** and **RpcServerUnregisterIf**.

The server function **RpcServerListen** does not return to the calling program until an exception occurs or until the server calls **RpcMgmtStopServerListening**. In Microsoft RPC, the client cannot directly call this stop-listening function. However, you can design the server application so that the user controls the server application as a service and directs another thread of the server application to call **RpcMgmtStopServerListening**. Or you can allow the client application to shut down the server application by making a remote procedure call to a function on the server that calls **RpcMgmtStopServerListening**. The following example uses the latter approach. The new remote function **Shutdown** is added to HELLOP.C:

```
/* hellop.c fragment */

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

void Shutdown(void)
{
    RPC_STATUS status;

    status = RpcMgmtStopServerListening(NULL);
     ...
    status = RPCServerUnregisterIf(NULL, NULL, FALSE);
     ...
}
```

The single null parameter to **RpcMgmtStopServerListening** indicates that the local application should stop listening for remote procedure calls. The two null parameters to **RpcServerUnregisterIf** indicate that no interfaces are registered. The FALSE parameter indicates that the interface should be removed from the registry immediately.

Because it is a remote procedure, **Shutdown** must also be added to the interface body section of the IDL file:

```
/* file: hello.idl */

[ uuid (6B29FC40-CA47-1067-B31D-00DD010662DA),
  version(1.0)
]
interface hello
{
void HelloProc([in, string] char * pszString);
void Shutdown(void);
}
```

Finally, the client application must add the call to the **Shutdown** function:

```
/* helloc.c (fragment) */

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

void main(void)
{
    char * pszString = "Hello, world";
```

```
    RpcStringBindingCompose(...);
    RpcBindingFromStringBinding(...);

    HelloProc(pszString);
    Shutdown();

    RpcStringFree(...);
    RpcBindingFree(...);
}
```

Summary: Basic Steps in RPC Development

A distributed application consists of client-side executables and server-side executables.

The RPC development process includes two more steps than the standard development process. You must specify the interface for the remote procedure call in the IDL and ACF files, which are compiled using the MIDL compiler. The MIDL compiler produces C source files and the stub files. The development process is then the same as for any application: compile the C-language files and link the objects with libraries to create the executables.

For the "Hello, world" distributed application, the developer creates the following source files:

- HELLOC.C
- HELLOS.C
- HELLOP.C
- HELLO.IDL
- HELLO.ACF

The MIDL compiler uses the HELLO.IDL and HELLO.ACF files to generate the client stub source file HELLO_C.C, the server stub source file HELLO_S.C, and the header file HELLO.H, which includes the RPC.H header file.

The client executable is built from the client run-time library and the following C-language header and source files:

- HELLO.H
- HELLOC.C
- HELLO_C.C (stub)

The server executable is built from the server run-time library and the following C-language header and source files:

- HELLO.H
- HELLOS.C
- HELLOP.C
- HELLO_S.C (stub)

The following figure illustrates the typical application-development cycle and the extra steps required by the RPC development cycle:

{ewc msdncd, EWGraphic, group10520 1 /a "SDK_a10.bmp"}

The following list summarizes the tasks in the development process when you use Microsoft RPC:

1. Create the Interface Definition Language file that specifies interface identification and the data types and function prototypes for the remote procedures.
2. Create the application configuration file.
3. Compile the interface definition using the MIDL compiler. The MIDL compiler generates C-language stub files and the header file for client and server.
4. Include the header file generated by the MIDL compiler in the client and server applications.
5. Create a server source program that calls RPC functions to make binding information available to the client, then calls **RpcServerListen** to start listening for client requests. Provide a method to shut down the server.
6. Link the client with the client stub file and the RPC client run-time libraries.
7. Link the server with the server stub file, the remote procedures, and the RPC server run-time library.

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 flavor of C; MIDL is a strongly typed formal language that allows you to control the data transmitted over the network. MIDL is designed to be similar to C so that developers familiar with C can learn it quickly.

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

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 types **int** and **void ***.

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

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 these 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. But 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 that 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 in the lowest memory address and the most significant byte in the highest address. Another architecture assigns the least significant byte in the highest memory address associated with that data and is known as "big endian."

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	8-bit data item guaranteed to be transmitted without any change
char	8-bit unsigned character data item
double	64-bit floating-point number
float	32-bit floating-point number
handle_t	Primitive handle that can be used for RPC binding or data serializing
hyper	64-bit integer that can be declared as either signed or unsigned. (Can also be referred to as **_int64**.)
long	32-bit integer that can be declared as either signed or unsigned
short	16-bit integer that can be declared as either signed or unsigned
small	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. (To use **wchar_t**, you must specify the **/ms_ext** switch when compiling the IDL file.)

The header file RPCNDR.H provides definitions for most of these base data types. MIDL does not recognize the keyword **int** for remotable objects unless it is accompanied by one of the modifiers that specify the integer's length: **hyper**, **long**, **short**, or **small**. The **int** keyword is optional and can be omitted.

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 follows:

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

This support is offered only as an extension to MIDL. To enable the use of **wchar_t** in remote operations, you must specify the **/ms_ext** switch when you compile the IDL file.

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.

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, but as in other type definitions, **int** is not recognized as a valid type unless it is accompanied by a modifier such as **long**, **small**, or **short**.

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);
```

Directional 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**. They describe the way parameters are passed between calling and called procedures. Directional attributes can be omitted for Microsoft extensions mode (**/ms_ext**) and C-language extensions mode (**/c_ext**). If no directional attribute is provided for a parameter, the MIDL compiler assumes a default value of **in**.

Use the following example to convert the following ANSI C function prototype to a remote procedure:

```
void MyFunction(int i);
```

MIDL requires that a directional attribute be associated with the parameter in the function prototype. In addition, **int** is not a valid MIDL type, so the type of the argument must be changed to the specific integer type. The converted function appears in the IDL file as a remote procedure call prototype:

```
void MyFunction([in] short i);
```

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 follows:

```
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, group10521 0 /a "SDK_a22.bmp"}

{ewc msdncd, EWGraphic, group10521 1 /a "SDK_a23.bmp"}

{ewc msdncd, EWGraphic, group10521 2 /a "SDK_a21.bmp"}

The transmit_as Attribute

The **transmit_as** attribute allows you 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.

For MIDL 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 to the transmitted type. The function is defined as follows:

```
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 to 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 follows:

```
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 follows:

```
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 follows:

```
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 of 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;
    }
}
```

Summary of 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 allowing you to reduce the amount of data transmitted over the network, the **transmit_as** attribute can make your application more efficient.

The represent_as Attribute

The **represent_as** attribute allows you to specify how a particular remotable data type is represented for the application. Specify the name of the represented type for a known transmittable type, and supply 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.

The **represent_as** attribute is similar to the **transmit_as** attribute. However, while **transmit_as** enables you to specify a data type that will be used for transmission, **represent_as** allows you to 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. 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.

Summary of represent_as Attribute

The **represent_as** attribute offers a way to present an application with a different and perhaps non-remotable data type, rather than 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.

Summary of Data and Attributes

MIDL was designed to solve the problem of transmitting data not only between different computers but between different computer architectures.

MIDL provides well-defined base types, replacing the implementation-dependent **int** type with the specific types **small**, **short**, **long**, and **hyper**. MIDL also supports the **float**, **double**, and **handle_t** base types.

MIDL assumes that **char** types are unsigned. If your C compiler assumes the use of **signed** **char**, use the MIDL compiler command-line switch **/char** to ensure that the header file explicitly declares all characters as unsigned.

MIDL enforces strong typing; the ANSI C generic pointer type **void** ***** is not supported, requiring the application to supply a pointer to a specific type.

MIDL also forces the parameters related to a union to be either defined as related parameters or included in the same structure.

You control the way data is transmitted over the network with the **transmit_as** attribute, and the way data is presented to the application with the **represent_as** attribute.

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 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 reflects 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 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. The size value provides information that the server stub requires to allocate the memory. The size parameter can also be **in, out**.

The length_is Attribute

The **size_is** attribute allows you to specify the maximum size of the array. When this is the only attribute, all elements of the array are transmitted. Rather than sending all elements of the array, you can specify the transmitted elements using the **length_is** attribute:

```
/* 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. 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:

```
/* 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:

- The **first_is** index value cannot be smaller than zero; **last_is** cannot be greater than **max_is**.
- Don't 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 that the size is zero or greater. If MIDL was invoked with the _error bounds_check flag, then the stub raises an exception when the size is less than 0, or the transmitted length is less than 0.
- You can't use the **length_is** and **last_is** attributes at the same time. You can't 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 permits you to 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.

Strings

You can use the **string** attribute for one-dimensional-character, wide-character, 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. When declaring a string that will hold up to 80 characters, for example, 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 multi-dimensional 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 multi-dimensional arrays. The attribute applies to the least-significant dimension, such as a conformant array of strings. You can also use multi-dimensional 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 multi-dimensional array is not equivalent to multiple levels of pointers. A multi-dimensional array is a single large block of memory and should not be confused with an array of pointers. Also, ANSI C syntax only allows the most significant (leftmost) array dimension to be unspecified in a multi-dimensional 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. In a distributed application, however, 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, 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 latest and greatest 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. It can change between null and non-null values or stay the same. But as long as the data referenced by the pointer is unique to the pointer, you can reduce the amount of processing by the stubs. Designate such a pointer 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. The contents of memory can change and the data returned from the server will be written into this area on the client. Designate such a pointer, known as a reference pointer, using the **ref** attribute.

Reference Pointers

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

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

{ewc msdncd, EWGraphic, group10522 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 can't be accessed by any other pointer or any other name in the function.

Unique Pointers

Unique pointers can change in value, but like 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, group10522 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 a unique pointer points to can't be accessed by any other pointer or any other 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, which means that multiple pointers can refer to the same data, as shown in the following figure:

{ewc msdncd, EWGraphic, group10522 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 provided as part of the run-time library. You must write your own versions of these functions and link them with your application. This design lets you decide how to manage memory.

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 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.

Using Arrays, Strings, and Pointers

The Doctor program is a good example of an application that demonstrates the design tradeoffs 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 the data can require a significant amount of processing so it is processed by the server.

The Doctor program uses a character array for input and returns another character array as output. We will examine the design tradeoffs of several alternative interfaces that manage these parameters.

Each section contains a short review of the attributes or the effects of combining the attributes.

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 string	**unique**

Knowing the constraints associated with these combinations of attributes, we can examine several alternative ways to send one character array from client to server and to return another character array from server to client.

Counted Character Arrays

The **size_is** attribute indicates the upper bound of the array. 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, byte, or character arrays just as they would be with arrays of other types.

in, out, size_is

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, then use the **length_is** parameter pcbSize to transmit the array elements into this memory. The developer must make sure that 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, which do not change in value and which 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:

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

Rather than 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

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 that 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:

```
/* 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:

```
/* 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. The **string** attribute directs the stub to calculate the string size, eliminating the overhead associated with the **size_is** parameters.

Strings

The **string** attribute indicates that the parameter is a pointer to an array of type **char**, **byte**, or **w_char**. Like 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 was 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 that 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

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.

```
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. Like 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:

```
/* 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:

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

in, string and out, string

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 that 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.

```
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.

Summary of Arrays and Pointers

MIDL defines three attributes that can be applied to pointers: full, unique, and reference. The different classifications help minimize the amount of processing by the stubs and make your distributed program as efficient as possible.

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**. Like these handles, the binding handle is opaque; your application cannot use it to directly access and manipulate data about that binding. The binding 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.

This section defines some 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 Using Encoding Services.

Binding-Handle Types

MIDL provides several types of handles so you can select the handle type that is best suited for your application. See Using Encoding Services for additional information on using a primitive handle as a serializing handle.

- Handles can be parameters that are passed to the remote procedure, or they can be global data structures that don't 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 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, producing such types as explicit primitive, explicit user-defined, implicit primitive, and implicit user-defined handles. You can select the handle type that is best suited for your application.

Binding

The server registers its interface, 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. This leads to a discussion of two principal 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 can't 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. The application-managed binding category includes all other types of handles besides auto handles: primitive, user-defined, and context handles.

The following figure shows this categorization of binding handles:

{ewc msdncd, EWGraphic, group10523 0 /a "SDK_a03.bmp"}

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 don't 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. 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:

```
/* 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 need only 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:

```
/* 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 follows:

```
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 demonstrates the parameters to the **RpcNsBindingExport** function that should be used with the Microsoft Locator, which 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**:

```
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 but be unavailable because the server is temporarily off line. 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.

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 certain information to identify the server:

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

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. In the C programming language, the backslash is an escape character, so 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 that four literal backslash characters will appear in the string-binding data structure after processing by the **sprintf** function.

```
/* 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 follows:

```
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 don't have to worry about putting the ampersand and 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.

```
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.

Handles Managed by the Application

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, however, these are not binding handles. See Using Encoding Services for additional information on serialization handles.

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, since an initial binding must be used before the server can return a context handle.

You 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. First, we define a structure that contains the file name and the file handle, as follows:

```
/* 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.

```
/* 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);
}
```

You must supply a context rundown routine that can be invoked when the connection is lost. A context handle will run down when the connection has closed and no RPC calls are in progress using the context handle. The context rundown routine uses the following syntax:

void *type***_rundown(***type***)**

type
Specifies the context-handle type.

In the following example, the context rundown routine cleans up by closing the file handle.

```
/* The rundown routine is associated with the contxt handle type. */

void __RPC_USER PCONTEXT_HANDLE_TYPE_rundown(PCONTEXT_HANDLE_TYPE phContext)
{
    FILE_CONTEXT_TYPE *pFileContext;

    printf("Context rundown routine\n");
    if (phContext)
    {
    pFileContext = (FILE_CONTEXT_TYPE *) phContext;
    if (pFileContext->hFile != NULL)
        fclose(pFileContext->hFile);
    midl_user_free (phContext);
    }
}
```

Binding 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, group10523 1 /a "SDK_a12.bmp"}

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.

Like 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 is not meaningful 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 the following:

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

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:

```
/* 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);
    ...
}
```

Implicit Handles

An implicit handle is a handle that is stored in a global variable. You usually initialize the handle, then don't 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:

```
/* 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:

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

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 follows:

```
/* 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);
}
```

Summary of Binding and Handles

Binding is the process of making a logical connection from a client to a server. 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 relies 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 might not appear explicitly in the stubs.

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***);**

cBytes
Specifies the count of bytes to allocate.

The **midl_user_allocate** function must be supplied by both client applications and server applications. Applications and generated stubs call **midl_user_allocate** directly or indirectly to manage allocated objects:

- The client and server applications should call **midl_user_allocate** to allocate memory for the application – for example, 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 follows:

- 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 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***);**

pBuffer
Specifies a pointer to the memory that is to be freed.

The **midl_user_free** function must be supplied by both client applications and server applications. 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:

- The server application should call **midl_user_free** to free memory allocated by the application – for example, 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. For additional information on **enable_allocate,** see MIDL Reference.

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
- Rpcss 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 posible. 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 follows:

```
/* 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 requires the use of the MIDL compiler switch **/ms_ext** at MIDL compilation time.

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 since 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. Unlike the **allocate(all_nodes)** attribute, however, the buffer parameter is not managed by the client stub, but by the client application.

The buffer must be an **out**-only parameter. The 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 requires the MIDL compiler switch **/ms_ext** at MIDL compilation time.

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 follows:

```
/* 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 big for the specified buffer size, an exception is raised.

Persistent Storage on the Server

You can optimize your application so that 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:

```
/* 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 especially 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. It provides the best overall stub performance for memory management. The default allocator/deallocator pair used by the stubs and runtime 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**.

The Rpcss package is enabled for MIDL-generated stubs automatically whenever full pointers are used, the arguments require memory allocation, or as a result of using the **enable_allocate** attribute. In **/ms_ext** 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 is not affected in default mode; in **/ms_ext** mode, the client side becomes alerted to the possibility that the Rpcss package may be enabled.

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 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 often 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.

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:

```
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. If array size is fixed, allocate the array size on the server, but transmit no elements. If array size is not bound, not legal: MIDL compiler error.	Transmit the number of elements indicated by the length. 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. If the array size is fixed, allocate the array size on the server, but transmit no elements. 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	Transmit the length.

		parameter. If the array size is bound, allocate the array size on the server, but transmit no elements. If array size is not bound, not legal: MIDL compiler error.	Transmit the number of 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. Transmit the number of elements indicated by the length. 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 parameter, client application must allocate; if embedded, can be null. Change from null to non-null causes client stub to allocate; change from non-null to null can cause orphaning.	
Full (**ptr**)	If parameter, client application must allocate; if embedded, can be null. Change from null to non-null causes client stub to allocate; change 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.

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.

```
/* 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. The server stub does not 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 that 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 doesn't have enough information to correctly duplicate the unique pointer in the server's address space, and the stub doesn't 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, but 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 non-encapsulated 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 larger indices 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 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 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, 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	Allocated on server by stubs

	memory for data	
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**)

Using 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 is only supported when using **/ms_ext** 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. Rather than 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. Use caution when doing so, however. 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*. The buffer is obtained according to 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, rather than 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. All serializing handles must be managed explicitly by you. First, you 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**. The 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. 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 as follows:

```
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
                                                      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 */
```

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 that 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 ACF specifies attributes that affect only local performance rather than the network contract.

Microsoft RPC also offers an option that allows you to combine the ACF and IDL attributes in a single IDL file. In **/ms_ext** mode, multiple interfaces can be combined 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 summarizes the attributes that are specified in the IDL and ACF files and the output files generated by the MIDL compiler. It is organized by topic. The same material is alphabetized and presented in more detail in the reference topics. For more information, see the MIDL Reference and the MIDL Command-Line Reference.

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**	Describes how the data object is used

The IDL File

The IDL file specifies an interface used by the client and the server. The interface consists of two parts, an interface header and an interface body.

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 exactly 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 3 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 short unsigned integers in the range between 0 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.

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 does not allow you to specify parameters of type **int** or**void *** because the size of these types is not known or because the size varies among different computers. 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 0 and TRUE as 1. When you use the Microsoft-extensions mode of the MIDL compiler, **boolean** initializations using 0 and 1 are allowed in addition to TRUE and FALSE. In DCE-compatibility mode, however, 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** without a modifier is not a valid MIDL type. The **hyper** keyword designates a 64-bit integer. The **long** keyword designates a 32-bit integer. The **short** keyword designates a 16-bit integer. The **small** keyword designates an 8-bit integer. For more information, see **signed** and **unsigned**.

The **float** keyword designates a 32-bit floating-point number. 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, instructing 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**.

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

Two fundamental types of discriminated unions are provided by MIDL: **non-encapsulated_union** and **encapsulated_union**. The discriminant of a non-encapsulated 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 field is the union.

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**, and **/ms_ext**.

Enumerated Types

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

Arrays

See **arrays**.

Type Attributes

Type attributes are the MIDL attributes that can be applied to type declarations: **transmit_as**, **handle**, **ignore**, **switch_type**, and **context_handle, string,** pointer attributes, and array attributes.

The **transmit_as** attribute instructs the compiler to associate a transmitted type with a presented type that client and server applications manipulate. You must supply the routines that carry out the conversion between the presented types and the transmitted types. Supply routines to release memory used to hold the converted data. Using the **transmit_as** attribute instructs the stub to call the supplied conversion routines before and after transmission. The **represent_as** attribute instructs the compiler to associate a local type with a transfer type that is transferred between client and server. You must supply the routines that convert between the local and the transfer types. Supply routines to release memory used to hold the converted data. Using the **represent_as** attribute instructs the stub to call the supplied conversion routines before and after transmission.

The **handle** attribute specifies that a type can occur as a user-defined, generic, or serialization 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 parameter.

The **ignore** attribute designates pointer fields to be ignored during the marshalling process. An ignored field is set to NULL on the receiver side when allocating memory. When used as a type attribute, the **ignore** attribute specifies that the ignored type can only be used as a field type.

The **switch_type** attribute designates the type of a union discriminator. This attribute applies only to a non-encapsulated union. The **context_handle** attribute allows the developer to write procedures that maintain state information between remote procedure calls. A context handle is a pointer with a **context_handle** attribute. 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.

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, another 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. 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. 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 pointer to a zero-terminated character or byte stream is to be treated as a string. The string attribute only applies 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.

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, 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. Pointer attributes are applied in the following order:

1. An explicit pointer attribute applied at the use site
2. When the unattributed pointer is a top-level pointer parameter, the **ref** attribute
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**.

Function Declarations

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's often desirable to place a local procedure declaration in the IDL file, just because 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**.

Parameter Declarations

The directional attributes **in** and **out** can be applied to parameters.

The **in** attribute indicates that the parameter is passed from the client to the server. The **out** attribute indicates that the parameter is passed from the server to the client. In DCE-compatibility mode, each parameter must have at least one explicit directional attribute. In Microsoft-extensions mode, a parameter without a directional attribute defaults to an **in** attribute.

Expressions

Expressions can be used as constant initializers, array bounds, and transmission-range specifiers, and union discriminators. For more information, see **version**.

Error Handling

Caller stubs report run-time errors to application code in one of the three following ways:

- When the return type of a remote procedure is **error_status_t,** the error is returned as the procedure return value.
- When the procedure has an **out** parameter of type **error_status_t ***, the error is returned as the value of this parameter.
- In any other case, the stub raises an exception.

Server stubs report run-time errors to application code by raising an exception. For more information, see **error_status_t**, **comm_status** and **fault_status**.

The ACF

The application configuration file (**ACF**) has two parts: an interface header similar to the interface header in the IDL file, and a sequence of attributes with values.

The ACF specifies behavior on the local computer and does not affect the data transmitted over the network. The ACF is used to specify details of a stub to be generated. In DCE-compatibility mode, 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 include Declaration

The **include** statement specifies one or more header files to be included into 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. It 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 parameter is not an explicit handle and that 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 that 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 that 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 allows the developer to 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 that contains 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

Run-Time RPC Functions

The 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

This section describes 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, a client 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. In addition, you can run several instances of the server on different hosts, thereby providing clients with random load balancing.

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 selects the appropriate implementation of the function.

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.

The set of RPC functions supported by Microsoft RPC overlaps the DCE RPC functions. The Microsoft RPC functions are optimized for use with the MS-DOS, Microsoft Windows, and Microsoft Windows NT 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 follows:

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**

Microsoft RPC function names are derived by converting the first character of the DCE RPC function name and every character that follows an underscore character to uppercase and then removing underscore characters. For example, the DCE function **rpc_server_use_all_protseqs_if** is named **RpcServerUseAllProtseqsIf** in Microsoft RPC.

Microsoft data-structure names are derived from the DCE names by converting all characters to uppercase and removing the trailing suffix **_t**. For example, the 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.

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

Binding Handle

One purpose of the RPC run-time functions is to provide the client with a binding handle to the server. A binding handle is a data structure that represents the logical connection between the client and the server.

Like a file handle or a window handle, a binding handle cannot be used to directly access and manipulate data about the binding itself. Instead, the binding handle is a unique identifier that the RPC run-time libraries use to access the appropriate data.

The components of a string binding offer a simple way to explain some of the data structures used to obtain a binding handle. Note, however, that creating a binding from a string binding is not recommended. Most applications should use the name-service functions.

String Binding

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. 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.

Endpoint

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 that an 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, rather than being dedicated to a distributed service when the service is developed.

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; 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, a database that is managed by a specific service that creates and deletes elements for applications. In Windows NT, the endpoint-mapping service is called RPCSS. The dynamic endpoint expires when the server instance stops running. Call **RpcEpUnregister** at application termination to remove the old endpoint from the endpoint mapper database.

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.

Fully and Partially Bound Handles

When you use dynamic endpoints, the run-time libraries obtain endpoint information when 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:

- From the client's interface specification
- From an endpoint-mapping service running on the server

When the client tries to use a partially bound handle, the endpoint information is not available in the interface specification, and there is no endpoint-mapping service, the client does not have enough information to make its remote procedure call and that call fails. You must provide the endpoint map when your distributed application uses partially bound handles. For more information about the endpoint map, 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.

Server-Application RPC API Calls

For most distributed applications, write your server application for calling the RPC functions in the following sequence:

1. Use the protocol sequence(s). Call one of the following RPC functions: **RpcServerUseProtseq**, **RpcServerUseAllProtseqs**, **RpcServerUseProtseqIf**, **RpcServerUseAllProtseqsIf**, and **RpcServerUseProtseqEp**.
2. Call **RpcServerInqBindings** to obtain the binding vector needed 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**.
4. Export to the name-service database. Call **RpcNsBindingExport**.
5. Clean up data structures. Call the RPC function **RpcBindingVectorFree**.
6. Register the interface. Call **RpcServerRegisterIf**.
7. Listen for clients. Call **RpcServerListen** or **RpcMgmtWaitServerListen**.

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**.
2. Remove the interface. Call the RPC function **RpcServerUnregisterIf**.
3. Remove endpoint-map database entries. Call the RPC function **RpcEpUnregister**.

When the server stops providing service for a short period of time, the server should not remove its registration from the name-service database. The RPC name-service function **RpcNsBindingUnexport** is called only when the service is permanently removed.

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 string that represents a valid combination of an RPC protocol (such as "ncacn"), a transport protocol (such as "tcp" and "nb"), and a network-address format (such as "ip"). Microsoft RPC supports the protocol sequences listed in the following list. Other protocol sequences will be supported in subsequent releases.

Protocol sequence	Description
ncacn_ip_tcp	Connection-oriented TCP/IP
ncacn_nb_tcp	NetBIOS over TCP/IP
ncacn_nb_nb	NetBIOS over NetBEUI
ncacn_np	Named pipes
ncacn_spx	Connection-oriented SPX
ncadg_ip_udp	Datagram-oriented TCP/IP
ncadg_ipx	Datagram-oriented IPX
ncalrpc	Local communication on Windows NT only

Note Windows 95 does not support **ncalrpc**, **ncacn_nb_ipx**, and **ncacn_nb_tcp**. The **ncacn_np** protocol is supported only on the client side. You must have an authentic Novell client to use the RPC SPX transport.

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 choices are listed following:

"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 that 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 (and the client has partially bound handles), the server application should also call **RpcEpRegister** or **RpcEpRegisterNoReplace**.

The dynamic endpoints are registered after the server application has specified the use of dynamic endpoints by calling the RPC function that starts with the prefix "RpcServerUseProtseq" or "RpcServerUseAllProtseqs."

The functions **RpcEpRegister** and **RpcEpRegisterNoReplace** store information about the dynamic endpoint in a database that contains local endpoint maps. The client run-time library gets the endpoint from the database by querying using a partially bound handle and the client interface specification. This allows the client to establish a binding to a server instance.

When the server run-time library selects the endpoint, it is available to the client application through the endpoint-map database.

An endpoint map stores handles that are partially bound to local server endpoints. Each database element contains an interface specification, an object UUID (which may be nil), the protocol sequence, and the associated endpoint.

When the client makes a remote procedure call using a partially bound handle, the client's run-time library asks the endpoint map for the endpoint of a compatible server instance. The client library supplies the interface UUID and the object UUID in the partially bound handle if it exists.

The endpoint map compares the client interface UUID and the object UUID to server entries in the endpoint-map database. If both the interface UUID and the object UUID match, the endpoint map tests compatibility of the version numbers, the protocol version, and the protocol sequence, as follows:

- The interface major version numbers must match, and the server minor version number must be greater than or equal to the client minor version number.
- At least one transfer syntax must match.
- The RPC protocol version must match.
- The server must have at least one protocol sequence that matches the client.

If all tests are successful, the endpoint map returns the valid endpoint, and the client run-time library returns a fully bound 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 DCE environment, the server application registers with the name-service database only once, when the application is installed. The Microsoft Locator maintains its database in transient memory on the server, while the DCE name service resides in permanent, replicated storage that is relatively expensive to update.

Using the Locator

This section describes how to best use the name service feature of Microsoft RPC.

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. In other words, 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, since they will change at each invocation of the server and will clutter up your name service entry. Use **RpcBindingReset** to remove such binding handles. For example, a reasonable sequence of server operations would be as follows:

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). This is a more robust name service. When using CDS, **RpcNsBindingExport** or **RpcNsBindingUnexport** will create a lot of network traffic for replication and distribution. Thus, the server should determine if the information already has been exported, and only export it if is has not.

What Happens During a Query

This section describes how the network handles the query when a 32-bit client looks 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 mailslot broadcast. If there is a match, the locator on each computer will respond by a directed mailslot (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, or
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 mailslot 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, for example, 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 doesn't 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 upper-case. When looking for a match, the locator is case-sensitive.
- Locator entry names are also case-sensitive.

Using CDS

If you have CDS, you can use it instead of the Locator. Change the registry entries as follows:

```
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 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.

The RPC Name-Service Database

A name service is a service that maps names to objects, usually maintaining the (name, object) pairs in a database. The name is usually 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, give your application users the ability to supply the name for registering the application in the name-service database. Provide a method, such as a data file, command-line input, or dialog box, that allows the application user or network administrator to specify the name.

The RPC name-service architecture supports various methods for organizing an application's server entries, but it is optimized for lookups. 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 allows the server to 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.

Design your application so that 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 might 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**. But if the server doesn't update the name-service entry, clients won't 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 that 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 doesn't require the object UUID of the resource. If it has the resource UUID but not the name, it can import from the null entry.

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 **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).

RpcServerRegisterIf sets 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 about supplying multiple EPVs to provide multiple implementations of the interface, see Multiple Interface Implementations.

Listening for Clients

After registering the protocol sequence, endpoint, and interface, and after 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 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 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. Two approaches are used 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 lookup 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 DCE CDS and the Microsoft Locator.

The 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.

The Microsoft Locator implementation of the name-service provider does not use DEFAULT_ENTRY and does not support 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.

Multiple Interface Implementations

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 **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**.

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 **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**.

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 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.

For 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 don't want to preload all object-UUID-to-type-UUID mappings.

Using Datagram Protocols

Microsoft RPC supports datagram, or connectionless, protocols; as well as connection-oriented protocols. Some of the features available when using datagram protocols are as follows:

- 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.
- Datagram RPC provides guaranteed, at-most-once delivery through its **non-idempotent** attribute. A **non-idempotent** routine is one that cannot be executed more than once because it will either return different results each time, or because it modifies some state. Contrast this with an **idempotent** routine which also provides guaranteed delivery, but does not ensure at-most-once delivery since transmission acknowledgement is not required.
- Datagram RPC supports the **broadcast** and **maybe** capabilities. **Broadcast** enables a client to issue messages to multiple servers at the same time. This allows the client to locate one of several available servers on the network, or to control multiple servers simultaneously.

Exception Handling

Microsoft RPC uses the same approach to exception handling as the Microsoft Win32 API.

With Microsoft Windows NT, 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 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*, and Windows NT. With Windows NT, RPC **try-except** support expands into Windows NT **try-except** support.

When you write distributed applications for Win32 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, group10525 0 /a "SDK_a20.bmp"}

The RPC exception handlers differ slightly from the DCE exception-handling macros TRY, FINALLY, and CATCH. Various vendors provide include files that map the 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 DCE exception counterparts, rpc_s_*, and map RPC_X_* error codes to rpc_x_*. For 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:

- Use a security package that can be accessed using the RPC functions.
- Use the security features built into Windows NT transport protocols.

The transport-level security method is not the preferred method. We recommend that 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 pipes transport, this version also implements the transport-independent security functions from 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 runtime library. This security information is called the client credentials. The client runtime library forwards the credentials to the server run-time library. The server runtime library 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 saver ensures 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

Depending on your security provider, invalid credentials may or may not be rejected. By default, the RPC runtime library will dispatch calls. It is your responsibility to protect yourself by either:

- Determining exactly who the client is, or
- Using impersonation.

Note Disable the guest account on all computers where security is a significant factor.

If you need additional information about 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.
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 For Windows 95 platforms, 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 a Windows 95 client communicating with a Windows NT server. These levels are never supported for a Windows 95 client communicating with a Windows 95 server.

The level of security required depends entirely on the application. When considering security levels, remember that the higher the protection level, the greater the overhead required to create and maintain the levels. Additionally, there are performance tradeoffs to consider. The checksum computation and encryption required by the RPC runtime 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. When selecting a security level, choose the elements needed by the application, mixing and matching only as required.

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 runtime 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, it 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 runtime library implicitly calls **RpcMgmtInqServerPrincName**; calls to management interfaces cannot be made to unbound handles. All RPC server processes have the same management interface, and 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 runtime library, since Windows 95 does not support the Windows NT security model.

Differences in Platforms

When developing applications for MS-DOS, you must feed in the password and credential information to **RpcBindingSetAuthInfo** manually. This is not the case for an NT or Windows-based platform where the password for the domain will be used. If a Windows For Workgroups or Windows 3.*x* workstation is not part of a domain, the user will be prompted for the password.

For MS-DOS applications, create a pointer to the SEC_WINNT_AUTH_IDENTITY structure and pass in the credential information under the *AuthIdentity* parameter.

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's 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 as follows:

- 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.

Security Packages

The RPC run-time API set includes several functions that let your application manage security.

A security package such as the Kerberos Authentication Protocol can be supplied as a DLL that interoperates with the Microsoft RPC run-time libraries.

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 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, rather than 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. **RpcRevertToSelf** restores 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 allows the client to 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 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. The security 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 that 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 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 Since it does not support the Windows NT security model, the Windows 95 runtime 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 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**.

Building RPC Applications

The procedure for building a distributed RPC application varies slightly, depending on the operating-system platform that you are developing on and the target platform, version of the MIDL and C or C++ compiler, and API libraries that you use. In all cases, however, the general procedure is the same: Develop the MIDL and C source files, compile the MIDL source files, then 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*, and Microsoft 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 for 32-bit environment	Produces C source code for 16- or 32-bit environment
C and MSVC for 16-bit environment (Microsoft C/C++ version 7.0)	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 RPC.
2. Develop the IDL file (and optional ACF) that specify 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 topic includes the following:

- a description of the **uuidgen** utility.
- a discussion about 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 eight hexadecimal digits followed by a hyphen, followed by three hyphen-separated groups of four hexadecimal digits, followed by a hyphen, followed by twelve 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 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.

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 don't allow unattributed pointers. If a pointer does not have an explicit attribute, the IDL file must have a **pointer_default** specification so that the pointer attribute can be set.

In Microsoft-extensions mode, you can specify a pointer's attribute in the IDL file that imports the defining IDL file. This mode, selected using **/ms_ext**, allows pointers defined in one IDL file to inherit attributes that are specified in other IDL files.

In Microsoft-extensions 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)

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 includes directives to include 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

Using the MIDL Compiler

The MIDL compiler offers a variety of command-line switches that allow you to control its input and output. This section describes some of the major command-line switches, options, and arguments that are supported in Microsoft RPC. For a complete description of each switch, see the reference documentation for the MIDL compiler.

In its default mode, the MIDL compiler generates files that are compatible with DCE RPC. In this mode, language features that are not compatible with DCE IDL are noted as warnings or errors.

Three other compiler modes can be activated with MIDL compiler switches. The modes are as follows:

Compiler mode	Description
/ms_ext	Specifies Microsoft extensions to DCE IDL compilers
/c_ext	Specifies C extension to IDL
/app_config	Supports selected ACF keywords in the IDL file, so you do not have to provide an ACF

The three compiler-mode switches permit different extensions to the DCE IDL language. Each set is independent of the other two and any combination of switches can be set. If no switch is turned on, the language and the compiler output are compatible with DCE IDL.

Each line in the following table shows one of the eight possible combinations of switch settings for the MIDL compiler:

/ms_ext	/c_ext	/app_config	Description
Off	Off	Off	DCE-compatibility mode
Off	Off	On	Allows some ACF attributes in the IDL file; otherwise DCE-compatible
Off	On	Off	DCE-compatible with Microsoft C extensions
Off	On	On	DCE-compatible with Microsoft C extensions and allow some ACF attributes in the IDL file
On	Off	Off	Allows MIDL extensions
On	Off	On	Allows MIDL extensions and allows ACF attributes in the IDL file
On	On	Off	Allows MIDL extensions and Microsoft C language extensions
On	On	On	Allows MIDL extensions, Microsoft C language extensions, and some ACF keywords in the IDL file

The following sections describe each of the compiler modes in detail.

Microsoft-Extensions Mode

The **/ms_ext** switch enables Microsoft extensions to the DCE IDL. The following features are supported in **/ms_ext** mode:

- Static callback functions on the client
- Explicit handle parameters in any position in the argument list rather than first only
- Pointers to tagged declaration types
- Expressions used as size and discriminator specifiers
- Enumerator initialization (sparse enums)
- **wchar_t** wide-character constants, strings, types, and parameters
- **cpp_quote**(*quoted_string*) and **#pragma midl_echo**
- Multiple interfaces in an IDL file
- Definitions outside of an interface

C-Extensions Mode

C-extensions mode, selected with the **/c_ext** switch, allows you to use existing C header files with your distributed application. The **/c_ext** switch allows you to use type qualifiers, such as **far** and **stdcall**, in type definitions and function prototypes specified in the IDL file.

The **/c_ext** switch instructs the MIDL compiler to accept C type qualifiers in type declarations as long as the type declarations are not used in a remote procedure call. This allows the compiler to ignore **int** and **void *** declarations that would otherwise generate MIDL compiler errors.

C-extensions mode also allows most Microsoft and ANSI C declarative syntax. The following features are supported in the **/c_ext** compiler mode as long as they are not used in remote procedures:

- **int**
- **void ***
- Declaration qualifiers in the IDL file: **_ _near, near, _ _far, far, _ _cdecl, cdecl, _ _pascal, pascal, _ _loadds, loadds, _ _volatile, volatile, _ _export, export**
- Complex declarators (such as **void ** _near _cdecl proc(int * _far * a[]);**)
- Ellipses in the procedure parameter list
- Bit fields in structures and unions
- Non-discriminated unions
- **extern** declarations
- Enumerator initialization

Application Configuration Mode

The **/app_config** switch allows you to put selected ACF attributes in an IDL file so you don't have to maintain two files that describe an interface. This mode supports the use of the **implicit_handle** and **auto_handle** interface attributes in the IDL file.

The following ACF attributes can appear in the IDL file when you use the **/app_config** compiler mode:

- Handle attributes in the IDL file: **implicit_handle**
- **auto_handle**

Help

A complete listing of MIDL compiler switches and options is available when you use the MIDL compiler **/help** and **/?** switches. The switches are organized by categories.

Response Files

As an alternative to placing all the options on the command line, the MIDL compiler accepts response files that contain switches and arguments. A response file is a text file containing one or more MIDL compiler command-line options. Unlike a command line, a response file allows multiple lines of options and filenames. This is important on systems such as MS-DOS that have a hard-coded limit on the length of a command line. You can specify a MIDL response file as follows:

midl **@***filename*

filename
Specifies the name of the response file. The response filename must immediately follow the @ character. No white space is allowed between the @ character and the response filename.

Options in a response file are interpreted as if they were present at that place in the MIDL command line.

Each argument in a response file must begin and end on the same line. The backslash character (\) cannot be used to concatenate lines.

MIDL supports command-line arguments that include one or more response files, combined with other command-line switches:

```
midl -pack 4 @midl1.rsp -env win32 @midl2.rsp itf.idl
```

The MIDL compiler does not support nested response files.

C-Compiler and C-Preprocessor Requirements

The MIDL compiler must interoperate with the C compiler and C preprocessor. The requirements for the C compiler and preprocessor are described in the following sections.

C-Preprocessor Requirements for MIDL

The MIDL compiler uses the C preprocessor during initial processing of the IDL file. The operating system used when you compile the IDL files is associated with a default C preprocessor. If you want to use a different C-preprocessor name, the MIDL compiler switch **/cpp_cmd** allows you to override the default C-preprocessor name:

midl **/cpp_cmd** cl386 *filename*

filename
Specifies the name of the IDL file.

During initial processing, the C preprocessor removes all preprocessor directives in the IDL file. After preprocessing, the only directive that can appear in a file is the **#line** directive in one of the following forms:

```
#line digit-sequence "filename" new-line

# digit-sequence "filename" new-line
```

Other directives should not appear in either the IDL file or any header file included by the IDL file. These other directives are not supported by the MIDL compiler and can cause errors. For a complete description of the **line** directive and other preprocessor directives, see your C-compiler documentation.

The MIDL compiler requires the C preprocessor to observe the following conventions:

- The input file must be the last argument on the command line.
- The preprocessor must direct output to the standard output device, *stdout*.

Preprocessor directives present in the IDL file do not appear in the header file generated by the MIDL compiler. For example, any values defined in the IDL file with the C **#define** statement are removed by the C preprocessor. These **#define** statements will not appear in the header file generated by the MIDL compiler. If such values are defined only in the MIDL file and are required by C source files, the C compiler will report errors when it tries to compile these source files.

Four workarounds are recommended:

- Use **cpp_quote** to reproduce **#define** in the generated header file
- Use **const** declaration specification
- Use header files that are included in the IDL file and the C source code
- Use enumeration constants in the IDL file

To get a declaration in the generated header file with **cpp_quote**, use the following statement:

```
cpp_quote ("#define ARRSIZE 10");
```

This statement results in the following line being generated in the header file:

```
#define ARRSIZE 10
```

You can reproduce manifest constants using the constant-declaration syntax:

```
const short ARRSIZE = 10
```

This syntax results in the following line being generated in the header file:

```
#define ARRSIZE 10
```

You can define separate header files that contain only preprocessor directives and include them in both the IDL file and the C source files. Although the directives will not be available in the header file

generated by the MIDL compiler, the C source program can include the separate header file.

You can choose to use enumeration constants in the IDL file. Enumeration constants are not removed during the early phases of MIDL compilation by the
C-compiler preprocessor, so these constants are available in the header file generated by the MIDL compiler. For example, the statement

```
typedef enum midlworkaround { MAXSTRINGCOUNT = 300 };
```

will not be removed during MIDL compilation by the C preprocessor. The constant MAXSTRINGCOUNT is available to C source programs that include the header file generated by the MIDL compiler.

Verifying Preprocessor Options

To verify the correct operation of **/cpp_opt** options, invoke the C preprocessor independently before including the command line as part of the MIDL compiler command line. When called independently, the C compiler correctly reports errors caused by invalid options.

Errors in **/cpp_opt** switch input to the MIDL compiler can produce error messages related to the IDL file. The errors are incorrectly reported by the MIDL compiler when operating with some C compilers.

For example, invalid command-line syntax to the Microsoft C compiler can be reported as a syntax error in the IDL file when that syntax is included as part of the MIDL compiler command line. The error is not in the IDL file but in the MIDL compiler **/cpp_opt** input.

The following MIDL compiler command line contains the **/cpp_opt** switch and related options:

```
midl /cpp_cmd "cl" /cpp_opt /E foo.idl
```

The options in this command line can be verified by invoking the compiler only, as follows:

```
cl /E foo.idl
```

C-Compiler Requirements for MIDL

The MIDL compiler requires the C compiler to support a packing level of 1, 2, 4, or 8. The command-line option for Microsoft C compilers that controls packing is **/Zp***level*, where *level* is the packing level: 1, 2, 4, or 8. The following rules govern the alignment of compound types:

- Base-type fields of *size* < *packing level* start on a (0 modulo *size*) address.
- Base-type fields of *size* >= *packing level* start on a (0 modulo *packing level*) address.
- The compound type itself (and any field of compound type) is aligned according to the strictest alignment requirement on any of its fields.
- Compound types are padded to the next (0 modulo *level*) address. This padding appears in the size returned by the C SIZEOF macro.

For example, consider a compound type consisting of a 1-byte character, an integer 4 bytes long, and a 1-byte character:

```
struct mystructtype {
    char c1;  /* requires 1 byte  */
    long l2;  /* requires 4 bytes */
    char c3;  /* requires 1 byte  */
 } mystruct;
```

For packing level 4, the structure *mystruct* is aligned on a (0 mod 4) boundary and sizeof(struct mystructtype) = 12.

For packing level 2, the structure *mystruct* is aligned on a (0 mod 2) boundary and sizeof(struct mystructtype) = 8.

C-Compiler Requirements for Callbacks in Microsoft Windows 3.*x*

When you use Microsoft Visual C/C++ version 1.5 to develop your RPC application for Microsoft Windows 3.x platforms, compile with the **/GA** switch. The **/GA** switch directs the compiler to generate code that loads the DS register from the SS register on entry to a far, exported function in a protected-mode application based on Windows 3.x.

For protected-mode Microsoft Windows applications, the **/GA** switch allows the C compiler to generate the code for performing the housekeeping chores required when switching between tasks. This code is needed when your RPC interface contains one or more callback functions. Without this code, these callback functions can fail at run time due to an incorrect DS value.

When you use compilers other than Microsoft Visual C/C++ version 1.5, use the compiler switch that is equivalent to **/GA**.

The **/GA** switch provides, in an optimized way, the same functionality that the **/Gw** switch and calls to the Windows 3.*x* function **MakeProcInstance** provided for previous versions of the Microsoft C compiler and previous versions of Microsoft Windows.

When you do not compile using the **/GA** switch − for example, when you are using a compiler that does not support the **/GA** switch − your application must:

1. Compile using the **/Gw** switch (or its equivalent).
2. Add the client stub functions to the EXPORTS section of the application's DEF file.
3. Replace function pointers in the the client stub function dispatch table with function pointers returned by **MakeProcInstance**.

The function dispatch table is part of the **RPC_CLIENT_INTERFACE** structure defined in the RPC header file RPCDCEP.H. For example, step 3 can be implemented using the following C code:

```
#include "hello.h"   // generated stub file
RPC_DISPATCH_FUNCTION Old, New;
HINSTANCE hInst;
RPC_CLIENT_INTERFACE * If = Hello_ClientIfHandle;
...
    for (i = 0; i < If->DispatchTable->DispatchTableCount; i++)
    {
       Old = If->DispatchTable->DispatchTable[i];
       New = (RPC_DISPATCH_FUNCTION) MakeProcInstance(Old, hInst);
       If->DispatchTable->DispatchTable[i] = New; // overwrite
    }
...
```

Link Libraries for MS-DOS

The following RPC client application static libraries are provided for MS-DOS:

Static library	Description
RPC.LIB	Base RPC functions and name-service functions
RPCNDR.LIB	NDR and other stub-helper functions
NDRLIB10.LIB	If you are using MIDL 1.0 on MS-DOS, you must connect to this library. All other users should disregard this library. Note that this library is meant to be a temporary solution, and MIDL 1.0 users should migrate to MIDL 2.0 at the earliest occasion.

The following RPC transports are provided for MS-DOS clients:

Pseudo-dynamic-link library	Description
RPC16C1.RPC	Client named-pipe transport
RPC16C3.RPC	Client TCP/IP transport
RPC16C5.RPC	NetBIOS transport
RPC16C6.RPC	Client SPX transport
RPCNS.RPC	Name-service functions
RPCNSLM.RPC	LAN Manager support functions
RPCNSMGM.RPC	Name-service management functions

Link Libraries for Microsoft Windows 3.*x*

The following RPC import libraries are provided for Microsoft Windows 16-bit clients:

Import library	Description
RPCW.LIB	RPC API and name-service functions
RPCNDRW.LIB	NDR and other stub-helper functions

The following RPC dynamic-link libraries are provided for Microsoft Windows 3.*x* clients:

Dynamic-link library	Description
RPCNS1.DLL	Name service
RPCRT1.DLL	Windows run-time library
RPC16C1.DLL	Client named-pipe transport
RPC16C3.DLL	Client TCP/IP transport
RPC16C5.DLL	Client NetBIOS transport

Link Libraries for Microsoft Windows NT

The following RPC import libraries are provided for Microsoft Windows NT clients and servers:

Import library	Description
RPCNDR.LIB	Helper functions
RPCNS4.LIB	Name-service functions
RPCRT4.LIB	Windows run-time functions

The following RPC libraries are provided for Microsoft Windows NT clients and servers:

Dynamic-link library	Description
RPCLTC1.DLL	Client named-pipe transport
RPCLTS1.DLL	Server named-pipe transport
RPCLTC3.DLL	Client TCP/IP transport
RPCLTS3.DLL	Server TCP/IP transport
RPCLTC5.DLL	Client NetBIOS transport
RPCLTS5.DLL	Server NetBIOS transport
RPCNS4.DLL	Name service
RPCRT4.DLL	Windows run-time library

Using the __midl Predefined Constant

When the MIDL compiler processes the input IDL and ACF files, __midl is defined by default. __midl is used for conditional compilation to attain consistency throughout the build. This phases out the use of defines in the header files, such as MIDL_PASS, and replaces them with a consistent flag.

Note that if you so choose, you can override this default by specifying the following on the command line:

```
-U__midl
```

Building 16-Bit Windows-Based Applications

You can build your 16-bit Windows-based client applications on a computer that is running Microsoft Windows 3.*x*, Microsoft Windows for Workgroups 3.1, or MS-DOS.

It is possible to install Microsoft RPC on Windows NT and to use this single platform for developing all server and client applications. However, cross-compilation of Windows 3.*x* and MS-DOS clients on Windows NT requires that you install a 16-bit C compiler in the Microsoft Windows NT environment. This development environment is not automatically installed as part of the Windows NT or Microsoft RPC installation.

Using Different Memory Models

Microsoft RPC supports the development of Microsoft Windows 3.*x* applications using small, medium, compact, and large memory models. You can build your application using any memory model when you use the macro definitions provided in the RPC header files.

Note Microsoft RPC for MS-DOS supports only large-model applications.

The following table lists some of the advantages and disadvantages of the different memory models:

Memory model	Advantage	Disadvantage
Small	Best performance; size and speed	Must add code to your application to handle the far pointers returned from remote procedures
Large	Easiest to build with the Microsoft RPC tools	Size/speed performance degradation; cannot run multiple instances
Large compiled with **/Gx**	Easy to build; can run multiple instances of the application	Size/speed performance degradation; limited to only one data segment

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 which macro definitions the MIDL compiler applies to function calls between RPC components:

{ewc msdncd, EWGraphic, group10526 0 /a "SDK_a29.bmp"}

The macro definitions are as follows:

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 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. You must use the macro **_ _RPC_API** when you define the functions associated with memory management, user-defined binding handles, and the **transmit_as** attribute. You must use the macro **_ _RPC_USER** when you define the context-rundown routine associated with the context handle. Specify the functions as follows:

_ _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**.

Two approaches 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.

Installing RPC

Microsoft® RPC consists of two products: one for developing distributed applications and one for executing distributed applications. Using the RPC development tools, you can develop RPC distributed applications in C/C++ on a computer with a Microsoft® Windows NT™ operating system. Distributed applications developed using RPC technology can be executed as servers in the Windows NT environment and as clients in the Windows NT, Windows 3.*x*, MS-DOS®, and Macintosh environments. The RPC user environment, which consists of the RPC run-time files and libraries, must be installed on servers and workstations that run RPC distributed applications.

Once RPC is installed, you can configure the network transport protocols and the name-service provider that RPC uses.

This section contains information on the following topics:

Installing the RPC Programming Environment

You develop RPC distributed applications on the 32-bit Windows NT platform. While the 16-bit MIDL compiler is not supported in this version of RPC, 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 Windows NT platform, no additional installation is required.

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

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*.
- MIDL compiler for Microsoft Windows NT

When you install the Win32 SDK, you install the following:

- C/C++ language header files (.H for the RPC run-time libraries) and run-time libraries (.LIB and .DLL) for Microsoft Windows NT
- Sample programs for Microsoft Windows NT
- RPC reference Help files
- The **uuidgen** utility

When you install Windows NT, you install the following:

- RPC Run-time .DLLs
- RPC Locator and RPC Endpoint-mapping services

Microsoft Windows NT

The Microsoft Win32 SDK contains the Microsoft Windows NT and Microsoft Windows 95 APIs. When the Win32 SDK is installed, Microsoft RPC is also installed.

Windows/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 toolkit. Microsoft RPC development for MS-DOS and Microsoft Windows requires:

- Microsoft C/C++ version 7.0 or MSVC++ version 1.*x*
- One of the following:
  - 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

The Setup program for installing RPC for Windows prompts you to specify one of two installation options: Install All Files or Custom Install. The Install All Files option quickly installs all the RPC files; all you have to do is select a NetBIOS protocol and a name-service provider. With the Custom Install option, you can specify all information about the setup.

{ewl msdncd, EWGraphic, group10529 0 /a "SDK.BMP"} To start the RPC Setup program for a Windows/MS-DOS Client

1. Insert the Setup disk in drive A.
2. From the File menu, choose Run.
3. In the Run box, type **a:\setup** and choose OK.

   After the Microsoft RPC dialog box is displayed, the Name Service Installation Options dialog box appears.
4. Choose Install Default Name Service Provider or Install Custom Name Service Provider, and then choose Continue.

   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.

   - In the Network box, type the network address. The network address is the name of the host computer that runs the NSI daemon (NSID).

     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. The network address can be 80 characters or less – for example, 11.1.9.169 is a valid address.
5. In the Installations Options dialog box, select the Install All Files option or the Custom Install option.
   - If you choose Install All Files, carry out the following procedure, "To run RPC Setup using the Install All Files option."
   - If you choose Custom Install, skip to To run RPC Setup using the Custom Install option.

{ewl msdncd, EWGraphic, group10529 1 /a "SDK.BMP"} To run RPC Setup using the Install All Files option

When you start the RPC Setup program and choose the Install All Files option, the Base Directory Path dialog box appears.

1. In the Path box, type the path where you want to install the RPC directory.

It is a good idea to install the RPC files into the same location as your 16-bit C compiler. Otherwise, you must set the environment variables INCLUDE, LIB, and PATH to point to the directories that contain the RPC header files, libraries, and DLLs and binaries.

2. Choose OK.

   The Custom NetBIOS Protocols dialog box appears.

3. In the Custom NetBIOS Protocols dialog box, select one of the following protocols:
   - Select Microsoft NetBEUI NetBIOS Protocol to install the NetBEUI NetBIOS protocol.
   - Select TCP/IP NetBIOS Protocol to install the TCP/IP protocol.
   - Select Custom NetBIOS Protocol if your computer uses more than one network card or more than one protocol. When you choose the Custom NetBIOS protocol, the NetBIOS Custom Protocol Mapping dialog box appears. Use the Current Mappings box to build a list of NetBIOS protocol mappings that associate the protocol you specify with a LAN adapter number. Add and delete NetBIOS protocol mappings using the New and Delete buttons.

     To add a NetBIOS protocol, choose New. In the Protocol box, type the name of the protocol using the protocol names "nb" and "tcpip." In the LAN Adapter # box, type the LAN adapter number. You can use LAN adapter (lana) numbers 0 through 9. If your configuration has only one card and protocol, the LAN adapter number is usually 0. For more information about NetBIOS settings, see Network-Transport Information for RPC.

     To delete a NetBIOS protocol, select the protocol from the Current Mappings box and choose Delete.

{ewl msdncd, EWGraphic, group10529 2 /a "SDK.BMP"} To run RPC Setup using the Custom Install option

When you start the RPC Setup program and choose the Custom Install option, the SDK Tools Options dialog box appears.

1. In the Base Installation Path box, choose Path to change the location where the RPC directory is installed.

   The Path box automatically displays the path to the Microsoft C/C++ version 7.0 directory. If you type a different path, it should point to the same location as this directory. Otherwise, you must set the environment variables INCLUDE, LIB, and PATH to point to the directories that contain the RPC header files, libraries, and DLLs and binaries.

2. In the Installation Options box, select one or more of the following files to install:
   - MS-DOS Libraries and Include Files. If you are developing a distributed application that will be used on MS-DOS clients, you must install these libraries and include files.
   - Windows Libraries and Include Files. If you are developing a distributed application that will be used on Windows clients, you must install these libraries.
   - Online Help Files. The Help files are not needed to develop applications. They are designed to answer questions during the development process.

3. Choose Continue.

   The Run-Time Custom Install Options dialog box appears.

4. In the Run-Time Custom Install Options dialog box, set one or more of the following options:
   - Under Registry Data File Path, choose Set Location to specify the location of the configuration file. The configuration file contains all of the configurable RPC components, which you can modify at a later time. Type the path in the Path box. The default is the root directory.
   - In the MS-DOS Options box, set the MS-DOS configuration options. If you did not install MS-DOS libraries and include files in the previous step, don't modify this section.

     Select the MS-DOS Run-time Support box to install the MS-DOS run-time libraries. This box must be marked if you will be programming or executing RPC in MS-DOS.

     Under Runtime DLL Path, choose Set Location to specify the path where the DLLs are stored.

Type the path in the Path box. Choose OK.

- To install Windows run-time libraries, select the Windows Run-time Support box. If you are programming in Microsoft Windows, you must install Windows run-time libraries.
- Under Network Drivers, select the Load All box to install all drivers.

  Choose the Set Location button to specify the drivers your distributed application uses. Select the protocol adapters you want to install. To select more than one driver, hold down the CTRL key as you select additional drivers. To undo your selection, choose Reset Selection.

5. Choose Continue.

   The Custom NetBios Protocols dialog box appears.
6. Follow step 3 in To run RPC Setup using the Install All Files option.

Writing 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++, version 2.0, cross development tools. In order to use rpc.lib, you must link against the C runtime and swapper library (swap.lib) provided with Visual C++, version 2.0.
- Macintosh RPC SDK, which can be found in the mac_rpc directory. 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.

Note There is no Macintosh support for the Windows 95 platform.

To set up the Windows NT server:

- Current 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.

To write an RPC client:

1. If you use **atexit** to perform cleanup at 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 following APIs are not supported by the Macintosh:

- **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

Installing RPC for Distributed-Application Users

When you use an RPC application on a Windows 3.*x* or MS-DOS system, the RPC run-time executables must be copied to the Windows 3.*x* or MS-DOS machines that will be using the application.

Microsoft RPC provides two utility programs to create RPC run-time installation disks to accompany your distributed application. These utilities provide the Setup program and the Microsoft RPC run-time libraries for use with Microsoft Windows/MS-DOS and MS-DOS. The RPC run-time libraries are automatically installed with Windows NT and no further RPC installation is required.

Microsoft RPC provides the following run-time installation utilities:

- The WRUNDISK.BAT utility, which creates a Microsoft Windows/MS-DOS run-time installation disk
- The DRUNDISK.BAT utility, which creates an MS-DOS-only run-time installation disk

Creating RPC Install Disks for Windows NT Servers and Clients

When Windows NT is installed on a server or client, the RPC run-time files are automatically installed as well. No further RPC installation is required.

However, to run MS-DOS or Microsoft Windows 16-bit clients on Windows NT, your application's install program must install the proper Windows/MS-DOS client DLLs.

Creating RPC Install Disks for Windows Clients

The WRUNDISK.BAT utility copies the RPC run-time libraries from an installed Microsoft Win32 SDK to create a Microsoft Windows/MS-DOS run-time install disk.

{ewl msdncd, EWGraphic, group10529 3 /a "SDK.BMP"} To create RPC install disks for a Windows client

1. Copy the compressed files from the Microsoft Win32 SDK directory \MSTOOLS\RPC_DOS\DISK1 to a floppy disk or directory on your hard disk.
2. Insert a formatted, blank floppy disk in the destination drive.
3. At the MS-DOS prompt, type:

   wrundisk *source destination*

   source

   Specifies the path to the disk or directory that contains the compressed files provided in the Microsoft Win32 SDK directory \MSTOOLS\RPC_DOS\DISK1.

   destination

   Specifies the name of the drive that contains the formatted, blank floppy disk. For example, the command

   ```
   wrundisk c:\nt\mstools\rpc_dos\disk1 a:
   ```

   copies the compressed files to a floppy disk in drive A.

Your application user can then use the Setup program on the run-time install disk to install the RPC run-time libraries.

Creating RPC Install Disks for MS-DOS Clients

The DRUNDISK.BAT utility copies files from an installed Microsoft Win32 SDK to create an MS-DOS-only run-time install disk. To ensure that all MS-DOS loadable transports are present on the run-time install disk, you must select all loadable transports at the time you install the Microsoft Win32 SDK.

{ewl msdncd, EWGraphic, group10529 4 /a "SDK.BMP"} To create RPC install disks for an MS-DOS client

1. Insert a formatted, blank floppy disk in the destination drive.
2. At the MS-DOS prompt, type:

   drundisk *directory destination*

   directory
   Specifies the name of the directory that contains all loadable transport (.RPC) files and all name-service DLLs – for example, C:\LANMAN.DOS\NETPROG

   destination
   Specifies the name of the drive that contains the formatted, blank floppy disk. For example, the command

   ```
   drundisk c:\lanman.dos\netprog a:
   ```

   copies the MS-DOS run-time libraries, the loadable transport files, and the name-service DLLs from the C drive to the floppy disk in drive A.

Your application user can then use the Setup program on the run-time install disk to install the RPC run-time libraries.

Configuring the Name-Service Provider

When a server registers its distributed application using a name-service provider, other computers using a name-service provider that want to interoperate with the server must use the same name-service provider as that of the server. 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 the DEC NSID.

In Microsoft RPC, the Microsoft Locator is the default name-service provider. It is designed for use in Windows environments.

Reconfiguring the Name Service for Microsoft Windows NT

When you install Microsoft Windows NT, the Microsoft Locator is automatically selected as the name-service provider. You can change the name-service provider through the Windows NT Control Panel.

{ewl msdncd, EWGraphic, group10529 5 /a "SDK.BMP"} To configure the name-service provider for Windows NT

1. In the Control Panel, choose the Networks icon.

   The Networks dialog box appears.
2. In the Networks dialog box, choose Add Software.
3. In the Network Software list, select Remote Procedure Call (RPC) Service, and then choose Configure.

   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 and the configuration process is 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 DCS 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/MS-DOS

When you install the Win32 SDK for Windows/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.

{ewl msdncd, EWGraphic, group10529 6 /a "SDK.BMP"} 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.

{ewl msdncd, EWGraphic, group10529 7 /a "SDK.BMP"} To configure the DCE CDS name-service provider

- You must have Digital Equipment Corporation's DCE DCS product to configure the DCE CDS as your name-service provider. See the documentation provided by Digital Equipment Corporation for information about DCE CDS.

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 RPC Services and Utilities on Microsoft Windows NT

On Microsoft Windows NT, the Microsoft Locator and the endpoint-mapping service are automatically started by the RPC run-time libraries when necessary. You can stop the Locator or the endpoint-mapping service on a machine. They will be restarted, as necessary, by the RPC run-time libraries.

Note Only administrators can start the RPC Locator and Endpoint-mapping services once they are stopped.

{ewl msdncd, EWGraphic, group10529 8 /a "SDK.BMP"} To stop and start the RPC endpoint-mapping service

1. From the Control Panel, select Services.
   The Services dialog box appears.
2. In the Service box, select Remote Procedure Call (RPC) Service and choose Start or Stop.

{ewl msdncd, EWGraphic, group10529 9 /a "SDK.BMP"} To stop and start the Microsoft Locator

1. From the Control Panel, select Services.
   The Services dialog box appears.
2. In the Service box, select Remote Procedure Call (RPC) Locator and choose Start or Stop.

Network-Transport Information for RPC

This section contains information describing the Windows NT registry entries and information relating to SPX1IPX installations.

Registry Information

When you install Microsoft Windows NT or you run the Windows/MS-DOS Setup programs, the RPC protocol information you specify is stored in the registry file. The Windows NT registry entries are automatically configured and no further configuration is necessary. With MS-DOS and Windows 3.1, 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\ ServerProtocols\ncalrpc	Specifies the name of the RPC 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 Microsoft Windows NT, the mapping string appears in the registry tree under \SOFTWARE\Microsoft\Rpc\NetBios. For MS-DOS and Windows, 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
```

SPX/IPX Installation

When using the **ncacn_spx** and **ncadg_ipx** transports, the server name is exactly the same as the Windows NT 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 described above:

Component	**Configured As**
Windows NT Server	NWCS
Windows NT Client	NWCS
Windows 3.*x*/MS-DOS Client	NetWare Redirector

The configuration in the table above 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 Server	SAP Agent
Windows NT 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.

MIDL Command-Line Reference

This section contains reference information for each command-line switch and switch option recognized by the Microsoft RPC MIDL compiler. Switch entries are arranged in alphabetical order. The topic **midl** describes the general command-line syntax.

midl

midl [*switch* [*switch-options*]] *filename*

switch
Specifies MIDL compiler command-line switches. Switches can appear in any sequence.

switch-options
Specifies options associated with *switch*. Valid options are described in the reference entry for each MIDL compiler switch.

filename
Specifies the name of the IDL file. This file usually has the extension .IDL, but it can have any extension or no extension.

Remarks

The MIDL compiler processes an IDL file and an optional ACF to generate a set of output files. The attributes specified in the IDL file's interface attribute list determine whether the compiler generates source files for an RPC interface or for a custom OLE interface. The following lists show the default names of the files generated for an IDL file named *name*.IDL. You can use command-line switches to override these default names. Note that the name of the IDL file can have no extension, or it can have an extension other than .IDL.

By default (that is, if the interface attribute list does not contain the **object** or **local** attribute), the compiler generates the following files for an RPC interface:

- Client stub (*name*_C.C)
- Server stub (*name*_S.C)
- Header file (*name*.H)

When the **object** attribute appears in the interface attribute list, you must use the **/ms_ext** command-line switch, and the compiler generates the following files for an OLE interface:

- Interface proxy file (*name*_P.C)
- Interface header file (*name*.H)
- Interface UUID file (*name*_I.C)

When the **local** attribute appears in the interface attribute list, the compiler generates only the interface header file, *name*.H.

The MIDL compiler provided with Microsoft RPC invokes the C preprocessor as needed to process the IDL file. It does not automatically invoke the C compiler to compile generated files.

Note The MIDL compiler provided with Microsoft RPC uses a different command-line syntax than the DCE IDL compiler uses.

Files Generated for an RPC Interface

The Client Stub

The client stub module provides surrogate entry points on the client for each of the operations defined in the input IDL file.

When the client application makes a call to the remote procedure, its call first goes to the surrogate routine in the client stub file. The client stub routine performs the following functions:

- Marshals arguments. The client stub packages input arguments into a form that can be transmitted to the server.
- Calls the client run-time library to transmit arguments to the remote address space and invoke the remote procedure in the server address space.
- Unmarshals output arguments. The client stub unpackages output arguments and returns to the caller.

The MIDL compiler switches **/client**, **/cstub**, and **/out** affect the client stub file.

The Server Stub

The server stub provides surrogate entry points on the server for each of the operations defined in the input IDL file.

When a server stub routine is invoked by the RPC run-time library, it performs the following functions:

- Unmarshals input arguments (unpacks the arguments from their transmitted formats)
- Calls the actual implementation of the procedure on the server
- Marshals output arguments (packages the arguments into the transmitted forms)

The MIDL compiler switches **/env**, **/server**, **/sstub**, and **/out** affect the server stub file.

The Header File

The header file contains definitions of all the data types and operations declared in the IDL file. The header file must be included by all application modules that call the defined operations, implement the defined operations, or manipulate the defined types.

The MIDL compiler switches **/header** and **/out** affect the header file.

Files Generated for an OLE Interface

This topic describes each of the files generated for a custom OLE interface, which you identify by including the **object** attribute in the interface attribute list of the IDL file. For OLE interfaces, the MIDL compiler combines all client and object server routines into a single interface proxy file. This file includes the surrogate entry points for both clients and servers. In addition, the MIDL compiler generates an interface header file, a private header file, and an interface UUID file. You will use all these files when creating a proxy DLL that contains the code to support the use of the interface by both client applications and object servers. You will also use the interface header file and the interface UUID file when creating the executable file for a client application that uses the interface.

The Interface Proxy File

The interface proxy file (*name*_P.C) is a C file that contains routines equivalent to those in the client stub, server stub, client and server files of an RPC interface. This file contains implementations of **CProxyInterface** and **CStubInterface** classes that are derived from the **CProxy** and **CStub** classes of the base interface. For example, an interface named ISomeInterface derived from the **IUnknown** interface is implemented in the CProxyISomeInterface and CStubISomeInterface classes derived from the **CProxyIUnknown** and **CStubIUnknown** classes.

The interface proxy file includes the following sections:

- The implementation of a **CProxyInterface** class. The virtual member functions of this class provide a client application's surrogate entry points for each of the interface functions. These member functions marshal the input arguments into a transmittable form, transmit the marshalled arguments along with information that identifies the interface and the operation, and then unmarshal the return value and any output arguments when the transmitted operation returns.
- The implementation of a **CStubInterface** class. The virtual member functions of this class provide an object server's surrogate entry points for each of the interface functions. These member functions unmarshal the input arguments, invoke the server's implementation of the interface function, and then marshal and transmit the return value and any output arguments. A **CStubInterface** class also includes a member function that is invoked by the RPC run-time library when a client application calls one of the interface functions. This routine calls the surrogate routine specified by the RPC message.
- Marshalling and unmarshalling support routines for complex data types.

Use the **/proxy** MIDL compiler switch to override the default name of the interface proxy file. The **/env** and **/out** switches affect this file.

The Header Files

The interface header file (*name*.H) contains type definitions and function declarations based on the interface definition in the IDL file. Include this file in the source files for the proxy DLL and for client applications that use the interface.

The **/header** MIDL compiler switch overrides the default name of the interface header file.

The Interface UUID File

The interface UUID file defines the constant **IID_Interface** and initializes it to the interface's UUID specified in the IDL file. Client applications and the proxy DLL use this constant to identify the interface.

The **/iid** MIDL compiler switch overrides the default name of the interface UUID file.

See Also

ACF, **/app_config**, **/c_ext**, **IDL**, **/import**, **/ms_ext**

@

midl @*response_file*

response_file
Specifies the name of a response file. The response filename must immediately follow the **@** character. No white space is allowed between the **@** character and the response filename.

Examples

```
midl @midl.rsp

midl /pack 4 @midl1.rsp /env win32 @midl2.rsp itf.idl
```

Remarks

As an alternative to placing all the options associated with a switch on the command line, the MIDL compiler accepts response files that contain switches and arguments.

A response file is a text file containing one or more MIDL compiler command-line options. Unlike a command line, a response file allows multiple lines of options and filenames. This is important on systems such as MS-DOS, which limit the number of characters in the command line.

Options in a response file are interpreted as if they are present at that place in the MIDL command line.

Each argument in a response file must begin and end on the same line. The backslash character (\) can't be used to concatenate lines.

When it is part of a quoted string in the response file, the backslash character (\) can only be used before another backslash (\) or before a double quotation mark character ("). When it is not part of a quoted string, the backslash character can only be used before a double quotation mark character (").

MIDL supports command-line arguments that include one or more response files combined with other command-line switches.

The MIDL compiler doesn't support nested response files.

See Also

midl

/acf

midl /acf *acf_filename*

acf_filename
Specifies the name of the ACF. White space may or may not be present between the **/acf** switch and the filename.

Example

```
midl /acf bar.acf foo.idl
```

Remarks

The **/acf** switch allows the user to supply an explicit ACF filename. The switch also allows the use of different interface names in the IDL and ACF files.

By default, the MIDL compiler constructs the name of the ACF by replacing the IDL filename extension (usually .IDL) with .ACF. When the **/acf** switch is present, the ACF takes its name from the specified filename. The **/acf** switch applies only to the IDL file specified on the MIDL compiler command line. It does not apply to imported files.

When the **/acf** switch is used, the interface name in the ACF need not match the MIDL interface name. This feature allows interfaces to share an ACF specification.

When an absolute path to an ACF is not specified, the MIDL compiler searches in the current directory, directories supplied by the **/I** option, and directories in the INCLUDE path. If the ACF is not found, the MIDL compiler assumes there is no ACF for this interface. For more information about the sequence of directories, see **/no_def_idir** switches. For more information related to **/acf**, see **IDL.**

See Also

midl,

/app_config

midl /app_config

Examples

```
midl /app_config foo.idl
midl /app_config /ms_ext foo.idl
midl /app_config /ms_ext /c_ext foo.idl
```

Remarks

The **/app_config** switch selects application-configuration mode, which allows you to use some ACF keywords in the IDL file. With this MIDL compiler switch, you can omit the ACF and specify an interface in a single IDL file.

This release of Microsoft RPC supports the use of the following ACF attributes in the IDL file:

- **implicit_handle**
- **auto_handle**
- **explicit_handle**

Future releases of Microsoft RPC may support the use of other ACF attributes in the IDL file.

For more information related to the **/app_config** switch, see **ACF** and **IDL.**

See Also

/c_ext, **midl**, **/ms_ext**

/c_ext

midl /c_ext

Examples

```
midl /c_ext foo.idl
midl /c_ext /ms_ext foo.idl
midl /c_ext /ms_ext /app_config foo.idl
```

Remarks

The **/c_ext** switch selects the MIDL compiler mode that supports the use of C-language extensions in the IDL file. The **/c_ext** switch makes it easier for you to use existing code and header files in your distributed application.

Many existing header files define types using qualifiers, such as **far** and **stdcall**, that are not part of the DCE IDL. DCE IDL compilers (and the MIDL compiler in DCE-compatibility mode), generate errors when they attempt to process these qualifiers. The MIDL compiler **/c_ext** switch allows you to compile IDL files that contain these qualifiers.

The type qualifiers do not affect the way the data is transmitted on the network.

The following C-language extensions are supported in the **/c_ext** compiler mode:

- Bit fields in structures and unions
- Comments that start with two backslash characters
- External declarations
- Procedures with ellipses in the parameter list
- Type **int** that is not used in remote operations
- Type **void *** that is not used in remote operations
- Type qualifiers, including the form with the ANSI-conformant prefix, contain two underscore characters: **_ _cdecl, cdecl, _ _const, const, _ _export, export, _ _far, far, _ _loadds, loadds, _ _near, near, _ _pascal, pascal, _ _stdcall, stdcall, _ _volatile,** and **volatile**.

Note that directional attributes can be omitted when using **/c_ext** mode.

The three compiler-mode switches, **/ms_ext, /c_ext,** and **/app_config,** are independent. Any combination of these mode switches, including none or all, can be used.

For more information about declaration qualifiers, see your Microsoft C/C++ documentation.

See Also

/app_config, **midl**, **/ms_ext**

/caux

This switch is obsolete and, if used, results in an error.

/char

midl /char { **signed** | **unsigned** | **ascii7** }

signed
Specifies that the default C-compiler type for **char** is signed. All occurrences of **char** not accompanied by a sign specification are generated as **unsigned char**.

unsigned
Specifies that the default C-compiler type for **char** is unsigned. All uses of **small** not accompanied by a sign specification are generated as **signed small**.

ascii7
Specifies that all **char** values are to be passed into the generated files without a specific sign keyword. All uses of **small** not accompanied by a sign specification are generated as **small**.

Examples

```
midl /char signed foo.idl
midl /char unsigned foo.idl
midl /char ascii7 foo.idl
```

Remarks

The **/char** switch helps you ensure that the MIDL compiler and C compiler operate together correctly for all **char** and **small** types. By definition, MIDL **char** is unsigned. **Small** is defined in terms of **char** (**#define small char**), and MIDL **small** is signed.

The **/char** switch directs the MIDL compiler to specify explicit **signed** or **unsigned** declarations in the generated files when the C-compiler sign declaration conflicts with the MIDL default for that type.

The following table summarizes the generated types:

midl /char option	Generated char type	Generated small type
midl /char signed	**unsigned char**	**small**
midl /char unsigned	**char**	**signed small**
midl /char ascii7	**char**	**small**

The **/char signed** option indicates that the C-compiler **char** type is signed. To match the MIDL default for **char**, the MIDL compiler must convert all uses of **char** not accompanied by a sign specification to **unsigned char**. The **small** type is not modified because this C-compiler default matches the MIDL default for **small**.

The **/char unsigned** option indicates that the C-compiler **char** type is unsigned. The MIDL compiler converts all uses of **small** not accompanied by a sign specification to **signed small**.

The **ascii7** option indicates that no explicit sign specification is added to **char** types. The type **small** is generated as **small**.

To avoid confusion, you should use explicit sign specifications for **char** and small **types** whenever possible in the IDL file. Explicit sign specification is allowed only when you use the **/ms_ext** switch. The use of explicitly signed **char** types in your IDL file is not supported by DCE IDL.

For more information related to **/char**, see **small**.

/client

midl /client { **stub** | **none** }

stub
Generates the client-side files.

none
Does not generate any client-side files.

Examples

```
midl /client none foo.idl
midl /client stub foo.idl
```

Remarks

The **/client** switch directs the MIDL compiler to generate client-side C source files for an RPC interface. When the **/client** switch is not specified, the MIDL compiler generates the client stub file. This switch does not affect OLE interfaces.

The **/client** switch takes precedence over the /**cstub** switch.

See Also

/cstub, **midl**, **/server**

/confirm

midl /confirm

Examples

```
midl /confirm
midl /confirm @response.rsp foo.idl
```

Remarks

The **/confirm** switch instructs the compiler to display all MIDL compiler options without processing the input IDL (and optional ACF) files.

See Also

/help, **midl**

/cpp_cmd

midl /cpp_cmd "*C_preprocessor_command***"**

C_preprocessor_command
Specifies the command that invokes the C preprocessor. This command allows you to override the default C preprocessor. By default, MIDL invokes the Microsoft C compiler for the build environment you are using.

Examples

```
midl /cpp_cmd "cl386" /cpp_opt "/E" foo.idl
midl /cpp_cmd "mycpp" /DFLAG=TRUE /Ic:\tmp foo.idl
midl /cpp_opt "/E /DFLAG=TRUE" foo.idl
```

Remarks

The **/cpp_cmd** switch specifies the C-compiler preprocessor that the MIDL compiler uses to preprocess the IDL and ACF files. When this switch is present, the *C_preprocessor_command* option is required.

When the specified C preprocessor does not direct its output to **stdout**, you must specify the C compiler switch that redirects output to **stdout** as part of the MIDL compiler **/cpp_opt** switch.

The C preprocessor is invoked by a command string that is formed from the information provided to the MIDL compiler **/cpp_cmd, /cpp_opt, /D, /I,** and **/U** switches. The following table summarizes how the command string is constructed for each combination of **/cpp_cmd** and **/cpp_opt** switches:

/cpp_cmd present?	**/cpp_opt present?**	**Description**
Yes	Yes	Invokes specified C compiler with specified options; you must supply **/E** as part of **/cpp_opt**
Yes	No	Invokes specified C compiler with settings obtained from MIDL **/I**, **/D**, **/U** switches; adds C compiler **/E** switch
No	Yes	Invokes Microsoft C compiler with specified options; does not use MIDL **/I**, **/D**, **/U** options; you must supply **/E** as part of **/cpp_opt**
No	No	Invokes Microsoft C compiler with **/E** option only

When the /**cpp_cmd** switch is not specified, the MIDL compiler invokes the Microsoft C/C++ compiler for that environment.

When the **/cpp_opt** switch is not present, the MIDL compiler concatenates the string specified by the /**cpp_cmd** switch with the information specified by the MIDL **/I**, **/D**, and **/U** options. The string **/E** is also concatenated to the C-compiler invocation string to indicate that the C compiler should perform preprocessing only. The MIDL compiler uses the concatenated string to invoke the C preprocessor for each IDL and ACF source file.

When the **/cpp_opt** switch is present, the MIDL compiler concatenates the string specified by the **/cpp_cmd** switch with the string specified by the **/cpp_opt** switch. The MIDL compiler uses the concatenated string to invoke the C preprocessor for each IDL and ACF source file. When the **/cpp_opt** switch is present, neither the MIDL compiler options specified by the **/I, /D,** and **/U** switches nor the C compiler switch **/E** is concatenated with the string. You must supply the **/E** option as part of

the string.

See Also

/cpp_opt, midl, /no_cpp

/cpp_opt

midl /cpp_opt "*C_preprocessor_option***"**

C_preprocessor_option
Specifies a command-line option associated with the C preprocessor. You must supply the C-compiler option **/E** as part of the *C_preprocessor_option* string.

Examples

```
midl /cpp_cmd "cl386" /cpp_opt "/E" foo.idl
midl /cpp_cmd "mycpp" /DFLAG=TRUE /Ic:\tmp foo.idl
midl /cpp_opt "/E /DFLAG=TRUE" foo.idl
```

Remarks

The **/cpp_opt** switch specifies options to pass to the C preprocessor. The **/cpp_opt** switch can be used with or without the **/cpp_cmd** switch. The following table summarizes how the C-preprocessor command string is constructed for each combination of **/cpp_cmd** and **/cpp_opt** switches:

/cpp_cmd present?	**/cpp_opt present?**	**Description**
Yes	Yes	Invokes specified C compiler with specified options; you must supply **/E** as part of **/cpp_opt**
Yes	No	Invokes specified C compiler with settings obtained from MIDL **/I**, **/D**, **/U** switches; adds C-compiler **/E** switch
No	Yes	Invokes Microsoft C compiler with specified options; does not use MIDL **/I**, **/D**, **/U** options; you must supply **/E** as part of **/cpp_opt**
No	No	Invokes Microsoft C compiler with **/E** option only

When the **/cpp_cmd** switch is present and the **/cpp_opt** switch is not, the MIDL compiler concatenates the string specified by the **cpp_cmd** switch with the **/I**, **/D**, and **/U** options and uses this concatenated string to invoke the C preprocessor for each IDL and ACF source file.

When the **/cpp_cmd** switch is not present, the preprocessor option is sent to the default C preprocessor. When the **/cpp_cmd** switch is present, the preprocessor option is sent to the specified C preprocessor.

See Also

/cpp_cmd, **midl**, **/no_cpp**

/cstub

midl /cstub *stub_file_name*

stub_file_name
Specifies a filename that overrides the default client stub filename. Filenames can be explicitly quoted using double quotes (") to prevent the shell from interpreting the special characters.

Example

```
midl /cstub my_cstub.c foo.idl
```

Remarks

The **/cstub** switch specifies the name of the client stub file for an RPC interface. The specified filename replaces the default filename. By default, the filename is obtained by adding the extension _C.C to the name of the IDL file. This switch does not affect OLE interfaces.

When you are importing files, the specified filename applies to only one stub file – the stub file that corresponds to the IDL file specified on the command line.

If *stub_file_name* does not include an explicit path, the file is written to the current directory or the directory specified by the **/out** switch. An explicit path in *stub_file_name* overrides the **/out** switch specification.

The **/client none** switch takes precedence over the **/cstub** switch.

See Also

/header, **midl**, **/out**, **/sstub**

/D

midl /D*name*=*definition*

name
Specifies a defined name that is passed to the C preprocessor when the **/cpp_cmd** switch is present and the **/cpp_opt** switch is not present.

definition
Specifies a value associated with the defined name.

Example

```
midl -DUNICODE foo.idl
```

Remarks

The **/D** switch defines a name and an optional value to be passed to the C preprocessor as if by a **#define** directive. Multiple **/D** directives can be used in a command line. White space between the **/D** switch and the defined name is optional.

When the **/cpp_cmd** switch is present and the **/cpp_opt** switch is not, the MIDL compiler concatenates the string specified by the /**cpp_cmd** switch with the **/I**, **/D**, and **/U** options and uses this concatenated string to invoke the C preprocessor for each IDL and ACF source file.

The MIDL compiler switch **/D** is ignored when the MIDL compiler switch **/no_cpp** or **/cpp_opt** is specified.

See Also

/cpp_cmd, **/cpp_opt**, **/I**, **midl**, **/no_cpp**, **/U**

/dlldata

midl /dlldata

Example

```
midl /dlldata data.c
```

Remarks

The **/dlldata** switch is used to specify the filename for the generated dlldata file for a proxy DLL. The default filename "dlldata.c" is used if the **/dlldata** switch is not specified.

The dlldata file must be linked to the proxy DLL. The dlldata file contains entry points and data structures required by the class factory for the proxy DLL. These data structures specify the object interfaces contained in the proxy DLL. The dlldata file also specifies the class ID of the class factory for the proxy DLL. This is always the UUID (IID) of the first interface of the first proxy file (by alphabetical order).

The same dlldata file should be specified when invoking MIDL on all the IDL files that will go into a particular proxy DLL. The dlldata file is created or updated during each MIDL compilation, incrementally building a list of the interfaces that will go into the proxy DLL.

See Also

midl

/env

midl /env { **dos** | **win16** | **win32** }

dos
Directs the MIDL compiler to generate stub files for an MS-DOS environment.

win16
Directs the MIDL compiler to generate stub files for the 16-bit Microsoft Windows environment such as Microsoft Windows 3.*x* or Microsoft Windows for Workgroups 3.1.

Note The server-stub file is not generated when you use the MIDL compiler switch **/env** with either the **dos** or **win16** option.

win32
Directs the MIDL compiler to generate stub files for a 32-bit Microsoft Windows environment such as Microsoft Windows NT.

Examples

```
midl /env dos foo.idl
midl /env win32 foo.idl
```

Remarks

The **/env** switch selects the environment in which the application runs. The **/env** switch primarily affects the packing level used for structures in that environment.

Specify the same packing-level setting for both the MIDL compiler and the C compiler.

The **/env** switch determines the packing level and other settings as follows:

/env	Packing level	Description
/env dos	2 or **/Zp** setting	**_ _far** precedes pointer declarations.
/env win16	2 or **/Zp** setting	**_ _far** precedes pointer declarations.
/env win32	8 or **/Zp** setting	No declaration qualifiers are added.

When **dos** is selected, **_ _far** precedes pointer declarations in the generated header file, and the stub files use packing-level 2 for all types involved in remote operations.

When **win16** is selected, **_ _far** precedes pointer declarations in the generated files, stub files assume C-compiler packing-level 2 for all types involved in remote operations, and **_ _export** is applied to callback stubs on the client side. You must compile the stubs with the **/GA** option.

When **win32** is selected, generated stubs assume C-compiler packing-level 8 for all types involved in remote operations.

When the 32-bit version of the MIDL compiler runs on Microsoft Windows NT, stubs are generated for the **win32** environment.

The compiler applies packing-level **/Zp4** to all types involved in a remote procedure call.

The **/pack** and **/Zp** switches take precedence over the **/env** settings.

See Also

midl, **/pack**, **/Zp**

/error

midl /error { allocation | stub_data | none | all }

allocation
Checks whether **midl_user_allocate** returns a null value, indicating an out-of-memory error.

stub_data
Generates a stub that catches unmarshalling exceptions on the server side and propogates them back to the client.

none
Performs no error checking.

all
Performs all error checking.

Examples

```
midl /error allocation foo.idl
midl /error none foo.idl
```

Remarks

The **/error** switch selects the amount of error checking to be performed by the generated stub files.

By default, the MIDL compiler generates code that checks for enum and memory-access errors. Enum errors are truncation errors caused by conversion between **long enum** types (32-bit integers) and **short enum** types (the network-data representation of **enum**). Memory-access errors are errors caused when array indexes are out of bounds or when a pointer exceeds the end of the buffer in marshalling code.

When you specify **/error allocate**, the stubs include code that raises an exception when **midl_user_allocate** returns 0.

The **/error stub_data** option prevents client data from crashing the server during unmarshalling; in effect providing a more robust method of handling the unmarshalling operation.

See Also

midl

/header

midl /header *filename*

filename
Specifies a header filename that overrides the default header filename. Filenames can be explicitly quoted using double quotes (") to prevent the shell from interpreting the special characters.

Example

```
midl /header "bar.h" foo.idl
```

Remarks

The **/header** switch specifies the name of the header file. The specified filename replaces the default filename. The default filename is obtained by replacing the IDL file extension (usually .IDL) with the extension .H. For OLE interfaces, the **/header** switch overrides the default name of the interface header file.

When you are importing files, the specified filename applies to only one header file– the header file that corresponds to the IDL file specified on the command line.

If *filename* does not include an explicit path, the file is written to the current directory or the directory specified by the **/out** switch. An explicit path in *filename* overrides the **/out** switch specification.

See Also

/cstub, **midl**, **/out**, **/sstub**, **/proxy**

/help (/?)

midl /help
midl /?

Example

```
midl /help
```

Remarks

The **/help** (**/?**) switch instructs the compiler to display a usage message detailing all available MIDL command-line switches and options.

The **/confirm** switch displays the MIDL compiler switch settings selected by the user.

See Also

/confirm, **midl**

/I

midl /I*include_path*

include_path
Specifies one or more directories that contain import, include, and ACF files. White space between the **/I** switch and *include_path* is optional. Separate multiple directories with a semicolon character (;).

Example

```
midl /I c:\include;c:\include\h /I\include2 foo.idl
```

Remarks

The **/I** switch specifies directories to be searched for imported IDL files, included header files, and ACF files. More than one directory can appear with each **/I** switch, and more than one **/I** switch can appear with each MIDL compiler invocation. Directories are searched in the order they are specified.

The **/I** switch setting is also passed by the MIDL compiler to the C compiler's C preprocessor. When the **/cpp_cmd** switch is present and the **/cpp_opt** switch is not, the MIDL compiler concatenates the string specified by the /**cpp_cmd** switch with the **/I**, **/D**, and **/U** options and uses this concatenated string to invoke the C preprocessor for each IDL and ACF source file. The MIDL compiler switch **/I** is not passed to the preprocessor when the MIDL compiler switch **/no_cpp** or **/cpp_opt** is specified.

In Microsoft operating-system environments (Windows NT, Windows 3.x, Windows for Workgroups, and MS-DOS), directories are searched in the following sequence:

1. Current directory
2. Directories specified by the **/I** switch (in order as they appear following the switch)
3. Directories specified by the INCLUDE environment variable

When directories are specified with the **/I** switch, the **/no_def_idir** switch instructs the MIDL compiler to ignore the current directory, ignore the directories specified by the INCLUDE environment variable, and search only the specified directories.

When no directories are specified with the **/I** switch, the **/no_def_idir** switch instructs the MIDL compiler to search only the current directory.

See Also

/acf, **/cpp_cmd**, **/cpp_opt**, **midl**, **/no_def_idir**

/iid

midl /iid *filename*

filename
Specifies an interface identifier filename that overrides the default interface identifier filename for an OLE interface. Filenames can be explicitly quoted using double quotes (") to prevent the shell from interpreting the special characters.

Example

```
midl /iid "foo_iid.c" foo.idl
```

Remarks

The **/iid** switch specifies the name of the interface identifier file for an OLE interface, overriding the default name obtained by adding _I.C to the IDL filename. The **/iid** switch does not affect RPC interfaces.

If *filename* does not include an explicit path, the file is written to the current directory or to the directory specified by the **/out** switch. An explicit path in *filename* overrides the **/out** switch specification.

See Also

/header, **midl**, **/out**, **/proxy**

/import

This switch is obsolete and if used, results in an error.

/ms_ext

midl /ms_ext

Examples

```
midl /ms_ext foo.idl
midl /ms_ext /app_config foo.idl
midl /ms_ext /app_config /c_ext foo.idl
```

Remarks

The **/ms_ext** switch enables Microsoft extensions to DCE IDL. The following features are supported in Microsoft-extensions mode:

- Interface definitions for OLE objects. For more information on the files generated for OLE interfaces, see **midl**.
- A **callback** attribute specifying a static callback function on the client
- **cpp_quote**(*quoted_string*) and **#pragma midl_echo**
- **wchar_t** wide-character types, constants, and strings
- **enum** initialization (sparse enumerators)
- Expressions as size and discriminator specifiers
- Handle extensions
- Pointer-attribute type inheritance
- Multiple interfaces
- Definitions outside of the interface block

Note that directional attributes can be omitted when using the **/ms_ext** mode.

Handle Extensions

The MIDL compiler switch **/ms_ext** allows Microsoft extensions to the IDL language that 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 and in Microsoft-extensions mode:

	DCE-compatibility mode		**Microsoft-extensions 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 extension 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 mode and Microsoft-extensions 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, [in] short s );`	An explicit handle of type **handle_t** is specified. The parameter H is the binding handle for the procedure.
`void proc3( [in] short s, [in] handle_t H );`	The first parameter is not a handle. In DCE-compatibility mode, implicit binding is used. An error is reported because the second parameter is expected to be transmissible, and **handle_t** cannot be transmitted. In Microsoft-extensions mode, the leftmost handle parameter, H, is the binding handle.
`typedef [handle] short * MY_HDL;` `void proc1([in] short s, [in] MY_HDL H );`	The first parameter is not a handle. In DCE-compatibility mode, implicit binding is used. The user-defined handle parameter H is treated as transmissible data. In Microsoft-extensions mode, the leftmost handle parameter, H, is the binding handle. The stubs call the user-supplied routines MY_HDL_bind and MY_HDL_unbind.
`typedef [handle] short * MY_HDL;` `void proc1([in] MY_HDL H, [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                     The binding handle is a context handle.
[context_handle]            The parameter H is the binding handle.
   void * CTXT_HDL;

void proc1([in] short
s,
      [in] long l,
      [in] CTXT_HDL H ,
      [in] char c);
```

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. DCE does not allow unattributed pointers. If a pointer does not have an explicit attribute, the IDL file must have a **pointer_default** specification so that the pointer attribute can be set.

In Microsoft-extensions mode, you can control pointer attributes from the base IDL file. This feature allows an IDL file to contain pointer types whose pointer attributes are not resolved until that IDL file is imported by another IDL file. These kinds of pointers are known as unattributed pointers. When 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 applied to the pointer at the definition or use site
2. **Pointer_default** in the IDL file that defines the type
3. **Pointer_default** in the IDL file that imports the type
4. **Unique** in Microsoft-extensions mode; **ptr** in DCE-compatibility mode

Two other switches control MIDL language features: **/app_config** and **/c_ext**. These switches can be used independently of the **/ms_ext** switch. For more information about these compiler modes, see **/app_config** and **/c_ext**.

See Also

midl

/ms_union

midl /ms_union

Example

```
midl /ms_union file.idl
```

Remarks

The **/ms_union** switch controls the NDR alignment of non-encapsulated unions.

The MIDL compiler mirrors the behavior of the OSF DCE IDL compiler for non-encapsulated unions. However, because earlier versions of the MIDL compiler did not do so, the **/ms_union** switch provides compatibility with interfaces built on previous versions of the MIDL compiler.

The ms_union feature can be used as a command line switch (**/ms_union**), an IDL interface attribute, or as an IDL type attribute.

See Also

IDL, **ms_union**

/no_cpp

midl /no_cpp

Example

```
midl /no_cpp foo.idl
```

Remarks

The **/no_cpp** switch specifies that the MIDL compiler does not call the C preprocessor to preprocess the IDL file.

The **/no_cpp** switch takes precedence over the **/cpp_cmd** and **/cpp_opt** switches.

See Also

/cpp_cmd, **/cpp_opt**, **/D**, **/I**, **midl**, **/U**

/no_default_epv

midl /no_default_epv

Example

```
midl /no_default_epv foo.idl
```

Remarks

The **/no_default_epv** switch directs the MIDL compiler not to generate a default epv. In this case, the application must register an epv with the RpcServerRegisterIf call. Compare this switch with the **/use_epv** switch described earlier in this chapter.

See Also

IDL, **/use_epv**, **RpcServerRegisterIf**

/no_def_idir

midl /no_def_idir

Examples

```
; search only the current directory
midl /no_def_idir foo.idl
; search only the specified directories
midl /no_def_idir /I c:\c700\include foo.idl
```

Remarks

When directories are specified with the **/I** switch, the **/no_def_idir** switch instructs the MIDL compiler to search only the directories specified with the **/I** switch, ignoring the current directory and ignoring the directories specified by the INCLUDE environment variable.

When no directories are specified with the **/I** switch, the **/no_def_idir** switch instructs the MIDL compiler to search only the current directory.

See Also

/acf, **/I**, **midl**

/no_warn

midl /no_warn

Examples

```
midl /no_warn foo.idl
midl /W0 foo.idl
```

Remarks

The **/no_warn** switch directs the MIDL compiler to suppress warning messages. The use of the **/no_warn** switch is equivalent to **/W0**.

See Also

midl, **/W**, **/WX**

/Oi

midl /Oi
midl /Oi1

Examples

```
midl /Oi foo.idl
midl /Oi /ms_ext foo.idl
midl /Oi /ms_ext /c_ext foo.idl
midl /Oi1 foo.idl
```

Remarks

The **/Oi** switch specifies the fully-interpreted method to marshal stub code passed between client and server.

The **/Oi1** option of this switch can be used to generate interpreted stub code if you are using version 3.51 of Windows NT. Note that this option will not work with the 3.5 version of Windows NT.

Note Stubs generated by the MIDL compiler in the interpreted method using the **/Oi** switch must be compiled as either a stdcall or a cdecl procedure during the C compilation. A PASCAL or Fastcall calling convention will not work. Additionally, the server stub must be compiled as __stdcall.

The number of parameters allowed is limited to 16 parameters on all platforms. Any procedure containing more than 16 parameters will automatically be processed in **/Os** mode.

There are important issues to consider before deciding on the method for marshalling code. These issues concern size and performance. The MIDL 2.0 compiler provides two methods for marshalling code: fully-interpreted (**/Oi**) and mixed-mode (**/Os**). Mixed-mode is the default.

The fully-interpreted method marshals data completely offline. This considerably reduces the size of the stub code. However, it also results in decreased performance.

If performance is a concern, the mixed-mode (**/Os**) method can be the best approach. In this mode, the compiler chooses to marshall some parameters inline in the generated stubs. While this results in larger stub size, it offers increased performance.

To further define the level of gradiation in how data is marshalled, this version of RPC provides an optimize attribute. This attribute is used as an ACF interface attribute or operation attribute to select the marshalling mode.

See Also

/Os, **optimize**

/oldnames

midl /oldnames

Example

```
midl /oldnames foo.idl
```

Remarks

The **/oldnames** switch directs the MIDL compiler to generate interface names which do not include the version number.

The MIDL 2.0 compiler incorporates the version number of the interface into the interface name that is generated in the stub (for example, foo_v1_0_ServerIfHandle). This naming format is consistent with the format used by the OSF DCE IDL compiler. However, it differs from the naming format used by the MIDL 1.0 compiler. The MIDL 1.0 compiler did not include version numbers in interface names (for example, foo_ServerIfHandle). The **/oldnames** switch allows you to instruct the MIDL compiler to generate interface names which do not include the version number. In this way, the format is consistent with names generated by the MIDL 1.0 compiler.

If you have server application code that was written for use with a stub generated by the MIDL 1.0 compiler, and it refers to the MIDL-generated interface name (for example, in a call to **RpcServerRegisterIf**), you must either change it to reference the MIDL 2.0 style of interface name or use the **/oldnames** switch when invoking the MIDL compiler.

See Also

IDL

/Os

midl /Os

Examples

```
midl /Os foo.idl
midl /Os /ms_ext foo.idl
midl /Os /ms_ext /c_ext foo.idl
```

Remarks

The **/Os** switch specifies the mixed-mode method to marshal stub code passed between client and server.

There are important issues to consider before deciding on the method for marshalling code. These issues concern size and performance. The MIDL 2.0 compiler provides two methods for marshalling code: mixed-mode (**/Os**) and fully-interpreted (**/Oi**). The fully-interpreted method marshals data completely offline. This reduces the size of the stub code considerably. However, it also results in decreased performance.

If performance is an important concern, the mixed-mode method (**/Os**) can be the best approach. In this mode, the compiler marshals some parameters inline in the generated stubs. While this results in larger stub size, it also offers increased performance. Because mixed-mode is the default, you need not explicitly select the **/Os** switch to accomplish mixed-mode marshalling.

To further define the level of gradiation in how data is marshalled, this version of RPC provides an **optimize** attribute. This attribute is used as an ACF interface attribute or operation attribute to select the marshalling mode.

See Also

/Oi, **optimize**

/out

midl /out *path-specification*

path-specification
Specifies the path to the directory in which the stub, header, and switch files are generated. A drive specification, an absolute directory path, or both can be specified. Paths can be explicitly quoted using double quotes (") to prevent the shell from interpreting the special characters.

Examples

```
midl /out c:\mydir\mysubdir\subdir2 deeper foo.idl
midl /out c:  foo.idl
midl /out \mydir\mysubdir\another foo.idl
```

Remarks

The **/out** switch specifies the default directory where the MIDL compiler writes output files. The output directory can be specified with a drive letter, an absolute path name, or both. The **/out** option can be used with any of the switches that enable individual output file specification.

When the **/out** switch is not specified, files are written to the current directory.

The default directory specified by the **/out** switch can be overridden by an explicit path name specified as part of the **/cstub, /header, /proxy,** or **/sstub** switch.

See Also

/cstub, **/header**, **midl**, **/proxy**, **/sstub**

/pack

midl /pack *packing_level*

packing_level
Specifies the packing level of structures in the target system. The packing-level value can be set to 1, 2, 4, or 8.

Examples

```
midl /pack 2 foo.idl
midl /pack 8 bar.idl
```

Remarks

The **/pack** switch is the same as the **/Zp** option. The **/pack** switch designates the packing level of structures in the target system. The packing-level value corresponds to the **/Zp** option value used by the Microsoft C/C++ version 7.0 compiler. For more information, see your Microsoft C/C++ programming documentation.

Specify the same packing level when you invoke the MIDL compiler and the C compiler. The default is 8.

For a discussion of the potential dangers in using non-standard packing levels, see the **/Zp** help topic.

See Also

midl, **/env**, **/Zp**

/prefix

midl /prefix { client | server | switch | all }

client
Affects only the client stub routine names.

server
Affects only the routine names called by the server stub routine.

switch
Affects an extra prototype added to the header file.

all
Affects both the client and server stub routine names.

Examples

```
midl /prefix client "c_" server "s_"
midl /prefix all "foo_"
midl /prefix client "bar_"
```

Remarks

The **/prefix** switch directs the MIDL compiler to add prefix strings to the client and/or server stub routine names. This can be used to allow a single program to be both a client and server of the same interface, without having the client- and server-side routine names conflict with each other. If the prefix for the client-side routines is different from the prefix for the server-side routines, the generated header file will have both client-side routine prototypes and server-side routine prototypes.

The **/prefix** switch is useful when a single header file will be used with stubs from multiple runs of the MIDL compiler. This forces additional routine prototypes in the header file.

In all cases, the **client**, **server**, and **switch** prefixes will override an **all** prefix.

See Also

midl

/proxy

midl /proxy *proxy_file_name*

proxy_file_name
Specifies a filename that overrides the default interface proxy filename. Filenames can be explicitly quoted using double quotes (") to prevent the shell from interpreting the special characters.

Example

```
midl /proxy my_proxy.c foo.idl
```

Remarks

The **/proxy** switch specifies the name of the interface proxy file for an OLE interface. The specified filename replaces the default filename obtained by adding _P.C to the name of the IDL file. The /**proxy** switch does not affect RPC interfaces.

If *proxy_file_name* does not include an explicit path, the file is written to the current directory or to the directory specified by the **/out** switch. An explicit path in *proxy_file_name* overrides the **/out** switch specification.

For a more detailed description of the interface proxy file and other files generated by the MIDL compiler, see **midl**.

See Also

/header, **/iid**, **midl**, **/out**

/rpcss

midl /rpcss

Example

```
midl /rpcss foo.idl
```

Remarks

The **/rpcss** switch, when specified, acts as though the **enable_allocate** attribute was specified on all operations of the interface. In osf mode, the server side stub enables the rpcss allocation package. However, when using the **ms_ext** mode of operation, the rpcss package is enabled only if either the procedure or interface has the **enable_allocate** attribute or the **/rpcss** switch is specified on the command line.

See Also

IDL, **enable_allocate**, **/ms_ext**

/saux

This switch is obsolete and if used, results in an error.

/server

midl /server { **stub** | **none** }

stub
Generates the server-side files.

none
Does not generate server-side files.

Examples

```
midl /server none
midl /server stub
```

Remarks

When the **/server** switch is not specified, the MIDL compiler generates the server stub file. This switch does not affect OLE interfaces.

The **none** option causes no files to be generated.

The **/server** switch takes precedence over the /**sstub** switch.

See Also

/client, **midl**, **/sstub**

/sstub

midl /sstub *stub_file_name*

stub_file_name
Specifies a filename that overrides the default server stub filename. Filenames can be explicitly quoted using double quotes (") to prevent the shell from interpreting the special characters.

Example

```
midl /sstub my_sstub.c foo.idl
```

Remarks

The **/sstub** switch specifies the name of the server stub file for an RPC interface. The specified filename replaces the default filename. By default, the filename is obtained by adding _S.C to the name of the IDL file. This switch does not affect OLE interfaces.

When you are importing files, the specified filename applies to only one stub file – the stub file that corresponds to the IDL file specified on the command line.

If *stub_file_name* does not include an explicit path, the file is written to the current directory or the directory specified by the **/out** switch. An explicit path in *stub_file_name* overrides the **/out** switch specification.

The **/server none** switch takes precedence over the **/sstub** switch.

See Also

/cstub, **/header**, **/out**

/syntax_check

midl /syntax_check
midl /Zs

Examples

```
midl /Zs foo.idl
midl /syntax_check foo.idl
```

Remarks

The **/syntax_check** switch indicates that the compiler checks only syntax and does not generate output files. This switch overrides all other switches that specify information about output files.

You can also specify syntax-checking mode with the MIDL compiler option **/Zs**.

See Also

midl, **/Zs**

/U

midl /**U***name*

name
Specifies a defined name to be passed to the C preprocessor as if by a **#undefine** directive. The name is predefined by the C preprocessor or previously defined by the user.

Example

```
midl /UUNICODE foo.idl
```

Remarks

The **/U** switch removes any previous definition of a name by passing the name to the C preprocessor as if by a **#undefine** directive. Multiple **/U** directives can be used in a command line. White space between the **/U** switch and the undefined name is optional.

When the **/cpp_cmd** switch is present and the **/cpp_opt** switch is not, the MIDL compiler concatenates the string specified by the /**cpp_cmd** switch with the **/I**, **/D**, and **/U** options and uses this concatenated string to invoke the C preprocessor for each IDL and ACF source file. The MIDL compiler switch **/U** is ignored when the MIDL compiler switch **/no_cpp** or **/cpp_opt** is specified.

See Also

/cpp_cmd, **/cpp_opt**, **/D**, **/I**, **midl**, **/no_cpp**

/use_epv

midl /use_epv

Example

```
midl /use_epv foo.idl
```

Remarks

The **/use_epv** switch directs the MIDL compiler to generate server stub code that calls the server application routine through an entry point vector (epv), rather than by a static call.

Typically, applications require static linkage to the server application routine. The MIDL compiler generates such a call by default. However, if an application requires the server stub to call the server application routine by using the epv, the **/use_epv** switch must be specified. When the /**use_epv** switch is specified, the MIDL compiler generates a default epv. This default epv is then used if the application does not register another epv through the **RpcServerRegisterIf** call.

See Also

IDL, **/no_default_epv**, **RpcServerRegisterIf**

/W

midl /W*level*

level
Specifies the warning level, an integer in the range 0 through 4. There is no space between the **/W** switch and the digit indicating the warning-level value.

Examples

```
midl /W2 foo.idl
midl /W4 bar.idl
```

Remarks

The **/W** switch specifies the warning level of the MIDL compiler; the warning level indicates the severity of the warning. Warning levels range from 1 to 4, with a value of zero meaning to display no warning information. The highest-severity warning is level 1. The following table describes the warnings for each warning level:

Warning level	Description	Example
W0	No warnings	
W1	Severe warnings that can cause application errors	No binding handle specified, unattributed pointers, conflicting switches
W2	May cause problems in the user's operating environment	Identifier length exceeds 31 characters; no default union arm specified
W3	Reserved	
W4	Lowest warning level	Non-ANSI C constructs

Warnings are different from errors. Errors cause the MIDL compiler to halt processing of the IDL file. Warnings cause the MIDL compiler to emit an informational message and continue processing the IDL file.

The warning level set by the **/W** switch can be used with the **/WX** switch to cause the MIDL compiler to halt processing of the IDL file.

The **/W** switch behaves the same as the **/warn** switch.

See Also

midl, **/warn**,

/warn

midl /warn*level*

level
Specifies the warning level, an integer in the range 0 through 4. There is no space between the **/warn** switch and the digit indicating the warning-level value.

Examples

```
midl /warn2 foo.idl
midl /warn4 bar.idl
```

Remarks

The **/warn** switch specifies the warning level of the MIDL compiler; the warning level indicates the severity of the warning. Warning levels range from 1 to 4, with a value of zero meaning to display no warning information. The highest severity warning is level 1. The following table describes the warnings for each warning level:

Warning level	Description	Example
0	No warnings	
1	Severe warnings that can cause application errors	No binding handle specified, unattributed pointers, conflicting switches
2	May cause problems in the user's operating environment	Identifier length exceeds 31 characters; no default union arm specified
3	Reserved	
4	Lowest warning level	Non-ANSI C constructs

Warnings are different from errors. Errors cause the MIDL compiler to halt processing of the IDL file. Warnings cause the MIDL compiler to emit an informational message and continue processing the IDL file.

The warning level set by the **/warn** switch can be used with the **WX** switch to cause the MIDL compiler to halt processing of the IDL file.

The **/warn** switch behaves the same as the **/W** switch.

See Also

midl

/WX

midl /WX

Examples

```
midl /WX foo.idl
midl /W3 /WX foo.idl
```

Remarks

The **/WX** switch instructs the MIDL compiler to handle all errors at the given warning level as errors. If the **/WX** switch is specified and the **/W*n*** switch is not specified, all warnings at the default level, level 1, are treated as errors.

The **/W*n*** switch directs the compiler to display all warnings at level *n* and **/WX** directs the compiler to handle all warnings as errors. The combination of these two switches directs the compiler to handle all warnings at level *n* as errors.

Errors are different from warnings. Errors cause the MIDL compiler to halt processing of the IDL file. Warnings cause the MIDL compiler to emit an informational message and continue processing the IDL file.

Warning-level zero (0) directs the MIDL compiler to suppress warning information. When the **/W0** and **/WX** switches are combined, the MIDL compiler suppresses all warning information. In this case, the **/WX** switch has no effect.

See Also

midl, **/W**

/Zp

midl /Zp*packing_level*

packing_level
Specifies the packing level of structures in the target system. The packing-level value can be set to 1, 2, 4, or 8.

Example

```
midl /Zp4 foo.idl
```

Remarks

The **/Zp** switch is the same as the **/pack** option.

The **/Zp** switch designates the packing level of structures in the target system. The packing-level value corresponds to the **/Zp** option value used by the Microsoft C/C++ compiler. For more information, see your Microsoft C/C++ programming documentation.

Specify the same packing level when you invoke the MIDL compiler and the C compiler.

The default packing level used when you do not specify the **/Zp** or **/pack** switch is 8.

Note Do not use /**Zp1** or /**Zp2** on MIPS or Alpha platforms, and do not use /**Zp4** or /**Zp8** on 16-bit platforms. Depending on the data type and memory location assigned by the C compiler at runtime, this can result in a data misalignment exception on MIPS and Alpha platforms. On MS-DOS platforms, the C compiler will not ensure the alignment at 4 or 8, and so the application may terminate.

See Also

midl, **/pack**

/Zs

midl /Zs
midl /syntax_check

Examples

```
midl /Zs foo.idl
midl /syntax_check foo.idl
```

Remarks

The **/Zs** switch indicates that the compiler only checks syntax and does not generate output files.

This switch overrides all other switches that specify information about output files.

You can also specify syntax-checking mode with the MIDL compiler switch **/syntax_check**.

See Also

midl, **/syntax_check**

MIDL Reference

This section provides a reference entry for each keyword in the Microsoft Interface Definition Language (MIDL) and application configuration file (ACF). Reference entries are also provided for important language productions and concepts. Each reference entry includes syntax, examples, descriptions, and cross-references.

The reference entries are arranged in alphabetical order. To examine the top-level structure of these files, start with the topics **ACF** and **IDL**.

ACF

[interface-attribute-list] interface *interface-name*
{
[**include** *filename-list* **; ...**]
[**typedef [***type-attribute-list***]** *typename***; ...**]

[[**[***function-attribute-list***]**] *function-name***(**
[[**[***parameter-attribute-list***]**] *parameter-name*]
...
);
]
...
}

.
.
.

interface-attribute-list
Specifies a list of one or more attributes that apply to the interface as a whole. Valid attributes include **auto_handle**, **implicit_handle**, **explicit_handle**, and **optimize**, **code** or **nocode**. When two or more interface attributes are present, they must be separated by commas.

interface-name
Specifies the name of the interface. In DCE-compatibility mode, the interface name must match the name of the interface specified in the IDL file. When you use the MIDL compiler switch **/acf**, the interface name in the ACF and the interface name in the IDL file can be different.

filename-list
Specifies a list of one or more C-language header filenames, separated by commas. The full filename, including the extension, must be supplied.

type-attribute-list
Specifies a list of one or more attributes, separated by commas, that apply to the specified type. Valid type attributes include **allocate** or **represent_as**.

typename
Specifies a type defined in the IDL file. Type attributes in the ACF can only be applied to types previously defined in the IDL file.

function-attribute-list
Specifies a list of one or more attributes, separated by commas, that apply to the function-return type. Valid function attributes include **allocate**, **optimize**, call_as, **code** or **nocode**.

function-name
Specifies the name of the function in the IDL file.

parameter-attribute-list
Specifies a list of zero or more attributes, separated by commas, that apply to the specified parameter. Valid parameter attributes include **byte_count.**

parameter-name
Specifies the name of the parameter in the IDL file. Only the name of the parameter must match the IDL file specification. The sequence of parameters is not significant.

Examples

```
/* example 1 */
[auto_handle] interface foo1 { }
```

```
/* example 2 */
[implicit_handle(handle_t h), code] interface foo2 {}

/* example 3 */
[code]
interface foo3;
{
     include "foo3a.h", "foo3b.h";
     typedef [allocate(all_nodes)] TREETYPE1;
     typedef [allocate(all_nodes, dont_free)] TREETYPE2;
     f1([byte_count(length)] pBuffer);
}
```

Remarks

The application configuration file, or ACF, is one of two files that define the interface for your distributed application. The second interface-defining file is the IDL file. The IDL file contains type definitions and function prototypes that describe how data is transmitted on the network. The ACF configures your application for a particular operating environment without affecting its network characteristics.

By using the IDL and ACF files, you separate the interface specification from environment-specific settings. The IDL file is meant to be portable to any other computer. When you move your distributed application to another computer, you should be able to reuse the IDL file. Environment-specific changes are made in the ACF.

Many distributed applications require no special configuration. For such applications, use the MIDL compiler switch **/app_config** to supply the ACF keywords **auto_handle** and **implicit_handle** in the IDL file and omit the ACF.

The ACF corresponds to the IDL file in the following ways:

- The interface name in the ACF must be the same as the interface name in the IDL file unless you compile with the MIDL compiler switch **/acf.**
- All type names and function names in the ACF must refer to types and functions defined in the IDL file.
- Function parameters need not appear in the same sequence in the ACF as in the IDL file, but parameter names in the ACF must match names in the IDL file.

Like the IDL file, the ACF consists of a header portion and a body portion, and in **/ms_ext** mode, it can contain multiple interfaces.

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, and 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 **include** directive specifies 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 allows you to 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 allow you to specify attributes that apply to the function as a whole. For more information, see **code**, **optimize**, and **nocode**.

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

See Also

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

allocate

typedef [allocate (*allocate-option-list***) [,** *type-attribute-list*] **]** *type-name***;**

allocate-option-list
Specifies one or more memory-allocation options. Select one of either **single_node** or **all_nodes**, or one of either **free** or **dont_free**, or one from each group. When you specify more than one option, separate options with commas.

type-attribute-list
Specifies other optional ACF type attributes. When you specify more than one type attribute, separate options with commas.

type-name
Specifies a type defined in the IDL file.

Examples

```
/* ACF file */
typedef [allocate(all_nodes, dont_free)] PTYPE1;
typedef [allocate(all_nodes)] PTYPE2;
typedef [allocate(dont_free)] PTYPE3;
```

Remarks

The ACF type attribute **allocate** allows you to customize memory allocation and deallocation for a type defined in the IDL file. Valid options are as follows:

Option	Description
all_nodes	Makes one call to allocate and free memory for all nodes
single_node	Makes many individual calls to allocate and free each node of memory
free	Frees memory on return from the server stub
dont_free	Doesn't free memory on return from the server stub

By default, the stubs may allocate storage for data referenced by a unique or full pointer by calling **midl_user_allocate** and **midl_user_free** individually for each pointer.

You can optimize the speed of your application by specifying the option **all_nodes**. This option directs the stub to compute the size of all memory referenced through the pointer of the specified type and to make a single call to **midl_user_allocate**. The stub releases the memory by making one call to **midl_user_free**.

The **dont_free** option directs the MIDL compiler to generate a server stub that does not call **midl_user_free** for the specified type. The **dont_free** option allows the pointer structures to remain accessible to the server application after the remote procedure call has completed and returned to the client.

Note that when applied to types used for **in, out** parameters, any parameter that is a pointer to a type qualified with the **all_nodes** option will cause a reallocation when the data is unmarshalled. It is the responsibility of the application to free the previously allocated memory corresponding to this parameter. For example:

```
typedef struct foo
{
[string] char * PFOO;
} * PFOO
void proc1 ( [in,out] PFOO * ppfoo);
```

The data type PFOO will be reallocated in the **out** direction by the stub before "unmarshalling." Therefore, the previously allocated area must be freed by the application. Otherwise, a memory leak will occur.

See Also

ACF, **midl_user_allocate**, **midl_user_free**, **typedef**

arrays

typedef [**[***type-attr-list***]**] *type-specifier* [*pointer-decl*] *array-declarator***;**

typedef [**[***type-attr-list***]**] **struct** [*tag*] **{**
[**[** *field-attribute-list* **]**] *type-specifier* [*pointer-decl*] *array-declarator***;**
...
}
typedef [**[***type-attr-list***]**] **union** [*tag*] **{**
[case (*limited-expression* [**, ...**] **)]**
[**[** *field_attribute-list* **]**] *type-specifier* [*pointer-decl*] *array-declarator***;**
[**[default]**
[**[** *field_attribute-list* **]**] *type-specifier* [*pointer-decl*] *array-declarator***;**
]

[**[***function-attribute-list***]**] *type-specifier* [*pointer-decl*] *function-name***(**
[**[***param-attr-list***]**] *type-specifier* [*pointer-decl*] *array-declarator*
, ...
);

type-attr-list
Specifies zero or more attributes that apply to the type. Valid type attributes include **handle**, **switch_type**, **transmit_as**; the pointer attribute **ref**, **unique**, or **ptr**; and the usage attributes **context_handle**, **string**, and **ignore**. Separate multiple attributes with commas.

type-specifier
Specifies the type identifier, base type, **struct**, **union**, or **enum** type. The type specification can include an optional storage specification.

pointer-decl
Specifies zero or more pointer declarators. A pointer declarator is the same as the pointer declarator used in C, constructed from the * designator, modifiers such as **far**, and the qualifier **const**.

array-declarator
Specifies the name of the array, followed by one of the following constructs for each dimension of the array: "**[ ]**", "**[*]**", "**[***const1***]**", or "**[***lower...upper***]**" where *lower* and *upper* are constant values that represent the lower and upper bounds. The constant *lower* must evaluate to zero.

tag
Specifies an optional tag for the structure or union.

field-attribute-list
Specifies zero or more field attributes that apply to the structure, union member, or function parameter. Valid field attributes include **first_is**, **last_is**, **length_is**, **max_is**, **size_is**; the usage attributes **string**, and **ignore**; the pointer attributes **ref**, **unique**, and **ptr**; and the union attribute **switch_type**. Separate multiple field attributes with commas. Note that of the attributes listed above, **first_is**, **last_is**, and **ignore** are not valid for unions.

limited-expression
Specifies a C-language expression supported by MIDL. Almost all C-language expressions are supported: The MIDL compiler supports conditional expressions, logical expressions, relational expressions, and arithmetic expressions. MIDL does not allow function invocations in expressions and does not allow pre- and post-increment and -decrement operators.

function-attribute-list
Specifies zero or more attributes that apply to the function. Valid function attributes are **callback**, **local**; the pointer attribute **ref**, **unique**, or **ptr**; and the usage attributes **string**, and **context_handle**.

function-name
Specifies the name of the remote procedure.

param-attr-list
Specifies the directional attributes and one or more optional field attributes that apply to the array parameter. Valid field attributes include **max_is**, **size_is**, **length_is**, **first_is**, and **last_is**.

Examples

```
/* IDL file interface body */
#define MAX_INDEX 10

typedef char  ATYPE[MAX_INDEX];
typedef short BTYPE[];        // Equivalent to [*];
typedef long  CTYPE[*][10];   // [][10]
typedef float DTYPE[0..10];   // Equivalent to [11]
typedef float ETYPE[0..(MAX_INDEX)];

typedef struct {
    unsigned short size;
    unsigned short length;
    [size_is(size), length_is(length)] char string[*];
} counted_string;

void MyFunction(
     [in, out] short * pSize,
     [in, out, string, size_is(*pSize)] char a[0..*]
);
```

Remarks

Array declarators appear in the interface body of the IDL file as part of a general declaration, as a member of a structure or union declarator, or as a parameter to a remote procedure call.

The bounds of each dimension of the array are expressed inside a separate pair of square brackets. An expression that evaluates to *n* signifies a lower bound of zero and an upper bound of *n* - *1*. If the square brackets are empty or contain a single asterisk (*), the lower bound is zero and the upper bound is determined at run time.

The array can also contain two values separated by an ellipsis that represent the lower and upper bounds of the array, as in [*lower...upper*]. Microsoft RPC requires a lower bound of 0. The MIDL compiler does not recognize constructs that specify nonzero lower bounds.

Arrays can be associated with the field attributes **size_is**, **max_is**, **length_is**, **first_is**, and **last_is** to specify the size of the array or the part of the array that contains valid data. These field attributes identify the parameter, structure field, or constant that specifies the array dimension or index.

The array must be associated with the identifier specified by the field attribute as follows: When the array is a parameter, the identifier must also be a parameter to the same function; when the array is a structure field, the identifier must be another structure field of that same structure.

An array is called "conformant" if the upper bound of any dimension is determined at run time. Only upper bounds can be determined at run time. To determine the upper bound, the array declaration must include a **size_is** or **max_is** attribute.

An array is called "varying" when its bounds are determined at compile time, but the range of transmitted elements is determined at run time. To determine the range of transmitted elements, the array declaration must include a **length_is**, **first_is**, or **last_is** attribute.

A conformant varying array (also called "open") is an array whose upper bound and range of transmitted elements are determined at run time.

At most, one conformant or conformant varying array can be nested in a C structure and must be the last element of the structure. Nonconformant varying arrays can occur anywhere in a structure.

Multidimensional Arrays

The user can declare types that are arrays and then declare arrays of objects of such types. The semantics of *m*-dimensional arrays of *n*-dimensional array types are the same as the semantics of *m+n*-dimensional arrays.

For example, the type RECT_TYPE can be defined as a two-dimensional array and the variable *rect* can be defined as an array of RECT_TYPE. This is equivalent to the three-dimensional array *equivalent_rect*:

```
typedef short int RECT_TYPE[10][20];
RECT_TYPE rect[15];
short int equivalent_rect[15][10][20];  // ~RECT_TYPE rect[15]
```

Microsoft RPC is C-oriented. Following C-language conventions, only the first dimension of a multidimensional array can be run-time - determined. Interoperation with DCE IDL arrays that support other languages is limited to:

- Multidimensional arrays with constant (compile-time - determined) bounds.
- Multidimensional arrays with all constant bounds except the first dimension. The upper bound and range of transmitted elements of the first dimension are run-time - dependent.
- Any one-dimensional arrays with a lower bound of zero.

When the **string** attribute is used on multidimensional arrays, the attribute applies to the rightmost array.

Arrays of Pointers

Reference pointers must point to valid data. The client application must allocate all memory for an **in** or **in, out** array of reference pointers, especially when the array is associated with **in**, or **in, out** **length_is**, or **last_is** values. The client application must also initialize all array elements before the call. Before returning to the client, the server application must verify that all array elements in the transmitted range point to valid storage.

On the server side, the stub allocates storage for all array elements, regardless of the **length_is** or **last_is** value at the time of the call. This feature can affect the performance of your application.

No restrictions are placed on arrays of unique pointers. On both the client and the server, storage is allocated for null pointers. When pointers are non-null, data is placed in preallocated storage.

An optional pointer declarator can precede the array declarator.

When embedded reference pointers are **out**-only parameters, the server-manager code must assign valid values to the array of reference pointers. For example,

```
typedef [ref] short * ARefPointer;
typedef ARefPointer ArrayOfRef[10];
void proc1( [out] ArrayOfRef Parameter );
```

The generated stubs allocate the array and assign NULL values to all pointers embedded in the array.

See Also

first_is, **IDL**, **last_is**, **length_is**, **max_is**, **ptr**, **ref**, **size_is**, **string**, **unique**

auto_handle

[auto_handle [**,** *interface-attribute-list*] **] interface** *interface-name*

interface-attribute-list
Specifies zero or more attributes that apply to the interface as a whole, such as **code** or **nocode**. Separate interface attributes with commas.

interface-name
Specifies the name of the interface.

Examples

```
[auto_handle] interface MyInterface { }
[auto_handle, code] interface MyInterface { }
```

Remarks

The ACF attribute **auto_handle** directs the stub to automatically establish the binding for a function that does not have an explicit binding-handle parameter.

The **auto_handle** attribute appears in the interface header of the ACF. It also appears in the interface header of the IDL file when you specify the MIDL compiler switch **/app_config**.

When the client calls a function that uses automatic binding, and no binding to a server exists, the stub automatically establishes the binding. The binding is reused for subsequent calls to other functions in the interface that use automatic binding. The client application program does not have to declare or perform any processing relating to the binding handle.

When the ACF is not present or does not include the **implicit_handle** attribute, the MIDL compiler uses **auto_handle** and issues an informational message. The MIDL compiler also uses **auto_handle** if needed to establish the initial binding for a **context_handle**.

The **auto_handle** attribute can occur only if the **implicit_handle** or **explicit_handle** attribute does not occur. The **auto_handle** attribute can occur in the ACF or IDL interface header at most once.

See Also

ACF, **/app_config**, **context_handle**, **IDL**, **implicit_handle**

base_types

Remarks

All data transmitted on the network during a remote procedure call must resolve to a base type or predefined type.

MIDL supports the following base types: **boolean**, **byte**, **char**, **double**, **float**, **handle_t**, **hyper**, **long**, **short**, **small**, and **void ***. The keywords **signed** and **unsigned** can be used to qualify integer and character types. MIDL also provides the predefined types **error_status_t** and **wchar_t**.

Base types can appear as type specifiers in **const** declarations, **typedef** declarations, general declarations, and as parameter type specifiers in function declarators.

The base and predefined types have the following default signs and default sizes:

Base type	Default sign	Description
boolean	**unsigned**	8-bit data item
byte	- (not applicable)	8-bit data item
char	**unsigned**	8-bit unsigned data item
double	**-**	64-bit floating-point number
float	**-**	32-bit floating-point number
handle_t	**-**	Primitive handle type
hyper	**signed**	64-bit signed integer
long	**signed**	32-bit signed integer
short	**signed**	16-bit signed integer
small	**signed**	8-bit signed integer
void *	**-**	32-bit context handle pointer type
wchar_t	**unsigned**	16-bit unsigned data item

Any other types in the interface must be derived from these base or predefined types. This requirement has the following two important effects:

- The type **int** cannot appear in remote functions without a size qualifier such as **short**, **small**, **long** or **hyper**.
- The type **void *** cannot appear in remote functions except when it is used to define a context handle.

DCE IDL compilers do not recognize the keyword **signed**. You can only use the keyword **unsigned** when you use the MIDL compiler in DCE-compatibility mode.

See Also

byte, **char**, **handle_t**, **long**, **/ms_ext**, **short**, **small**, **wchar_t**

boolean

Remarks

The keyword **boolean** indicates that the expression or constant expression associated with the identifier takes the value TRUE or FALSE.

The **boolean** type is one of the base types of the IDL language. The **boolean** type can appear as a type specifier in **const** declarations, **typedef** declarations, general declarations, and function declarators (as a function-return-type specifier and as a parameter-type specifier). For the context in which type specifiers appear, see **IDL**.

See Also

base_types, **IDL**

broadcast

[[[*IDL-operation-attributes* **]]]** *operation-attribute* **, ...**

IDL-operation-attributes
Specifies zero or more IDL operation attributes, such as **broadcast** and **idempotent**. Operation attributes are enclosed in square brackets.

Remarks

The keyword **broadcast** specifies that remote procedure calls be sent to all servers on a local network. Rather than being delivered to one particular server, the routine is always broadcast to all the servers on the network. The client receives output from the first reply to return successfully. Subsequent replies are discarded.

The **broadcast** attribute specifies that the routine can be called multiple times and at the same time be sent to multiple servers as the result of one RPC. This is different from the **idempotent** attribute, which specifies that a call can be retried if it does not complete. However, an operation with the **broadcast** attribute is implicitly an **idempotent** operation. It ensures that the data for an RPC is received and processed zero or more times.

The **broadcast** attribute is supported only by connectionless protocols (datagrams). If a remote procedure broadcasts its call to all hosts on a local network, it must use the datagram protocol sequence **ncadg_ip_udp**. Note that the size of a **broadcast** packet is determined by the datagram service in use.

See Also

idempotent, **IDL**, **maybe**, **non-idempotent**

byte

Remarks

The **byte** keyword specifies an 8-bit data item.

A **byte** data item does not undergo any conversion for transmission on the network, as can occur for a **char** type.

The **byte** type is one of the base types of the interface definition language (IDL). The **byte** type can appear as a type specifier in **const** declarations, **typedef** declarations, general declarations, and function declarators (as a function-return-type specifier and as a parameter-type specifier). For the context in which type specifiers appear, see **IDL**.

See Also

base_types, **char**

byte_count

[*function-attribute-list*] *function-name*(
    [byte_count(*length-variable-name***)]** *pointer-parameter-name*);
    ...
);

function-attribute-list
Specifies zero or more ACF function attributes.

function-name
Specifies the name of the function defined in the IDL file. The function name is required.

length-variable-name
Specifies the name of the **in**-only parameter that specifies the size, in bytes, of the memory area referenced by *pointer-parameter-name*.

pointer-parameter-name
Specifies the name of the **out**-only pointer parameter defined in the IDL file.

Examples

```
/* IDL file */
void proc1([in] unsigned long length, [out] struct foo * pFoo);

/* ACF file */
proc1([byte_count(length)] pFoo);
```

Remarks

Note The ACF attribute **byte_count** represents a Microsoft extension to DCE IDL.

The **byte_count** attribute is a parameter attribute that associates a 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 allows you to create a persistent buffer area in client memory that can be reused during more than one call to the remote procedure.

The ability to turn off the client stub memory allocation allows you to tune the application for efficiency. For example, the **byte_count** attribute can be used by service-provider functions that use Microsoft RPC. When a user application calls the service-provider API and provides a pointer to a buffer, the service provider can pass the buffer pointer on to the remote function and reuse the buffer during multiple remote calls without forcing the user to reallocate the memory area.

The memory area can contain complex data structures that consist of multiple pointers. Because the memory area is contiguous, the application does not have to make many calls to individually free each pointer and structure. The memory area can be allocated or freed with one call to the memory allocation or free routine.

The buffer must be an **out**-only parameter. The buffer length in bytes must be an **in**-only parameter.

Note Specify a buffer that is large enough to contain all the **out** parameters. Pointers are unmarshalled on a 4-byte aligned boundary. Therefore, alignment padding that the stubs will perform must be accounted for in the space for the buffer. In addition, packing levels used during C-language compilation can vary. Use a byte count value that accounts for additional packing bytes added for the packing level used during C-language compilation.

See Also

ACF, in, length_is, out, size_is

call_as

[call_as (local-proc**),** [**,** operation-attribute-list]] operation-name **;**

local-proc
Specifies an operation-defined routine.

operation-attribute-list
Specifies one or more attributes that apply to the operation. Separate multiple attributes with commas.

operation-name
Specifies the named operation presented to the application.

Remarks

The **call_as** attribute enables a non-remotable function to be mapped to a remote function. This is particularly helpful in interfaces that have numerous non-remotable types as parameters. Rather than using many **represent_as** and **transmit_as** types, you can combine all the conversions using **call_as** routines. You supply the two **call_as** routines (client side and server side) to bind the routine between the application calls and the remote calls. The **call_as** attribute can be used for object interfaces, where the interface definition can be used for local calls as well as remote calls because it allows a non-remotable interface to be remoted transparently. The **call_as** attribute can only be used for **/ms_ext** mode.

For example, assume that the routine **f1** in object interface **IFace** requires numerous conversions between the user calls and what is actually transmitted. The following examples describe the IDL and ACF files for interface **IFace**:

In the IDL file for interface **IFace**:

```
[local] HRESULT f1 ( <users parameter list> )
[call_as( f1 )] long Remf1 ( <remotable parameter list> );
```

In the ACF for interface **IFace**:

```
[call_as( f1 )] Remf1();
```

This would cause the generated header file to define the interface using the definition of f1, yet it would also provide stubs for Remf1:

Generated Vtable in the header file for interface **IFace**:

```
struct IFace_vtable        {
          ...
          HRESULT ( * f1) ( <users parameter list>);
          ...
          };
```

The client-side proxy would then have a typical MIDL-generated proxy for Remf1, while the server side stub for Remf1 would be the same as the typical MIDL-generated stub:

```
void IFace_Remf1_Stub ( . . .)
    {
    ...
    invoke IFace_f1_Stub ( <remotable parameter list> )   /* instead
                of IFace_f1 */
    ...
    }
```

Then, the two **call_as** bond routines (client side and server side) must be manually coded:

```
HRESULT f1_Proxy ( <users parameter list> )
   {
   ...
   Remf1_Proxy ( <remotable parameter list> );
   ...
   }

long IFace_f1_Stub ( <remotable parameter list> )
   {
   ...
   IFace_f1 ( <users parameter list> );
   ...
   }
```

For object interfaces, the prototypes for the bond routines are:

For client side:

```
<local_return_type>  <interface>_<local_routine>_proxy
( <local_parameter_list> );
```

For server side:

```
<remote_return_type>  <interface>_<local_routine>_stub
( <remote_parameter_list> );
```

For non-object interfaces, the prototypes for the bond routines are:

For client side:

```
<local_return_type>  <local_routine> ( <local_parameter_list> );
```

For server side:

```
<local_return_type>  <interface>_v<maj>_<min>_<local_routine>
( <remote_parameter_list> );
```

See Also

represent_as, **transmit_as**

callback

[callback [**,** *function-attr-list*] **]** *type-specifier* [*ptr-declarator*] *function-name***(**
[**[***parameter-attribute-list***]**] *type-specifier* [*declarator*]
, ...
);

function-attr-list
Specifies zero or more attributes that apply to the function. Valid function attributes are **local**; the pointer attribute **ref**, **unique**, or **ptr**; and the usage attributes **string**, **ignore**, and **context_handle**. Separate multiple attributes with commas.

type-specifier
Specifies a **base_type**, **struct**, **union, enum** type, or type identifier. An optional storage specification can precede *type-specifier*.

ptr-declarator
Specifies zero or more pointer declarators. A pointer declarator is the same as the pointer declarator used in C; it is constructed from the ***** designator, modifiers such as **far**, and the qualifier **const**.

function-name
Specifies the name of the remote procedure.

parameter-attribute-list
Specifies zero or more directional attributes, field attributes, usage attributes, and pointer attributes appropriate for the specified parameter type. Separate multiple attributes with commas.

declarator
Specifies a standard C declarator such as identifiers, pointer declarators, and array declarators. For more information, see **pointers** and **arrays**. The parameter-name identifier is optional.

Example

```
[callback] void DisplayString([in, string] char * p1);
```

Remarks

The **callback** attribute declares a static callback function that exists on the client side of the distributed application. Callback functions provide a way for the server to execute code on the client.

The callback function is useful when the server must obtain information from the client. If server applications were supported on Windows 3.*x*, the server could make a call to a remote procedure on the Windows 3.*x* server to obtain the needed information. The callback function accomplishes the same purpose. The callback allows the server to query the client for information in the context of the original call.

Callbacks are special cases of remote calls that execute as part of a single thread. A callback is issued in the context of a remote call. Any remote procedure defined as part of the same interface as the static callback function can call the callback function.

Handles cannot be used as parameters in callback functions. Because callbacks always execute in the context of a call, the binding handle used by the client to make the call to the server is also used as the binding handle from the server to the client.

Callbacks can nest to any depth.

See Also

IDL, **/ms_ext**

char

Remarks

The keyword **char** identifies a data item that has 8 bits. To the MIDL compiler, a **char** is unsigned by default and is synonymous with **unsigned char**.

In this version of Microsoft RPC, the character translation tables that convert between ASCII and EBCDIC are built into the run-time libraries and cannot be changed by the user.

The **char** type is one of the base types of the interface definition language (IDL). The **char** type can appear as a type specifier in **const** declarations, **typedef** declarations, general declarations, and function declarators (as a function-return-type specifier and a parameter-type specifier). For the context in which type specifiers appear, see **IDL**.

DCE IDL compilers do not accept the keyword **signed** applied to **char** types. To allow this construct, use the MIDL compiler **/ms_ext** switch.

See Also

base_types, **byte**, **/char**, **/ms_ext**, **signed**, **string**, **wchar_t**

code

[code [**,** *ACF-interface-attributes*] **] interface** *interface-name*
{
[**include** *filename-list* **;**] **...**
[**typedef [***type-attribute-list***]** *typename***;**] **...**

[**[code** [**,** *ACF-function-attributes*] **]** *function-name* **(**
[*ACF-parameter-attributes* **]** *parameter-name* **;**
...
);
]
...
}

ACF-interface-attributes
Specifies a list of one or more attributes that apply to the interface as a whole. Valid attributes include either **auto_handle** or **implicit_handle** and either **code**, **nocode**, or **optimize**. When two or more interface attributes are present, they must be separated by commas.

interface-name
Specifies the name of the interface. In DCE-compatibility mode, the interface name must match the name of the interface specified in the IDL file. When you use the MIDL compiler switch **/acf**, the interface name in the ACF and the interface name in the IDL file can be different.

filename-list
Specifies a list of one or more C-header filenames, separated by commas. You must supply the full filename, including the extension.

type-attribute-list
Specifies a list of one or more attributes, separated by commas, that apply to the specified type. Valid type attributes include **allocate** and **represent_as**.

typename
Specifies a type defined in the IDL file. Type attributes in the ACF can only be applied to types previously defined in the IDL file.

ACF-function-attributes
Specifies zero or more other attributes that apply to the function as a whole, such as **comm_status**. Function attributes are enclosed in square brackets. Separate multiple function attributes with commas.

function-name
Specifies the name of the function as defined in the IDL file.

ACF-parameter-attributes
Specifies ACF attributes that apply to a parameter. Note that 0, 1, or more attributes can be applied to the parameter. Separate multiple parameter attributes with commas. ACF parameter attributes are enclosed in square brackets.

parameter-name
Specifies a parameter of the function as defined in the IDL file. Each parameter for the function must be specified in the same sequence and using the same name as defined in the IDL file.

Remarks

The **code** attribute can appear in the ACF header, or it can be applied to an individual function.

When the **code** attribute appears in the ACF header, client stub code is generated for all remote functions that do not have the **nocode** function attribute. You can override the **code** attribute in the header for an individual function by specifying the **nocode** attribute as a function attribute.

When the **code** attribute appears in the remote function's attribute list, client stub code is generated for the function.

Client stub code is not generated when:

- The ACF header includes the **nocode** attribute.
- The **nocode** attribute is applied to the function.
- The **local** attribute applies to the function in the interface file.

Either **code** or **nocode** can appear in the interface or function attribute list, and the one you choose can appear exactly once in the list.

See Also

ACF, **nocode**

comm_status

[comm_status [**,** *ACF-function-attributes*] **]** *function-name***(**
    [**[** *ACF-parameter-attributes* **]**] *parameter-name*
    , ...
);

[**[** *ACF-function-attributes* **]**] *function-name***(**
    [comm_status [**,** *ACF-parameter-attributes*] **]** *parameter-name*
    ...);

ACF-function-attributes
Specifies zero or more ACF function attributes, such as **comm_status** and **nocode**. Function attributes are enclosed in square brackets. Note that 0, 1, or more attributes can be applied to a function. Separate multiple function attributes with commas. Note that if **comm_status** appears as a function attribute, it cannot also appear as a parameter attribute.

function-name
Specifies the name of the function as defined in the IDL file.

ACF-parameter-attributes
Specifies attributes that apply to a parameter. Note that 0, 1, or more attributes can be applied to the parameter. Separate multiple parameter attributes with commas. Parameter attributes are enclosed in square brackets. IDL parameter attributes, such as directional attributes, are not allowed in the ACF. Note that if **comm_status** appears as a parameter attribute, it cannot also appear as a function attribute.

parameter-name
Specifies the parameter for the function as defined in the IDL file. Each parameter for the function must be specified in the same sequence, using the same name as defined in the IDL file.

Remarks

The **comm_status** attribute can be used as either a function attribute or as a parameter attribute, but it can appear only once per function. It can be applied either to the function or to one parameter in each function.

The **comm_status** attribute can only be applied to functions that return the type **error_status_t**. When a communication error occurs during the execution of the function, an error code is returned.

When **comm_status** is used as a parameter attribute, the parameter must be defined in the IDL file and must be an **out** parameter of type **error_status_t**. When a communication error occurs during the execution of the function, the parameter is set to the error code. When the remote call completes successfully, the procedure sets the value.

It is possible for both the **comm_status** and **fault_status** attributes to appear in a single function, either as function attributes or parameter attributes. If both attributes are function attributes or if they apply to the same parameter, and no error occurs, the function or parameter has the value **error_status_ok**. Otherwise, it contains the appropriate **comm_status** or **fault_status** value. Because values returned for **comm_status** are different from the values returned for **fault_status**, the returned values are readily interpreted.

See Also

ACF, error_status_t, fault_status,

const

const *const-type identifier* **=** *const-expression* **;**

/* IDL file **typedef** syntax */
[typedef [, *type-attribute-list*] **]** **const** *const-type declarator-list*;
[typedef [, *type-attribute-list*] **]** *pointer-type* **const** *declarator-list*;

[**[** *function-attr-list* **]**] *type-specifier* [*ptr-decl*] *function-name*(
[**[** *parameter-attribute-list* **]**] **const** *const-type* [*declarator*],
[**[** *parameter-attribute-list* **]**] *pointer-type* **const** [*declarator*]
, ...
);

const-type
Specifies a valid MIDL integer, character, string, or boolean type. Valid MIDL types include **small**, **short**, **long**, **char**, **char ***, **wchar_t**, **wchar_t ***, **byte**, **byte ***, and **void ***. The integer and character types can be **signed** or **unsigned**.

identifier
Specifies a valid MIDL identifier. Valid MIDL identifiers consist of up to 31 alphanumeric and/or underscore characters and must start with an alphabetic or underscore character.

const-expression
Specifies an expression, identifier, or numeric or character constant appropriate for the specified type: constant integer literals or constant integer expressions for integer constants; boolean expressions that can be computed at compilation for **boolean** types; single-character constants for **character** types; and string constants for **string** types. The **void *** type can be initialized only to NULL.

type-attribute-list
Specifies one or more attributes that apply to the type.

pointer-type
Specifies a valid MIDL pointer type.

declarator and *declarator-list*
Specifies standard C declarators, such as identifiers, pointer declarators, and array declarators. For more information, see **pointers** and **arrays**. The *declarator-list* consists of one or more declarators, separated by commas. The parameter-name identifier in the function declarator is optional.

function-attr-list
Specifies zero or more attributes that apply to the function. Valid function attributes are **callback**, **local**; the pointer attribute **ref**, **unique**, or **ptr**; and the usage attributes **string**, **ignore**, and **context_handle**.

type-specifier
Specifies a **base_type**, **struct**, **union**, **enum** type, or type identifier. An optional storage specification can precede *type-specifier*.

ptr-decl
Specifies zero or more pointer declarators. A pointer declarator is the same as the pointer declarator used in C. It is constructed from the ***** designator, modifiers such as **far**, and the qualifier **const**.

function-name
Specifies the name of the remote procedure.

parameter-attribute-list
Specifies zero or more directional attributes, field attributes, usage attributes, and pointer attributes appropriate for the specified parameter type. Separate multiple attributes with commas.

Examples

```
const void *  p1        = NULL;
```

```
const char    my_char1  = 'a';
const char    my_char2  = my_char1;
const wchar_t my_wchar3 = L'a';
const wchar_t * pszNote = L"Note";
const unsigned short int x = 123;

typedef [string] const char *LPCSTR;

HRESULT GetName([out] wchar_t * const pszName );
```

Remarks

MIDL allows you to declare constant integer, character, string, and boolean types in the interface body of the IDL file. You can use the **const** keyword to modify the type of a type declaration or the type of a function parameter. **Const** type declarations are reproduced in the generated header file as **#define** directives.

DCE IDL compilers do not support constant expressions. To enable constant expressions, use the MIDL compiler switch **/ms_ext**.

A previously defined constant can be used as the assigned value of a subsequent constant.

The value of a constant integral expression is automatically converted to the respective integer type in accordance with C conversion rules.

The value of a character constant must be a single-quoted ASCII character. When the character constant is the single-quote character itself ('), the backslash character (\) must precede the single-quote character, as in \'.

The value of a character-string constant (**char ***) must be a double-quoted string. Within a string, the backslash (\) character must precede a literal double-quote character (**"**), as in **\"**. Within a string, the backslash character (\) represents an escape character. String constants can consist of up to 255 characters.

The value NULL is the only valid value for constants of type **void ***.

Any attributes associated with the **const** declaration are ignored.

The MIDL compiler does not check for range errors in **const** initialization. For example, when you specify "const short x = 0xFFFFFFFF;" the MIDL compiler does not report an error and the initializer is reproduced in the generated header file.

See Also

base_types, **IDL**, **/ms_ext**

context_handle

typedef [context_handle [**,** *type-attribute-list*] **]** *type-specifier declarator-list***;**

[context_handle [**,** *function-attr-list*] **]** *type-specifier* [*ptr-decl*] *function-name***(**
[**[***parameter-attribute-list***]**] *type-specifier* [*declarator*]
, ...
);

[**[** *function-attr-list* **]**] *type-specifier* [*ptr-decl*] *function-name***(**
[context_handle [**,** *parameter-attribute-list*] **]** *type-specifier* [*declarator*]
, ...
);

[**void __RPC_USER** *context-handle-type***_rundown** (*context-handle-type*)**;**]

type-attribute-list
Specifies one or more attributes that apply to the type.

type-specifier
Specifies a pointer type or a type identifier. An optional storage specification can precede *type-specifier*.

declarator and *declarator-list*
Specifies standard C declarators, such as identifiers, pointer declarators, and array declarators. The declarator for a context handle must include at least one pointer declarator. For more information, see **pointers** and **arrays**. The *declarator-list* consists of one or more declarators, separated by commas. The parameter-name identifier in the function declarator is optional.

function-attr-list
Specifies zero or more attributes that apply to the function. Valid function attributes are **callback**, **local**; the pointer attribute **ref**, **unique**, or **ptr**; and the usage attributes **string**, **ignore**, and **context_handle**.

ptr-decl
Specifies zero or more pointer declarators. A pointer declarator is the same as the pointer declarator used in C; it is constructed from the ***** designator, modifiers such as **far**, and the qualifier **const**.

function-name
Specifies the name of the remote procedure.

parameter-attribute-list
Specifies zero or more directional attributes, field attributes, usage attributes, and pointer attributes appropriate for the specified parameter type. Separate multiple attributes with commas.

context-handle-type
Specifies the identifier that specifies the context handle type as defined in a **typedef** declaration that takes the **context_handle** attribute. The rundown routine is optional.

Example

```
typedef [context_handle] void * PCONTEXT_HANDLE_TYPE;
short RemoteFunc1([out] PCONTEXT_HANDLE_TYPE * pCxHandle);
short RemoteFunc2([in, out] PCONTEXT_HANDLE_TYPE * pCxHandle);
void __RPC_USER PCONTEXT_HANDLE_TYPE_rundown (PCONTEXT_HANDLE_TYPE);
```

Remarks

The **context_handle** attribute identifies a binding handle that maintains context, or state information, on the server between remote procedure calls. The attribute can appear as an IDL **typedef** type attribute, as a function return type attribute, or as a parameter attribute.

DCE IDL compilers restrict context handles to pointers of type **void *.** When you use the MIDL compiler switch **/ms_ext** to specify the Microsoft-extensions mode, a context handle can be any pointer type selected by the user, as long as it complies with the requirements for context handles described following. The data associated with such a context handle type is not transmitted on the network, and so should only be manipulated by the server application.

Like other handle types, the context handle is opaque to the client application. Any data associated with the context handle type is not transmitted. On the server, the context handle serves as a handle on active context and all data associated with the context handle type is accessible.

To create a context handle, the client passes to the server an **out, ref** pointer to a context handle. (The context handle itself can have a null or non-null value, as long as its value is consistent with its pointer attributes. For example, when the context handle type has the **ref** attribute applied to it, it cannot have a null value.) Another binding handle must be supplied to accomplish the binding until the context handle is created. When no explicit handle is specified, implicit binding is used. When no **implicit_handle** attribute is present, an auto handle is used.

The remote procedure on the server creates an active context handle. The client must use that context handle as an **in** or **in, out** parameter in subsequent calls. An **in**-only context handle can be used as a binding handle, so it must have a non-null value. An **in**-only context handle does not reflect state changes on the server.

On the server, the called procedure can interpret the context handle as needed. For example, the called procedure can allocate heap storage and use the context handle as a pointer to this storage.

To close a context handle, the client passes the context handle as an **in, out** argument. The server must return a null context handle when it is no longer maintaining context on behalf of the caller. For example, if the context handle represents an open file and the call closes the file, the server must set the context handle to NULL and return it to the client. A null value is invalid as a binding handle on subsequent calls.

A context handle is only valid for one server. When a function has two handle parameters and the context handle is not null, the binding handles must refer to the same address space.

When a function has an **in** or an **in, out** context handle, its context handle can be used as the binding handle. In this case, implicit binding is not used and the **implicit_handle** or **auto_handle** attribute is ignored.

The following restrictions apply to context handles:

- Context handles cannot be array elements, structure members, or union members. They can only be parameters.
- Context handles cannot have the **transmit_as** or **represent_as** attribute.
- Parameters that are pointers to **out** context handles must be **ref** pointers.
- An **in** context handle can be used as the binding handle and cannot be null.
- An **in**, **out** context handle can be null on input, but only if the procedure has another explicit handle parameter.
- A context handle cannot be used with callbacks.

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. When it is supplied, it 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 as follows:

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**.

See Also

auto_handle, **handle**, **handles**

cpp_quote

cpp_quote("*string***")**

string
Specifies a quoted string that is emitted in the generated header file. The string must be quoted to prevent expansion by the C preprocessor.

Examples

```
cpp_quote("#include \"foo.h\" ")
cpp_quote("#define UNICODE")
```

Remarks

The **cpp_quote** keyword instructs MIDL to emit the specified string, without the quote characters, into the generated header file.

C-language preprocessing directives that appear in the IDL file are eaten (that is, processed) by the C compiler's preprocessor. The **#define** directives in the IDL file are available during MIDL compilation but are not available to the C compiler.

For example, when the preprocessor encounters the directive "**#define** WINDOWS 4", the preprocessor replaces all occurrences of "WINDOWS" in the IDL file with "4". The symbol "WINDOWS" is not available during C-language compilation.

To allow the C-preprocessor macro definitions to pass through the MIDL compiler to the C compiler, use the **#pragma midl_echo** or **cpp_quote** directive. These directives instruct the MIDL compiler to generate a header file that contains the parameter string with the quotation marks removed. The **#pragma midl_echo** and **cpp_quote** directives are equivalent.

The MIDL compiler places the strings specified in the **cpp_quote** and **pragma** directives in the header file in the sequence in which they are specified in the IDL file and relative to other interface components in the IDL file. The strings should usually appear in the IDL file interface body section after all **import** operations.

See Also

IDL, **pragma**

decode

[decode [**,** *interface-attribute-list*] **] interface** *interface-name*
[decode [**,** *op-attribute-list*] **]** *proc-name*
typedef [decode [**,** *type-attribute-list*] **]** *type-name*

interface-attribute-list
Specifies other attributes that apply to the interface as a whole.

interface-name
Specifies the name of the interface.

op-attribute-list
Specifies other operational attributes that apply to the procedure, such as **encode**.

proc-name
Specifies the name of the procedure.

type-attribute-list
Specifies other attributes, such as **encode** and **allocate**.

typename
Specifies a type defined in the IDL file.

Remarks

The **decode** attribute specifies that a procedure or a type needs de-serialization support. This attribute causes the MIDL compiler to generate code that an application can use to retrieve serialized data from a buffer. The **encode** attribute provides serialization support, generating the code to serialize data into a buffer.

Use the **encode** and **decode** attributes in an ACF to generate serialization code for procedures or types defined in the IDL file of an interface. When used as an interface attribute, **decode** applies to all types and procedures defined in the IDL file. When used as a type attribute, **decode** applies only to the specified type. When used as an operational attribute, **decode** applies only to that procedure.

For more information about using this serialization support, see Using Encoding Services and **encode**.

double

Remarks

The **double** keyword designates a 64-bit floating-point number.

The **double** type is one of the base types of the interface definition language (IDL). The **double** type can appear as a type specifier in **typedef** declarations, general declarations, and function declarators (as a function-return-type specifier and a parameter-type specifier). For the context in which type specifiers appear, see **IDL**.

The **double** type cannot appear in **const** declarations.

See Also

base_types, **float**

enable_allocate

Remarks

The keyword **enable_allocate** specifies that the server stub code should enable the stub memory management environment. In default mode, the stub enables this environment automatically when the remote operation includes full pointers, or pointers that provide for the stub or the user to allocate memory, or on request when the **enable_allocate** attribute is used. When using **ms_ext** mode, the server stub enables the memory environment only when the **enable_allocate** attribute is used. The memory management environment must be enabled before memory can be allocated using **RpcSmAllocate**.

The client side stub may be sensitive to the **Rpcss** memory management environment. If a sensitive client stub is executed when the **Rpcss** package is disabled, the default user allocator/deallocators are called (for example, **midl_user_allocate**/**midl_user_free**). When enabled, the **Rpcss** package uses the allocator/deallocator pair from the package. In the default mode, the client is always sensitive to the **Rpcss** memory management environment and therefore, the **enable_allocate** attribute will not affect the client stubs. In the **/ms_ext** mode, the client is sensitive only when the **enable_allocate** attribute is used. Typically, the client side stub operates in the disabled environment.

See Also

ACF, **RpcSmDisableAllocate**, **RpcSmEnableAllocate**, **RpcSmFree**

encapsulated_union

typedef [**[***type-attribute-list***]**]
union [*struct-name*] **switch (***switch-type switch-name***)** [*union-name*] **{**
[case (*limited-expression-list* **)]**
[**[** *field-attribute-list* **]**] *type-specifier declarator-list* **;**
...
}

type-attribute-list
Specifies zero or more other attributes that apply to the union type. Valid type attributes include **handle**, **transmit_as**; the pointer attribute **ref**, **unique**, or **ptr**; and the usage attributes **context_handle** and **ignore**. Separate multiple attributes with commas.

struct-name
Specifies an optional tag that names the structure generated by the MIDL compiler.

switch-type
Specifies an **int, char, enum** type, or an identifier that resolves to one of these types.

switch-name
Specifies the name of the variable of type *switch-type* that acts as the union discriminant.

union-name
Specifies an optional identifier that names the union in the structure, generated by the MIDL compiler, that contains the union and the discriminant.

limited-expression-list
Specifies one or more C-language expressions that are supported by MIDL. Almost all C-language expressions are supported: The MIDL compiler supports conditional expressions, logical expressions, relational expressions, and arithmetic expressions. MIDL does not allow function invocations in expressions and does not allow pre- and post-increment and -decrement operators.

field-attribute-list
Specifies zero or more field attributes that apply to the union member. Valid field attributes include **first_is**, **last_is**, **length_is**, **max_is**, **size_is**; the usage attributes **string**, **ignore**, and **context_handle**; the pointer attribute **unique** or **ptr**; and, for members that are themselves nonencapsulated unions, the union attribute **switch_type**. Separate multiple field attributes with commas.

type-specifier
Specifies a **base_type**, **struct**, **union**, **enum** type, or type identifier. An optional storage specification can precede *type-specifier*.

declarator-list
One or more standard C declarators, such as identifiers, pointer declarators, and array declarators. (Function declarators and bit-field declarations are not allowed in unions that are transmitted in remote procedure calls. When you use the MIDL compiler switch **/ms_ext**, these declarators are allowed in unions that are not transmitted.) Separate multiple declarators with commas.

Examples

```
typedef union _S1_TYPE switch (long l1) U1_TYPE {
    case 1024:
        float f1;
    case 2048:
        double d2;
} S1_TYPE;

/* in generated header file */
typedef struct _S1_TYPE {
```

```
    long l1;
    union {
        float f1;
        double d2;
    } U1_TYPE;
} S1_TYPE;
```

Remarks

The encapsulated union is indicated by the presence of the **switch** keyword. This type of union is so named because the MIDL compiler automatically encapsulates the union and its discriminant in a structure for transmission during a remote procedure call.

If the union tag is missing (U1_TYPE in the example above), the compiler will generate the structure with the union field named *tagged_union*.

The shape of unions must be the same across platforms to ensure interconnectivity.

See Also

IDL, **ms_union**, **non-encapsulated_union**, **switch_is**, **switch_type**, **union**

encode

[encode [**,** *interface-attribute-list*] **] interface** *interface-name*
[encode [**,** *op-attribute-list*] **]** *proc-name*
typedef [encode [**,** *type-attribute-list*] **]** *type-name*

interface-attribute-list
Specifies other attributes that apply to the interface as a whole.

interface-name
Specifies the name of the interface.

op-attribute-list
Specifies other operational attributes that apply to the procedure, such as **decode**.

proc-name
Specifies the name of the procedure.

type-attribute-list
Specifies other attributes that apply to the type, such as **decode** and **allocate**.

typename
Specifies a type defined in the IDL file.

Examples

```
/*
    ACF file example;
    Assumes MyType1, MyType2, MyType3, MyProc1, MyProc2, MyProc3 defined
     in IDL file
    MyType1, MyType2, MyProc1, MyProc2 have encode and decode
     serialization support
    MyType3 and MyProc3 have encode serialization support only
*/
[ encode, implicit_handle(handle_t bh) ]  interface regress
{
     typedef [ decode ] MyType1;
     typedef [ encode, decode ] MyType2;
     [ decode ] MyProcc1();
     [ encode ] MyProc2();
}
```

Remarks

The **encode** attribute specifies that a procedure or a data type needs serialization support. This attribute causes the MIDL compiler to generate code that an application can use to serialize data into a buffer. The **decode** attribute provides deserialization support, generating the code for retrieving data from a buffer.

Use the **encode** and **decode** attributes in an ACF to generate serialization code for procedures or types defined in the IDL file of an interface. When used as an interface attribute, **encode** applies to all the types and procedures defined in the IDL file. When used as an operational attribute, **encode** applies only to the specified procedure. When used as a type attribute, **encode** applies only to the specified type.

When the **encode** or **decode** attribute is applied to a procedure, the MIDL compiler generates a serialization stub in a similar fashion as remote stubs are generated for remote routines. A procedure can be either a remotable or a serializing procedure, but it cannot be both. The prototype of the generated routine is sent to the STUB.H file, while the stub itself goes into the STUB_C.C file.

The MIDL compiler generates two functions for each type the **encode** attribute applies to, and one

additional function for each type the **decode** attribute applies to. 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.

For additional information about serialization handles and encoding or decoding data, see Using Encoding Services.

See Also

decode

endpoint

endpoint("*protocol-sequence***:[***endpoint-port***]" [, ...])**

protocol-sequence
Specifies a valid transport family name. The following names are recognized by the run-time libraries provided with Microsoft RPC:

Family string	Description
ncacn_ip_tcp	TCP/IP
ncacn_nb_nb	NetBIOS over Microsoft NetBEUI
ncacn_nb_tcp	NetBIOS over TCP
ncacn_np	Named pipes
ncacn_spx	Connection-oriented SPX
ncadg_ip_upd	Datagram-oriented TCP/IP
ncadg_ipx	Datagram-oriented IPX
ncalrpc	Local RPC communication

Note Windows 95 does not support **ncalrpc**. The **ncacn_np** protocol is supported only on the client side.

endpoint-port
Specifies a string that represents the endpoint designation for the specified protocol family. The syntax of the port string is specific to each protocol sequence.

Examples

```
endpoint("ncacn_np:[\\pipe\\rainier]")

endpoint("ncacn_ip_tcp:[1044]", "ncacn_np:[\\pipe\\shasta]")
```

Remarks

The **endpoint** attribute specifies a well-known port or ports (communication endpoints) on which servers of the interface listen for calls.

The endpoint specifies a transport family such as the TCP/IP connection-oriented protocol, a NetBIOS connection-oriented protocol, or the named-pipe connection-oriented protocol.

The *protocol-sequence* value determines the valid values for the *endpoint-port*. The MIDL compiler checks only general syntax for the *endpoint-port* entry. Port specification errors are reported by the run-time libraries. For information about the allowed values for each protocol sequence, see the topic for that protocol sequence.

The following protocol sequences specified by DCE are not supported by the MIDL compiler provided with Microsoft RPC: **ncacn_osi_dna** and **ncadg_dds**.

The user must correctly quote backslash characters in endpoints. This commonly occurs when the endpoint is a named pipe.

Endpoint information specified in the IDL file is used by the RPC run-time functions **RpcServerUseProtseqIf** and **RpcServerUseAllProtseqsIf**.

See Also

IDL, **ncacn_dnet_nsp**, **ncacn_nb_nb**, **ncacn_np**, **ncacn_spx**, **ncalrpc**

enum

enum [*tag*] **{** *identifier* [**=***integer-value*] [**, ...**] **}**

tag
Specifies an optional tag for the enumerated type.

identifier
Specifies the particular enumeration.

integer-value
Specifies a constant integer value.

Examples

```
typedef enum {Monday=2, Tuesday, Wednesday, Thursday, Friday} workdays;

typedef enum {Clemens=21, Palmer=22, Ryan=34} pitchers;
```

Remarks

The keyword **enum** is used to identify an enumerated type. **Enum** types can appear as type specifiers in **typedef** declarations, general declarations, and function declarators (either as the function-return-type or as a parameter-type specifier). For the context in which type specifiers appear, see **IDL**.

With Microsoft RPC, you can assign integer values to enumerators. To enable this mode, compile with the switch **/ms_ext**.

When assignment operators are not provided, identifiers are mapped to consecutive integers from left to right, starting with 0. When assignment operators are provided, assigned values start from the most recently assigned value.

Like C-language enumerators, enumerator names must be unique but the enumerator values need not be.

The maximum number of identifiers is 65,535. Objects of type **enum** are **int** types, and their size is system-dependent. However, for transmission over the network, objects of **enum** types are treated as 2-byte objects of type **unsigned short** and must be positive values less than or equal to 32,767. Values outside this range cause the run-time exception RPC_X_ENUM_VALUE_OUT_OF_RANGE.

See Also

IDL, **typedef**, v1_enum

error_status_t

Remarks

The **error_status_t** keyword designates a type for an object that contains communication-status or fault-status information.

The **error_status_t** type is used as a part of the exception handling architecture in IDL. This type maps to an unsigned long. Applications that catch error situations have an **out** parameter or a return type of a procedure specified as **error_status_t**, and qualify the **error_status_t** with the **comm_status** or **fault_status** attributes in the ACF. If the parameter or return type was not qualified with the **comm_status** or **fault_status** attributes, then the parameter operates as though it were an unsigned long.

The MIDL 2.0 compiler generates stubs that contain the proper error handling architecture. However, earlier versions of the MIDL compiler handled a parameter or return type of **error_status_t** as though the **comm_status** and **fault_status** attributes were applied, even if they were not. With the MIDL 2.0 compiler, you must explicitly apply the **comm_status** and **fault_status** attributes to the parameter or procedure in the ACF.

The **error_status_t** type is one of the predefined types of the interface definition language. Predefined types can appear as type specifiers in **typedef** declarations, in general declarations, and in function declarators (either as the function-return-type or as parameter-type specifiers).

See Also

comm_status, **fault_status**, **IDL**

explicit_handle

[explicit_handle] {...}

Example

```
/* ACF File */
[explicit_handle]
{
};
```

Remarks

The **explicit_handle** attribute specifies that each procedure has, as its first parameter, a primitive handle, such as a **handle_t** type. This is the case even if the IDL file does not contain the handle in its parameter list. The prototypes emitted to the header file and stub routines contain the additional parameter, and that parameter is used as the handle for directing the remote call. The **explicit_handle** attribute affects both remote procedures and serialization procedures. For type serialization, the support routines are generated with the initial parameter as an explicit (serialization) handle. If the **explicit_handle** attribute is not used, the application can still specify that an operation have an explicit handle (binding or serialization) directing the call. To do so, a prototype with the first argument containing a handle type is supplied to the IDL file. Note that in **/ms_ext** mode, an argument that does not appear first can also be used as a handle directing the call. So, while the **explicit_handle** attribute is a way of giving the IDL prototype a primitive **explicit_handle** attribute, it doesn't necessarily require a change to the IDL file.

The **explicit_handle** attribute can be used as either an interface attribute or an operation attribute. As an interface attribute, it affects all the operations in the interface and all the types that require serialization support. If, however, it is used as an operation attribute, it affects only that particular operation.

See Also

ACF, **auto_handle**, **implicit_handle**

fault_status

[fault_status [**,** *ACF-function-attributes*] **]** *function-name***(**
[**[** *ACF-parameter-attributes* **]**] *parameter-name*
, ...
);

[**[** *ACF-function-attributes* **]**] *function-name***(**
[fault_status [**,** *ACF-parameter-attributes*] **]** *parameter-name*
...);

ACF-function-attributes
Specifies zero or more ACF function attributes, such as **fault_status** and **nocode**. Function attributes are enclosed in square brackets. Note that zero or more attributes can be applied to a function. Separate multiple function attributes with commas. Note that if **fault_status** appears as a function attribute, it cannot also appear as a parameter attribute.

function-name
Specifies the name of the function as defined in the IDL file.

ACF-parameter-attributes
Specifies attributes that apply to a parameter. Note that zero or more attributes can be applied to the parameter. Separate multiple parameter attributes with commas. Parameter attributes are enclosed in square brackets. IDL parameter attributes, such as directional attributes, are not allowed in the ACF. Note that if **fault_status** appears as a parameter attribute, it cannot also appear as a function attribute.

parameter-name
Specifies the parameter for the function as defined in the IDL file. Each parameter for the function must be specified in the same sequence, using the same name as defined in the IDL file.

Remarks

The **fault_status** attribute can be used as either a function attribute or as a parameter attribute, but it can appear only once per function. It can be applied either to the function or to one parameter in each function.

The **fault_status** attribute can only be applied to functions that return the type **error_status_t**. When the remote procedure fails in a way that causes a fault PDU to be returned, an error code is returned.

When **fault_status** is used as a parameter attribute, the parameter must be an **out** parameter of type **error_status_t**. If a server error occurs, the parameter is set to the error code. When the remote call completes successfully, the procedure sets the value.

The parameter associated with the **fault_status** attribute does not have to be specified in the IDL file. When the parameter is not specified, a new **out** parameter of type **error_status_t** is generated following the last parameter defined in the DCE IDL file.

It is possible for both the **fault_status** and **comm_status** attributes to appear in a single function, either as function attributes or parameter attributes. If both attributes are function attributes or if they apply to the same parameter, and no error occurs, the function or parameter has the value **error_status_ok**. Otherwise, it contains the appropriate status code value. Because values returned for **fault_status** are different from the values returned for **comm_status**, the returned values are readily interpreted.

See Also

ACF, **comm_status**, **error_status_t**

field_attributes

[**[** *field-attribute-list* **]**] *type-specifier declarator-list***;**

field-attribute-list
Specifies zero or more field attributes that apply to the structure or union member, array, or function parameter. Valid field attributes include **first_is**, **last_is**, **length_is**, **max_is**, **size_is**; the usage attributes **string**, **ignore**, and **context_handle**; the pointer attribute **ref**, **unique**, or **ptr**; and the union attribute **switch_type**. Separate multiple field attributes with commas.

type-specifier
Specifies a **base_type**, **struct, union,** or **enum** type or type identifier. An optional storage specification can precede *type-specifier*.

declarator-list
Specifies one or more standard C declarators, such as identifiers, pointer declarators, and array declarators. Separate multiple declarators with commas.

Remarks

Field attributes are used in structure, union, array, and function-parameter declarators to define transmission characteristics of the declarator during a remote procedure call.

Field-attribute keywords include **size_is**, **max_is**, **length_is**, **first_is**, and **last_is**; the usage attributes **string**, **ignore**, and **context_handle**; the union switch **switch_is**;and the pointer attributes **ref**, **unique**, and **ptr**.

The field attributes **size_is**, **max_is**, **length_is**, **first_is**, and **last_is** specify the size or range of valid data for the declarator. These field attributes associate another parameter, structure member, union member, or constant expression with the declarator.

Field attributes that are parameters must associate with declarators that are parameters; field attributes that are members of structures or unions must associate with declarators that are members of the same structure or union.

For information about the context in which field attributes appear, see **arrays**, **struct**, and **union**.

See Also

arrays, **first_is**, **IDL**, **last_is**, **length_is**, **max_is**, **size_is**

first_is

first_is(*limited-expression-list***)**

limited-expression-list
Specifies one or more C-language expressions supported by MIDL. The expression evaluates to an integer that represents the array index of the first array element to be transmitted. Separate multiple expressions with commas.

Example

```
void Proc1(
     [in] short First,
     [first_is(iFirst)] A[10]);
```

Remarks

The **first_is** attribute specifies the index of the first array element to be transmitted. If the **first_is** attribute is not present, or if the specified index is a negative number, array element 0 is the first element transmitted.

The **first_is** attribute can also help determine the values of the array indexes corresponding to the **last_is** or **length_is** attribute when these attributes are not specified. The relationship between these array indexes is as follows:

```
length = last - first + 1
```

The following relationship must also hold**:**

```
0 <= first_is <= max_is
```

The following relationship must hold when **max_is <= 0:**

```
first_is == 0
```

The **first_is** attribute cannot be used at the same time as the **string** attribute.

See Also

field_attributes, **IDL**, **last_is**, **length_is**, **max_is**, **min_is**, **size_is**

float

Remarks

The **float** keyword designates a 32-bit floating-point number.

The **float** type is one of the base types of the interface definition language (IDL). The **float** type can appear as a type specifier in **typedef** declarations, general declarations, and function declarators (as a function-return-type specifier and a parameter-type specifier). For the context in which type specifiers appear, see **IDL**.

The **float** type cannot appear in **const** declarations.

See Also

base_types, **double**.

handle

typedef [handle] *typename*;
handle_t __RPC_USER *typename*_**bind (***typename***);**
void __RPC_USER *typename*_**unbind (***typename***, handle_t);**

typename
Specifies the name of the user-defined binding-handle type.

Examples

```
typedef [handle] struct {
    char machine[8];
    char nmpipe[256];
 } h_service;

handle_t __RPC_USER h_service_bind(h_service);
void __RPC_USER h_service_unbind(h_service, handle_t);
```

Remarks

The **handle** attribute specifies a user-defined or "customized" handle type. User-defined handles permit developers to design handles that are meaningful to the application.

A user-defined handle can only be defined in a type declaration, not in a function declarator.

A parameter of a type defined by the **handle** attribute is used to determine the binding for the call and is transmitted to the called procedure.

The user must provide binding and unbinding routines to convert between primitive and user-defined handle types. Given a user-defined handle of type *typename*, the user must supply the routines *typename*_**bind** and *typename*_**unbind**. For example, if the user-defined handle type is named MYHANDLE, the routines are named MYHANDLE_**bind** and MYHANDLE_**unbind**.

If successful, the *typename*_**bind** routine should return a valid primitive binding handle and if unsuccessful, a NULL. If the routine returns NULL, the *typename*_**unbind** routine will not be called. If the binding routine returns an invalid binding handle different from NULL, the stub behavior is undefined.

When the remote procedure has a user-defined handle as a parameter or as an implicit handle, the client stubs call the binding routine before calling the remote procedure. The client stubs call the unbinding routine after the remote call.

In DCE IDL, a parameter with the **handle** attribute must appear as the first parameter in the remote procedure argument list. Subsequent parameters, including other **handle** attributes, are treated as ordinary parameters. Microsoft supports an extension to DCE IDL that allows the user-defined **handle** parameter to appear in positions other than the first parameter.

See Also

handles, **IDL**, **implicit_handle**, **typedef**

handles

Remarks

Binding handles are data objects that represent the binding between the client and the server.

MIDL supports the base type **handle_t**. Handles of this type are known as "primitive handles."

You can define your own handle types using the **handle** attribute. Handles defined in this way are known as "user-defined or customized handles" or "generic handles."

You can also define a handle that maintains state information using the **context_handle** attribute. Handles defined in this way are known as "context handles."

If no state information is needed and you do not choose to call the RPC run-time libraries to manage the handle, you can request that the run-time libraries provide automatic binding. This is done by using the ACF keyword **auto_handle**.

You can specify a global variable as the binding handle by using the ACF keyword **implicit_handle**.

The **explicit_handle** keyword is used to state that each remote function has an explicitly specified handle.

See Also

auto_handle, **base_types**, **context_handle**, **explicit_handle**, **handle**, **handle_t**, **implicit_handle**

handle_t

Remarks

The **handle_t** keyword declares an object to be of the primitive handle type **handle_t**. A primitive binding handle is a data object that can be used by the application to represent the binding.

The **handle_t** type is one of the predefined types of the interface definition language (IDL). It can appear as a type specifier in **typedef** declarations, general declarations, and function declarators (as a function-return-type specifier and a parameter-type specifier). For the context in which type specifiers appear, see **IDL**

In Microsoft RPC, parameters of type **handle_t** can occur only as **in** parameters. Primitive handles cannot have the **unique** or **ptr** attribute.

Parameters of type **handle_t** (primitive handle parameters) are not transmitted on the network.

See Also

base_types, **handles**

heap

The DCE ACF keyword **heap** is not implemented in Microsoft RPC.

hyper

Remarks

The keyword **hyper** indicates a 64-bit integer that can be declared as either signed or unsigned.

The **hyper** type is one of the base types of the interface definition language (IDL). The **hyper** type can appear as a type specifier in **const** declarations, **typedef** declarations, general declarations, and function declarators (as a function-return-type specifier and as a parameter-type specifier). For the context in which type specifiers appear, see **IDL**.

Note For 16-bit platforms, the MIDL compiler replaces unsigned hyper integers with **MIDL_uhyper**. This allows interfaces with unsigned hyper integers to be defined on platforms that do not directly support 64-bit integers. **MIDL_uhyper** is defined in the RPC header files.

See Also

base_types

idempotent

[[**[** *IDL-operation-attributes* **]**]] *operation-attribute* **, ...**

IDL-operation-attributes
Specifies zero or more IDL operation attributes, such as **idempotent** and **broadcast**. Operation attributes are enclosed in square brackets.

Remarks

An **idempotent** operation is one that does not modify state information and returns the same results each time it is performed. Performing the routine more than once has the same effect as performing it once.

RPC supports two types of remote call semantics: calls to **idempotent** operations and calls to **non-idempotent** operations. An **idempotent** operation can be carried out more than once with no ill effect. Conversely, a **non-idempotent** operation (at-most-once) cannot be executed more than once because it will either return different results each time or because it modifies some state.

To ensure that a procedure is automatically re-executed if the call does not complete, use the **idempotent** attribute. If the **idempotent**, **broadcast**, or **maybe** attributes are not present, the procedure will use **non-idempotent** semantics by default. In this case, the operation is executed only once.

See Also

broadcast, **IDL**, **maybe**, **non-idempotent**

IDL

[*interface-attribute-list* **]** **interface** *interface-name*
{
[**import** *import-file-list* **;** **...**]
[**cpp_quote(**"*string*"*)* **...**]

[**const** *const-type identifier* **=** *const-expression* **;** **...**]

[[**typedef**] [**[***type-attribute-list***]**] *type-specifier declarator-list***;** **...**]

[[**[***function-attr-list***]**] *type-specifier* [*pointer-declarator*] *function-name***(**
[**[***parameter-attribute-list***]**] *type-specifier* [*declarator*]
, ...
);
...
]
}

.
.
.

interface-attribute-list
Specifies either the attribute **uuid** or the attribute **local** and other optional attributes that apply to the interface as a whole. The attributes **endpoint**, **version**, and **pointer_default** are optional. When you compile with the **/app_config** switch, either **implicit_handle** or **auto_handle** can also be present. Separate multiple attributes with commas. The *interface-attribute-list* does not have to be present for imported IDL files but must be present for the base IDL file.

interface-name
Specifies the name of the interface. The identifier must be unique or different from any type names. It also must be 17 characters or less because it is used to form the name of the interface handle. The same interface name must be supplied in the ACF, except when you compile with the **/acf** switch.

import-file-list
Specifies one or more IDL files to import. Separate filenames with commas.

string
Specifies a string that is emitted in the generated header file.

const-type
Specifies the name of an integer, character, **boolean**, **void** *, **byte**, or string (**char** *, **byte** *, **wchar_t** *) type. Only these types can be assigned **const** values in the IDL file.

identifier
Specifies a valid MIDL identifier. Valid MIDL identifiers consist of up to 31 alphanumeric and/or underscore characters and must start with an alphabetic or underscore character.

const-expression
Specifies a constant declaration. The *const-expression* must evaluate to the type specified by *const-type.* For more information, see **const**.

type-attribute-list
Specifies one or more attributes that apply to the type. Valid type attributes include **handle**, **switch_type**, **transmit_as**; the pointer attribute **ref**, **unique**, or **ptr**; and the usage attributes **context_handle**, **ignore**, and **string**. Separate multiple attributes with commas.

type-specifier
Specifies a **base_type**, **struct**, **union, enum** type, or type identifier. An optional storage

specification can precede *type-specifier*.

declarator and *declarator-list*
Specify standard C declarators, such as identifiers, pointer declarators, and array declarators. For more information, see **pointers** and **arrays**. The *declarator-list* consists of one or more declarators separated by commas. The parameter declarator in the function declarator, such as the parameter name, is optional.

function-attr-list
Specifies zero or more attributes that apply to the function. Valid function attributes are **callback**, **local**; the pointer attribute **ref**, **unique**, or **ptr**; and the usage attributes **string**, **ignore**, and **context_handle**.

pointer-declarator
Specifies zero or more pointer declarators. A pointer declarator is the same as the pointer declarator used in C; it is constructed from the ***** designator, modifiers such as **far**, and the qualifier **const**.

function-name
Specifies the name of the remote procedure.

parameter-attribute-list
Specifies zero or more attributes appropriate for the specified parameter type. Parameter attributes can take the directional attributes **in** and **out**; the field attributes **first_is**, **last_is**, **length_is**, **max_is**, **size_is**, and **switch_type**; the pointer attribute **ref**, **unique**, or **ptr**; and the usage attributes **context_handle** and **string**. The usage attribute **ignore** cannot be used as a parameter attribute. Separate multiple attributes with commas.

Examples

```
[ uuid(12345678-1234-1234-1234-123456789ABC),
  version(3.1),
  pointer_default(unique)
] interface IdlGrammarExample
{
import "windows.idl", "other.idl";
const wchar_t * NAME = L"Example Program";
typedef char * PCHAR;

void DictCheckSpelling(
     [in, string] PCHAR   word,     // word to look up
     [out]        short * isPresent // 0 if not present
);
}
```

Remarks

The IDL file contains the specification for the interface. The interface includes the set of data types and the set of functions to be executed from a remote location. Interfaces specify the function prototypes for remote functions and for many aspects of their behavior from the point of view of interface users.

Another file, the application configuration file (ACF), contains attributes that tailor the application for a specific operating environment. For more information, see **ACF**.

An interface specification consists of an interface header followed by an interface body. The interface header includes an attribute list describing characteristics that apply to the interface as a whole. The interface body contains the remote data types and function prototypes.

The interface body contains zero or more import lists, constant declarations, general declarations, and function declarators.

In **/ms_ext** mode, an IDL file can contain multiple interfaces. Type definitions, construct declarations,

and imports can occur outside of the interface body. All definitions from the main IDL file will appear in the generated header file, and all the procedures from all the interfaces in the main IDL file will generate stub routines. This enables applications that support multiple interfaces to merge IDL files into a single, combined IDL file. As a result, it requires less time to compile the files and also allows MIDL to reduce redundancies in the generated stubs. This can significantly improve **object** interfaces by the ability to share common code for base interfaces and derived interfaces. For non-**object** interfaces, the procedure names must be unique across all the interfaces. For **object** interfaces, the procedure names need only be unique within an interface.

The syntax for declarative constructs in the IDL file is similar to that for C. MIDL supports all Microsoft C version 7.0 declarative constructs except:

- Older style declarators that allow a declarator to be specified without a type specifier, such as:

```
x (y)
short x (y)
```

- Declarations with initializers (MIDL only accepts declarations that conform to the MIDL **const** syntax)
- Floating-point constants

The **import** keyword specifies the names of one or more IDL files to import. The import directive is similar to the C **include** directive, except that only data types are assimilated into the importing IDL file.

The constant declaration specifies **boolean**, integer, character, wide-character, string, and **void *** constants. For more information, see **const**.

A general declaration is similar to the C **typedef** statement with the addition of IDL type attributes. In **/ms_ext** mode, the MIDL compiler also allows an implicit declaration in the form of a variable definition.

The function declarator is a special case of the general declaration. You can use IDL attributes to specify the behavior of the function return type and each of the parameters.

See Also

arrays, **const**, **enum**, **import**, **in**, **interface**, **midl**, **out**, **pointers**, **struct**, **union**

ignore

[ignore] *pointer-member-type pointer-name*;

pointer-member-type
Specifies the type of the pointer member of the structure or union.

pointer-name
Specifies the name of the pointer member that is to be ignored during marshalling.

Example

```
typedef struct _DBL_LINK_NODE_TYPE {
    long value;
    struct _DBL_LINK_NODE_TYPE * next;
    [ignore] struct _DBL_LINK_NODE_TYPE * previous;
} DBL_LINK_NODE_TYPE;
```

Remarks

The **ignore** attribute designates that a pointer contained in a structure or union and the object indicated by the pointer is not transmitted. The **ignore** attribute is restricted to pointer members of structures or unions.

The value of a structure member with the **ignore** attribute is undefined at the destination. An **in** parameter is not defined at the remote computer. An **out** parameter is not defined at the local computer.

The **ignore** attribute allows you to prevent transmisison of data. This is useful in situations such as a double-linked list. The following example includes a double-linked list that introduces data aliasing:

```
/* IDL file */
typedef struct _DBL_LINK_NODE_TYPE {
    long value;
    struct _DBL_LINK_NODE_TYPE * next;
    struct _DBL_LINK_NODE_TYPE * previous;
} DBL_LINK_NODE_TYPE;

void remote_op([in] DBL_LINK_NODE_TYPE * list_head);

/* application */
DBL_LINK_NODE_TYPE * p, * q

p = (DBL_LINK_NODE_TYPE *)
     midl_user_allocate(sizeof(DBL_LINK_NODE_TYPE));
q = (DBL_LINK_NODE_TYPE *)
     midl_user_allocate(sizeof(DBL_LINK_NODE_TYPE));

p->next = q;
q->previous = p;
p->previous = q->next = NULL;
...
remote_op(p);
```

Aliasing occurs in the preceding example because the same memory area is available from two different pointers in the function **p** and **p->next->previous**.

Note that **ignore** cannot be used as a type attribute.

See Also

pointers, **ptr**, **ref**, **unique**

iid_is

[iid_is(*limited-expression***)]**

limited-expression
Specifies a limited expression. MIDL does not allow function invocations in expressions and does not allow pre- and post-increment and -decrement operators.

Example

```
HRESULT    CreateInstance(
     [in] REFIID riid,
     [out, iid_is(riid)] IUnknown ** ppvObject);
```

Remarks

The **iid_is** pointer attribute specifies the IID of the OLE interface pointed to by an interface pointer. You can use **iid_is** in attribute lists for function parameters and for structure or union members. The stubs use the IID to determine how to marshal the interface pointer. This is useful for an interface pointer that is typed as a base class parameter.

Files that use the **iid_is** attribute must be compiled with the MIDL compiler switch **/ms_ext**.

See Also

object, **uuid**

implicit_handle

implicit_handle(*handle-type handle-name***)**

handle-type
Specifies the handle data type, such as the base type **handle_t** or a user-defined handle type.

handle-name
Specifies the name of the handle.

Example

```
/* ACF file */
[implicit_handle(handle_t hMyHandle)]
{
}
```

Remarks

The **implicit_handle** attribute specifies the handle used for functions that do not include an explicit handle as a procedure parameter. If the procedure is remote, the handle will be used as the binding handle for the remote call. The implicit handle may also be used to establish an initial binding for a function that uses a context handle. If the procedure is a serializing procedure, the handle is used as a serializing handle controlling the operation. In the case of type serialization, the handle is used as the serialization handle for all the serialized types.

The **implicit_handle** attribute specifies a global variable that contains a handle used by any function needing implicit handles.

The implicit binding handle type must be either **handle_t** (or a type based on **handle_t**) or a user-defined handle type specified with the **handle** attribute. The implicit serializing handle must be a type based on **handle_t**.

If the implicit handle type is not defined in the IDL file or in any files included and imported by the IDL file for the MIDL computer, you must supply the file containing the handle-type definition when you compile the stubs. Use the ACF **include** statement to include the file containing the handle-type definition.

The **implicit_handle** attribute can occur once, at most. The **implicit_handle** attribute can occur only if the **auto_handle** or **explicit_handle** attribute does not occur.

See Also

ACF, **auto_handle**, **explicit_handle**, **include**

import

import "*filename***" [, ...] ;**

filename
Specifies the name of the IDL file to import.

Example

```
import "foo1.idl";
import "foo2.idl", "foo3.idl", "foo4.idl";
```

Remarks

The **import** directive is similar to the C-language preprocessor macro **#include**. The **import** directive directs the compiler to include data types defined in the imported IDL files. In contrast to the C-language **#include** macro, the **import** directive ignores procedure prototypes.

The **import** keyword is optional and can appear zero or more times in the IDL file. Each **import** keyword can be associated with more than one filename. Multiple filenames are separated by comma characters. You must enclose the filename within quotation marks and end the import statement with the semicolon character (;). 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 C-language header (.H) file generated for the interface does not directly contain the imported types but instead generates a **#include** directive for the header file corresponding to the imported interface. For example, when FOO.IDL imports BAR.IDL, the generated header file FOO.H includes BAR.H (FOO.H contains the directive **#include** BAR.H).

The **import** function is **idempotent** – that is, importing an interface more than once has the same effect as importing it once.

The behavior of the import directive is independent of the MIDL compiler mode switches **/ms_ext, /c_ext,** and **/app_config**.

See Also

IDL

in

[**[***function-attribute-list***]**] *type-specifier* [*pointer-declarator*] *function-name***(**
[in [**,** *parameter-attribute-list*] **]** *type-specifier* [*declarator*]
, ...
);

function-attribute-list
Specifies zero or more attributes that apply to the function. Valid function attributes are **callback**, **local**, the pointer attribute **ref**, **unique**, or **ptr**, and the usage attributes **string**, **ignore**, and **context_handle**.

type-specifier
Specifies a **base_type**, **struct, union,** or **enum** type or type identifier. An optional storage specification can precede *type-specifier*.

pointer-declarator
Specifies zero or more pointer declarators. A pointer declarator is the same as the pointer declarator used in C; it is constructed from the ***** designator, modifiers such as **far**, and the qualifier **const**.

function-name
Specifies the name of the remote procedure.

parameter-attribute-list
Specifies zero or more attributes appropriate for the specified parameter type. Parameter attributes with the **in** attribute can also take the directional attribute **out**; the field attributes **first_is**, **last_is**, **length_is**, **max_is**, **size_is** and **switch_type**; the pointer attribute **ref**, **unique**, or **ptr**; and the usage attributes **context_handle** and **string**. The usage attribute **ignore** cannot be used as a parameter attribute. Separate multiple attributes with commas.

declarator
Specifies standard C declarators, such as identifiers, pointer declarators, and array declarators. For more information, see **pointers** and **arrays**. The parameter declarator in the function declarator, such as the parameter name, is optional.

Example

```
void MyFunction([in] short count);
```

Remarks

The **in** attribute indicates that a parameter is to be passed from the calling procedure to the called procedure.

A related attribute, **out**, indicates that a parameter is to be returned from the called procedure to the calling procedure. The **in** and **out** attributes are known as directional parameter attributes because they specify the direction in which parameters are passed. A parameter can be defined as **in**, **out**, or **in, out**.

The **in** attribute identifies parameters that are marshalled by the client stub for transmission to the server.

The **in** attribute is applied to a parameter by default when no directional parameter attribute is specified.

See Also

IDL, **midl_user_allocate**, **out**

include

include *filenames*;

filenames
Specifies the name of one or more C-language header files. The .H extension must be supplied in the MS-DOS, Windows, and Windows NT environments. Separate multiple C-language header filenames with commas.

Remarks

The body of the ACF can contain **include** directives, ACF **typedef** attributes, and ACF function and parameter attributes.

The ACF **include** statement specifies one or more header files included in the generated stub code. The stub code contains a C-preprocessor **#include** statement, and the user supplies the C-language header file when compiling the stubs. **Include** statements rely on the C-compiler mechanism of searching the directory structure for included files.

Note Use the **import** directive rather than the **include** directive for system files, such as WINDOWS.H, that contain data types you want to make available to the IDL file. The **import** directive ignores function prototypes and allows you to use MIDL compiler switches that optimize the generation of support routines.

See Also

ACF, **import**, **typedef**

in_line

The DCE IDL keyword **in_line** is not supported in Microsoft RPC.

See Also

IDL

int

[*type-specifier*] [**signed | unsigned**] *integer-modifier* [**int**] *declarator-list*;

type-specifier
Specifies a **base_type**, **struct**, **union**, **enum** type, or type identifier. An optional storage specification can precede *type-specifier*.

integer-modifier
Specifies the keyword **small**, **short**, **long**, or **hyper**, which selects the size of the integer data. The size qualifier must be present.

declarator-list
Specifies one or more standard C declarators, such as identifiers, pointer declarators, and array declarators. (Function declarators and bit-field declarations are not allowed in structures that are transmitted in remote procedure calls. These declarators are allowed in structures that are not transmitted.) Separate multiple declarators with commas.

Examples

```
signed short int i = 0;
short int j = i;
typedef struct {
    small int          i1;
    short              i2;
    unsigned long int i3;
} INTSIZETYPE;

void MyFunc([in] long int lCount);
```

Remarks

The keyword **int** is an optional keyword that can accompany the keywords **small**, **short**, and **long**.

Integer types are among the base types of the interface definition language (IDL). They can appear as type specifiers in **typedef** declarations, general declarations, and function declarators (as a function-return-type specifier and as a parameter-type specifier). For the context in which type specifiers appear, see **IDL**.

The keyword **int** cannot be used alone. It must always appear with one of the valid integer modifiers. These modifiers specify the number of bits used to describe the integer data.

If no integer sign specification is provided, the integer type defaults to **signed**.

DCE IDL compilers do not allow the keyword **signed** to specify the sign of integer types. To allow the use of the keyword **signed**, use the MIDL compiler switch **/ms_ext**.

See Also

base_types, **long**, **/ms_ext**, **short**, **small**

__int64

The keyword **__int64** specifies a valid integer supported by the MIDL compiler. For a discussion of how to use **__int64**, see **hyper**.

See Also

IDL, **int**

interface

[*interface-attribute-list* **]** **interface** *interface-name* [**:** *base-interface*]

/*IDL file **typedef** syntax */
typedef interface interface-name declarator-list

interface-attribute-list

Specifies attributes that apply to the interface as a whole. Valid interface attributes for an IDL file include **endpoint**, **local**, **object**, **pointer_default**, **uuid**, and **version**. Valid interface attributes for an ACF include **encode**, **decode**, either **auto_handle** or **implicit_handle**, and either **code** or **nocode**.

interface-name
Specifies the name of the interface. The identifier must start with an alphabetic or underscore character and can consist of up to 17 alphanumeric and underscore characters. The identifier must be 17 characters or less because it is used to form the name of the interface handle.

base-interface
Specifies the name of an interface from which this derived interface inherits member functions, status codes, and interface attributes. The derived interface does not inherit type definitions. To do this, use the **import** keyword to import the IDL file of the base interface.

declarator-list
Specifies standard C declarators, such as identifiers, pointer declarators, and array declarators. For more information, see **pointers** and **arrays**. The *declarator-list* consists of one or more declarators, separated by commas.

Examples

```
/* use of interface keyword in IDL file for an RPC interface */
  [ uuid (00000000-0000-0000-0000-000000000000),
    version (1.0) ]
  interface remote_if_2
  {
  }

/* use of interface keyword in ACF for an RPC interface */
  [ implicit_handle( handle_t xa_bhandle ) ]
  interface remote_if_2
  {
  }

/* use of interface keyword in IDL file for an OLE interface */
  [ object, uuid (00000000-0000-0000-0000-000000000000) ]
  interface IDerivedInterface : IBaseInterface
  {
    import "base.idl"
    Save();
  }

/* use of interface keyword to define an interface pointer type */
  typedef interface IStorage *LPSTORAGE;
```

Remarks

The **interface** keyword specifies the name of the interface. The interface name must be provided in

both the IDL file and the ACF.

The interface names in the IDL file and ACF must be the same, except when you use the MIDL compiler switch **/acf**. For more information, see **/acf**.

The interface name forms the first part of the name of interface-handle data structures that are parameters to the RPC run-time functions. For more information, see **RPC_IF_HANDLE**.

If the interface header includes the **object** attribute to indicate an OLE interface, it must also include the **uuid** attribute and must specify the base OLE interface from which it is derived. For more information about OLE interfaces, see **object**.

You can also use the **interface** keyword with the **typedef** keyword to define an interface data type.

See Also

ACF, **endpoint**, **IDL**, **local**, **pointer_default**, **uuid**, **version**

last_is

[last_is(*limited-expression-list* **)]**

limited-expression-list
Specifies one or more C-language expressions supported by MIDL. The expression evaluates to an integer that represents the array index of the last array element to be transmitted. Separate multiple expressions with commas.

Example

```
proc1(
     [in] short Last,
     [in, last_is(Last)] short asNumbers[MAXSIZE]);
```

Remarks

The field attribute **last_is** specifies the index of the last array element to be transmitted. When the specified index is zero or negative, no array elements are transmitted.

The **last_is** attribute determines the value of the array index corresponding to the **length_is** attribute when **length_is** is not specified. The relationship between these array indexes is as follows:

```
length = last - first + 1
```

If the value of the array index specified by **first_is** is larger than the value specified by **last_is**, zero elements are transmitted.

The **last_is** attribute can be used only if the array has a fixed allocation size. The **last_is** attribute cannot be used as a field attribute at the same time as the **string** attribute.

When the value specified by **max_is** is equal to or greater than zero, the following relationship must be true:

```
0 <= last_is <= max_is
```

When **min_is** is greater than or equal to **max_is**, the following relationship must be free:

```
last_is <= max_is
```

See Also

field_attributes, **first_is**, **IDL**, **length_is**, **max_is**, **size_is**

length_is

[length_is(*limited-expression-list* **)]**

limited-expression-list
Specifies one or more C-language expressions that are supported by MIDL. The expression evaluates to an integer that represents the number of array elements to be transmitted. Separate multiple expressions with commas.

Examples

```
/* counted string holding at most "size" characters */
typedef struct {
     unsigned short size;
     unsigned short length;
     [size_is(size), length_is(length)] char string[*];
 } COUNTED_STRING_TYPE;

/* counted string holding at most 80 characters */
typedef struct {
     unsigned short length;
     [length_is(length)] char string[80];
 } STATIC_COUNTED_STRING_TYPE;

void Proc1(
     [in] short iLength;
     [in, length_is(iLength)] short asNumbers[10];
```

Remarks

The **length_is** attribute specifies the number of array elements to be transmitted. A non-negative value must be specified.

The **length_is** attribute determines the value of the array indexes corresponding to the **last_is** attribute when **last_is** is not specified. The relationship between these array indexes is as follows:

```
length = last - first + 1
```

The **length_is** attribute cannot be used at the same time as the **last_is** attribute or the **string** attribute.

To define a counted string with a **length_is** or **last_is** attribute, use a character array or pointer without the **string** attribute.

See Also

field_attributes, **first_is**, **IDL**, **last_is**, **max_is**, **min_is**, **size_is**

local

[local [**,** *interface-attribute-list*] **] interface** *interface-name*

[object, uuid(*string-uuid***), local** [**,** *interface-attribute-list*] **]**
interface *interface-name*

[local [**,** *function-attribute-list*] **]** *function-declarator* **;**

interface-attribute-list
Specifies other attributes that apply to the interface as a whole. The attributes **endpoint**, **version**, and **pointer_default** are optional. When you compile with the **/app_config** switch, either **implicit_handle** or **auto_handle** can also be present. Separate multiple attributes with commas.

interface-name
Specifies the name of the interface.

string-uuid
Specifies a UUID string generated by the **uuidgen** utility. You can enclose the UUID string in quotes when you use the MIDL compiler switch **/ms_ext**.

function-attribute-list
Specifies zero or more attributes that apply to the function. Valid function attributes are **callback**; the pointer attribute **ref**, **unique**, or **ptr**; and the usage attributes **string**, **ignore**, and **context_handle**. Separate multiple attributes with commas.

function-declarator
Specifies the type specifier, function name, and parameter list for the function.

Examples

```
/* IDL file #1 */
[local] interface local_procs
{ void MyLocalProc(void);}

/* IDL file #2 */
[object,
 uuid(12345678-1234-1234-123456789ABC),
 local] interface local_object_procs
{ void MyLocalObjectProc(void);}

/* IDL file #3 */
[uuid(12345678-1234-1234-123456789ABC)]
interface mixed_procs
{
[local] void MyLocalProc(void);
void MyRemoteProc([in] short sParam);
}
```

Remarks

The **local** attribute can be applied to individual functions or to the interface as a whole.

When used in the interface header, the **local** attribute allows you to use the MIDL compiler as a header generator. The compiler does not generate stubs for any functions and does not ensure that the header can be transmitted.

For an RPC interface, the **local** attribute cannot be used at the same time as the **uuid** attribute. Either **uuid** or **local** must be present in the interface header, and the one you choose must occur exactly

once.

For an OLE interface (identified by the **object** interface attribute), the interface attribute list can include the **local** attribute even though the **uuid** attribute is present.

When used in an individual function, the **local** attribute designates a local procedure for which no stubs are generated. Using **local** as a function attribute is a Microsoft extension to DCE IDL and requires the MIDL compiler switch **/ms_ext.**

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.

See Also

IDL, /ms_ext, object, uuid

long

The **long** keyword designates a 32-bit integer. It can be preceded by either the keyword **signed** or the keyword **unsigned**. The **int** keyword is optional and can be omitted. To the MIDL compiler, a long integer is signed by default and is synonymous with **signed long int**.

The **long** integer type is one of the base types of the IDL language. The **long** integer type can appear as a type specifier in **const** declarations, **typedef** declarations, general declarations, and function declarators (as a function-return-type specifier and as a parameter-type specifier). For the context in which type specifiers appear, see **IDL**.

See Also

base_types, **hyper**, **int**, **short**, **small**

max_is

[max_is(*limited-expression-list* **)]**

limited-expression-list
Specifies one or more C-language expressions that are supported by MIDL. The expression evaluates to an integer that represents the highest valid array index. Separate multiple expressions with commas.

Examples

```
void Proc1(
     [in] short m,
     [in, max_is(m)] short a[]);  /* if m = 10,
           there are 11 transmitted      values of (a[0]...a[10])*/
void Proc2(
     [in] short m,
     [in, max_is(m)] short b[][20];  /* if m = 10,
           the valid range for b   is b[0...10][20] */
```

Remarks

The **max_is** attribute designates the maximum value for a valid array index. For an array of size *n* in C, where the first array element is element number 0, the maximum value for a valid array index is *n*-1.

The **max_is** attribute cannot be used as a field attribute at the same time as the **size_is** attribute.

See Also

field_attributes, **IDL**, **min_is**, **size_is**

maybe

[[**[** *IDL-operation-attributes* **]**]] *operation-attribute* **, ...**

IDL-operation-attributes
Specifies zero or more IDL operation attributes, such as **maybe** and **idempotent**. Operation attributes are enclosed in square brackets.

Remarks

The keyword **maybe** indicates that the remote procedure call does not need to execute every time it is called and the client does not expect a response. Note that the **maybe** protocol ensures neither delivery nor completion of the call.

A call with the **maybe** attribute cannot contain output parameters and is implicitly an **idempotent** call.

See Also

broadcast, **idempotent**, **IDL**, **non-idempotent**

midl_user_allocate

void __RPC_FAR * __RPC_API midl_user_allocate (size_t *cBytes*);

cBytes
Specifies the count of bytes to allocate.

Example

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

Remarks

The **midl_user_allocate** function must be supplied by both client applications and server applications. Applications and generated stubs call **midl_user_allocate** when dealing with objects referenced by pointers:

- The server application should call **midl_user_allocate** to allocate memory for the application – for example, when creating a new node.
- The server stub calls **midl_user_allocate** when unmarshalling pointed-at 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,** and **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 must return a null pointer.

It is recommended that **midl_user_allocate** return a pointer that is 8 bytes aligned.

See Also

allocate, **midl_user_free**, **pointers**, **ptr**, **ref**, **unique**

midl_user_free

void __RPC_API midl_user_free(void __RPC_FAR * p);

Example

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

Remarks

The **midl_user_free** function must be supplied by both client applications and server applications. 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 objects referenced by pointers:

- The server application should call **midl_user_free** to free memory allocated by the application – for example, when deleting a specified node.
- The server stub calls **midl_user_free** to release memory on the server after marshalling all **out** arguments, **in, out** arguments, and the return value.

See Also

midl_user_allocate, **pointers**, **unique**

min_is

The DCE IDL attribute **min_is** is not implemented in Microsoft RPC. The value of the minimum valid array index is zero.

See Also

arrays, **IDL**, **max_is**

ms_union

[..., ms_union, ...] interface-name {...}

interface-name
Specifies the name of the interface.

Example

```
[ms_union] long procedure (...);
```

Remarks

The keyword **ms_union** is used to control the NDR alignment of non-encapsulated unions.

The MIDL compiler in this version of Microsoft RPC mirrors the behavior of the OSF DCE IDL compiler for non-encapsulated unions. However, because earlier versions of the MIDL compiler did not do so, the **/ms_union** switch provides compatibility with interfaces built on previous versions of the MIDL compiler.

The ms_union feature can be used as an IDL interface attribute, an IDL type attribute, or as a command-line switch (**/ms_union**).

See Also

IDL, **/ms_union**

ncacn_dnet_nsp

endpoint("ncacn_dnet_nsp:*server-name***[***port-name***]")**

server-name
Specifies the name or internet address for the server, or host, computer. The name is a character string.

port-name
Specifies a DECnet object name or object number. The object name can consist of up to 16 characters. Object numbers can be prefixed by the pound sign (#).

Examples

```
[   uuid(12345678-4000-2006-0000-20000000001a),
    version(1.1),
    endpoint("ncacn_dnet_nsp:node[RPC0034501]")

[   uuid(12345678-4000-2006-0000-20000000001a),
    version(1.1),
    endpoint("ncacn_dnet_nsp:node[#41]")
]
```

Remarks

The **ncacn_dnet_nsp** keyword identifies DECnet as the protocol family for the endpoint.

The syntax of the DECnet transport port string, like all port strings, is defined independently of the IDL specification. The compiler performs some syntax checking but does not guarantee that the endpoint specification is correct. Some errors may be reported at run time.

See Also

endpoint, **ncacn_ip_tcp**, **ncacn_nb_nb**, **ncacn_nb_tcp**, **ncacn_np**, **ncacn_spx**, **ncalrpc**, **string_binding**

ncacn_ip_tcp

endpoint("ncacn_ip_tcp:*server-name***[***port-name***]")**

server-name
Specifies the name or Internet address for the server, or host, computer. The name is a character string. The Internet address is a common Internet address notation.

port-name
Specifies an optional 16-bit number. Values of less than 1024 are usually reserved. If no value is specified, the endpoint-mapping service selects a valid *port-name* value.

Example

```
[   uuid(12345678-4000-2006-0000-20000000001a),
    version(1.1),
    endpoint("ncacn_ip_tcp:rainier[1404]")
] interface foo

endpoint("ncacn_ip_tcp:128.10.2.30[1404]")
```

Remarks

The **ncacn_ip_tcp** keyword identifies TCP/IP as the protocol family for the endpoint.

Only the **ncacn_spx** and **ncacn_ip_tcp** protocols support cancels. For all other protocols, the cancel routines will return RPC_S_OK, but there will be no effect. Specifically:

RpcCancelThread will alert the specified thread, but will not interrupt a pending RPC.

RpcTestCancel will return RPC_S_OK if the current thread has been alerted.

RpcMgmtSetCancelTimeout has no visible effect.

RpcCancelThread, **RpcTestCancel**, and **RpcMgmtSetCancelTimeout** are only supported on Windows NT platforms; all other platforms return RPC_S_CANNOT_SUPPORT.

The syntax of the TCP/IP transport port string, like all port strings, is defined independently of the IDL specification. The compiler performs some syntax checking but does not guarantee that the endpoint specification is correct. Some errors may be reported at run time rather than during compilation.

See Also

endpoint, **IDL**, **ncacn_nb_tcp**, **ncacn_np**, **ncacn_spx**, **ncalrpc**, **string_binding**

ncacn_nb_nb

endpoint("ncacn_nb_nb:[*port-name***]")**

port-name
Specifies an optional 8-bit value ranging from 0 to 255. Values of less than 0x20 are reserved. If the *port-name* value is not specified, the endpoint-mapping service selects the port value.

Example

```
[   uuid(12345678-4000-2006-0000-20000000001a),
    version(1.1),
    endpoint("ncacn_nb_nb:[100]")
]
```

Remarks

The **ncacn_nb_nb** keyword identifies NetBEUI over NetBIOS as the protocol family for the endpoint.

The syntax of the NetBIOS port string, like all port strings, is defined by the transport implementation and is independent of the IDL specification. The MIDL compiler performs limited syntax checking but does not guarantee that the endpoint specification is correct. Some classes of errors may be reported at run time rather than during compilation.

See Also

endpoint, **IDL**, **ncacn_ip_tcp**, **ncacn_nb_tcp**, **ncacn_np**, **ncacn_spx**, **ncalrpc**, **string_binding**

ncacn_nb_tcp

endpoint("ncacn_nb_tcp:[*port-name***]")**

port-name
Specifies an optional 8-bit value ranging from 0 to 255. Values of less than 0x20 are reserved. If the *port-name* value is not specified, the endpoint-mapping service selects the port value.

Example

```
[   uuid(12345678-4000-2006-0000-20000000001a),
    version(1.1),
    endpoint("ncacn_nb_tcp:[100]")
]
```

Remarks

The **ncacn_nb_tcp** keyword is used to identify TCP over NetBIOS as the protocol family for the endpoint.

The syntax of the NetBIOS port string, like all port strings, is defined by the transport implementation and is independent of the IDL specification. The MIDL compiler performs limited syntax checking but does not guarantee that the endpoint specification is correct. Some classes of errors may be reported at run time rather than compile time.

See Also

endpoint, **IDL**, **ncacn_ip_tcp**, **ncacn_nb_nb**, **ncacn_np**, **ncacn_spx**, **ncalrpc**, **string_binding**

ncacn_np

endpoint("ncacn_np:*server-name***[\\pipe***pipe-name***]")**

server-name
Specifies the name of the server. The *server-name* is optional. If the *server-name* is present, it must be preceded by four backslash characters, as in "\\\\myserver".

pipe-name
Specifies a valid pipe name. A valid pipe name is a string containing identifiers separated by backslash characters. The first identifier must be **pipe**. Each identifier must be separated by two backslash characters.

Example

```
[   uuid(12345678-4000-2006-0000-20000000001a),
    version(1.1),
    endpoint("ncacn_np:[\\pipe\\stove\\hat]")
]
```

Remarks

The **ncacn_np** keyword identifies named pipes as the protocol family for the endpoint.

Note For Windows 95 platforms, **ncacn_np** is not supported.

The syntax of the named-pipe port string, like all port strings, is defined by the transport implementation and is independent of the IDL specification. The MIDL compiler performs limited syntax checking but does not guarantee that the endpoint specification is correct. Some classes of errors may be reported at run time rather than during compilation.

See Also

endpoint, **IDL**, **ncacn_ip_tcp**, **ncacn_nb_nb**, **ncacn_nb_tcp**, **ncalrpc**, **string_binding**

ncacn_spx

endpoint("ncacn_spx:*link-address***[***port-name***]")**

link-address
Specifies 8 hexadecimal digits (4 bytes) that represent the network address and 12 hexadecimal digits (6 bytes) that represent the node address. A null string specifies the local computer.

port-name
Specifies an optional 16-bit number that represents the socket address. Values can range from 1 to 65,535. When no value is specified, the endpoint-mapping service selects a valid *port-name* value.

Example

```
[   uuid(12345678-4000-2006-0000-20000000001a),
    version(1.1),
    endpoint("ncacn_spx:[1000]")
] interface foo
```

Remarks

The **ncacn_spx** keyword identifies SPX as the protocol family for the endpoint.

The syntax of the SPX transport port string, like all port strings, is defined independently of the IDL specification. The compiler performs some syntax checking but does not guarantee that the endpoint specification is correct. Some errors may be reported at run time rather than during compilation.

When using the **ncacn_spx** transport, the server name is exactly the same as the Windows NT 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** transport. The following is a partial list of characters prohibited in Novell server names:

" * + . / : ; < = > ? [ ] \ |

The **ncacn_spx** transport is not supported by the version of NWLink supplied with MS Client 3.0.

Only the **ncacn_spx** and **ncacn_ip_tcp** protocols support cancels. For all other protocols, the cancel routines will return RPC_S_OK, but there will be no effect. Specifically:

RpcCancelThread will alert the specified thread, but will not interrupt a pending RPC.

RpcTestCancel will return RPC_S_OK if the current thread has been alerted.

RpcMgmtSetCancelTimeout has no visible effect.

RpcCancelThread, **RpcTestCancel**, and **RpcMgmtSetCancelTimeout** are only supported on Windows NT platforms; all other platforms return RPC_S_CANNOT_SUPPORT.

Note The network and node addresses that specify the network-address portion of the **ncacn_spx** endpoint specification are available from the **comcheck** utility. With Windows NT, the network address is stored in the registry and the node address is displayed by the command **net config rdr**.

See Also

endpoint, **IDL**, **ncacn_ip_tcp**, **ncacn_nb_nb**, **ncacn_nb_tcp**, **ncacn_np**, **ncalrpc**, **string_binding**

ncadg_ip_udp

endpoint("ncadg_ip_udp:*server-name*[*port-name*]")

server-name
Specifies the name or internet address for the server, or host, computer. The name is a character string. The internet address is a common internet address notation.

port-name
Specifies an optional 16-bit number. Values of less than 1024 are usually reserved. If no value is specified, the endpoint-mapping service selects a valid *port-name* value.

Example

```
[   uuid(12345678-4000-2006-0000-20000000001a),
    version(1.1),
    endpoint("ncadg_ip_udp:rainier[1404]")
] interface foo

endpoint("ncadg_ip_udp:128.10.2.30[1404]")
```

Remarks

The **ncadg_ip_udp** keyword identifies the datagram version of TCP/IP as the protocol family for the endpoint.

The datagram protocols (**ncadg_ipx** and **ncadg_ip_udp**) have the following limitations:

They do not support callbacks. Any functions using the callback attribute will fail.

They do not support the RPC security API (**RpcBindingSetAuthInfo**, **RpcImpersonateClient**, etc.).

The syntax of the TCP/IP transport port string, like all port strings, is defined independently of the IDL specification. The compiler performs some syntax checking but does not guarantee that the endpoint specification is correct. Some errors may be reported at run time rather than during compilation.

See Also

endpoint, **IDL**, **ncacn_nb_tcp**, **ncacn_np**, **ncacn_spx**, **ncalrpc**, **string_binding**

ncadg_ipx

endpoint("ncadg_ipx:*link-address***[***port-name***]")**

link-address
Specifies eight hexadecimal digits (4 bytes) that represent the network address and 12 hexadecimal digits (6 bytes) that represent the node address. A null string specifies the local computer.

port-name
Specifies an optional 16-bit number that represents the socket address. Values can range from 1 to 65535. When no value is specified, the endpoint-mapping service selects a valid *port-name* value.

Example

```
[   uuid(12345678-4000-2006-0000-20000000001a),
    version(1.1),
    endpoint("ncadg_ipx:[1000]")
] interface foo
```

Remarks

The **ncadg_ipx** keyword identifies IPX as the protocol family for the endpoint.

The syntax of the IPX transport port string, like all port strings, is defined independently of the IDL specification. The compiler performs some syntax checking but does not guarantee that the endpoint specification is correct. Some errors may be reported at run time rather than during compilation.

When using the **ncadg_ipx** transport, the server name is exactly the same as the Windows NT 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 **ncadg_ipx** transport. The following is a partial list of characters prohibited in Novell server names:

" * + . / : ; < = > ? [ ] \ |

The **ncadg_ipx** transport is supported by the version of NWLink supplied with MS Client 3.0.

The datagram protocols (**ncadg_ipx** and **ncadg_ip_udp**) have the following limitations:

They do not support callbacks. Any functions using the callback attribute will fail.

They do not support the RPC security API (**RpcBindingSetAuthInfo**, **RpcImpersonateClient**, etc.).

Only the **ncacn_spx** and **ncacn_ip_tcp** protocols support cancels. For all other protocols, the cancel routines will return RPC_S_OK, but there will be no effect. Specifically:

RpcCancelThread will alert the specified thread, but will not interrupt a pending RPC.

RpcTestCancel will return RPC_S_OK if the current thread has been alerted.

RpcMgmtSetCancelTimeout has no visible effect.

RpcCancelThread, **RpcTestCancel**, and **RpcMgmtSetCancelTimeout** are only supported on Windows NT platforms; all other platforms return RPC_S_CANNOT_SUPPORT.

Note The network and node addresses that specify the network-address portion of the **ncadg_ipx** endpoint specification are available from the **comcheck** utility. With Windows NT, the network address is stored in the registry and the node address is displayed by the command **net config rdr**.

See Also

endpoint, **IDL**, **ncacn_ip_tcp**, **ncacn_nb_nb**, **ncacn_nb_tcp**, **ncacn_np**, **ncalrpc**, **string_binding**

ncalrpc

endpoint("ncalrpc:[*port-name***]")**

port-name
Specifies a string that represents a Windows NT object name. The string should not contain any backslash (\) characters.

Example

```
[   uuid(12345678-4000-2006-0000-20000000001a),
    version(1.1),
    endpoint("ncalrpc:[myapplicationname]")
]
```

Remarks

The **ncalrpc** keyword identifies local interprocess communication as the protocol family for the endpoint. This keyword is one of the valid protocol family names that must be used with the **endpoint** attribute.

Note The **ncalrpc** endpoint is only supported by Windows NT systems. It is not supported by Windows 95.

The computer name must not be used with the **ncalrpc** keyword.

The syntax of the local interprocess-communication port string, like all port strings, is defined by the transport implementation and is independent of the IDL specification. The MIDL compiler performs limited syntax checking but does not guarantee that the endpoint specification is correct. Some classes of errors may be reported at run time rather than during compilation.

See Also

endpoint, **IDL**, **ncacn_ip_tcp**, **ncacn_nb_nb**, **ncacn_nb_tcp**, **ncacn_np**, **ncacn_spx**, **string_binding**

nocode

[nocode [**,** *ACF-interface-attributes*] **] interface** *interface-name*
{
[**include** *filename-list* **;**] **...**
[**typedef [***type-attribute-list***]** *typename***;**] **...**

[**[nocode** [**,** *ACF-function-attributes*] **]** *function-name* **(**
[*ACF-parameter-attributes* **]** *parameter-name* **;**
...
);
]
...
}

ACF-interface-attributes
Specifies a list of one or more attributes that apply to the interface as a whole. Valid attributes include either **auto_handle** or **implicit_handle** and either **code** or **nocode**. When two or more interface attributes are present, they must be separated by commas.

interface-name
Specifies the name of the interface. In DCE-compatibility mode, the interface name must match the name of the interface specified in the IDL file. When you use the MIDL compiler switch **/acf**, the interface name in the ACF and the interface name in the IDL file can be different.

filename-list
Specifies a list of one or more C-language header filenames, separated by commas. The full filename, including the extension, must be supplied.

type-attribute-list
Specifies a list of one or more attributes, separated by commas, that apply to the specified type. Valid type attributes include **allocate**.

typename
Specifies a type defined in the IDL file. Type attributes in the ACF can only be applied to types previously defined in the IDL file.

ACF-function-attributes
Specifies attributes that apply to the function as a whole, such as **comm_status**. Function attributes are enclosed in square brackets. Separate multiple function attributes with commas.

function-name
Specifies the name of the function as defined in the IDL file.

ACF-parameter-attributes
Specifies ACF attributes that apply to a parameter. Note that zero or more attributes can be applied to the parameter. Separate multiple parameter attributes with commas. ACF parameter attributes are enclosed in square brackets.

parameter-name
Specifies a parameter of the function as defined in the IDL file. Each parameter for the function must be specified in the same sequence and using the same name as defined in the IDL file.

Remarks

The **nocode** attribute can appear in the ACF header, or it can be applied to an individual function.

When the **nocode** attribute appears in the ACF header, client stub code is not generated for any remote function unless it has the **code** function attribute. You can override the **nocode** attribute in the header for an individual function by specifying the **code** attribute as a function attribute.

When the **nocode** attribute appears in the function's attribute list, no client stub code is generated for

the function.

Client stub code is not generated when:

- The ACF header includes the **nocode** attribute.
- The **nocode** attribute is applied to the function.
- The **local** attribute applies to the function in the interface file.

Either **code** or **nocode** can appear in a function's attribute list, and the one you choose can appear exactly once.

The **nocode** attribute is ignored when server stubs are generated. You cannot apply it when generating server stubs in DCE-compatibility mode.

See Also

ACF, **code**

non-encapsulated_union

typedef [switch_type(*switch-type-specifier***)** [**,** *type-attr-list*] **] union** [*tag*] **{**
    [case (*limited-expr-list***)]**
        [**[** *field-attribute-list* **]**] *type-specifier declarator-list* **;**
    [**[default]**
        [**[** *field-attribute-list* **]**] *type-specifier declarator-list* **;**
    ]
}

switch-type-specifier
Specifies an integer, character, or **enum** type or an identifier of such a type.

type-attr-list
Specifies zero or more other attributes that apply to the union type. Valid type attributes include **handle**, **transmit_as**; the pointer attribute **unique**, or **ptr**; and the usage attributes **context_handle** and **ignore**. Separate multiple attributes with commas.

tag
Specifies an optional tag.

limited-expr-list
Specifies one or more C-language expressions that are supported by MIDL. Almost all C expressions are supported. The MIDL compiler supports conditional expressions, logical expressions, relational expressions, and arithmetic expressions. MIDL does not allow function invocations in expressions and does not allow pre- and post-increment and -decrement operators.

field-attribute-list
Specifies zero or more field attributes that apply to the union member. Valid field attributes include **first_is**, **last_is**, **length_is**, **max_is**, **size_is**; the usage attributes **string**, **ignore**, and **context_handle**; the pointer attribute **ref**, **unique**, or **ptr**; and, for members that are themselves unions, the union attribute **switch_type**. Separate multiple field attributes with commas.

type-specifier
Specifies a **base_type**, **struct, union, enum** type, or type identifier. An optional storage specification can precede *type-specifier*.

declarator-list
Specifies one or more standard C declarators, such as identifiers, pointer declarators, and array declarators. (Function declarators and bit-field declarations are not allowed in unions that are transmitted in remote procedure calls. These declarators are allowed in unions that are not transmitted.) Separate multiple declarators with commas.

Remarks

The non-encapsulated union is indicated by the presence of the type attribute **switch_type** and the field attribute **switch_is**.

The shape of unions must be the same across platforms to ensure interconnectivity.

For more information, see **switch_is** and **switch_type**.

See Also

field_attributes, **IDL**, **union**

non-idempotent

Non-idempotent (at most once) indicates that the remote procedure call cannot be executed more than once because it will return a different value or change a state. **Non-idempotent** is the default for remote procedure calls.

All **non-idempotent** calls are executed by the server "at most once"; that is, not at all or exactly once. The protocol used to enforce this is the **callback** function. **Non-idempotent** requests require the server to perform a **callback** when it receives a request from a client about which it has no information. The server makes the **callback** request by issuing a remote procedure call to the client. When it receives the **callback**, the server's boot time and the client's sequence number are used as the basis of comparison to validate the request. If a match is made, the server executes the original request. Otherwise, the request is ignored.

A non-idempotent call ensures that the data is received and processed at most one time.

See Also

broadcast, **callback**, **idempotent**, **IDL**, **maybe**

notify

[notify] ... ;

Example

```
[notify] ProcedureFoo;
```

Remarks

The **notify** attribute instructs the MIDL compiler to generate a server stub that contains a call to a specially-named procedure on the server side of the application. This procedure is called the **notify** procedure. It is called after all the output arguments have been marshalled and any memory associated with the parameters is freed.

The **notify** attribute is useful to develop applications acquiring resources in the server manager routine. These resources are then freed in the **notify** procedure after the output parameters are fully marshalled.

The **notify** procedure name is the name of the remote procedure suffixed by **_notify**. The **notify** procedure does not require any parameters and does not return a result. A prototype of this procedure is also generated in the header file. For example, if the IDL file contains the following:

```
ProcedureFoo [in] short S);
```

Specify the following in the ACF for MIDL to generate the **notify** call:

```
[notify] ProcedureFoo();
```

The generated code will contain the following call to the **notify** procedure:

```
ProcedureFoo_notify();
```

The header file will contain a prototype:

```
void ProcedureFoo_notif(void);
```

See Also

ACF

object

[object, uuid(*string-uuid***)[,** *interface-attribute-list*] **]**
interface *interface-name* **:** *base-interface*

string-uuid
Specifies a UUID string generated by the **uuidgen** utility. You can enclose the UUID string in quotes when you use the MIDL compiler switch **/ms_ext**.

interface-attribute-list
Specifies other attributes that apply to the interface as a whole.

interface-name
Specifies the name of the interface.

base-interface
Specifies the name of an OLE interface from which this derived interface inherits member functions, status codes, and interface attributes. All OLE interfaces are derived from the **IUnknown** interface or some other OLE interface.

Remarks

The **object** interface attribute identifies a custom OLE interface. An interface attribute list that does not include the **object** attribute indicates a DCE RPC interface. An interface attribute list for an OLE interface must include the **uuid** attribute, but it cannot include the **version** attribute.

By default, compiling an OLE interface with the MIDL compiler generates the files needed to build a proxy DLL that contains the code to support the use of the custom OLE interface by both client applications and object servers. However, if the interface attribute list for an OLE interface specifies the **local** attribute, the MIDL compiler generates only the interface header file.

The MIDL compiler automatically generates an interface data type for an OLE interface. As an alternative, you can use **typedef** with the **interface** keyword to explicitly define an interface data type. The interface specification can then use the interface data type in function parameters and return values, **struct** and **union** members, and other type declarations. The following example illustrates the use of an automatically generated **IStream** data type:

```
[object, uuid (ABCDEFOO-1234-1234-5678-ABCDEF123456]
     interfaace IStream{
           typedef IStream * LPSTREAM;
}
}
```

In an OLE interface, all the interface member functions are assumed to be virtual functions. A virtual function has an implicit **this** pointer as the first parameter. The virtual function table contains an entry for each interface member function.

Files that use the **object** attribute must be compiled with the **/ms_ext** MIDL compiler switch.

See Also

IDL, **iid_is**, **local**, **/ms_ext**, **uuid**, **version**

optimize

optimize ("*optimization-options***")**

optimization-options
Specifies the method of marshalling data. Use either "s" for mixed-mode marshalling or "i" for interpreted marshalling.

Examples

```
optimize ("s") void FasterProcedure(...);
optimize ("i") void SmallerProcedure(...);
{
};
```

Remarks

The keyword **optimize** is used to fine-tune the level of gradation for marshalling data.

This version of RPC provides two methods for marshalling data: mixed-mode ("s") and interpreted ("i"). These methods correspond to the **/Os** and **/Oi** command-line switches. The interpreted method marshals data completely offline. While this can reduce the size of the stub considerably, performance can be affected.

If performance is a concern, the mixed-mode method can be the best approach. Mixed-mode allows the MIDL compiler to make the determination between which data will be marshalled inline and which will be marshalled by a call to an offline dynamic-link library. If many procedures use the same data types, a single procedure can be called repeatedly to marshal the data. In this way, data that is most suited to inline marshalling is processed inline while other data can be more efficiently marshalled offline.

Note that the **optimize** attribute can be used as an interface attribute or as an operation attribute. If it is used as an interface attribute, it sets the default for the entire interface, overriding command-line switches. If, however, it is used as an operation attribute, it affects only that operation, overriding command-line switches and the interface default.

See Also

ACF, **/Oi**, **/Os**

out

[**[***function-attribute-list***]**] *type-specifier* [*pointer-declarator*] *function-name***(**
[out [**,** *parameter-attribute-list*] **]** *type-specifier* [*declarator*]
, ...
);

function-attribute-list
Specifies zero or more attributes that apply to the function. Valid function attributes are **callback**, **local**; the pointer attribute **ref**, **unique**, or **ptr**; and the usage attributes **string**, **ignore**, and **context_handle**.

type-specifier
Specifies a **base_type**, **struct**, **union**, or **enum** type or type identifier. An optional storage specification can precede *type-specifier*.

pointer-declarator
Specifies zero or more pointer declarators. A pointer declarator is the same as the pointer declarator used in C; it is constructed from the ***** designator, modifiers such as **far**, and the qualifier **const**.

function-name
Specifies the name of the remote procedure.

parameter-attribute-list
Specifies zero or more attributes appropriate for a specified parameter type. Parameter attributes with the **out** attribute can also take the directional attribute **out**; the field attributes **first_is**, **last_is**, **length_is**, **max_is**, **size_is**, and **switch_type**; the pointer attribute **ref**, **unique**, or **ptr**; and the usage attributes **context_handle** and **string**. The usage attribute **ignore** cannot be used as a parameter attribute. Separate multiple attributes with commas.

declarator
Specifies standard C declarators, such as identifiers, pointer declarators, and array declarators. For more information, see **pointers** and **arrays**. The parameter declarator in the function declarator, such as the parameter name, is optional.

Example

```
void MyFunction([out] short * pcount);
```

Remarks

The **out** attribute identifies pointer parameters that are returned from the called procedure to the calling procedure (from the server to the client).

The **out** attribute indicates that a parameter that acts as a pointer and its associated data in memory are to be passed back from the called procedure to the calling procedure.

The **out** attribute must be a pointer. DCE IDL compilers require the presence of an explicit ***** as a pointer declarator in the parameter declaration. Microsoft IDL offers an extension that drops this requirement and allows an array or a previously defined pointer type.

A related attribute, **in**, indicates that the parameter is passed from the calling procedure to the called procedure. The **in** and **out** attributes specify the direction in which the parameters are passed. A parameter can be defined as **in**-only, **out**-only, or **in, out**.

An **out**-only parameter is assumed to be undefined when the remote procedure is called and memory for the object is allocated by the server. Since top-level pointer/parameters must always point to valid storage, and therefore cannot be null, **out** cannot be applied to top-level **unique** or **ptr** pointers. Parameters that are **ref** pointers must be either **in** or **in, out** parameters.

See Also

in, ref

out_of_line

The DCE IDL keyword **out_of_line** is not supported in Microsoft RPC.

See Also

IDL

pipe

The DCE IDL keyword **pipe** is not supported in Microsoft RPC.

See Also

IDL

pointer_default

pointer_default (ptr | ref | unique)

Example

```
[uuid(6B29FC40-CA47-1067-B31D-00DD010662DA),
version(3.3),
pointer_default(unique)]
interface dictionary
```

Remarks

The **pointer_default** attribute specifies the default pointer attribute for all pointers except top-level pointers that appear in parameter lists. This includes embedded pointers – pointers that appear in structures, unions, and arrays. The **pointer_default** attribute can also apply to pointers returned by functions.

MIDL generates an error during compilation when you do not supply a pointer attribute in an interface that includes embedded pointers.

The default does not apply to pointers that appear as top-level parameters, such as individual pointers used as function parameters. A **pointer** attribute must be supplied for these pointers. The default is always overridden when a **pointer** attribute is supplied. If all pointers are supplied with **pointer** attributes, the default attribute is ignored.

The **pointer_default** attribute is an optional attribute in the IDL file. The **pointer_default** attribute is required only in the interface header when:

- A parameter with more than one asterisk (*) appears in a function.
- A structure member or union arm with a pointer declarator does not have a pointer attribute.
- A function returns a pointer type and does not have a pointer attribute as a function attribute.

If the **pointer_default** attribute appears in the interface header and is not required, it is ignored.

See Also

interface, **pointers**, **ptr**, **ref**, **unique**

pointers

MIDL supports three kinds of pointers: reference pointers, unique pointers, and full pointers. These pointers are specified by the pointer attributes **ref**, **unique,** and **ptr**.

A pointer attribute can be applied as a type attribute; as a field attribute that applies to a structure member, union member, or parameter; or as a function attribute that applies to the function return type. The pointer attribute can also appear with the **pointer_default** keyword.

To allow a pointer parameter to change in value during a remote function, you must provide another level of indirection by supplying multiple pointer declarators. The explicit pointer attribute applied to the parameter affects only the rightmost pointer declarator for the parameter. When there are multiple pointer declarators in a parameter declaration, the other declarators default to the pointer attribute specified by the **pointer_default** attribute. To apply different pointer attributes to multiple pointer declarators, you must define intermediate types that specify the explicit pointer attributes.

Default Pointer-Attribute Values

When no pointer attribute is associated with a pointer that is a parameter, the pointer is assumed to be a **ref** pointer.

When no pointer attribute is associated with a pointer that is a member of a structure or union, the MIDL compiler assigns pointer attributes using the following priority rules (1 is highest):

1. Attributes explicitly applied to the pointer type
2. Attributes explicitly applied to the pointer parameter or member
3. **Pointer_default** attribute in the IDL file that defines the type
4. **Pointer_default** attribute in the IDL file that imports the type
5. **Ptr** (DCE-compatibility mode); **unique** (Microsoft-extensions mode)

When the IDL file is compiled using Microsoft-extensions (**/ms_ext**) mode, imported files can inherit pointer attributes from importing files. For more information, see **import**.

Function Return Types

A pointer returned by a function must be a **unique** pointer or a full pointer. The MIDL compiler reports an error if a function result is a reference pointer, either explicitly or by default. The returned pointer always indicates new storage.

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

Pointer Attributes in Type Definitions

When you specify a pointer attribute at the top level of a **typedef** statement, the specified attribute is applied to the pointer declarator, as expected. When no pointer attribute is specified, pointer declarators at the top level of a **typedef** statement inherits the pointer attribute type when used.

DCE IDL does not allow the same pointer attribute to be explicitly applied twice – for example, in both the **typedef** declaration and the parameter attribute list. When you use the Microsoft-extensions mode of the MIDL compiler (**/ms_ext**), this constraint is relaxed.

See Also

allocate, **IDL**, **import**, **/ms_ext**, **pointer_default**, **ptr**, **ref**, **unique**

pragma

#pragma midl_echo("*string***")**
#pragma *token-sequence*
#pragma pack *(n)*
#pragma pack *([***push***] [, id] [, n})*
#pragma pack *([***pop***] [, id] [, n})*

string
Specifies a string that is inserted into the generated header file. The quotation marks are removed during the insertion process.

token-sequence
Specifies a sequence of tokens that are inserted into the generated header file as part of a **#pragma** directive without processing by the MIDL compiler.

n
Specifies the current pack size. Valid values are 1, 2, 4, 8, and 16.

id
Specifies the user id.

Examples

```
/* IDL file */
#pragma midl_echo("#define UNICODE")
cpp_quote("#define __DELAYED_PREPROCESSING__ 1")
#pragma hdrstop
#pragma check_pointer(on)

/* generated header file */
#define UNICODE
#define __DELAYED_PREPROCESSING__ 1
#pragma hdrstop
#pragma check_pointer(on)
```

Remarks

The **#pragma midl_echo** directive instructs MIDL to emit the specified string, without the quote characters, into the generated header file.

C-language preprocessing directives that appear in the IDL file are processed by the C compiler's preprocessor. The **#define** directives in the IDL file are available during MIDL compilation, although not to the C compiler.

For example, when the preprocessor encounters the directive "#define WINDOWS 4", the preprocessor replaces all occurrences of "WINDOWS" in the IDL file with "4". The symbol "WINDOWS" is not available at C-compile time.

To allow the C-preprocessor macro definitions to pass through the MIDL compiler to the C compiler, use the **#pragma midl_echo** or **cpp_quote** directive. These directives instruct the MIDL compiler to generate a header file that contains the parameter string with the quotation marks removed. The **#pragma midl_echo** and **cpp_quote** directives are equivalent.

The **#pragma pack** directive is used by the MIDL compiler to control the packing of structures. It overrides the **/Zp** command-line switch. The pack (*n*) option sets the current pack size to a specific value: 1, 2, 4, 8, or 16. The pack (push) and pack (pop) options have the following characteristics:

- The compiler maintains a packing stack. The elements of the packing stack include a pack size and an optional *id*. The stack is limited only by available memory with the current pack size at the top of

the stack.

- Pack (push) results in the current pack size pushed onto the packing stack. The stack is limited by available memory.
- Pack (push, *n*) is the same as pack (push) followed by pack (*n*).
- Pack (push, *id*) also pushes *id* onto the packing stack along with the pack size.
- Pack (push, *id*, *n*) is the same as pack (push, *id*) followed by pack (*n*).
- Pack (pop) results in popping the packing stack. Unbalanced pops cause warnings and set the current pack size to the command-line value.
- If pack (pop, *id*, *n*) is specified, then *n* is ignored.

The MIDL compiler places the strings specified in the **cpp_quote** and **pragma** directives in the header file in the sequence in which they are specified in the IDL file and relative to other interface components in the IDL file. The strings should usually appear in the interface-body section of the IDL file after all **import** operations.

The MIDL compiler does not attempt to process **#pragma** directives that do not start with the prefix "midl_." Other **#pragma** directives in the IDL file are passed into the generated header file without changes.

See Also

cpp_quote, **IDL**, **/Zp**

ptr

pointer_default(ptr)

typedef [ptr [**,** *type-attribute-list*] **]** *type-specifier declarator-list***;**

typedef *struct-or-union-declarator* **{**
[ptr [**,** *field-attribute-list*] **]** *type-specifier declarator-list***;**
...}

[ptr [**,** *function-attribute-list*] **]** *type-specifier ptr-decl function-name***(**
[**[** *parameter-attribute-list* **]**] *type-specifier* [*declarator*]
, ...
);
[**[** *function-attribute-list* **]**] *type-specifier* [*ptr-decl*] *function-name***(**
[ptr [**,** *parameter-attribute-list*] **]** *type-specifier* [*declarator*]
, ...
);

type-attribute-list
Specifies one or more attributes that apply to the type. Valid type attributes include **handle**, **switch_type**, **transmit_as**; the pointer attribute **ref**, **unique**, or **ptr**; and the usage attributes **context_handle**, **string**, and **ignore**. Separate multiple attributes with commas.

type-specifier
Specifies a **base_type**, **struct**, **union,** or **enum** type or type identifier. An optional storage specification can precede *type-specifier*.

declarator and *declarator-list*
Specifies standard C declarators, such as identifiers, pointer declarators, and array declarators. For more information, see **pointers** and **arrays**. The *declarator-list* consists of one or more declarators separated by commas. The parameter-name identifier in the function declarator is optional.

struct-or-union-declarator
Specifies a MIDL **struct** or **union** declarator. *field-attribute-list*
Specifies zero or more field attributes that apply to the structure or union member or function parameter. Valid field attributes include **first_is**, **last_is**, **length_is**, **max_is**, **size_is**; the usage attributes **string**, **ignore**, and **context_handle**; the pointer attribute **ref**, **unique**, or **ptr**; and the union attribute **switch_type**. Separate multiple field attributes with commas.

function-attribute-list
Specifies zero or more attributes that apply to the function. Valid function attributes are **callback**, **local**; the pointer attribute **ref**, **unique**, or **ptr**; and the usage attributes **string**, **ignore**, and **context_handle**.

ptr-decl
Specifies at least one pointer declarator to which the **ptr** attribute applies. A pointer declarator is the same as the pointer declarator used in C; it is constructed from the ***** designator, modifiers such as **far,** and the qualifier **const**.

function-name
Specifies the name of the remote procedure.

parameter-attribute-list
Consists of zero or more attributes appropriate for the specified parameter type. Parameter attributes can take the directional attributes **in** and **out**; the field attributes **first_is**, **last_is**, **length_is**, **max_is**, **size_is**, and **switch_type**; the pointer attribute **ref**, **unique**, or **ptr**; and the usage attributes **context_handle** and **string**. The usage attribute **ignore** cannot be used as a parameter attribute. Separate multiple attributes with commas.

Examples

```
pointer_default(ptr)

typedef [ptr, string] unsigned char * MY_STRING_TYPE;

[ptr] char * MyFunction([in, out, unique] long * plNumber);
```

Remarks

The **ptr** attribute designates a pointer as a full pointer. The full pointer approaches the full functionality of the C-language pointer. The full pointer can have the value NULL and can change during the call from null to non-null. Storage pointed to by full pointers can be reached by other names in the application supporting aliasing and cycles. This functionality requires more overhead during a remote procedure call to identify the data referred to by the pointer, determine whether the value is NULL, and to discover if two pointers point to the same data.

Use full pointers for:

- Remote return values.
- Double pointers, when the size of an output parameter is not known.
- NULL pointers.

Full (and unique) pointers cannot be used to describe the size of an array or union because these pointers can have the value NULL. This restriction by MIDL prevents an error that can result when a NULL value is used as the size.

Reference and unique pointers are assumed to cause no aliasing of data. A directed graph obtained by starting from a unique or reference pointer and following only unique or reference pointers contains neither reconvergence nor cycles.

To avoid aliasing, all pointer values should be obtained from an input pointer of the same class of pointer. If more than one pointer points to the same memory location, all such pointers must be full pointers.

In some cases, full and unique pointers can be mixed. A full pointer can be assigned the value of a unique pointer, as long as the assignment does not violate the restrictions on changing the value of a unique pointer. However, when you assign a unique pointer the value of a full pointer, you may cause aliasing.

Mixing full and unique pointers can cause aliasing, as demonstrated in the following example:

```
typedef struct {
    [ptr] short * pdata;          // full pointer
} GRAPH_NODE_TYPE;

typedef struct {
    [unique] graph_node * left;   // unique pointer
    [unique] graph_node * right;  // unique pointer
} TREE_NODE_TYPE;

// application code:
short a = 5;
TREE_NODE_TYPE * t;
GRAPH_NODE_TYPE g, h;

g.pdata = h.pdata = &a;
t->left = &g;
t->right = &h;
```

```
// t->left->pdata == t->right->pdata == &a
```

Although "t->left" and "t->right" point to unique memory locations, "t->left->pdata" and "t->right->pdata" are aliased. For this reason, aliasing-support algorithms must follow all pointers (including unique and reference pointers) that may eventually reach a full pointer.

See Also

IDL, **pointer_default**, **pointers**, **ref**, **unique**

ref

pointer_default(ref)

typedef [ref [**,** *type-attribute-list*] **]** *type-specifier declarator-list***;**

typedef *struct-or-union-declarator* **{**
[ref [**,** *field-attribute-list*] **]** *type-specifier declarator-list***;**
...}

[**[** *function-attribute-list* **]**] *type-specifier* [*ptr-decl*] *function-name***(**
[ref [**,** *parameter-attribute-list*] **]** *type-specifier* [*declarator*]
, ...
);

type-attribute-list
Specifies one or more attributes that apply to the type. Valid type attributes include **handle**, **switch_type**, **transmit_as**; the pointer attributes **ref**, **unique**, or **ptr**; and the usage attributes **context_handle**, **string**, and **ignore**. Separate multiple attributes with commas.

type-specifier
Specifies a **base_type**, **struct**, **union**, or **enum** type or type identifier. An optional storage specification can precede *type-specifier*.

declarator and *declarator-list*
Specifies standard C declarators, such as identifiers, pointer declarators, and array declarators. For more information, see **pointers** and **arrays**. The *declarator-list* consists of one or more declarators separated by commas. The parameter-name identifier in the function declarator is optional.

struct-or-union-declarator
Specifies a MIDL **struct** or **union** declarator.

field-attribute-list
Specifies zero or more field attributes that apply to the structure, union member, or function parameter. Valid field attributes include **first_is**, **last_is**, **length_is**, **max_is**, **size_is**; the usage attributes **string**, **ignore**, and **context_handle**; the pointer attribute **ref**, **unique**, or **ptr**; and the union attribute **switch_type**. Separate multiple field attributes with commas.

function-attribute-list
Specifies zero or more attributes that apply to the function. Valid function attributes are **callback**, **local**; the pointer attribute **ref**, **unique**, or **ptr**; and the usage attributes **string**, **ignore**, and **context_handle**.

ptr-decl
Specifies at least one pointer declarator to which the **ref** attribute applies. A pointer declarator is the same as the pointer declarator used in C; it is constructed from the ***** designator, modifiers such as **far**, and the qualifier **const**.

function-name
Specifies the name of the remote procedure.

parameter-attribute-list
Consists of zero or more attributes appropriate for the specified parameter type. Parameter attributes can take the directional attributes **in** and **out**; the field attributes **first_is**, **last_is**, **length_is**, **max_is**, **size_is**, and **switch_type**; the pointer attribute **ref**, **unique**, or **ptr**; and the usage attributes **context_handle** and **string**. The usage attribute **ignore** cannot be used as a parameter attribute. Separate multiple attributes with commas.

Example

```
[unique] char * GetFirstName(
```

```
    [in, ref] char * pszFullName
);
```

Remarks

The **ref** attribute identifies a reference pointer. It is used simply to represent a level of indirection.

A pointer attribute can be applied as a type attribute, as a field attribute that applies to a structure member, union member, or parameter; or as a function attribute that applies to the function return type. The pointer attribute can also appear with the **pointer_default** keyword.

A reference pointer has the following characteristics:

- Always points to valid storage; never has the value NULL. A reference pointer can always be dereferenced.
- Never changes during a call. A reference pointer always points to the same storage on the client before and after the call.
- Does not allocate new memory on the client. Data returned from the server is written into existing storage specified by the value of the reference pointer before the call.
- Does not cause aliasing. Storage pointed to by a reference pointer cannot be reached from any other name in the function.

A reference pointer cannot be used as the type of a pointer returned by a function.

If no attribute is specified for a top-level pointer parameter, it is treated as a reference pointer.

See Also

pointers, **ptr**, **unique**

represent_as

typedef [represent_as(*repr-type***)** [**,** *type-attribute-list*] **]**
*named-type***;**

void __RPC_USER *named-type***_from_local (**
repr-type **__RPC_FAR * ,**
named-type **__RPC_FAR * __RPC_FAR *);**

void __RPC_USER *named-type***_to_local (**
named-type **__RPC_FAR * ,**
repr-type **__RPC_FAR *);**

void __RPC_USER *named-type***_free_inst (**
named-type **__RPC_FAR *);**

void __RPC_USER *named-type***_free_local (**
*repr-type***__RPC_FAR *);**

named-type
Specifies the named transfer data type that is transferred between client and server.

type-attribute-list
Specifies one or more attributes that apply to the type. Valid type attributes include **handle**, **switch_type**; the pointer attribute **ref**, **unique**, or **ptr**; and the usage attributes **string** and **ignore**. Separate multiple attributes with commas.

repr-type
Specifies the represented local type in the target language that is presented to the client and server applications.

Examples

```
typedef struct _TREE_NODE_TYPE {
    unsigned short data;
    struct _TREE_NODE_TYPE * left;
    struct _TREE_NODE_TYPE * right;
} TREE_NODE_TYPE;

typedef [represent_as(TREE_NAMED_TYPE)] TREE_TYPE; /*in ACF */

void __RPC_USER TREE_TYPE_from_local(
     TREE_TYPE __RPC_FAR * ,
     TREE_NAMED_TYPE __RPC_FAR * __RPC_FAR *);

void __RPC_USER TREE_TYPE_to_local (
     TREE_NAMED_TYPE __RPC_FAR *,
     TREE_TYPE __RPC_FAR *);

void __RPC_USER TREE_TYPE_free_inst(
     TREE_NAMED_TYPE __RPC_FAR *);

void __RPC_USER TREE_TYPE_free_local(
     TREE_TYPE __RPC_FAR *);
```

Remarks

The **represent_as** attribute associates a named local type in the target language *repr-type* with a transfer type *named-type* that is transferred between client and server. You must supply routines that

convert between the local and the transfer types and that free memory used to hold the converted data. The **represent_as** attribute instructs the stubs to call the user-supplied conversion routines.

The transferred type *named-type* must resolve to a MIDL base type, predefined type, or to a type identifier. For more information, see **base_types**.

The user must supply the following routines:

Routine name	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)

The client stub calls *named-type*_**from_local** to allocate space for the transmitted type and to translate the data from the local type to the network type. The server stub allocates space for the original data type and calls *named-type*_**to_local** to translate the data from the network type to the local type.

Upon return from the application code, the client and server stubs call *named-type*_**free_inst** to deallocate the storage for network type. The client stub calls *named-type*_**free_local** to deallocate storage returned by the routine.

The following types cannot have a **represent_as** attribute:

- Pipe types.
- Types used as the base type in a pipe definition.
- Conformant, varying, or conformant varying arrays.
- Structures in which the last member is a conformant array (a conformant structure).
- Pointers or types that contain a pointer.
- Predefined types **handle_t**, **void**.

See Also

ACF, **arrays**, **base_types**, **typedef**

shape

The DCE IDL keyword **shape** is not supported in Microsoft RPC.

See Also

IDL

short

The **short** keyword designates a 16-bit integer. The **short** keyword can be preceded by either the keyword **signed** or the keyword **unsigned**. The **int** keyword is optional and can be omitted. To the MIDL compiler, a short integer is signed by default and is synonymous with **signed short int.**

The **short** integer type is one of the base types of the IDL language. The **short** integer type can appear as a type specifier in **const** declarations, **typedef** declarations, general declarations, and function declarators (as a function-return-type specifier and a parameter-type specifier). For the context in which type specifiers appear, see **IDL**.

See Also

base_types, **IDL**, **int**, **long**, **small**

signed

The **signed** keyword indicates that the most significant bit of an integer variable represents a sign bit rather than a data bit. This keyword is optional and can be used with any of the character and integer types **char**, **wchar_t**, **long**, **short**, and **small**.

When you use the MIDL compiler switch **char**, character and integer types that appear in the IDL file without explicit sign keywords can appear with the **signed** or **unsigned** keywords in the generated header file. To avoid confusion, explicitly specify the sign of the integer and character types.

See Also

base_types, **IDL**, **int**, **long**, **short**, **small**, **unsigned**

size_is

[size_is(*limited-expression***)]**

limited-expression
Specifies a C-language expression that evaluates to an integer that represents the maximum allocation size, in elements, of the leftmost dimension of an array. The MIDL compiler supports conditional expressions, logical expressions, relational expressions, and arithmetic expressions. MIDL does not allow function invocations in expressions and does not allow pre- and post-increment and -decrement operators.

Examples

```
void Proc1(
     [in, short] m;
     [in, size_is(m)] short a[]);  /* if m = 10, a[10] */
void Proc2(
     [in, short] m;
     [in, size_is(m)] short b[][20]);  /* if m = 10, b[10][20] */
```

Remarks

The **size_is** attribute is used to specify an expression or identifier that designates the maximum size, in elements, of an array dimension. The specified value must be non-negative. You can use either **size_is** or **max_is** (but not both in the same attribute list) to specify the size of an array whose upper bounds are determined at run time. Note, however, that the **size_is** attribute cannot be used on array dimensions that are fixed. The **max_is** attribute specifies the maximum valid array index. As a result, specifying **size_is(***n***)** is equivalent to specifying **max_is**(*n*-1).

An **in** or **in, out** conformant-array parameter with the **string** attribute need not have the **size_is** or **max_is** attribute. In this case, the size of the allocation is determined from the null terminator of the input string. All other conformant arrays with the **string** attribute must have a **size_is** or **max_is** attribute.

In an OLE interface (when the **object** interface attribute is specified), you can use the **size_is** attribute with a **void *** type. In this case, the declarator is treated as a pointer to an array of bytes. This usage requires using the MIDL compiler with the **/ms_ext** switch.

See Also

arrays, **field_attributes**, **first_is**, **IDL**, **last_is**, **length_is**, **max_is**, **min_is**

small

The **small** keyword designates an 8-bit integer number. The **small** keyword can be preceded by either the keyword **signed** or the keyword **unsigned**. The **int** keyword is optional and can be omitted. To the MIDL compiler, a small integer is **signed** by default and is synonymous with **signed small int**.

The **small** integer type is one of the base types of the IDL language. The **small** integer type can appear as a type specifier in **const** declarations, **typedef** declarations, general declarations, and function declarators (as a function-return-type specifier and as a parameter-type specifier). For the context in which type specifiers appear, see **IDL**.

The sign of the **small** type can be modified by the MIDL compiler switch **/char**. To avoid confusion, specify the sign of the integer type with the keywords **signed** and **unsigned**.

See Also

/char, **int**, **long**, **short**

string

typedef [string [**,** *type-attribute-list*] **]** *type-specifier declarator-list***;**

typedef *struct-or-union-declarator* **{**
[string [**,** *field-attribute-list*] **]** *type-specifier declarator-list***;**
...}

[string [**,** *function-attribute-list*] **]** *type-specifier ptr-decl function-name***(**
[**[** *parameter-attribute-list* **]**] *type-specifier* [*declarator*]
, **...**
);
[**[** *function-attribute-list* **]**] *type-specifier* [*ptr-decl*] *function-name***(**
[string [**,** *parameter-attribute-list*] **]** *type-specifier* [*declarator*]
, **...**
);

type-attribute-list
Specifies one or more attributes that apply to a type. Valid type attributes include **handle**, **switch_type**, **transmit_as**; the pointer attribute **ref**, **unique**, or **ptr**; and the usage attributes **context_handle**, **string**, and **ignore**. Separate multiple attributes with commas.

type-specifier
Specifies a **base_type**, **struct**, **union**, or **enum** type or type identifier. An optional storage specification can precede *type-specifier*.

declarator and *declarator-list*
Specify standard C declarators, such as identifiers, pointer declarators, and array declarators. For more information, see **pointers** and **arrays**. The *declarator-list* consists of one or more declarators separated by commas. The parameter-name identifier in the function declarator is optional.

struct-or-union-declarator
Specifies a MIDL **struct** or **union** declarator.

field-attribute-list
Specifies zero or more field attributes that apply to the structure, union member, or function parameter. Valid field attributes include **first_is**, **last_is**, **length_is**, **max_is**, **size_is**; the usage attributes **string**, **ignore**, and **context_handle**, the pointer attribute **ref**, **unique**, or **ptr**, and the union attribute **switch_type**. Separate multiple field attributes with commas.

function-attribute-list
Specifies zero or more attributes that apply to the function. Valid function attributes are **callback**, **local**; the pointer attribute **ref**, **unique**, or **ptr**; and the usage attributes **string**, **ignore**, and **context_handle**.

ptr-decl
Specifies an optional pointer declarator to which the **string** attribute applies. A pointer declarator is the same as the pointer declarator used in C; it is constructed from the ***** designator, modifiers such as **far**, and the qualifier **const**.

function-name
Specifies the name of the remote procedure.

parameter-attribute-list
Consists of zero or more attributes appropriate for the specified parameter type. Parameter attributes can take the directional attributes **in** and **out**; the field attributes **first_is**, **last_is**, **length_is**, **max_is**, **size_is**, and **switch_type**; the pointer attribute **ref**, **unique**, or **ptr**; and the usage attributes **context_handle** and **string**. The usage attribute **ignore** cannot be used as a parameter attribute. Separate multiple attributes with commas.

Example

```
/* a string type that can hold up to 80 characters */
typedef [string] char line[81];

void Proc1([in, string] char * pszName);
```

Remarks

The **string** attribute indicates that the one-dimensional **char**, **wchar_t**, **byte** (or equivalent) array or the pointer to such an array must be treated as a string.

The string can also be an array (or a pointer to an array) of constructs whose fields are all of the type "byte."

If the **string** attribute is used with an array whose bounds are determined at run time, you must also specify a **size_is** or **max_is** attribute.

The **string** attribute cannot be used with attributes that specify the range of transmitted elements, such as **first_is**, **last_is**, and **length_is**.

When used on multidimensional arrays, the **string** attribute applies to the rightmost array.

To define a counted string, do not use the **string** attribute. Use a character array or character-based pointer such as the following:

```
typedef struct {
    unsigned short size;
    unsigned short length;
    [size_is(size), length_is(length)] char string[*];
} counted_string;
```

The **string** attribute specifies that the stub should use a language-supplied method to determine the length of strings.

When declaring strings in C, you must allocate space for an extra character that marks the end of the string.

See Also

arrays, **char**, **wchar_t**

struct

struct [*struct-tag*] **{**
    [**[** *field-attribute-list* **]**] *type-specifier declarator-list*;
    ...
}

struct-tag
Specifies an optional tag for the structure.

field-attribute-list
Specifies zero or more field attributes that apply to the structure member. Valid field attributes include **first_is**, **last_is**, **length_is**, **max_is**, **size_is**; the usage attributes **string**, **ignore**, and **context_handle**; the pointer attribute **ref**, **unique**, or **ptr**; and the union attribute **switch_type**. Separate multiple field attributes with commas.

type-specifier
Specifies a **base_type**, **struct, union,** or **enum** type or type identifier. An optional storage specification can precede *type-specifier*.

declarator-list
Specifies one or more standard C declarators, such as identifiers, pointer declarators, and array declarators. (Function declarators and bit-field declarations are not allowed in structures that are transmitted in remote procedure calls. These declarators are allowed in structures that are not transmitted.) Separate multiple declarators with commas.

Example

```
typedef struct _PITCHER_RECORD_TYPE {
    short flag;
    [switch_is(flag)] union PITCHER_STATISTICS_TYPE p;
} PITCHER_RECORD_TYPE;
```

Remarks

The **struct** keyword is used in a structure type specifier. The IDL structure type specifier differs from the standard C type specifier in the following ways:

- Each structure member can be associated with optional field attributes that describe characteristics of that structure member for the purposes of a remote procedure call.
- Bit fields and function declarators are not allowed in structures that are used in remote procedure calls. These standard C declarator constructs can be used only if the structure is not transmitted on the network.

The shape of structures must be the same across platforms to ensure interconnectivity.

See Also

arrays, **base_types**, **/c_ext**, **IDL**, **/ms_ext**, **pointers**

switch

switch (*switch-type switch-name***)**

switch-type
Specifies an **int**, **char**, **enum** type, or an identifier that resolves to one of these types.

switch-name
Specifies the name of the variable of type *switch-type* that acts as the union discriminant.

Examples

```
typedef union _S1_TYPE switch (long l1) U1_TYPE {
    case 1024:
        float f1;
    case 2048:
        double d2;
} S1_TYPE;

/* in generated header file */
typedef struct _S1_TYPE {
    long l1;
    union {
        float f1;
        double d2;
    } U1_TYPE;
} S1_TYPE;
```

Remarks

The **switch** keyword selects the discriminant for an **encapsulated_union**.

See Also

IDL, **non-encapsulated_union**, **switch_is**, **switch_type**, **union**

switch_is

typedef struct [*struct-tag*] **{**
    [switch_is(*limited-expr***)** [**,** *field-attr-list*] **]** *union-type-specifier declarator***;**
    ...
}

[**[***function-attribute-list***]**] *type-specifier* [*pointer-declarator*] *function-name***(**
    [switch_is(*limited-expr***)** [**,** *param-attr-list*] **]** *union-type* [*declarator*]
    , ...
);

struct-tag
Specifies an optional tag for a structure.

limited-expr
Specifies a C-language expression supported by MIDL. Almost all C-language expressions are supported. The MIDL compiler supports conditional expressions, logical expressions, relational expressions, and arithmetic expressions. MIDL does not allow function invocations in expressions and does not allow pre- and post-increment and -decrement operators.

field-attr-list
Specifies zero or more field attributes that apply to a union member. Valid field attributes include **first_is**, **last_is**, **length_is**, **max_is**, **size_is**; the usage attributes **string**, **ignore**, and **context_handle**; the pointer attribute **ref**, **unique**, or **ptr**; and for members that are themselves unions, the union attribute **switch_type**. Separate multiple field attributes with commas.

union-type-specifier
Specifies the **union** type identifier. An optional storage specification can precede *type-specifier*.

declarator and *declarator-list*
Specifies a standard C declarator, such as an identifier, pointer declarator, and array declarator. (Function declarators and bit-field declarations are not allowed in unions that are transmitted in remote procedure calls. These declarators are allowed in unions that are not transmitted.) Separate multiple declarators with commas.

function-attribute-list
Specifies zero or more attributes that apply to the function. Valid function attributes are **callback**, **local**; the pointer attribute **ref**, **unique**, or **ptr**; and the usage attributes **string**, **ignore**, and **context_handle**.

type-specifier
Specifies a **base_type**, **struct**, **union**, **enum** type, or type identifier. An optional storage specification can precede *type-specifier*.

pointer-declarator
Specifies zero or more pointer declarators. A pointer declarator is the same as the pointer declarator used in C; it is constructed from the ***** designator, modifiers such as **far**, and the qualifier **const**.

function-name
Specifies the name of the remote procedure.

param-attr-list
Specifies zero or more attributes appropriate for the specified parameter type. Parameter attributes can take the directional attributes **in** and **out**, the field attributes **first_is**, **last_is**, **length_is**, **max_is**, **size_is**, and **switch_type**; the pointer attribute **ref**, **unique**, or **ptr**; and the usage attributes **context_handle** and **string**. The usage attribute **ignore** cannot be used as a parameter attribute. Separate multiple attributes with commas.

union-type
Identifies the **union** type specifier.

Examples

```
typedef [switch_type(short)] union _WILLIE_UNION_TYPE {
    [case(24)]
        float fMays;
    [case(25)]
        double dMcCovey;
    [default]
        ;
} WILLIE_UNION_TYPE;

typedef struct _WINNER_TYPE {
    [switch_is(sUniformNumber)] union WILLIE_UNION_TYPE w;
    short sUniformNumber;
} WINNER_TYPE;
```

Remarks

The **switch_is** attribute specifies the expression or identifier acting as the union discriminant that selects the union member. The discriminant associated with the **switch_is** attribute must be defined at the same logical level as the union:

- When the union is a parameter, the union discriminant must be another parameter.
- When the union is a field of a structure, the discriminant must be another field of the same structure.

The sequence in a structure or a function parameter list is not significant. The union can either precede or follow the discriminant.

The **switch_is** attribute can appear as a field attribute or as a parameter attribute.

See Also

encapsulated_union, **non-encapsulated_union**, **switch_type**, **union**

switch_type

switch_type(*switch-type-specifier***)**

switch-type-specifier
Specifies an integer, character, boolean, or **enum** type, or an identifier of such a type.

Examples

```
typedef [switch_type(short)] union _WILLIE_UNION_TYPE {
    [case(24)]
        float fMays;
    [case(25)]
        double dMcCovey;
    [default]
        ;
} WILLIE_UNION_TYPE;

typedef struct _WINNER_TYPE {
    [switch_is(sUniformNumber)] union WILLIE_UNION_TYPE w;
    short sUniformNumber;
} WINNER_TYPE;
```

Remarks

The **switch_type** attribute identifies the type of the variable used as the union discriminant. The switch type can be an integer, character, boolean, or **enum** type.

The **switch_is** attribute specifies the name of the parameter that is the union discriminant. The **switch_is** attribute applies to parameters or members of structures or unions.

The union and its discriminant must be specified at the same logical level. When the union is a parameter, the union discriminant must be another parameter. When the union is a field of a structure, the discriminant must be another field of the structure at the same level as the union field.

See Also

encapsulated_union, **IDL**, **non-encapsulated_union**, **switch_is**, **union**

transmit_as

typedef [transmit_as(*xmit-type***)** [**,** *type-attribute-list*] **]**
*type-specifier declarator-list***;**

void __RPC_USER *type-id***_to_xmit (**
type-id **__RPC_FAR *,**
xmit-type **__RPC_FAR * __RPC_FAR *);**

void __RPC_USER *type-id***_from_xmit (**
xmit-type **__RPC_FAR *,**
type-id **__RPC_FAR *);**

void __RPC_USER *type-id***_free_inst (**
type-id **__RPC_FAR *);**

void __RPC_USER *type-id***_free_xmit (**
*xmit-type***__RPC_FAR *);**

xmit-type
Specifies the data type that is transmitted between client and server.

type-attribute-list
Specifies one or more attributes that apply to the type. Valid type attributes include **handle**, **switch_type**; the pointer attribute **ref**, **unique**, or **ptr**; and the usage attributes **string** and **ignore**. Separate multiple attributes with commas.

type-specifier
Specifies a **base_type**, **struct**, **union**, **enum** type, or type identifier. An optional storage specification can precede *type-specifier*.

declarator-list
Specifies standard C declarators, such as identifiers, pointer declarators, and array declarators. For more information, see **pointers** and **arrays**. The *declarator-list* consists of one or more declarators separated by commas. The parameter declarator in the function declarator, such as the parameter name, is optional.

type-id
Specifies the name of the data type that is presented to the client and server applications.

Examples

```
typedef struct _TREE_NODE_TYPE {
    unsigned short data;
    struct _TREE_NODE_TYPE * left;
    struct _TREE_NODE_TYPE * right;
} TREE_NODE_TYPE;

typedef [transmit_as(TREE_XMIT_TYPE)] TREE_NODE_TYPE * TREE_TYPE;

void __RPC_USER TREE_TYPE_to_xmit(
     TREE_TYPE __RPC_FAR * ,
     TREE_XMIT_TYPE __RPC_FAR * __RPC_FAR *);

void __RPC_USER TREE_TYPE_from_xmit (
     TREE_XMIT_TYPE __RPC_FAR *,
     TREE_TYPE __RPC_FAR *);
```

```
void __RPC_USER TREE_TYPE_free_inst(
     TREE_TYPE __RPC_FAR *);

void __RPC_USER TREE_TYPE_free_xmit(
     TREE_XMIT_TYPE __RPC_FAR *);
```

Remarks

The **transmit_as** attribute instructs the compiler to associate *type-id,* a presented type that client and server applications manipulate, with a transmitted type *xmit-type*. The user must supply routines that convert data between the presented and the transmitted types; these routines must also free memory used to hold the converted data. The **transmit_as** attribute instructs the stubs to call the user-supplied conversion routines.

The transmitted type *xmit-type* must resolve to a MIDL base type, predefined type, or a type identifier. For more information, see **base_types**.

The user must supply the following routines:

Routine name	Description
*type-id*_**to_xmit**	Converts data from the presented type to the transmitted type
*type-id*_**from_xmit**	Converts data from the transmitted type to the presented type
*type-id*_**free_inst**	Frees storage used by the callee for the presented type
*type-id*_**free_xmit**	Frees storage used by the caller for the transmitted type

The client stub calls *type-id*_**to_xmit** to allocate space for the transmitted type and to translate the data into objects of type *xmit-type.* The server stub allocates space for the original data type and calls *type-id*_**from_xmit** to translate the data from its transmitted type to the presented type.

Upon return from the application code, the server stub calls *type-id*_**free_inst** to deallocate the storage for *type-id* on the server side. The client stub calls *type-id*_**free_xmit** to deallocate the *xmit-type* storage on the client side.

The following types cannot have a **transmit_as** attribute:

- Context handles (types with the **context_handle** type attribute and types that are used as parameters with the **context_handle** attribute)
- Parameters that are conformant, varying, or open arrays
- Structures that contain conformant arrays
- The predefined type **handle_t, void**

When a pointer attribute appears as one of the type attributes with the **transmit_as** attribute, the pointer attribute is applied to the *xmit_type* parameter of the *type-id*-**to_xmit** and *type-id*-**from_xmit** routines.

See Also

arrays, **base_types**, **context_handle**, **IDL**, **typedef**

typedef

/* IDL file **typedef** syntax */
typedef [**[** *idl-type-attribute-list* **]**] *type-specifier declarator-list***;**

/* ACF **typedef** syntax */
typedef [*acf-type-attribute-list* **]** *typename*;

idl-type-attribute-list
Specifies one or more attributes that apply to the type. Valid type attributes in an IDL file include **handle**, **switch_type**, **transmit_as**; the pointer attribute **ref**, **unique**, or **ptr**; and the usage attributes **context_handle**, **string**, and **ignore**. Separate multiple attributes with commas.

type-specifier
Specifies a **base_type**, **struct**, **union**, **enum** type, or type identifier. An optional storage specification can precede *type-specifier*. The **const** keyword can precede *type-specifier*.

declarator-list
Specifies standard C declarators, such as identifiers, pointer declarators, and array declarators. For more information, see **pointers** and **arrays**. The *declarator-list* consists of one or more declarators, separated by commas.

acf-type-attribute-list
Specifies one or more attributes that apply to the type. Valid type attributes in an ACF include **allocate**, **encode**, and **decode**.

typename
Specifies a type defined in the IDL file.

Remarks

The IDL **typedef** keyword allows **typedef** declarations that are very similar to C-language **typedef** declarations. The IDL **typedef** declaration is augmented to allow you to associate type attributes with the defined types. Valid type attributes include **handle**, **switch_type**, **transmit_as**; the pointer attribute **ref**, **unique**, or **ptr**; and the usage attributes **context_handle**, **string**, and **ignore**.

The **typedef** keyword in an ACF applies attributes to types that are defined in the corresponding IDL file. For example, the **allocate** type attribute allows you to customize memory allocation and deallocation by both the application and the stubs.

The ACF **typedef** statement appears as part of the **ACF_body**. Note that the ACF **typedef** syntax is different from the IDL **typedef** syntax and the C-language **typedef** syntax. No new types can be introduced in the ACF.

See Also

ACF, **allocate**, **decode**, **encode**, **IDL**

union

The **union** keyword appears in functions that relate to discriminated unions.

MIDL supports two types of discriminated unions: encapsulated unions and non-encapsulated unions. The encapsulated union is compatible with previous implementations of RPC (NCA version 1). The non-encapsulated union eliminates some of the restrictions of the encapsulated union and provides a more visible discriminant than the encapsulated union.

The encapsulated union is identified by the **switch** keyword and the absence of other union-related keywords.

The non-encapsulated union, also known as a union, is identified by the presence of the **switch_is** and **switch_type** keywords, which identify the discriminant and its type.

When you use **in, out** unions, be aware that changing the value of the union switch during the call can make the remote call behave differently from a local call. On return, the stubs copy the **in, out** parameter into memory that is already present on the client. When the remote procedure modifies the value of the union switch and consequently changes the data object's size, the stubs can overwrite valid memory with the **out** value. When the union switch changes the data object from a base type to a pointer type, the stubs can overwrite valid memory when they copy the pointer referent into the memory location indicated by the **in** value of a base type.

The shape of unions must be identical across platforms to ensure interconnectivity.

See Also

encapsulated_union, **IDL**, **non-encapsulated_union**, **switch_is**, **switch_type**

unique

pointer_default(unique)

typedef [unique [**,** *type-attribute-list*] **]** *type-specifier declarator-list***;**

typedef *struct-or-union-declarator* **{**
[unique [**,** *field-attribute-list*] **]** *type-specifier declarator-list***;**
...}

[unique [**,** *function-attribute-list*] **]** *type-specifier ptr-decl function-name***(**
[**[** *parameter-attribute-list* **]**] *type-specifier* [*declarator*]
, ...
);
[**[** *function-attribute-list* **]**] *type-specifier* [*ptr-decl*] *function-name***(**
[unique [**,** *parameter-attribute-list*] **]** *type-specifier* [*declarator*]
, ...
);

type-attribute-list
Specifies one or more attributes that apply to the type. Valid type attributes include **handle**, **switch_type**, **transmit_as**; the pointer attribute **ref**, **unique**, or **ptr**; and the usage attributes **context_handle**, **string**, and **ignore**. Separate multiple attributes with commas.

type-specifier
Specifies a **base_type**, **struct**, **union**, **enum** type, or type identifier. An optional storage specification can precede *type-specifier*.

declarator and *declarator-list*
Specifies standard C declarators, such as identifiers, pointer declarators, and array declarators. For more information, see **pointers** and **arrays**. The *declarator-list* consists of one or more declarators separated by commas. The parameter-name identifier in the function declarator is optional.

struct-or-union-declarator
Specifies a MIDL **struct** or **union** declarator. *field-attribute-list*
Specifies zero or more field attributes that apply to the structure member, union member, or function parameter. Valid field attributes include **first_is**, **last_is**, **length_is**, **max_is**, **size_is**; the usage attributes **string**, **ignore**, and **context_handle**; the pointer attribute **ref**, **unique**, or **ptr**; and the union attribute **switch_type**. Separate multiple field attributes with commas.

function-attribute-list
Specifies zero or more attributes that apply to the function. Valid function attributes are **callback**, **local**; the pointer attribute **ref**, **unique**, or **ptr**; and the usage attributes **string**, **ignore**, and **context_handle**.

ptr-decl
Specifies at least one pointer declarator to which the **unique** attribute applies. A pointer declarator is the same as the pointer declarator used in C; it is constructed from the ***** designator, modifiers such as **far**, and the qualifier **const**.

function-name
Specifies the name of the remote procedure.

parameter-attribute-list
Consists of zero or more attributes appropriate for the specified parameter type. Parameter attributes can take the directional attributes **in** and **out**; the field attributes **first_is**, **last_is**, **length_is**, **max_is**, **size_is**, and **switch_type**; the pointer attribute **ref**, **unique**, or **ptr**; and the usage attributes **context_handle** and **string**. The usage attribute **ignore** cannot be used as a parameter attribute. Separate multiple attributes with commas.

Example

```
pointer_default(unique)

typedef [unique, string] unsigned char * MY_STRING_TYPE;

[unique] char * MyFunction([in, out, unique] long * plNumber);
```

Remarks

The **unique** attribute specifies a unique pointer.

Pointer attributes can be applied as a type attribute; as a field attribute that applies to a structure member, union member, or parameter; or as a function attribute that applies to the function return type. The pointer attribute can also appear with the **pointer_default** keyword.

A unique pointer has the following characteristics:

- Can have the value NULL.
- Can change during a call from NULL to non-null, from non-null to NULL, or from one non-null value to another.
- Can allocate new memory on the client. When the unique pointer changes from NULL to non-null, data returned from the server is written into new storage.
- Can use existing memory on the client without allocating new memory. When a unique pointer changes during a call from one non-null value to another, the pointer is assumed to point to a data object of the same type. Data returned from the server is written into existing storage specified by the value of the unique pointer before the call.
- Can orphan memory on the client. Memory referenced by a non-null unique pointer may never be freed if the unique pointer changes to NULL during a call and the client does not have another means of dereferencing the storage.
- Does not cause aliasing. Like storage pointed to by a reference pointer, storage pointed to by a unique pointer cannot be reached from any other name in the function.

The stubs call the user-supplied memory-management functions **midl_user_allocate** and **midl_user_free** to allocate and deallocate memory required for unique pointers and their referents.

The following restrictions apply to unique pointers:

- The **unique** attribute cannot be applied to binding-handle parameters (**handle_t**) and context-handle parameters.
- The **unique** attribute cannot be applied to **out**-only top-level pointer parameters (parameters that have only the **out** directional attribute).
- Unique pointers cannot be used to describe the size of an array or union arm because unique pointers can have the value NULL. This restriction prevents the error that results if a null value is used as the array size or the union-arm size.

See Also

pointer_default, **pointers**, **ptr**, **ref**

unsigned

The **unsigned** keyword indicates that the most significant bit of an integer variable represents a data bit rather than a signed bit. This keyword is optional and can be used with any of the character and integer types **char**, **wchar_t**, **long**, **short**, and **small**.

When you use the MIDL compiler switch **/char**, character and integer types that appear in the IDL file without explicit sign keywords can appear with the **signed** or **unsigned** keyword in the generated header file. To avoid confusion, explicitly specify the sign of the integer and character types.

See Also

base_types, **/char**, **IDL**, **int**, **long**, **short**, **signed**, **small**

uuid

uuid (*string_uuid***)**
uuid ("*string-uuid***")**

string-uuid
Specifies a string consisting of eight hexadecimal digits followed by a hyphen, then three groups of four hexadecimal digits each followed by a hyphen, then twelve hexadecimal digits. You can enclose the UUID string in quotes when you use the MIDL compiler switch **/ms_ext**.

Examples

```
uuid(6B29FC40-CA47-1067-B31D-00DD010662DA)

uuid("6B29FC40-CA47-1067-B31D-00DD010662DA")
```

Remarks

The **uuid** interface attribute designates a universally unique identifier (UUID) that is assigned to the interface and that distinguishes it from other interfaces. The run-time library uses the interface UUID to help establish communication between the client and server applications.The **uuid** attribute can appear in the interface attribute list for either an RPC interface or an OLE interface.

For an RPC interface, the interface attribute list must include either the **uuid** attribute or the **local** attribute, and the one you choose must occur exactly once. If the list includes the **uuid** attribute, it can also include the **version** attribute.

For an OLE interface (identified by the **object** interface attribute), the interface attribute list must include the **uuid** attribute, but it cannot include the **version** attribute. The list for an OLE interface can include the **local** attribute even though the **uuid** attribute is present.

Microsoft RPC supports an extension to DCE IDL that allows the UUID to be enclosed in double quotation marks. The quoted form is needed for
C-compiler preprocessors that interpret UUID numbers as floating-point numbers.

All UUID values should be computer-generated to guarantee uniqueness. Use the **uuidgen** utility to generate unique UUID values.

The UUID and version numbers of the interface are used to determine whether the client can bind to the server. For the client to bind to the server, the UUID specified in the client and server interfaces must be the same.

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.

See Also

IDL, **interface**, **local**, **/ms_ext**, **version**

v1_enum

[v1_enum] enum {...}

Example

```
typedef [v1_enum] enum {label1, label2, ...};
```

Remarks

The keyword **v1_enum** is used to specify enumerations to be transmitted as a 32-bit entity.

Enumeration types are transmitted as 16-bits by default. When applied to an enumerator, the **v1_enum** attribute ensures that the enumerator is transmitted as 32-bit, rather than 16-bit.

Note that the use of the **v1_enum** attribute is recommended for enumerators on 32-bit systems. This increases the efficiency of marshalling and unmarshalling data when such an enumerator is embedded in structures or unions.

See Also

enum, **IDL**

version

version (*major-value*[**.** *minor-value*] **)**

major-value
Specifies a short unsigned integer between 0 and 65,535, inclusive, that represents the major version number.

minor-value
Specifies a short unsigned integer between 0 and 65,535, inclusive, that represents the minor version number. The minor version value is optional. If present, the minor version value is separated from the major version number by a period character (.). If not specified, the minor version value is zero.

Remarks

The **version** interface attribute identifies a particular version among multiple versions of an RPC interface. With the version attribute, you ensure that only compatible versions of client and server software are allowed to bind.

The MIDL compiler does not support multiple versions of an OLE interface. As a result, an interface attribute list that includes the **object** attribute cannot include the **version** attribute. To create a new version of an existing OLE interface, use interface inheritance. A derived OLE interface has a different UUID but inherits the interface member functions, status codes, and interface attributes of the base interface.

In combination with the **uuid** value, the **version** value uniquely identifies the interface. The run-time library passes the **version** and **uuid** values to the server when the client calls a remote function. A client can bind to a server for a given interface if:

- The **uuid** value is the same.
- The major version number is the same.
- The client's minor version number is less than or equal to the server's minor version number.

It is to your benefit and your users' benefit to retain upward compatibility among versions – that is, to modify the interface so that only the minor version number changes. You can retain upward compatibility when you add new data types that are not used by existing functions and when you add new functions without changing the interface specification for existing functions.

Change the major version number if any one of the following conditions apply:

- If you change a data type that is used by an existing function.
- If you change the interface specification for an existing function (such as adding or removing a parameter).
- If you add callbacks that are called by existing functions.

Change the minor version number if all of the following conditions apply:

- If you add type definitions or constants that are not used by any existing functions or callbacks.
- If you do not change any existing functions and you add new functions to the interface.
- If you add callbacks that are not called by any existing functions and the new callbacks follow any existing functions.

If your modifications qualify as an upward-compatible change to the interface, use the following procedure to modify the interface (IDL) file:

1. Add new constant and type definitions to the interface file.
2. Add callback functions to the end of the interface file.

3. Add new functions to the end of the interface file.

The **version** attribute can occur at most once in the interface header.

When the version attribute is not present, the interface has a default version of 0.0.

The period character between the major and minor numbers is a delimiter and does not represent a decimal point. The minor number is treated as an integer. Leading zeroes are not significant. Trailing zeroes are significant.

For example, the version setting 1.11 represents a major value of one and a minor value of eleven. Version 1.11 does not represent a value between 1.1 and 1.2.

See Also

IDL, **interface**, **uuid**

void

void *function* **(***parameter-list***);**
*return-type function***(void);**
typedef [context_handle] void * *context-handle-type***;**
return-type function **(...[context_handle] void * *** *context-handle-type***...);**

function
Specifies the name of the remote procedure.

parameter-list
Specifies the list of parameters passed to the function along with the associated parameter types and parameter attributes.

return-type
Specifies the name of the type returned by the function.

context-handle-type
Specifies the name of the type that takes the **context_handle** attribute.

Examples

```
void VoidFunc1(void);
void VoidFunc2([in, out] short s1);
typedef [context_handle] void * MY_CX_HNDL_TYPE;
void InitHandle([out] MY_CX_HNDL_TYPE * ppCxHndl);
```

Remarks

The base type **void** indicates a procedure with no arguments or a procedure that does not return a result value.

The pointer type **void ***, which in C describes a generic pointer that can be cast to represent any pointer type, is limited in MIDL to its use with the **context_handle** keyword.

See Also

base_types, **context_handle**, **IDL**

wchar_t

The **wchar_t** keyword designates a wide-character type. The **wchar_t** type is defined by MIDL as an **unsigned short** (16-bit) data object.

The MIDL compiler allows redefinition of **wchar_t**, but only if it is consistent with the preceding definition.

The wide-character type is one of the predefined types of MIDL. The wide-character type can appear as a type specifier in **const** declarations, **typedef** declarations, general declarations, and function declarators (as a function-return-type specifier and as a parameter-type specifier). For the context in which type specifiers appear, see **IDL**.

The **string** attribute can be applied to a pointer or array of type **wchar_t**.

Use the **L** character before a character or a string constant to designate the wide-character-type constant.

See Also

base_types, **char**, **IDL**

MIDL Compiler Errors and Warnings

This section lists MIDL compiler error and warning messages.

An error or warning message sometimes specifies the name of one or more MIDL compiler mode switches. The MIDL compiler accepts an IDL file when you use some mode switches but generates errors for the same file when you do not use mode switches. For example, abstract declarators are valid in Microsoft-extensions mode but generate errors when the **/ms_ext** switch is not specified.

Command-line errors appear in the following format:

```
Command line error : MIDLnnnn : <error text>
[<additional error information>]
```

The additional-error information field provides context-specific information about the error. The information in this field depends on the error message. For example, when an unresolved type-declaration error occurs, the additional-information field displays the name of the type that could not be resolved.

Compile-time warnings appear in the following format:

```
<FileName>(line#) : warning MIDLnnnn :
<warning text>
[optional context information] :
```

Compile-time errors appear in the following format:

```
<FileName>(line#) : error MIDLnnnn :
<error text>
[optional context information] :
```

Optional context information refers to the context in which the error occurred. The MIDL compiler reports this information to help you quickly find the error in the IDL file. Context information is generated when the MIDL compiler discovers an error during semantic analysis of type and procedure signatures.

MIDL1000 : missing source file name
No input file has been specified in the MIDL compiler command line.

MIDL1001 : cannot open input file
The input file specified could not be opened.

MIDL1002 : error while reading input file
The system returned an error while reading the input file.

MIDL1003 : error returned by the C preprocessor
The preprocessor returned an error. The error message is directed to the output stream.

MIDL1004 : cannot execute C preprocessor
The operating system reported an error when it tried to spawn the preprocessor. With MS-DOS, this error can occur when the argument list exceeds 128 bytes. You can reduce the size of the argument list by using a response file.

MIDL1005 : cannot find C preprocessor
The MIDL compiler cannot locate the preprocessor in the specified path or in the path specified by the PATH environment variable.

MIDL1006 : invalid C preprocessor executable
The specified preprocessor is not executable or has an invalid executable-file format.

MIDL1007 : switch specified more than once on command line

A switch has been redefined. The redefined switch is displayed after the error message.

MIDL1008 : unknown switch
An unknown switch has been specified on the command line.

MIDL1009 : unknown argument ignored
The MIDL compiler does not recognize the command-line argument as either a switch, a switch argument, or a filename. The compiler discards the unknown argument and attempts to continue processing.

MIDL1010 : switch not implemented
The switch is defined as part of the IDL compiler but is not implemented in Microsoft RPC.

MIDL1011 : argument(s) missing for switch
The switch expected an argument and the argument is not present. Check the syntax documentation for the specified switch.

MIDL1012 : argument illegal for switch /
The argument supplied to the specified switch is illegal.

MIDL1013 : illegal syntax for switch
Several command-line switches require a space between the switch and the argument, while other switches require no space between the switch and the argument. The specified command line violates the defined syntax for that switch.

MIDL1014 : /no_cpp overrides /cpp_cmd and /cpp_opt
The **cpp_opt** command has been supplied along with the **/no_cpp** switch. The **/no_cpp** switch takes precedence over the other switches.

MIDL1015 : /no_warn overrides warning-level switch
The **no_warn** option has been specified along with the warning-level switch W1, W2, or W3.
The /**no_warn** switch takes precedence over all other warning-level switches.

MIDL1016 : cannot create intermediate file
The system returned an error when the compiler tried to create an intermediate file.

MIDL1018 : out of system file handles
The MIDL compiler ran out of file handles while opening a file. This error can occur if too many import files are open and the compiler tries to open an IDL file or an intermediate file.

MIDL1020 : cannot open response file
The specified response file could not be opened. The file probably does not exist.

MIDL1021 : illegal character(s) found in response file
A non-printable character has been detected in the response file. The response file should contain valid MIDL command-line switches and arguments.

MIDL1023 : nested invocation of response files is illegal
A response file cannot contain the **@** command that directs the MIDL compiler to process another response file. Although there is no limit on the number of response files, response files cannot be nested.

MIDL2000 : must specify /c_ext for abstract declarators
Abstract declarators represent a Microsoft extension to RPC and are not defined in DCE RPC. To compile a file that includes abstract declarators, you must use the **/c_ext** switch.

MIDL2001 : instantiation of data is illegal; you must use "extern" or "static"
Declaration and initialization in the IDL file are not compatible with DCE RPC. To instantiate data, use the Microsoft extensions to RPC by compiling with either the **/ms_ext** or **/c_ext** compiler switch.

MIDL2002 : compiler stack overflow
The compiler ran out of stack space while processing the IDL file. This problem can occur when the compiler is processing a complex declaration or expression. To solve the problem, simplify the complex

declaration or expression.

MIDL2003 : redefinition
This error message can appear under the following circumstances: a type has been redefined; a procedure prototype has been redefined; a member of a structure or union of the same name already exists; a parameter of the same name already exists in the prototype.

MIDL2004 : [auto_handle] binding will be used for procedure
No handle type has been defined as the default handle type. The compiler assumes that an auto handle will be used as the binding handle for the specified procedure.

MIDL2005 : out of memory
The compiler ran out of memory during compilation. Reduce the size or complexity of the IDL file or allocate more memory to the process.

MIDL2006 : recursive definition
A structure or union has been recursively defined. This error can occur when a pointer specification in a nested structure definition is missed.

MIDL2007 : import ignored; file already imported
Importing an IDL file is an idempotent operation. All but the first import operation are ignored.

MIDL2008 : sparse enums require /c_ext or /ms_ext switch
Assigning to enumeration constants is not compatible with DCE RPC. To use the extensions to RPC that permit assigning values to enumeration constants, use the **/c_ext** or **/ms_ext** switch.

MIDL2009 : undefined symbol
An undefined symbol has been used in an expression. This error can occur when you use an **enum** label that is not defined.

MIDL2010 : type used in ACF file not defined in IDL file
An undefined type is being used.

MIDL2011 : unresolved type declaration
The type reported in the additional-information field has not been defined elsewhere in the IDL file.

MIDL2012 : use of wide-character constants requires /ms_ext or /c_ext
Wide-character constants are a Microsoft extension to DCE IDL. To enable the use of the data type **wchar_t**, use the MIDL compiler switch **/ms_ext** or **/c_ext**.

MIDL2013 : use of wide-character strings requires /ms_ext or /c_ext
Wide-character string constants are a Microsoft extension to DCE IDL. To enable the use of the data type **wchar_t**, use the MIDL compiler switch **/ms_ext** or **/c_ext**.

MIDL2014 : inconsistent redefinition of type wchar_t
The type **wchar_t** has been redefined as a type that is not equivalent to **unsigned short**.

MIDL2017 : syntax error
The compiler detected a syntax error at the specified line.

MIDL2018 : cannot recover from earlier syntax errors; aborting compilation
The MIDL compiler automatically tries to recover from syntax errors by adding or removing syntactic elements. This message indicates that despite these attempts to recover, the compiler detected too many errors. Correct the specified error(s) and recompile.

MIDL2019 : unknown pragma option
The specified C pragma is not supported in MIDL. Remove the pragma from the IDL file.

MIDL2020 : feature not implemented
The MIDL feature, although part of the language definition, is not implemented in Microsoft RPC and is not supported by the MIDL compiler. For example, the following language features are not implemented: bitset, pipe, and the international character type. The unimplemented language feature

appears in the additional-information field of the error message.

MIDL2021 : type not implemented
The specified data type, although a legal MIDL keyword, is not implemented in Microsoft RPC.

MIDL2022 : non-pointer used in a dereference operation
A data type that is not a pointer has been associated with pointer operations. You cannot access the object through the specified non-pointer.

MIDL2023 : expression has a divide by zero
The constant expression contains a divide by zero.

MIDL2024 : expression uses incompatible types
The left and right sides of the operator in an expression are of incompatible types.

MIDL2025 : non-array expression uses index operator
The expression uses the array-indexing operation on a data item that is not of the array type.

MIDL2026 : left-hand side of expression does not evaluate to struct/union/enum
The direct or indirect reference operator "." or "->" has been applied to a data object that is not a structure, union, or **enum**. You cannot obtain a direct or indirect reference using the specified object.

MIDL2027 : constant expression expected
A constant expression was expected in the syntax. For example, array bounds require a constant expression. The compiler issues this error message when the array bound is defined with a variable or undefined symbol.

MIDL2028 : expression cannot be evaluated at compile time
The compiler cannot evaluate an expression at compile time.

MIDL2029 : expression not implemented
A feature that was supported in previous releases of the MIDL compiler is not supported in the version of the compiler supplied with Microsoft RPC. Remove the specified feature.

MIDL2030 : no [pointer_default] specified, assuming [unique] for all unattributed pointers
The MIDL compiler offers three different default cases for pointers that do not have pointer attributes. Function parameters that are top-level pointers default to **ref** pointers. Pointers embedded in structures and pointers to other pointers (not top-level pointers) default to the type specified by the **pointer_default** attribute. When no **pointer_default** attribute is supplied, these non-top-level pointers default to unique pointers. This error message indicates the last case: no **pointer_default** attribute is supplied and there is at least one non-top-level pointer that will be treated as a unique pointer.

MIDL2031 : [out] only parameter cannot be a pointer to an open structure
An **out**-only parameter has been used as a pointer to a structure, known as an open structure, whose transmitted range and size are determined at run time. The server stub does not know how much space to allocate for an open structure. Use a pointer to a pointer to the open structure and ensure that the server application allocates sufficient space for it.

MIDL2032 : [out] only parameter cannot be an unsized string
An array with the string attribute has been declared as an **out**-only parameter without any size specification. The server stub needs size information to allocate memory for the string. You can remove the string attribute and add the **size_is** attribute, or you can change the parameter to an **in, out** parameter.

MIDL2033 : [out] parameter is not a pointer
All **out** parameters must be pointers, in keeping with the call-by-value convention of the C programming language. The **out** directional parameter indicates that the server transmits a value to the client. With the call-by-value convention, the server can transmit data to the client only if the function argument is a pointer.

MIDL2034 : open structure cannot be a parameter
A structure or union is truncated when the last element of that structure or union is a conformant array.

MIDL2035 : [out] context handle/generic handle must be specified as a pointer to that handle type
A context-handle or user-defined handle parameter with the **out** directional attribute must be a pointer to a pointer.

MIDL2036 : [context_handle] must not derive from a type that has the [transmit_as] attribute
Context handles must be transmitted as context-handle types. They cannot be transmitted as other types.

MIDL2037 : cannot specify a variable number of arguments to a remote procedure
Remote procedure calls that specify the number of variable arguments at compile time are not compatible with the DCE RPC definition. You cannot use a variable number of arguments in Microsoft RPC.

MIDL2038 : named parameter cannot be "void"
A parameter with the base type **void** is specified with a name.

MIDL2040 : cannot use [comm_status] on both a parameter and a return type
Both the procedure and one of its parameters have the **comm_status** attribute. The **comm_status** attribute specifies that only one data object can be of type **error_status_t** at a time.

MIDL2041 : [local] attribute on a procedure requires /ms_ext
A procedure uses the **local** attribute as a function attribute, which is not compatible with DCE RPC. To enable the Microsoft RPC extensions, use the MIDL compiler switch **/ms_ext**.

MIDL2042 : field deriving from a conformant array must be the last member of the structure
The structure contains a conformant array that is not the last element in the structure. The conformant array must appear as the last structure element.

MIDL2043 : duplicate [case] label
A duplicate case label has been specified. The duplicate label is displayed.

MIDL2044 : no [default] case specified for discriminated union
A discriminated union has been specified without a default case.

MIDL2045 : attribute expression cannot be resolved
The expression associated with the attribute cannot be resolved. This error usually occurs when a variable that appears in the expression is not defined. For example, the error can occur when the variable *s* is not defined and is used by the attribute **size_is**(*s*).

MIDL2046 : attribute expression must be of integral type
The specified attribute variable or expression must be an integral type. This error occurs when the attribute-expression type does not resolve to an integer.

MIDL2047 : [byte_count] requires /ms_ext
The **byte_count** attribute represents an extension to DCE RPC. To enable the Microsoft RPC extensions, use the MIDL compiler switch **/ms_ext**.

MIDL2048 : [byte_count] can be applied only to out parameters of pointer type
The **byte_count** attribute can only be applied to **out** parameters, and all **out** parameters must be pointer types.

MIDL2049 : [byte_count] cannot be specified on a pointer to a conformant array or structure
The **byte_count** attribute cannot be applied to a conformant array or structure.

MIDL2050 : parameter specifying the byte count is not [in]
The value associated with the **byte_count** must be transmitted from the client to the server; it must be an **in** parameter. The **byte_count** parameter does not need to be an **in, out** parameter.

MIDL2051 : parameter specifying the byte count is not an integral type
The value associated with the byte count must be the integer type **small**, **short**, or **long**.

MIDL2052 : [byte_count] cannot be specified on a parameter with size attributes

The **byte_count** attribute cannot be used with other size attributes such as **size_is** or **length_is**.

MIDL2053 : [case] expression is not constant
The expression specified for the case label is not a constant.

MIDL2054 : [case] expression is not of integral type
The expression specified for the case label is not an integer type.

MIDL2055 : specifying [context handle] on a type other than void * requires /ms_ext
For DCE RPC compatibility, the context handle must be a pointer of type **void ***. To use the Microsoft RPC extensions that allow context handles to be associated with types other than **void ***, use the MIDL compiler switch **/ms_ext**.

MIDL2056 : cannot specify more than one parameter with each of comm_status/fault_status
The **comm_status** attribute may only appear once, and the **fault_status** attribute may only appear once per procedure.

MIDL2057 : error_status_t parameter must be an [out] only pointer parameter
The error-code type **error_status_t** is transmitted from server to client and therefore must be specified as an **out** parameter. Due to the constraints in the C programming language, all **out** parameters must be pointers.

MIDL2058 : endpoint syntax error
The endpoint syntax is incorrect.

MIDL2059 : inapplicable attribute
The specified attribute cannot be applied in this construct. For example, the string attribute applies to **char** arrays or **char** pointers and cannot be applied to a structure that consists of two **short** integers:

```
typedef [string] struct foo {
    short x;
    short y;
};
```

MIDL2060 : [allocate] requires /ms_ext
The **allocate** attribute represents a Microsoft extension that is not defined as part of DCE RPC. To enable the Microsoft extensions, use the **/ms_ext** switch.

MIDL2061 : invalid [allocate] mode
An invalid mode for the **allocate** attribute construct has been specified. The four valid modes are **single_node**, **all_nodes**, **on_null**, and **always**.

MIDL2062 : length attributes cannot be applied with string attribute
When the string attribute is used, the generated stub files call the **strlen** function to determine the string length. Don't use the length attribute and the string attribute for the same variable.

MIDL2063 : [last_is] and [length_is] cannot be specified at the same time
Both **last_is** and **length_is** have been specified for the same array. These attributes are related as follows: length = last - first + 1. Because each value can be derived from the other, don't specify both.

MIDL2064 : [max_is] and [size_is] cannot be specified at the same time
Both **max_is** and **size_is** have been specified for the same array. These attributes are related as follows: max = size + 1. Because each value can be derived from the other, don't specify both.

MIDL2065 : no [switch_is] attribute specified at use of union
No discriminant has been specified for the union. The **switch_is** attribute indicates the discriminant used to select among the union fields.

MIDL2066 : no [uuid] specified for interface
No UUID has been specified for the interface.

MIDL2067 : cannot specify both [local] and [uuid] as interface attributes

The local and UUID keywords cannot be used at the same time, except on [object] interfaces.

MIDL2068 : type mismatch between length and size attribute expressions
The length and size attribute expressions must be of the same types. For example, this warning is issued when the attribute variable for the **size_is** expression is of type **unsigned long** and the attribute variable for the **length_is** expression is of type **long**.

MIDL2069 : [string] attribute must be specified "byte", "char", or "wchar_t" array or pointer
A string attribute cannot be applied to a pointer or array whose base type is not a **byte**, **char**, or **struct** in which the members are all of the **byte** or **char** type.

MIDL2070 : mismatch between the type of the [switch_is] expression and the switch type of the union
If the union **switch_type** is not specified, the switch type is the same type as the **switch_is** field.

MIDL2071 : [transmit_as] cannot be applied to a type that derives from a context handle
Context handles cannot be transmitted as other types.

MIDL2072 : [transmit_as] must specify a transmissible type
The specified **transmit_as** type derives from a type that cannot be transmitted by Microsoft RPC, such as **void**, **void ***, or **int**. Use a defined RPC base type; in the case of **int**, add size specifiers like **small**, **short**, or **long** to qualify the **int**.

MIDL2073 : transmitted type must not be a pointer or derive from a pointer
The transmitted type cannot be a pointer or derive from a pointer.

MIDL2074 : presented type must not derive from a conformant/varying array, its pointer equivalent, or a conformant/varying structure
The type to which **transmit_as** has been applied cannot derive from a conformant array or structure (an array or structure whose size is determined at run time).

MIDL2075 : [uuid] format is incorrect
The UUID format does not conform to specification. The UUID must be a string that consists of five sequences of hexadecimal digits of length 8, 4, 4, 4, and 12 digits. "12345678-1234-ABCD-EF01-28A49C28F17D" is a valid UUID. Use the function **UuidCreate** or a utility to generate a valid UUID.

MIDL2076 : uuid is not a hex number
The UUID specified for the interface contains characters that are invalid in a hexadecimal number representation. The characters 0 through 9 and A through F are valid in a hexadecimal representation.

MIDL2077 : interface name specified in the ACF file does not match that specified in the IDL file
In this compiler mode, the name that follows the interface keyword in the ACF must be the same as the name that follows the interface keyword in the IDL file. The interface names in the IDL and ACF files can be different when you compile with the MIDL compiler switch **/acf**.

MIDL2078 : conflicting attributes
Conflicting attributes have been specified. This error often occurs when two attributes are mutually exclusive. For example, the attributes **code** and **nocode** cannot be used at the same time.

MIDL2080 : [local] procedure cannot be specified in ACF file
A local procedure has been specified in the ACF. The local procedure can only be specified in the IDL file.

MIDL2081 : specified type is not defined as a handle
The type specified in the **implicit_handle** attribute is not defined as a handle type. Change the type definition or the type name specified by the attribute.

MIDL2082 : procedure undefined
An attribute has been applied to a procedure in the ACF and that procedure is not defined in the IDL file.

MIDL2083 : this parameter does not exist in the IDL file

A parameter specified in the ACF does not exist in the definition in the IDL file. All parameters, functions, and type definitions that appear in the ACF must correspond to parameters, functions, and types previously defined in the IDL file.

MIDL2084 : this array bounds construct is not supported
MIDL currently supports array-bounds constructs of the form *Array[Lower .. Upper]* only when the constant that specifies the lower bound of the array resolves to the value zero.

MIDL2085 : array bound specification is illegal
The user specification of array bounds for the fixed-size array is illegal. For example:

```
typedef short Array[-1]
```

MIDL2087 : pointee / array does not derive any size
A conformant array has been specified without any size specification. You can specify the size with the **max_is** or **size_is** attribute.

MIDL2088 : badly formed character constant
The end-of-line character is not allowed in character constants.

MIDL2089 : end of file found in comment
The end-of-file character has been encountered in a comment.

MIDL2090 : end of file found in string
The end-of-file character has been encountered in a string.

MIDL2091 : identifier length exceeds 31 characters
Identifiers are limited to 31 alphanumeric characters. Identifier names longer than 31 characters are truncated.

MIDL2092 : end of line found in string
The end-of-line character has been encountered in the string. Verify that you have included the double-quote character that terminates the string.

MIDL2093 : string constant exceeds limit of 255 characters
The string exceeded the maximum allowable length of 255 characters.

MIDL2094 : constant too big
The constant is too large to be represented internally.

MIDL2095 : error in opening file
The operating system reported an error while trying to open an output file. This error can be caused by a name that is too long for the file system or by a duplicate filename.

MIDL2096 : [out] only parameter must not derive from a top-level [unique] or [ptr] pointer/array
A unique pointer cannot be an **out**-only parameter. By definition, a unique pointer can change from null to non-null. No information about the **out**-only parameter is passed from client to server.

MIDL2097 : attribute is not applicable to this non-rpcable union
The **switch_is** and **switch_type** attributes apply to a union that is transmitted as part of a remote procedure call.

MIDL2098 : expression used for a size attribute must not derive from an [out] only parameter
The value of an **out**-only parameter is not transmitted to the server and cannot be used to determine the length or size of the **in** parameter.

MIDL2099 : expression used for a length attribute for an [in] parameter cannot derive from an [out] only parameter
The value of an **out**-only parameter is not transmitted to the server and cannot be used to determine the length or size of the **in** parameter.

MIDL2100 : use of "int" needs /c_ext
MIDL is a strongly typed language. All parameters transmitted over the network must be derived from

one of the MIDL base types. The type **int** is not defined as part of MIDL. Transmitted data must include a size specifier: **small**, **short**, or **long**. Data that is not transmitted over the network can be included in an interface; use the **/c_ext** switch.

MIDL2101 : struct/union field must not be void
Fields in a structure or union must be declared to be of a specific base type supported by MIDL or a type that is derived from the base types. **Void** types are not allowed in remote operations.

MIDL2102 : array element must not be void
An array element cannot be void.

MIDL2103 : use of type qualifiers and/or modifiers needs /c_ext
Type modifiers such as **_cdecl** and **_far** can be compiled only if you specify the **/c_ext** switch.

MIDL2104 : struct/union field must not derive from a function
The fields of a structure or union must be MIDL base types or types that are derived from these base types. Functions are not legal in structure or union fields.

MIDL2105 : array element must not be a function
An array element cannot be a function.

MIDL2106 : parameter must not be a function
The parameter to a remote procedure must be a variable of a specified type. A function cannot be a parameter to the remote procedure.

MIDL2107 : struct/union with bit fields needs /c_ext
You must specify the MIDL compiler switch **/c_ext** to allow bit fields on data that is not transmitted in a remote procedure call.

MIDL2108 : bit field specification on a type other that "int" is a non ANSI-compatible extension
The ANSI C programming language specification does not allow bit fields to be applied to non-integer types.

MIDL2109 : bit field specification can be applied only to simple, integral types
The ANSI C programming language specification does not allow bit fields to be applied to non-integer types.

MIDL2110 : struct/union field must not derive from handle_t or a context_handle
Context handles cannot be transmitted as part of another structure. They must be transmitted as context handles.

MIDL2111 : array element must not derive from handle_t or a context handle
Context handles cannot be transmitted as part of an array.

MIDL2112 : this specification of union needs /c_ext
A union that appears in the interface definition must be associated with the discriminant or declared as local. Data that is not transmitted over the network can be implicitly declared as local when you use the **/c_ext** switch.

MIDL2113 : parameter deriving from an "int" must have size specifier "small", "short", or "long" with the "int"
The type **int** is not a valid MIDL type unless it is accompanied by a size specification. Use one of the size specifiers **small**, **short**, or **long**.

MIDL2114 : type of the parameter cannot derive from void or void*
MIDL is a strongly typed language. All parameters transmitted over the network must be derived from one of the MIDL base types. MIDL does not support **void** as a base type. You must change the declaration to a valid MIDL type.

MIDL2115 : parameter deriving from a struct/union containing bit fields is not supported
Bit fields are not defined as a valid data type by DCE RPC.

MIDL2116 : use of a parameter deriving from a type containing type-modifiers/type-qualifiers needs /c_ext
Such keywords as **far**, **near**, **const**, and **volatile** can appear in the IDL file only when you activate the **/c_ext** extension to the MIDL compiler.

MIDL2117 : parameter must not derive from a pointer to a function
The RPC run-time libraries transmit a pointer and its associated data between client and server. Pointers to functions cannot be transmitted as parameters because the function cannot be transmitted over the network.

MIDL2118 : parameter must not derive from a non-rpcable union
The union must be associated with a discriminant. Use the **switch_is** and **switch_type** attributes.

MIDL2119 : return type derives from an "int". You must use size specifiers with the "int"
The type **int** is not a valid MIDL type unless it is accompanied by a size specification. Use one of the size specifiers **small**, **short**, or **long**.

MIDL2120 : return type must not derive from a void pointer
MIDL is a strongly typed language. All parameters transmitted over the network must be derived from one of the MIDL base types. **Void** types are not defined as part of MIDL. You must change the declaration to a valid MIDL type.

MIDL2121 : return type must not derive from a struct/union containing bit-fields
Bit fields are not defined as a valid data type by DCE RPC.

MIDL2122 : return type must not derive from a non-rpcable union
The union must be associated with a discriminant. Use the **switch_is** and **switch_type** attributes.

MIDL2123 : return type must not derive from a pointer to a function
The RPC run-time libraries transmit a pointer and its associated data between client and server. Pointers to functions cannot be transmitted as parameters because RPC does not define a method to transmit the associated function over the network.

MIDL2124 : compound initializers are not supported
DCE RPC supports simple initialization only. The structure or array cannot be initialized in the IDL file.

MIDL2125 : ACF attributes in the IDL file need the /app_config switch
A Microsoft extension allows you to specify ACF attributes in the IDL file. Use the **/app_config** switch to activate this extension.

MIDL2126 : single line comment needs /ms_ext or /c_ext
Single-line comments that use two backslash characters (\\) represent a Microsoft extension to DCE RPC. You must use one of the mode-extension switches for a single-line comment.

MIDL2127 : [version] format is incorrect
The interface version number in the interface header must be specified in the format *major*.*minor*, where each number can range from 0 to 65535.

MIDL2128 : "signed" needs /ms_ext or /c_ext
The use of the **signed** keyword is a Microsoft extension to DCE RPC. You must use one of the mode-extension switches to activate this extension.

MIDL2129 : mismatch in assignment type
The type of the variable does not match the type of the value that is assigned to the variable.

MIDL2130 : declaration must be of the form: const <type><declarator> = <initializing expression>
The declaration is not compatible with DCE RPC syntax. Use the **/ms_ext** or **/c_ext** MIDL compiler mode switch.

MIDL2131 : declaration must have "const"
Declarations in the IDL file must be constant expressions that use the keyword **const**. For example:

```
const short x = 2;
```

MIDL2132 : struct/union/enum must not be defined in a parameter type specification
The structure, union, or enumerated type must be explicitly specified outside of the function prototype.

MIDL2133 : [allocate] attribute must be applied only on non-void pointer types
The **allocate** attribute is designed for complex pointer-based data structures. When the **allocate** attribute is specified, the stub file traverses the data structure to compute the total size of all objects accessible from the pointer and all other pointers in the data structure. Change the type to a non-void pointer type or remove the **allocate** attribute and use another method to determine its allocation size, such as the **sizeof** operator.

MIDL2134 : array or equivalent pointer construct cannot derive from a non-encapsulated union
Each union must be associated with a discriminant. Arrays of unions are not permitted because they do not provide the associated discriminant. Arrays of structures are permitted because each structure consists of the union and its discriminant.

MIDL2135 : field must not derive from an error_status_t type
The **error_status_t** type can only be used as a parameter or a return type. It cannot be embedded in the field of a structure or union.

MIDL2136 : union has at least one arm without a case label
The union declaration does not match the required MIDL syntax for the union. Each union arm requires a case label or default label that selects that union arm.

MIDL2137 : a parameter or a return value must not derive from a type which has [ignore] applied to it
The **ignore** attribute is a field attribute that can only be applied to fields, such as fields of structures and arrays. The **ignore** attribute indicates that the stub should not dereference the pointer during transmission and is not allowed when it conflicts with other attributes that must be dereferenced, such as **out** parameters and function return values.

MIDL2138 : pointer already has a pointer-attribute applied to it
Only one of the pointer attributes, **ref**, **unique**, or **ptr**, can be applied to a pointer.

MIDL2139 : field/parameter must not derive from a structure that is recursive through a ref pointer
By definition, a reference pointer cannot be set to NULL. A recursive data structure defined with a reference pointer has no null elements and by convention is non-terminating. Use a **unique** pointer attribute to allow the data structure to specify a null element or redefine the data structure as a non-recursive data structure.

MIDL2140 : use of field deriving from a void pointer needs /c_ext
The type **void *** and other types and type qualifiers that are not supported by DCE IDL are only allowed in the IDL file when you use the MIDL compiler switch **/c_ext**. Redefine the pointer type or recompile using the **/c_ext** switch.

MIDL2141 : use of this attribute needs /ms_ext
This language feature is a Microsoft extension to DCE IDL. You must specify the MIDL compiler switch **/ms_ext**.

MIDL2142 : use of wchar_t needs /ms_ext or /c_ext
The wide-character type represents an extension to DCE IDL. The MIDL compiler accepts the wide-character type only when you specify the **/ms_ext** or **/c_ext** switch.

MIDL2143 : unnamed fields need /ms_ext or /c_ext
DCE IDL does not support the use of unnamed structures or unions embedded in other structures or unions. In DCE IDL, all such embedded fields must be named. To enable this Microsoft extension to IDL, supply the MIDL compiler switch **/ms_ext** or **/c_ext**.

MIDL2144 : unnamed fields can derive only from struct/union types

The Microsoft extension to the DCE IDL that supports unnamed fields applies only to structures and unions. You must assign a name to the field or redefine the field to comply with this restriction.

MIDL2145 : field of a union cannot derive from a varying/conformant array or its pointer equivalent
The conformant array cannot appear alone in the union but must be accompanied by the value that specifies the size of the array. Instead of using the array as the union arm, use a structure that consists of the conformant array and the identifier that specifies the size.

MIDL2146 : no [pointer_default] attribute specified, assuming [ptr] for all unattributed pointers in interface
The DCE IDL implementation specifies that all pointers in each IDL file must be associated with pointer attributes. When an explicit pointer attribute is not assigned to the parameter or pointer type and no **pointer_default** attribute is specified in the IDL file, the full pointer attribute **ptr** is associated with the pointer. You can change the pointer attributes by using explicit pointer attributes, by specifying a **pointer_default** attribute, or by specifying the **/ms_ext** switch to change the default for unattributed pointers to **unique**.

MIDL2147 : initializing expression must resolve to a constant expression
The use of initializing expressions is limited to constant expressions in all MIDL compiler modes. The expression must be resolvable at compile time. Specify a literal constant, or an expression that resolves to a constant, rather than a variable.

MIDL2148 : attribute expression must be of type integer, char, byte, boolean or enum
The specified type does not resolve to a valid switch type. Use an integer, character, **byte**, **boolean**, or **enum** type, or a type that is derived from one of these types.

MIDL2149 : illegal constant
The specified constant is out of the valid range for the specified type.

MIDL2150 : attribute not implemented; ignored
The attribute specified is not implemented in this release of Microsoft RPC. The MIDL compiler continues processing the IDL file as if the attribute were not present.

MIDL2151 : return value must not derive from a [ref] pointer
Function return values that are defined to be pointer types must be specified as unique or full pointers. Reference pointers cannot be used.

MIDL2152 : attribute expression must be a variable name or a pointer dereference expression in this mode. You must specify the /ms_ext switch
The DCE IDL compiler requires the size associated with the **size_is** attribute to be specified by a variable or pointer variable. To enable the Microsoft extension that allows the **size_is** attribute to be defined by a constant expression, use the **/ms_ext** switch.

MIDL2153 : parameter must not derive from a recursive non-encapsulated union
A union must include a discriminant, so a union cannot have another union as an element. A union can be embedded in another union only when it is part of a structure that includes the discriminant.

MIDL2154 : binding-handle parameter cannot be [out] only
The handle parameter identified by the MIDL compiler as the binding handle for this operation must be an **in** parameter. **Out**-only parameters are undefined on the client stub, and the binding handle must be defined on the client.

MIDL2155 : pointer to a handle cannot be [unique] or [ptr]
The unique and full pointer attributes allow the value NULL. The binding handle cannot be null. Use the **ref** attribute to derive the binding-handle parameter from reference pointers.

MIDL2156 : parameter that is not a binding handle must not derive from handle_t
The primitive handle type **handle_t** is not a valid data type that is transmitted over the network. Change the parameter type to a type other than **handle_t** or remove the parameter.

MIDL2157 : unexpected end of file found
The MIDL compiler found the end of the file before it was able to successfully resolve all syntactical elements of the file. Verify that the terminating right brace character (}) is present at the end of the file, or check the syntax.

MIDL2158 : type deriving from handle_t must not have [transmit_as] applied to it
The primitive handle type **handle_t** is not transmitted over the network.

MIDL2159 : [context_handle] must not be applied to a type that has [handle] applied to it
The **context_handle** and **handle** attributes cannot be applied to the same type.

MIDL2160 : [handle] must not be specified on a type deriving from void or void *
A type specified with the **handle** attribute can be transmitted over the network, but the type **void *** is not a transmissible type. The handle type must resolve to a type that derives from the valid base types.

MIDL2161 : parameter must have either [in], [out] or [in,out] in this mode. You must specify /ms_ext or /c_ext
The DCE IDL compiler requires all parameters to have explicit directional parameters. To use the Microsoft extensions to DCE IDL, where you can omit explicit directional attributes, use the MIDL compiler switch **/ms_ext** or **/c_ext**.

MIDL2162 : [transmit_as] must not be specified on void type
The **transmit_as** attribute applies only to pointer types. Use the type **void *** in place of **void**.

MIDL2163 : void must be specified on the first and only parameter specification
The keyword **void** incorrectly appears with other function parameters. To specify a function without parameters, the keyword **void** must be the only element of the parameter list, as in the following example:

```
void Foo(void)
```

MIDL2164 : [switch_is] must be specified only on a type deriving from a non-encapsulated union
The **switch_is** keyword is incorrectly applied. It can only be used with non-encapsulated union types. For more information, see the syntax section in the reference entry for non-encapsulated unions.

MIDL2165 : stringable structures are not implemented in this version
DCE IDL allows the attribute string to apply to a structure whose elements consist only of characters, bytes, or types that resolve to characters or bytes. This functionality is not supported in Microsoft RPC. The string attribute cannot be applied to the structure as a whole; it can be applied to each individual array.

MIDL2166 : switch type can only be integral, char, byte, boolean or enum
The specified type does not resolve to a valid switch type. Use an integer, character, **byte**, **boolean**, or **enum** type, or a type that is derived from one of these types.

MIDL2167 : [handle] must not be specified on a type deriving from handle_t
A handle type must be defined using one and only one of the handle types or attributes. Use the primitive type **handle_t** or the attribute **handle**, but not both. The user-defined handle type must be transmissible, but the **handle_t** type is not transmitted on the network.

MIDL2168 : parameter deriving from handle_t must not be an [out] parameter
A handle of the primitive type **handle_t** is meaningful only to the side of the application in which it is defined. The type **handle_t** is not transmitted on the network.

MIDL2169 : expression specifying size or length attributes derives from [unique] or [ptr] pointer dereference
Although the unique and full pointer attributes allow pointers to have null values, the expression that defines the size or length attribute must never have a null value. When pointers are used, MIDL constrains expressions to **ref** pointers.

MIDL2170 : "cpp_quote" requires /ms_ext
The **cpp_quote** attribute is a Microsoft extension to DCE IDL. Use the MIDL compiler switch **/ms_ext**.

MIDL2171 : quoted uuid requires /ms_ext
The ability to specify a UUID value within quotation marks is a Microsoft extension to DCE IDL. Use the MIDL compiler switch **/ms_ext**.

MIDL2172 : return type cannot derive from a non-encapsulated union
The non-encapsulated union cannot be used as a function return type. To return the union type, specify the union type as an **out** or **in, out** parameter.

MIDL2173 : return type cannot derive from a conformant structure
The size of the return type must be a constant. You cannot specify as a return type a structure that contains a conformant array even when the structure also includes its size specifier. To return the conformant structure, specify the structure as an **out** or **in, out** parameter.

MIDL2174 : [transmit_as] must not be applied to a type deriving from a generic handle
In this release, the **handle** and **transmit_as** attributes cannot be combined on the same type.

MIDL2175 : [handle] must not be applied to a type that has [transmit_as] applied to it
In this release, the **handle** and **transmit_as** attributes cannot be combined on the same type.

MIDL2176 : type specified for the const declaration is invalid
Const declarations are limited to integer, character, wide-character, string, and boolean types.

MIDL2177 : operand to the sizeof operator is not supported
The MIDL compiler supports the **sizeof** operation for simple types only.

MIDL2178 : this name already used as an const identifier name
The identifier has previously been used to identify a constant in a **const** declaration. Change the name of one of the identifiers so that the identifiers are unique.

MIDL2179 : inconsistent redefinition of type error_status_t
The type **error_status_t** must resolve to the type **unsigned long**. Other type definitions cannot be used.

MIDL2180 : [case] value out of range of switch type
The value associated with the switch statement case is out of range for the specified switch type. For example, this error occurs when a long integer value is used in the case statement for a short integer type.

MIDL2181 : parameter deriving from wchar_t needs /ms_ext
The wide-character type **wchar_t** is a Microsoft extension to DCE IDL. Use the MIDL compiler switch **/ms_ext**.

MIDL2182 : this interface has only callbacks
Callbacks are valid only in the context of a remote procedure call. The interface must include at least one function prototype for a remote procedure call that does not include the **callback** attribute.

MIDL2183 : redundantly specified attribute; ignored
The specified attribute has been applied more than once. Multiple instances of the same attribute are ignored.

MIDL2184 : context handle type used for an implicit handle
A type that was defined using the **context_handle** attribute has been specified as the handle type in an **implicit_handle** attribute. The attributes cannot be combined in this way.

MIDL2185 : conflicting options specified for [allocate]
The options specified for the ACF attribute **allocate** represent conflicting directives. For example, specify either the option **all_nodes** or the option **single_node**, but not both.

MIDL2186 : error while writing to file
An error occurred while writing to the file. This condition can be caused by errors relating to disk space,

file handles, file-access restrictions, and hardware failures.

MIDL2187 : no switch type found at definition of union, using the [switch_is] type
The union definition does not include an explicit **switch_type** attribute. The type of the variable specified by the **switch_is** attribute is used as the switch type.

MIDL2188 : semantic check incomplete due to previous errors
The MIDL compiler makes two passes over the input file(s) to resolve any forward declarations. Due to errors encountered during the first pass, checking for the second pass has not been performed. Unreported errors relating to forward declarations may still be present in the file.

MIDL2189 : handle parameter or return type is not supported on a [callback] procedure
A callback procedure occurs in the context of a call from a client to the server and uses the same binding handle as the original call. Explicit binding-handle parameters or return types are not permitted in callback functions.

MIDL2192 : [context_handle] must not derive from handle_t
The three handle characteristics − the type **handle_t**, the attribute **handle**, and the attribute **context_handle** − are mutually exclusive. Only one can be applied to a type or parameter at a time.

MIDL2193 : array size exceeds 65536 bytes
On some Microsoft platforms, the maximum transmissible data size is 64K. Redesign your application so that all transmitted data fits within the maximum transmissible size.

MIDL2194 : field of a non-encapsulated union cannot be another non-encapsulated union
Unions that are transmitted as part of a remote procedure call require an associated data item, the discriminant, that selects the union arm. Unions nested in other unions do not offer a discriminant; as a result, they cannot be transmitted in this form. Create a structure that consists of the union and its discriminant.

MIDL2195 : pointer attribute(s) applied on an embedded array; ignored
A pointer attribute can be applied to an array only when the array is a top-level parameter. Other pointer attributes applied to arrays embedded in other data structures are ignored.

MIDL2196 : [allocate] is illegal on a type that has [transmit_as] applied to it
The **transmit_as** and **allocate** attributes cannot both be applied to the same type. The **transmit_as** attribute distinguishes between presented and transmitted types, while the **allocate** attribute assumes that the presented type is the same as the transmitted type.

MIDL2198 : [implicit_handle] type undefined; assuming primitive handle
The handle type specified in the ACF is not defined in the IDL file. The MIDL compiler assumes that the handle type resolves to the primitive handle type **handle_t**. Add the **handle** attribute to the type definition if you want the handle to behave like a user-defined, or generic, handle.

MIDL2199 : array element must not derive from error_status_t
In this release of Microsoft RPC, the type **error_status_t** can only appear as a parameter or a return type. It cannot appear in arrays.

MIDL2200 : [allocate] illegal on a type deriving from a primitive/generic/context handle
By design, the ACF attribute **allocate** cannot be applied to handle types.

MIDL2201 : transmitted or presented type must not derive from error_status_t
In this release of Microsoft RPC, the type **error_status_t** cannot be used with the **transmit_as** attribute.

MIDL2202 : discriminant of a union must not derive from a field with [ignore] applied to it
A union used in a remote procedure call must be associated with another data item, called the discriminant, that selects the union arm. The discriminant must be transmitted. The **ignore** attribute cannot be applied to the union discriminant.

MIDL2203 : [nocode] must be specified with "/server none" in this mode
Some DCE IDL compilers generate an error when the **nocode** attribute is applied to a procedure in an

interface for which server stub files are being generated. Because the server must support all operations, **nocode** must not be applied to a procedure in this mode or you must use the MIDL compiler switch **/server none** to explicitly specify that no server routines are to be generated.

MIDL2204 : no remote procedures specified, no client/server stubs will be generated
The provided interface does not have any remote procedures, so only header files will be generated.

MIDL2205 : too many default cases specified for encapsulated union
An encapsulated union may only have one default: arm.

MIDL2206 : union specification with no fields is illegal
Unions must have at least one field.

MIDL2207 : value out of range
The provided case value is out of the range of the switch type.

MIDL2208 : [context_handle] must be applied on a pointer type
Context handles must always be pointer types. DCE specifies that all context handles must be typed as "void *".

MIDL2209 : return type must not derive from handle_t
Handle_t may not be returned.

MIDL2210 : [handle] must not be applied to a type deriving from a context handle
A type may not be both a context handle and a generic handle.

MIDL2211 : field deriving from an \"int\" must have size specifier \"small\", \"short\", or \"long\" with the \"int\"
The use of "int" is not remotable, since the size of "int" may be different accross machines.

MIDL2212 : field must not derive from a void or void *
Void and void * are not remotable types.

MIDL2213 : field must not derive from a struct containing bit-fields
bit fields in structs are not remotable.

MIDL2214 : field must not derive from a non-rpcable union
A union must be specified as a non-encapsulated union or encapsulated union in order to be remoted. Ordinary C unions lack the discriminant needed to remote the union.

MIDL2215 : field must not derive from a pointer to a function
Pointers to functions may not be remoted.

MIDL2216 : cannot use [fault_status] on both a parameter and a return type
[fault_status] may only be used once per procedure, although [comm_status] may be used independently.

MIDL2217 : return type too complicated for /Oi, using /Os
Large by-value return types may only be handled by /Os optimization stubs. The stubs for this routine will be generated using /Os optimization.

MIDL2218 : generic handle type too large for /Oi, using /Os
Large by-value generic handle types may only be handled by /Os optimization stubs. The stubs for this routine will be generated using /Os optimization.

MIDL2219 : [allocate(all_nodes)] on an [in,out] parameter may orphan the original memory
Use of [allocate(all_nodes)] on an [in,out] parameter must re-allocate contiguous memory for the [out] direction, thus orphaning the [in] parameter. This usage is not recommended.

MIDL2220 : cannot have a [ref] pointer as a union arm
Ref pointers must always point to valid memory, but an [in,out] union with a ref pointer may return a ref pointer when the [in] direction used another type.

MIDL2222 : use of [comm_status] or [fault_status] not supported for /Oi, using /Os

[comm_status] and [fault_status] may only be handled by /Os optimization stubs. The stubs for this routine will be generated using /Os optimization.

MIDL2223 : use of an unknown type for [represent_as] not supported for /Oi, using /Os
Use of a represent_as with a local type that is not defined in the idl file or an imported idl file may only be handled by /Os optimization stubs. The stubs for this routine will be generated using /Os optimization.

MIDL2224 : array types with [transmit_as] or [represent_as] not supported on return type for /Oi, using /Os
Returning an array with [transmit_as] or [represent_as] applied may only be handled by /Os optimization stubs. The stubs for this routine will be generated using /Os optimization.

MIDL2226 : [callback] requires /ms_ext
[callback] is a Microsoft extension and requires use of the /ms_ext switch.

MIDL2227 : circular interface dependency
This interface uses itself (directly or indirectly) as a base interface.

MIDL2228 : only IUnknown may be used as the root interface
Currently, all interfaces must have IUnknown as the root interface.

MIDL2229 : [IID_IS] may only be applied to pointers to interfaces
[iid_is] can only be applied to interface pointers, although they may be specified as IUnknown *.

MIDL2230 : interfaces may only be used in pointer-to-interface constructs
Interface names may not be used except as base interfaces or interface pointers.

MIDL2231 : interface pointers must have a UUID/IID
The base type of the iid_is expression must be a UUID/GUID/IID type.

MIDL2232 : definitions and declarations outside of interface body requires /ms_ext
Putting declarations and definitions outside of any interface body is a Microsoft extension and requires the use of the /ms_ext switch.

MIDL2233 : multiple interfaces in one file requires /ms_ext
Using multiple interfaces in a single idl file is a Microsoft extension and requires the use of the /ms_ext switch.

MIDL2234 : only one of [implicit_handle], [auto_handle], or [explicit_handle] allowed
Each interface may only have one of the above.

MIDL2235 : [implicit_handle] references a type which is not a handle
Implicit handles must be of one of the handle types.

MIDL2236 : [object] procs may only be used with "/env win32"
[object] interfaces may not be used with 16-bit environments.

MIDL2237 : [callback] with -env dos/win16 not supported for /Oi, using /Os
Callbacks in 16-bit environments may only be handled by /Os optimization stubs. The stubs for this routine will be generated using /Os optimization.

MIDL2238 : float/double not supported as top-level parameter for /Oi, using /Os
Float and double as parameters may only be handled by /Os optimization stubs. The stubs for this routine will be generated using /Os optimization. Float and double within structs/arrays/etc. May still be handled with /Oi.

MIDL2239 : pointers to context handles may not be used as return values
Context handles must be used as direct return values, not indirect return values.

MIDL2240 : procedures in an object interface must return an HRESULT
All non-[local] procedures in an object interface must return an HRESULT/SCODE.

MIDL2241 : duplicate UUID

UUIDs must be unique.

MIDL2242 : [object] interfaces must derive from other [object] interfaces
Interface inheritance is only allowed using object interfaces.

MIDL2243 : [IID_IS] expression must be a pointer to IID structure
The base type of the iid_is expression must be a UUID/GUID/IID type.

MIDL2244 : [call_as] type must be a [local] procedure
Tthe target of a call_as, if defined, must have [local] applied.

MIDL2245 : undefined [call_as] must not be used in an object interface [call_as]: in_list
Another routine defined in the ACf is attempting to use the same call_as routine as the previous routine.

MIDL2246 : [auto_handle] may not be used with [encode] or [decode]
[encode] and [decode] may only be used with explicit handles or implicit handles.

MIDL2247 : normal procs are not allowed in an interface with [encode] or [decode]
Interfaces containing [encode] or [decode] procedures may not also have remoted procedures.

MIDL2248 : top-level conformance or variance not allowed with [encode] or [decode]
Types that have top-level conformance or variance may not use type serialization, since there is no way to provide sizing/lengthing. Structs containing them are, however, allowed.

MIDL2249 : [out] parameters may not have \"const\"
Since an [out] parameter is altered, it may not have const.

MIDL2250 : return values may not have \"const\"
Since a function value is set, it must not have const.

MIDL2251 : multiple calling conventions illegal
Only one calling convention may be applied to a single procedure.

MIDL2252 : attribute illegal on [object] procedure
The above attribute only applies to procedures in interfaces that do not have [object].

MIDL2253 : [out] interface pointers must use double indirection
Since the altered value is the pointer to the interface, there must be another level of indirection above it to allow it to be returned.

MIDL2254 : procedure used twice as the caller in [call_as]
A given [local] procedure may only be used once as the target of a [call_as], in order to avoid name clashes.

MIDL2255 : [call_as] target must have [local] in an object interface
The target of a call_as must be a defined, [local] procedure in the current interface.

MIDL2256 : [code] and [nocode] may not be used together
These two attributes are contradictory, and may not be used together.

MIDL2257 : [maybe] procedures may not have a return value or [out] params
Since [maybe] procedures may never return, there is no way to get returned values.

MIDL2258 : pointer to function must be used
Although function type definitions are allowed in /c_ext mode, they may only be used as pointers to functions (and may never be remoted).

MIDL2259 : functions may not be passed in an RPC operation
Functions and function pointers may not be remoted.

MIDL2260 : hyper/double not supported as return value for /Oi, using /Os
Hyper and double return values may only be handled by /Os optimization stubs. The stubs for this routine will be generated using /Os optimization.

MIDL2261 : #pragma pack(pop) without matching #pragma pack(push)
#pragma pack(push) and #pragma pack(pop) must appear in matching pairs. At least one too many #pragma pack(push)'s were specified.

MIDL2262 : stringable structure fields must be byte/char/wchar_t
[string] may only be applied to a struct whose fields are all of type byte, or a type definition equivalent of byte.

MIDL2263 : [notify] not supported for /Oi, using /Os
The [notify] attribute may only be processed by /Os optimization stubs.

MIDL2264 : handle parameter or return type is not supported on a procedure in an [object] interface
Handles may not be used with [object] interfaces.

MIDL2265 : ANSI C only allows the leftmost array bound to be unspecified
In an conformant array, ANSI C only allows the leftmost (most significant) array bound to be unspecified. If multiple dimensions are conformant, MIDL will attempt to put a "1" in the other conformant dimensions. If the other dimensions are defined in a different typedef, this may not be possible. Try putting all the array dimensions on the use of the array to avoid this. In any case, beware of the array indexing calculations done by the compiler; you may need to do your own calculations using the actual sizes.

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_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 For Windows 95 platforms, 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 a Windows 95 client communicating with a Windows NT server. These levels are never supported for a Windows 95 client communicating with a Windows 95 server.

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	NT LM SSP (NT Security Service)

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

```
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

A protocol sequence is a character string identifying the network protocols used to establish a relationship between a client and server. The protocol sequence contains a set of options that must be defined to the RPC run-time library. There are three options in this set:

- The RPC protocol used for communications. (Available options are ncacn and ncadg.)
- The format used in the network address supplied in the binding. (Available options are ip, dnet, and osi.)
- The transport protocol used for communications. (Available options are tcp, udp, nsp, dna, np, and nb.)

The following predefined strings represent valid combinations:

Protocol sequence	Description
ncacn_ip_tcp	NCA connection over TCP/IP (transmission control protocol/internet protocol)
ncacn_nb_nb	NCA connection over NetBEUI over NetBIOS
ncacn_nb_tcp	NCA connection over TCP/IP over NetBIOS
ncacn_np	NCA connection over named pipes
ncacn_spx	Connection-oriented SPX
ncadg_ip_udp	Datagram-oriented TCP/IP
ncadg_ipx	Datagram-oriented IPX
ncalrpc	Local procedure call

Note Windows 95 does not support **ncalrpc**, **ncacn_nb_ipx**, and **ncacn_nb_tcp**. The **ncacn_np** protocol is supported only on the client side. You must have an authentic Novell client to use the RPC SPX transport.

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 Windows NT applications.

The **ncacn_dnet_nsp** protocol sequence is supported only for MS-DOS, Microsoft Windows 3.*x*, and Microsoft Windows for Workgroups 3.1 client applications. This release of Microsoft RPC does not include support for the **ncacn_dnet_nsp** protocol sequence with Microsoft Windows NT client or server applications.

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 short *        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.

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

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

RpcServerRegisterIf

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_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.

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.

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.

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:

typedef struct _SEC_WINNT_AUTH_IDENTITY

```
  unsigned short __RPC_FAR   *     User;
  unsigned long  __Userlength;
  unsigned short __RPC_FAR   *     Domain;
  unsigned long  Domian Length;
  unsigned short __RPC_FAR   *     Password;
  unsigned long  Password length;
SEC_WINNT_AUTH_IDENTITY,     *
PSEL_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.

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.

For Windows 3.*x* and MS-DOS, the strings are ANSI; for Windows NT, the strings are Unicode. For Windows NT, the values for *Userlength*; *DomainLength*, and *Password Length* 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 mapping object UUIDs to manager entry-point vectors, see **RpcServerRegisterIf**..

ProtocolSequence

Specifies the protocol sequence for making remote procedure calls. The following protocol sequences are supported by Microsoft RPC:

- **ncacn_ip_tcp**
- **ncacn_np**
- **ncacn_nb_nb**, **ncacn_nb_tcp**
- **ncacn_spx**
- **ncalrpc**
- **ncadg_ip_udp**
- **ncadg_ipx**

Note Windows 95 does not support **ncalrpc**, **ncacn_nb_ipx**, and **ncacn_nb_tcp**. The **ncacn_np** protocol is supported only on the client side. The **ncacn_dnet_nsp** protocol sequence is supported only for MS-DOS, Microsoft Windows 3.*x*, and Microsoft Windows for Workgroups 3.1 client applications. This release of Microsoft RPC does not include support for the **ncacn_dnet_nsp** protocol sequence with Microsoft Windows NT client or server applications.

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	Example
ncacn_dnet_nsp	Area and node syntax	4.120
ncacn_ip_tcp	Common internet address notation	128.10.2.30
ncacn_ip_tcp	Host name	ko
ncacn_nb_nb, ncacn_nb_tcp	Windows NT server name	marketing
ncacn_np	Windows NT server name	\\marketing
ncacn_spx	Network number and node number	~0000000108002B3061 2C
ncadg_ip_udp	Host name	ko or 11.101.9.127
ncadg_ipx	Network number and node number	~0000000108002B3061 2C
ncalrpc	None	

The network-address field is optional. When you do not specify a network address, the string binding refers to your local host. No network address should be specified when you use the **ncalrpc** protocol sequence.

When the specified host name is multi-homed, the function **RpcBindingFromStringBinding** returns

a binding handle for the first host address it finds. To specify the host address, use the common internet address notation, an area node syntax, or an NSP instead of a host name.

The string representing the NSP is a percent sign followed by the letter X followed by hexadecimal digits. No separators are allowed. Because the NSP specifies the transport to be used as part of its value, any transport options in the string binding are ignored.

Endpoint

Specifies the endpoint, or address, of the process to receive remote procedure calls. An endpoint can be preceded by the keyword **endpoint=**. The endpoint field is optional.

The format and content of an endpoint depend on the specified protocol sequence as follows:

Protocol sequence	Endpoint	Example
ncacn_dnet_nsp	DECnet phase IV object number (must be preceded by the # character)	#17
ncacn_ip_tcp	Internet port number	1025
ncacn_nb_nb, ncacn_nb_tcp	Integer between 0 and 255. Many values between 0 and 32 are reserved by Microsoft.	100
ncacn_np	Windows NT named pipe. Name must start with "\\pipe".	\\pipe\\pipename
ncacn_spx	Integer between 1 and 65535.	5000
ncadg_ip_udp	Internet port number	1025
ncadg_ipx	Integer between 1 and 65535	5000
ncalrpc	String specifying Windows NT object name. The string cannot include any backslash characters.	my_object

Option

Specifies options associated with *Endpoint*. 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 follows:

Protocol sequence	Option name
ncacn_ip_tcp	None
ncacn_nb_nb, ncacn_nb_tcp	None
ncacn_np	Security
ncacn_spx	None
ncadg_ip_udp	None
ncadg_ipx	None
ncalrpc	Security

The Security option name, supported for the **ncalrpc** and **ncacn_np** 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_dnet_nsp:took[elf_server]
obj-uuid@ncacn_dnet_nsp:took[endpoint=elf_server]
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=anonymous dynamic true]
obj-uuid@ncalrpc:
obj-uuid@ncalrpc:[object1_name_demonstrating_that_these_can_be_lengthy]
obj-uuid@ncalrpc:[object2_name,Security=anonymous dynamic true]
```

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 (via mail, on paper, or some other means) the string binding to the client.

See Also

RpcBindingFromStringBinding, **RpcBindingToStringBinding**

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 QuickInfo

The **DceErrorInqText** function returns the message text for a status code.

```
#include <rpc.h>
   RPC_STATUS RPC_ENTRY
   DceErrorInqText(
      unsigned long   StatusToConvert,
      unsigned char *   ErrorText);
```

This function is supported by both the 32-bit Windows NT and Windows 95 platforms. Note that it is supported in ANSI only on the Windows 95 platform.

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

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

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

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

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

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

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

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);
```

Parameter

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

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

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 QuickInfo

The **RpcAbnormalTermination** function determines whether termination statements are being executed due to an exception or not.

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 QuickInfo

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 QuickInfo

The **RpcBindingFree** function releases binding-handle resources.

```
#include <rpc.h>
   RPC_STATUS RPC_ENTRY
   RpcBindingFree(
      RPC_BINDING_HANDLE *    Binding);
```

Parameter

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 QuickInfo

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 QuickInfo

The **RpcBindingInqAuthClient** function returns authentication and authorization information from an authenticated client's binding handle.

```
#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);
```

This function is supported by both the 32-bit Windows NT and Windows 95 platforms.

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 securing package. 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 services, see **RpcMgmtInqDefaultProtectLevel**.

Specify a null value to prevent **RpcBindingInqAuthClient** from returning the *AuthnSvc* argument. This parameter is not used by the RPC_C_AUTHN_WINNT security package. 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 QuickInfo

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 the RPC data types and structures reference entry for 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 **RpcMgmtInqDefaultProtectLevel**.

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 possible returns, see **RpcMgmtInqDefaultProtectLevel**.

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 QuickInfo

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 QuickInfo

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);
```

Parameter

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 QuickInfo

The **RpcBindingServerFromClient** function converts a client binding handle to a server binding handle.

```
#include <rpc.h>
   RPC_STATUS RPC_ENTRY
   RpcBindingServerFromClient(
      RPC_BINDING_HANDLE   ClientBinding,
      RPC_BINDING_HANDLE *   ServerBinding);
```

This function is supported by both the 32-bit Windows NT and Windows 95 platforms.

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.

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.

Note This routine is only supported for TCP or SPX.

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 QuickInfo

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 RPC-supported authentication levels, see the RPC data types and structures reference entry for authentication-level constants.

AuthnSvc
Specifies the authentication service to use. For a list of RPC-supported authentication services, see the RPC data types and structures reference entry for 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 constants for the *AuthzSvc* argument, see the RPC data types and structures reference entry for authorization-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 QuickInfo

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 QuickInfo

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 QuickInfo

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 QuickInfo

The **RpcCancelThread** function cancels a thread.

```
#include <rpc.h>
   RPC_STATUS RPC_ENTRY
   RpcCancelThread(
      HANDLE   ThreadHandle);
```

This function is supported only by 32-bit Windows NT platforms.

Parameter

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 QuickInfo

See

RpcExcept

RpcEndFinally QuickInfo

See

RpcFinally

RpcEpRegister QuickInfo

The **RpcEpRegister** function adds to or replaces server address information in the local endpoint-map database.

```
#include <rpc.h>
   RPC_STATUS RPC_ENTRY
   RpcEpRegister(
      RPC_IF_HANDLE    IfSpec,
      RPC_BINDING_VECTOR *    BindingVector,
      UUID_VECTOR *    UuidVector,
      unsigned char *    Annotation);
```

This function is supported by both the 32-bit Windows NT and Windows 95 platforms.

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**, , **RpcServerUseProtseqIf**

RpcEpRegisterNoReplace QuickInfo

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 QuickInfo

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_BINDING	Wrong kind of binding for operation

See Also

RpcBindingFromStringBinding, RpcBindingReset, RpcEpRegister, RpcEpRegisterNoReplace, RpcNsBindingImportBegin, RpcNsBindingImportDone, RpcNsBindingImportNext

RpcEpUnregister QuickInfo

The **RpcEpUnregister** function removes server-address information from the local endpoint-map database.

```
#include <rpc.h>
  RPC_STATUS RPC_ENTRY
  RpcEpUnregister(
    RPC_IF_HANDLE   IfSpec,
    RPC_BINDING_VECTOR *   BindingVector,
    UUID_VECTOR *   UuidVector);
```

This function is supported by both the 32-bit Windows NT and Windows 95 platforms.

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 QuickInfo

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 QuickInfo

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 QuickInfo

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 QuickInfo

The **RpcIfIdVectorFree** function frees the vector and the interface-identification structures contained in the vector.

```
#include <rpc.h>
   RPC_STATUS RPC_ENTRY
   RpcIfIdVectorFree(
      RPC_IF_ID_VECTOR * *   IfIdVec);
```

This function is supported by both the 32-bit Windows NT and Windows 95 platforms.

Parameter

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 QuickInfo

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 Value

Value	Meaning
RPC_S_OK	Success

See Also

RpcServerInqIf, RpcServerRegisterIf

RpcImpersonateClient QuickInfo

A server thread that is processing client remote procedure calls can call the **RpcImpersonateClient** function to impersonate the active client.

```
#include <rpc.h>
   RPC_STATUS RPC_ENTRY
   RpcImpersonateClient(
      RPC_BINDING_HANDLE        CallHandle);
```

This function is supported only by 32-bit Windows NT platforms.

Parameter

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_NO_CONTEXT_AVAIL	The server does not have permission to impersonate the client

See Also

RpcRevertToSelf

RpcMacSetYieldInfo

#include <rpc.h>
RPC_STATUS RPC_ENTRY
RpcMacSetYieldInfo(
MACYIELDCALLBACK *pfnCallback***);**

Parameter

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 Value

Value	Meaning
RPC_S_OK	The information was set successfully.

RpcMgmtEnableIdleCleanup QuickInfo

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.

```
#include <rpc.h>
   RPC_STATUS RPC_ENTRY
   RpcMgmtEnableIdleCleanup(void);
```

This function is supported by the 32-bit 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.

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 QuickInfo

The **RpcMgmtEpEltInqBegin** function creates an inquiry context for viewing the elements in an endpoint map.

```
#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);
```

This function is supported by both the 32-bit Windows NT and Windows 95 platforms.

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 QuickInfo

The **RpcMgmtEpEltInqDone** function deletes the inquiry context for viewing the elements in an endpoint map.

```
#include <rpc.h>
   RPC_STATUS RPC_ENTRY
   RpcMgmtEpEltInqDone(
      RPC_EP_INQ_HANDLE *    InquiryContext);
```

This function is supported by both the 32-bit Windows NT and Windows 95 platforms.

Parameter

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 QuickInfo

The **RpcMgmtEpEltInqNext** function returns one element from an endpoint map.

```
#include <rpc.h>
  RPC_STATUS RPC_ENTRY
  RpcMgmtEpEltInqNext(
    RPC_EP_INQ_HANDLE   InquiryContext,,
    RPC_IF_ID *  Ifld,
    UUID *   ObjectUuid,
    unsigned char * *   Annotation);
```

This function is supported by both the 32-bit Windows NT and Windows 95 platforms.

Parameters

InquiryContext
Specifies an inquiry context. The inquiry context is returned from **RpcMgmtEpEltInqBegin**.

Ifld
Returns the interface identifier of the endpoint map element.

Binding
Returns the binding handle from the endpoint map element.

ObjectUuid
Returns the object UUID from the endpoint map element.

Annotation
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_S_NO_MORE_ELEMENTS 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 QuickInfo

The **RpcMgmtEpUnregister** function removes server address information from an endpoint map.

```
#include <rpc.h>
  RPC_STATUS RPC_ENTRY
  RpcMgmtEpUnregister(
    RPC_BINDING_HANDLE   EpBinding,
    RPC_IF_ID *   IfId,
    RPC_BINDING_HANDLE  Binding,
    UUID *   ObjectUuid);
```

This function is supported by both the 32-bit Windows NT and Windows 95 platforms.

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 QuickInfo

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 QuickInfo

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	Windows NT 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 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 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_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 For Windows 95 platforms, 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 a Windows 95 client communicating with a Windows NT server. These levels are never supported for a Windows 95 client communicating with a Windows 95 server.

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_SERVICE	Unknown authentication service

RpcMgmtInqIfIds QuickInfo

The **RpcMgmtInqIfIds** function returns a vector containing the identifiers of the interfaces offered by the server.

```
#include <rpc.h>
  RPC_STATUS RPC_ENTRY
  RpcMgmtInqIfIds(
    RPC_BINDING_HANDLE   Binding,
    RPC_IF_ID_VECTOR * *   IfIdVector);
```

This function is supported by both the 32-bit Windows NT and Windows 95 platforms.

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 QuickInfo

The **RpcMgmtInqServerPrincName** function returns a server's principal name.

```
#include <rpc.h>
  RPC_STATUS RPC_ENTRY
  RpcMgmtInqServerPrincName(
    RPC_BINDING_HANDLE   Binding,
    unsigned int   AuthnSvc,
    unsigned char * *   ServerPrincName);
```

This function is supported by both the 32-bit Windows NT and Windows 95 platforms. Note that it is supported only in ANSI on the Windows 95 platform.

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 QuickInfo

The **RpcMgmtInqStats** function returns RPC run-time statistics.

```
#include <rpc.h>
  RPC_STATUS RPC_ENTRY
  RpcMgmtInqStats(
    RPC_BINDING_HANDLE   Binding,
    RPC_STATS_VECTOR * *   Statistics);
```

This function is supported by both the 32-bit Windows NT and Windows 95 platforms.

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_BINDING	Wrong kind of binding for operation

See Also

RpcEpResolveBinding, **RpcMgmtStatsVectorFree**

RpcMgmtIsServerListening QuickInfo

The **RpcMgmtIsServerListening** function tells whether a server is listening for remote procedure calls.

```
#include <rpc.h>
  RPC_STATUS RPC_ENTRY
  RpcMgmtIsServerListening(
    RPC_BINDING_HANDLE   Binding);
```

This function is supported by both the 32-bit Windows NT and Windows 95 platforms.

Parameter

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 QuickInfo

The **RpcMgmtSetAuthorizationFn** function establishes an authorization function for processing remote calls to a server's management routines.

#include <rpc.h>
RPC_STATUS RPC_ENTRY
RpcMgmtSetAuthorizationFn(
RPC_MGMT_AUTHORIZATION_FN *AuthorizationFn***);**

This function is supported only by 32-bit Windows NT platforms.

Parameter

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 QuickInfo

The **RpcMgmtSetCancelTimeout** function sets the lower bound on the time to wait before timing out after forwarding a cancel.

```
#include <rpc.h>
   RPC_STATUS RPC_ENTRY
   RpcMgmtSetCancelTimeout(
      signed int   Seconds);
```

This function is supported only by 32-bit Windows NT platforms.

Parameter

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 QuickInfo

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 QuickInfo

The **RpcMgmtSetServerStackSize** function specifies the stack size for each server thread.

```
#include <rpc.h>
   RPC_STATUS RPC_ENTRY
   RpcMgmtSetServerStackSize(
      unsigned int       ThreadStackSize);
```

This function is supported by both 32-bit Windows NT and Windows 95 platforms.

Parameter

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 QuickInfo

The **RpcMgmtStatsVectorFree** function frees a statistics vector.

```
#include <rpc.h>
   RPC_STATUS RPC_ENTRY
   RpcMgmtStatsVectorFree(
      RPC_STATS_VECTOR * *    StatsVector);
```

This function is supported by both the 32-bit Windows NT and Windows 95 platforms.

Parameter

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 QuickInfo

The **RpcMgmtStopServerListening** function tells a server to stop listening for remote procedure calls.

```
#include <rpc.h>
  RPC_STATUS RPC_ENTRY
  RpcMgmtStopServerListening(
    RPC_BINDING_HANDLE   Binding);
```

This function is supported by both the 32-bit Windows NT and Windows 95 platforms.

Parameter

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

See Also

RpcEpResolveBinding, **RpcMgmtWaitServerListen**, **RpcServerListen**

RpcMgmtWaitServerListen QuickInfo

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 the 32-bit Windows NT and Windows 95 platforms.

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 QuickInfo

The **RpcNetworkInqProtseqs** function returns all protocol sequences supported by both the RPC run-time library and the operating system.

```
#include <rpc.h>
   RPC_STATUS RPC_ENTRY
   RpcNetworkInqProtseqs(
      RPC_PROTSEQ_VECTOR * *   ProtSeqVector);
```

This function is supported by both the 32-bit Windows NT and Windows 95 platforms.

For a list of procotol sequences supported by RPC, see RPC Data Types and Structures.

Parameter

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 QuickInfo

The **RpcNetworkIsProtseqValid** function tells whether the specified protocol sequence is supported by both the RPC run-time library and the operating system.

```
#include <rpc.h>
   RPC_STATUS RPC_ENTRY
   RpcNetworkIsProtseqValid(
      unsigned char *   Protseq);
```

For a list of procotol sequences supported by RPC, see RPC Data Types and Structures.

Parameter

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 RPC-supported protocol sequences, see String Binding.

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 QuickInfo

The **RpcNsBindingExport** function establishes a name-service database entry with multiple binding handles and multiple objects for a server.

```
#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);
```

This function is supported by both the 32-bit Windows NT and Windows 95 platforms.

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 QuickInfo

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 QuickInfo

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);
```

Parameter

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 QuickInfo

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 QuickInfo

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 QuickInfo

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 QuickInfo

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);
```

Parameter

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 QuickInfo

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 QuickInfo

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 QuickInfo

The **RpcNsBindingUnexport** function removes the binding handles for an interface and objects from an entry in the name-service database.

```
#include <rpc.h>
   RPC_STATUS RPC_ENTRY
   RpcNsBindingUnexport(
      unsigned long   EntryNameSyntax,
      unsigned char *   EntryName,
      RPC_IF_HANDLE   IfSpec,
      UUID_VECTOR *   ObjectUuidVec);
```

This function is supported by both the 32-bit Windows NT and Windows 95 platforms.

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 QuickInfo

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 QuickInfo

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 QuickInfo

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);
```

Parameter

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 QuickInfo

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 QuickInfo#include <rpc.h>

```
RPC_STATUS RPC_ENTRY
RpcNsGroupDelete(
    unsigned long    GroupNameSyntax,
    unsigned char *  GroupName);
```

The **RpcNsGroupDelete** function deletes a group attribute.

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 QuickInfo#include <rpc.h>

```
RPC_STATUS RPC_ENTRY
RpcNsGroupMbrAdd(
    unsigned long   GroupNameSyntax,
    unsigned char *   GroupName,
    unsigned long   MemberNameSyntax,
    unsigned char *   MemberName);
```

The **RpcNsGroupMbrAdd** function adds an entry name to a group. If necessary, it creates the entry.

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

QuickInfo#include <rpc.h>

```
RPC_STATUS RPC_ENTRY
RpcNsGroupMbrInqBegin(
    unsigned long    GroupNameSyntax,
    unsigned char *  GroupName,
    unsigned long    MemberNameSyntax,
    RPC_NS_HANDLE *  InquiryContext);
```

The **RpcNsGroupMbrInqBegin** function creates an inquiry context for viewing group members.

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

QuickInfo#include <rpc.h>

```
RPC_STATUS RPC_ENTRY
RpcNsGroupMbrInqDone(
    RPC_NS_HANDLE *    InquiryContext);
```

The **RpcNsGroupMbrInqDone** function deletes the inquiry context for a group.

Parameter

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

QuickInfo#include <rpc.h>

```
RPC_STATUS RPC_ENTRY
RpcNsGroupMbrInqNext(
    RPC_NS_HANDLE    InquiryContext,
    unsigned char * *    MemberName);
```

The **RpcNsGroupMbrInqNext** function returns one entry name from a group at a time.

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 QuickInfo#include <rpc.h>

```
RPC_STATUS RPC_ENTRY
RpcNsGroupMbrRemove(
    unsigned long   GroupNameSyntax,
    unsigned char *   GroupName,
    unsigned long   MemberNameSyntax,
    unsigned char *   MemberName);
```

The **RpcNsGroupMbrRemove** function removes an entry name from a group.

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 QuickInfo

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 QuickInfo

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 QuickInfo

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 QuickInfo

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 QuickInfo

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 QuickInfo

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);
```

Parameter

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 Value

Value	Meaning
RPC_S_OK	Success

See Also

RpcNsMgmtHandleSetExpAge, **RpcNsMgmtSetExpAge**

RpcNsMgmtSetExpAge QuickInfo

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);
```

Parameter

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_UNAVAILABLE	Name service unavailable

See Also

RpcNsMgmtHandleSetExpAge

RpcNsProfileDelete QuickInfo

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 QuickInfo

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 QuickInfo

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 QuickInfo

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);
```

Parameter

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 Value

Value	Meaning
RPC_S_OK	Success

See Also

RpcNsProfileEltInqBegin, **RpcNsProfileEltInqNext**

RpcNsProfileEltInqNext QuickInfo

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 QuickInfo

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 QuickInfo

The **RpcObjectInqType** function returns the type of an object.

```
#include <rpc.h>
  RPC_STATUS RPC_ENTRY
  RpcObjectInqType(
    UUID *  ObjUuid,
    UUID *  TypeUuid);
```

This function is supported by both the 32-bit Windows NT and Windows 95 platforms.

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 QuickInfo

The **RpcObjectSetInqFn** function registers an object-inquiry function. A null value turns off a previously registered object-inquiry function.

#include <rpc.h>
RPC_STATUS RPC_ENTRY
RpcObjectSetInqFn(
RPC_OBJECT_INQ_FN *InquiryFn*);

This function is supported by both the 32-bit Windows NT and Windows 95 platforms.

Parameter

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 Value

Value	Meaning
RPC_S_OK	Success

See Also

RpcObjectInqType, RpcObjectSetType

RpcObjectSetType QuickInfo

The **RpcObjectSetType** function assigns the type of an object.

```
#include <rpc.h>
   RPC_STATUS RPC_ENTRY
   RpcObjectSetType(
      UUID *   ObjUuid,
      UUID *   TypeUuid);
```

This function is supported by both the 32-bit Windows NT and Windows 95 platforms.

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 QuickInfo

The **RpcProtseqVectorFree** function frees the protocol sequences contained in the vector and the vector itself.

```
#include <rpc.h>
   RPC_STATUS RPC_ENTRY
   RpcProtseqVectorFree(
      RPC_PROTSEQ_VECTOR * *   ProtSeqVector);
```

This function is supported by both the 32-bit Windows NT and Windows 95 platforms.

For a list of procotol sequences supported by RPC, see RPC Data Types and Structures.

Parameter

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 Value

Value	Meaning
RPC_S_OK	Success

See Also

RpcNetworkInqProtseqs

RpcRaiseException QuickInfo

Use the **RpcRaiseException** function to raise an exception. The **RpcRaiseException** function does not return to the caller.

void RPC_ENTRY
RpcRaiseException (
RPC_STATUS *Exception*);

Parameter

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 QuickInfo

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.

```
#include <rpc.h>
   RPC_STATUS RPC_ENTRY
   RpcRevertToSelf (void);
```

This function is supported only by 32-bit Windows NT platforms.

Return Values

Value	Meaning
RPC_S_OK	Success
RPC_S_NO_CALL_ACTIVE	Server does not have a client to impersonate
RPC_S_CANNOT_SUPPORT	Not supported for this operating system, this transport, or this security subsystem

See Also

RpcImpersonateClient

RpcServerInqBindings QuickInfo

The **RpcServerInqBindings** function returns the binding handles over which remote procedure calls can be received.

```
#include <rpc.h>
   RPC_STATUS RPC_ENTRY
   RpcServerInqBindings(
      RPC_BINDING_VECTOR * *   BindingVector);
```

This function is supported by both the 32-bit Windows NT and Windows 95 platforms.

Parameter

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.

```
#include <rpc.h>
  RPC_STATUS RPC_ENTRY
  RpcServerInqDefaultPrincName(
    unsigned long   AuthnSvc,
    RPC_CHAR * *    PrincName,
    );
```

This function is supported only by Windows 95 platforms.

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 QuickInfo

The **RpcServerInqIf** function returns the manager entry-point vector (EPV) registered for an interface.

```
#include <rpc.h>
  RPC_STATUS RPC_ENTRY
  RpcServerInqIf(
    RPC_IF_HANDLE   IfSpec,
    UUID *   MgrTypeUuid,
    RPC_MGR_EPV * *   MgrEpv);
```

This function is supported by both the 32-bit Windows NT and Windows 95 platforms.

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 QuickInfo

The **RpcServerListen** function tells the RPC run-time library to listen for remote procedure calls.

```
#include <rpc.h>
  RPC_STATUS RPC_ENTRY
  RpcServerListen(
    unsigned int     MinimumCallThreads,
    unsigned int     MaxCalls,
    unsigned int     DontWait);
```

This function is supported by both the 32-bit Windows NT and Windows 95 platforms.

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**, **RpcServerUseAllProtseqs**, **RpcServerUseAllProtseqsIf**, **RpcServerUseProtseq**, **RpcServerUseProtseqEp**, **RpcServerUseProtseqIf**

RpcServerRegisterAuthInfo QuickInfo

The **RpcServerRegisterAuthInfo** function registers authentication information with the RPC run-time library.

```
#include <rpc.h>
  RPC_STATUS RPC_ENTRY
  RpcServerRegisterAuthInfo(
    unsigned char *   ServerPrincName,
    unsigned long   AuthnSvc,
    RPC_AUTH_KEY_RETRIEVAL_FN   GetKeyFn,
    void *   Arg);
```

This function is supported by both the 32-bit Windows NT and Windows 95 platforms.

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 QuickInfo

The **RpcServerRegisterIf** function registers an interface with the RPC run-time library.

```
#include <rpc.h>
  RPC_STATUS RPC_ENTRY
  RpcServerRegisterIf(
    RPC_IF_HANDLE   IfSpec,
    UUID *   MgrTypeUuid,
    RPC_MGR_EPV *   MgrEpv);
```

This function is supported by both the 32-bit Windows NT and Windows 95 platforms.

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.

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 **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

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 **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 **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 depicted in the preceding tables when called by a client with interface UUID *uuid2* and object UUID *uuidC*:

1. The server calls **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**, **RpcServerUnregisterIf**

RpcServerUnregisterIf QuickInfo

The **RpcServerUnregisterIf** function unregisters an interface from the RPC run-time library.

```
#include <rpc.h>
  RPC_STATUS RPC_ENTRY
  RpcServerUnregisterIf(
    RPC_IF_HANDLE   IfSpec,
    UUID *   MgrTypeUuid,
    unsigned int        WaitForCallsToComplete);
```

This function is supported by both the 32-bit Windows NT and Windows 95 platforms.

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** 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.

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

RpcServerUseAllProtseqs QuickInfo

The **RpcServerUseAllProtseqs** function tells the RPC run-time library to use all supported protocol sequences for receiving remote procedure calls.

```
#include <rpc.h>
   RPC_STATUS RPC_ENTRY
   RpcServerUseAllProtseqs(
      unsigned int       MaxCalls,
      void *   SecurityDescriptor);
```

This function is supported by both the 32-bit Windows NT and Windows 95 platforms.

For a list of procotol sequences supported by RPC, see RPC Data Types and Structures.

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.

Remarks

Note The Microsoft RPC implementation of **RpcServerUseAllProtseqs** includes a new, additional parameter, *SecurityDescriptor*, that does not appear in the DCE specification.

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.

After registering protocol sequences, a server typically calls the following routines:

- **RpcServerInqBindings** to obtain a vector containing all of the server's binding handles.
- **RpcEpRegister** or **RpcEpRegisterNoReplace** to register the binding handles with the endpoint-mapping service. During implementation and debugging, server developers can communicate their binding information to clients using string bindings. This allows them 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, via mail, on paper, or by some other means, the string binding to the client.
- **RpcNsBindingExport** to place the binding handles in the name-service database for access by any client.
- **RpcBindingVectorFree** to free the vector of server binding handles.
- **RpcServerRegisterIf** to register the interfaces offered by the server with the RPC run-time library. This is a required call.

- **RpcServerListen** to begin receiving remote procedure call requests. This is a required call.

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**

RpcServerUseAllProtseqsIf QuickInfo

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.

```
#include <rpc.h>
  RPC_STATUS RPC_ENTRY
  RpcServerUseAllProtseqsIf(
    unsigned int   MaxCalls,
    RPC_IF_HANDLE   IfSpec,
    void *   SecurityDescriptor);
```

This function is supported by both the 32-bit Windows NT and Windows 95 platforms.

For a list of procotol sequences supported by RPC, see RPC Data Types and Structures.

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.

Remarks

Note The Microsoft RPC implementation of **RpcServerUseAllProtseqsIf** includes a new, additional parameter, *SecurityDescriptor*, that does not appear in the DCE specification.

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 **RpcServerUseAllProtseqs** for the list of routines a server typically calls after calling **RpcServerUseAllProtseqsIf**.

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**, **RpcServerRegisterIf**, **RpcServerUseAllProtseqs**, **RpcServerUseProtseq**, **RpcServerUseProtseqEp**, **RpcServerUseProtseqIf**

RpcServerUseProtseq QuickInfo

The **RpcServerUseProtseq** function tells the RPC run-time library to use the specified protocol sequence for receiving remote procedure calls.

```
#include <rpc.h>
   RPC_STATUS RPC_ENTRY
   RpcServerUseProtseq(
      unsigned char *   ProtSeq,
      unsigned int      MaxCalls,
      void *   SecurityDescriptor);
```

This function is supported by both the 32-bit Windows NT and Windows 95 platforms.

For a list of procotol sequences supported by RPC, see RPC Data Types and Structures.

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.

Remarks

Note The Microsoft RPC implementation of **RpcServerUseProtseq** includes a new, additional parameter, *SecurityDescriptor*, that does not appear in the DCE specification.

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 **RpcServerUseAllProtseqs** for the list of routines a server typically calls after calling **RpcServerUseProtseq**.

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**, **RpcServerRegisterIf**, **RpcServerUseAllProtseqs**, **RpcServerUseAllProtseqsIf**, **RpcServerUseProtseqEp**, **RpcServerUseProtseqIf**

RpcServerUseProtseqEp QuickInfo

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.

```
#include <rpc.h>
   RPC_STATUS RPC_ENTRY
   RpcServerUseProtseqEp(
      unsigned char *   Protseq,
      unsigned int      MaxCalls,
      unsigned char *   Endpoint,
      void *    SecurityDescriptor);
```

This function is supported by both the 32-bit Windows NT and Windows 95 platforms.

For a list of procotol sequences supported by RPC, see RPC Data Types and Structures.

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.

Remarks

Note The Microsoft RPC implementation of **RpcServerUseProtseqEp** includes a new, additional parameter, *SecurityDescriptor*, that does not appear in the DCE specification.

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 **RpcServerUseAllProtseqs** for the list of routines a server typically calls after calling **RpcServerUseProtseqEp**.

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**

RpcServerUseProtseqIf QuickInfo

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.

```
#include <rpc.h>
   RPC_STATUS RPC_ENTRY
   RpcServerUseProtseqIf(
      unsigned char *   Protseq,
      unsigned int      MaxCalls,
      RPC_IF_HANDLE     IfSpec,
      void *    SecurityDescriptor);
```

This function is supported by both the 32-bit Windows NT and Windows 95 platforms.

For a list of procotol sequences supported by RPC, see RPC Data Types and Structures.

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 **RpcServerUseAllProtseqs** for the list of routines a server typically calls after calling **RpcServerUseProtseqIf**.

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**

RpcSmAllocate

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

The **RpcSmClientFree** function frees memory returned from a client stub.

```
#include <rpc.h>
   RPC_STATUS RPC_ENTRY
   RpcSmClientFree(
      void   *   NodeToFree);
```

Parameter

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 Value

Value	Meaning
RPC_S_OK	Success

See Also

RpcSmFree, **RpcSmGetThreadHandle**, **RpcSmSetClientAllocFree**, **RpcSmSetThreadHandle**, **RpcSmSwapClientAllocFree**

RpcSmDestroyClientContext

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);
```

Parameter

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

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 Value

Value	Meaning
RPC_S_OK	Success

See Also

RpcSmAllocate, **RpcSmEnableAllocate**

RpcSmEnableAllocate

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, the **RpcSmEnableAllocate** routine is called by applications to establish the stub memory management environment. This environment must be established prior to making a call to **RpcSmAllocate**. In default 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. For **ms_ext** or **c_ext** modes, 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

The **RpcSmFree** function releases memory allocated by **RpcSmAllocate**.

```
#include <rpc.h>
   RPC_STATUS RPC_ENTRY
   RpcSmFree(
      void   *   NodeToFree);
```

Parameter

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 Value

Value	Meaning
RPC_S_OK	Success

See Also

RpcSmAllocate, **RpcSmGetThreadHandle**, **RpcSmSetThreadHandle**

RpcSmGetThreadHandle

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)
```

Parameter

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 Value

Value	Meaning
RPC_S_OK	Success

See Also

RpcSmAllocate, **RpcSmFree**, **RpcSmSetThreadHandle**

RpcSmSetClientAllocFree

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

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);
```

Parameter

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 Value

Value	Meaning
RPC_S_OK	Success

See Also

RpcSmAllocate, **RpcSmGetThreadHandle**, **RpcSmFree**

RpcSmSwapClientAllocFree

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

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);
```

Parameter

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 Value

Value	Meaning
RPC_S_OUT_OF_MEMORY	Out of memory

See Also

RpcSmAllocate, **RpcSsDisableAllocate**, **RpcSsEnableAllocate**, **RpcSsFree**, **RpcSsGetThreadHandle**, **RpcSsSetThreadHandle**

RpcSsDestroyClientContext QuickInfo

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,
   );
```

Parameter

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

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**

RpcSsEnableAllocate

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 Value

Value	Meaning
RPC_S_OUT_OF_MEMORY	Out of memory

See Also

RpcSmEnableAllocate, **RpcSsAllocate**, **RpcSsDisableAllocate**

RpcSsFree

The **RpcSsFree** function releases memory allocated by **RpcSsAllocate**.

```
#include <rpc.h>
   void RPC_ENTRY
   RpcSsFree(
      void *     NodeToFree);
```

Parameter

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

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 Value

Value	Meaning
RPC_S_OK	Success

See Also

RpcSmGetThreadHandle, **RpcSsAllocate**, **RpcSsFree**, **RpcSsSetThreadHandle**

RpcSsSetClientAllocFree

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 Value

Value	Meaning
RPC_S_OUT_OF_MEMORY	Out of memory

See Also

RpcSmSetClientAllocFree, **RpcSsAllocate**, **RpcSsFree**

RpcSsSetThreadHandle

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);
```

Parameter

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

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 QuickInfo

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 more information, see the RPC data types and structures reference entry for string binding. For a list of procotol sequences supported by RPC, see 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 QuickInfo

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 procotol sequences supported by RPC, see 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 QuickInfo

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);
```

Parameter

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 Value

Value	Meaning
RPC_S_OK	Success

See Also

RpcBindingToStringBinding, **RpcNsBindingInqEntryName**, **RpcStringBindingParse**

RpcTestCancel QuickInfo

The **RpcTestCancel** function checks for a cancel indication.

```
#include <rpc.h>
   RPC_STATUS RPC_ENTRY
   RpcTestCancel(
      );
```

This function is supported only by 32-bit Windows NT platforms.

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 QuickInfo

See

RpcExcept

RpcTryFinally QuickInfo

See

RpcFinally

RpcWinSetYieldInfo

The **RpcWinSetYieldInfo** function configures Microsoft Windows 3.x client applications to yield to other applications during remote procedure calls.

```
#include <rpc.h>
  RPC_STATUS
  RpcWinSetYieldInfo(HWND      hWnd,
    BOOL   fCustomYield,
    WORD   wMsg,
    DWORD   dwOtherInfo);
```

This function is only supported by Windows 3.*x* applications.

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.

Yielding is not necessary for Win32 applications. Environments that support Win32 applications are preemptively multitasked and are multithreaded. For efficiency, a Win32-only application should not call **RpcWinSetYieldInfo**.

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

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 supported by Windows 3.*x* applications.

```
#include <rpc.h>
   RPC_STATUS RPC_ENTRY
   RpcWinSetYieldTimeout (
   unsigned int        Timeout );
```

Parameter

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 QuickInfo

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 QuickInfo

The **UuidCreate** function creates a new UUID.

```
#include <rpc.h>
   RPC_STATUS RPC_ENTRY
   UuidCreate(
      UUID *    Uuid);
```

Parameter

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 QuickInfo

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 QuickInfo

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 QuickInfo

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 QuickInfo

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 QuickInfo

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 QuickInfo

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.

Legal Information

CLUUID

This directory contains the files for the sample distributed application "cluuid":

File Description

README.TXT Readme file for the cluuid sample CLUUID.IDL Interface definition language file CLUUID.ACF Attribute configuration file CLUUIDC.C Client main program CLUUIDS.C Server main program CLUUIDP.C Remote procedures MAKEFILE Nmake file to build for NT MAKEFILE.DOS Nmake file to build for MS-DOS

This sample program demonstrates how to supply multiple implementations of the remote procedure specified in the interface. It also demonstrates how the client selects among the implementations by providing a client object uuid.

The server calls RpcObjectSetType to associate a client object uuid with the object uuid in the Object Registry Table. The server initializes a manager entry point vector (manager epv) and then calls RpcRegisterIf to associate the interface uuid and the object uuid with the manager epv in the Interface Registry Table.

When the client makes a remote procedure call, the client object uuid is mapped to the object uuid in the Object Registry Table. The resulting object uuid and the interface uuid are mapped to a manager entry point vector in the Interface Registry Table.

By default, in this example, the server registers two implementations of the "hello, world" function HelloProc and HelloProc2. The HelloProc2 implementation is associated with the object uuid "11111111-1111-1111-1111-111111111111". When the client makes a procedure call with a null uuid, the client's request is mapped to the original HelloProc. When the client makes a procedure call with the client object uuid "11111111-1111-1111-1111-11111111111", the client's request is mapped to HelloProc2 (which prints the string in reverse).

---------- BUILDING CLIENT AND SERVER APPLICATIONS FOR MICROSOFT WINDOWS NT: ----------

The following environment variables should be set for you already. set CPU=i386

```
set INCLUDE=c:\mstools\h
set LIB=c:\mstools\lib
set PATH=c:\winnt\system32;c:\mstools\bin;
```

For mips, set CPU=mips For alpha, set CPU=alpha

Build the sample distributed application:

```
nmake cleanall
nmake
```

This builds the executable programs cluuidc.exe (client) and cluuids.exe (server).

---------- BUILDING THE CLIENT APPLICATION FOR MS-DOS ----------

After installing the Microsoft Visual C/C++ version 1.50 development environment and the Microsoft RPC version 2.0 toolkit on a Windows NT computer, you can build the sample client application from Windows NT.

```
nmake -f makefile.dos cleanall
nmake -f makefile.dos
```

This builds the client application cluuidc.exe.

You may also execute the Microsoft Visual C/C++ compiler under MS-DOS. This requires a two step build process.

Step One: Compile the .IDL files under Windows NT
```
nmake -a -f makefile.dos cluuid.h
```

Step Two: Compile the C sources (stub and application) under MS-DOS
```
nmake -f makefile.dos
```

---------- RUNNING THE CLIENT AND SERVER APPLICATIONS ----------

On the server, enter

```
cluuids
```

On the client, enter

```
net start workstation
cluuidc
```

To call the second implementation of the function, on the client, enter

```
cluuidc -u "11111111-1111-1111-1111-111111111111"
```

Note: The client and server applications can run on the same Microsoft Windows NT computer when you use different screen groups.

Several command line switches are available to change settings for this program. For a listing of the switches available from the client program, enter

```
cluuidc -?
```

For a listing of switches available from the server program, enter

```
cluuids -?
```

MAKEFILE (CLUUID RPC Sample)

```
#************************************************************#
#**                                                        **#
#**                  Microsoft RPC Examples                **#
#**                    cluuid Application                  **#
#**            Copyright(c) Microsoft Corp. 1992           **#
#**                                                        **#
#************************************************************#

!include <ntwin32.mak>


all : cluuidc cluuids

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

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

# cluuidc stub
cluuid_c.obj : cluuid_c.c cluuid.h
   $(cc) $(cdebug) $(cflags) $(cvars) $*.c

# Make the server side application
cluuids : cluuids.exe
cluuids.exe : cluuids.obj cluuidp.obj cluuid_s.obj
    $(link) $(linkdebug) $(conflags) -out:cluuids.exe \
      cluuids.obj cluuid_s.obj cluuidp.obj \
      rpcrt4.lib $(conlibsmt)

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

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

# cluuids stub file
cluuid_s.obj : cluuid_s.c cluuid.h
   $(cc) $(cdebug) $(cflags) $(cvarsmt) $*.c

# Stubs and header file from the IDL file
cluuid.h cluuid_c.c cluuid_s.c : cluuid.idl cluuid.acf
    midl -oldnames -use_epv -no_cpp cluuid.idl

# Clean up everything
```

```
cleanall : clean
    -del *.exe

# Clean up everything but the .EXEs
clean :
    -del *.obj
    -del *.map
    -del cluuid_c.c
    -del cluuid_s.c
    -del cluuid.h
```

CLUUIDC.C (CLUUID RPC Sample)

```
/
*********************************************************************
                  Microsoft RPC Version 2.0
           Copyright Microsoft Corp. 1992, 1993, 1994
                      Cluuid Example

    FILE:        cluuidc.c

    USAGE:       cluuidc  -n network_address
                          -p protocol_sequence
                          -e endpoint
                          -o options
                          -s string_displayed_on_server
                          -u client object uuid

    PURPOSE:     Client side of RPC distributed application

    FUNCTIONS:   main() - binds to server and calls remote procedure

    COMMENTS:    This distributed application prints a string such as
                 "hello, world" on the server. The client manages its
                 connection to the server. The client uses the implicit
                 binding handle ImpHandle defined in the file cluuid.h.

*********************************************************************
/

#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include "cluuid.h"    // header file generated by MIDL compiler

void Usage(char * pszProgramName)
{
    fprintf(stderr, "Usage:  %s\n", pszProgramName);
    fprintf(stderr, " -p protocol_sequence\n");
    fprintf(stderr, " -n network_address\n");
    fprintf(stderr, " -e endpoint\n");
    fprintf(stderr, " -o options\n");
    fprintf(stderr, " -s string\n");
    fprintf(stderr, " -u uuid\n");
    exit(1);
}

void _CRTAPI1 main(int argc, char **argv)
{
    RPC_STATUS status;
    unsigned char * pszUuid             = NULL;
    unsigned char * pszProtocolSequence = "ncacn_np";
    unsigned char * pszNetworkAddress   = NULL;
    unsigned char * pszEndpoint         = "\\pipe\\cluuid";
    unsigned char * pszOptions          = NULL;
```

```
unsigned char * pszStringBinding    = NULL;
unsigned char * pszString           = "hello, world";
unsigned long ulCode;
int i;

/* allow the user to override settings with command line switches */
for (i = 1; i < argc; i++) {
    if ((*argv[i] == '-') || (*argv[i] == '/')) {
        switch (tolower(*(argv[i]+1))) {
        case 'p':  // protocol sequence
            pszProtocolSequence = argv[++i];
            break;
        case 'n':  // network address
            pszNetworkAddress = argv[++i];
            break;
        case 'e':
            pszEndpoint = argv[++i];
            break;
        case 'o':
            pszOptions = argv[++i];
            break;
        case 's':
            pszString = argv[++i];
            break;
        case 'u':
            pszUuid = argv[++i];
            break;
        case 'h':
        case '?':
        default:
            Usage(argv[0]);
        }
    }
    else
        Usage(argv[0]);
}

/* Use a convenience function to concatenate the elements of  */
/* the string binding into the proper sequence.               */
status = RpcStringBindingCompose(pszUuid,
                                 pszProtocolSequence,
                                 pszNetworkAddress,
                                 pszEndpoint,
                                 pszOptions,
                                 &pszStringBinding);
printf("RpcStringBindingCompose returned 0x%x\n", status);
printf("pszStringBinding = %s\n", pszStringBinding);
if (status) {
    exit(status);
}

/* Set the binding handle that will be used to bind to the server. */
status = RpcBindingFromStringBinding(pszStringBinding,
                                     &ImpHandle);
printf("RpcBindingFromStringBinding returned 0x%x\n", status);
```

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

    printf("Calling the remote procedure 'HelloProc'\n");
    printf("  print the string '%s' on the server\n", pszString);

    RpcTryExcept {
        HelloProc(pszString);  /* make call with user message */
        printf("Calling the remote procedure 'Shutdown'\n");
        Shutdown();             // shut down the server side
    }
    RpcExcept(1) {
        ulCode = RpcExceptionCode();
        printf("Runtime reported exception 0x%lx = %ld\n", ulCode, ulCode);
    }
    RpcEndExcept

    /*  The calls to the remote procedures are complete. */
    /*  Free the string and the binding handle           */

    status = RpcStringFree(&pszStringBinding);  // remote calls done; unbind
    printf("RpcStringFree returned 0x%x\n", status);
    if (status) {
        exit(status);
    }

    status = RpcBindingFree(&ImpHandle);  // remote calls done; unbind
    printf("RpcBindingFree returned 0x%x\n", status);
    if (status) {
        exit(status);
    }

    exit(0);
}


/*********************************************************************/
/*                 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);
}


/* end file cluuidc.c */
```

CLUUIDP.C (CLUUID RPC Sample)

```
/
*********************************************************************
                  Microsoft RPC Version 2.0
          Copyright Microsoft Corp. 1992, 1993, 1994
                       Cluuid Example

    FILE:       cluuidp.c

    PURPOSE:    Remote procedures that are linked with the server
                side of RPC distributed application

    FUNCTIONS:  HelloProc() - prints "hello, world" or other string
                HelloProc2() - prints string backwards

    COMMENTS:   This sample program demonstrates how to supply
                multiple implementations of the remote procedure
                specified in the interface. It also demonstrates
                how the client selects among the implementations
                by providing a client object uuid.

                The server calls RpcObjectSetType to associate a
                client object uuid with the object uuid in the
                Object Registry Table. The server initializes a
                manager entry point vector (manager epv) and
                then calls RpcRegisterIf to associate the interface
                uuid and the object uuid with the manager epv in the
                Interface Registry Table.

                When the client makes a remote procedure call,
                the client object uuid is mapped to the object uuid
                in the Object Registry Table. The resulting
                object uuid and the interface uuid are mapped to
                a manager entry point vector in the Interface
                Registry Table.

                By default, in this example, the server registers
                two implementations of the "hello, world" function
                HelloProc and HelloProc2.  The HelloProc2
                implementation is associated with the object uuid
                "11111111-1111-1111-1111-111111111111". When
                the client makes a procedure call with a null
                uuid, the client's request is mapped to the
                original HelloProc.  When the client makes a
                procedure call with the client object uuid
                "11111111-1111-1111-1111-11111111111", the
                client's request is mapped to HelloProc2 (which
                prints the string in reverse).

*********************************************************************
/

#include <stdlib.h>
```

```
#include <stdio.h>
#include <string.h>
#include "cluuid.h"    // header file generated by MIDL compiler

void HelloProc(unsigned char * pszString)
{
    printf("%s\n", pszString);
}

void HelloProc2(unsigned char * pszString)
{
    printf("%s\n", strrev(pszString));
}

void Shutdown(void)
{
    RPC_STATUS status;

    printf("Calling RpcMgmtStopServerListening\n");
    status = RpcMgmtStopServerListening(NULL);
    printf("RpcMgmtStopServerListening returned: 0x%x\n", status);
    if (status) {
        exit(status);
    }

    printf("Calling RpcServerUnregisterIf\n");
    status = RpcServerUnregisterIf(NULL, NULL, FALSE);
    printf("RpcServerUnregisterIf returned 0x%x\n", status);
    if (status) {
        exit(status);
    }
}

/* end of file cluuidp.c */
```

CLUUIDS.C (CLUUID RPC Sample)

```
/
*********************************************************************
                  Microsoft RPC Version 2.0
          Copyright Microsoft Corp. 1992, 1993, 1994
                     Cluuid Example

    FILE:       cluuids.c

    USAGE:      cluuids  -p protocol_sequence
                         -e endpoint
                         -m max calls
                         -n min calls
                         -f flag for RpcServerListen
                         -1 client object uuid
                         -2 manager epv uuid

    PURPOSE:    Server side of RPC distributed application hello

    FUNCTIONS:  main() - registers server as RPC server

    COMMENTS:   Print "hello, world" on the server.
                When you supply a type UUID, the client must
                supply the same UUID.

*********************************************************************
/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include "cluuid.h"    // header file generated by MIDL compiler

// the second implementation of the remote procedure
extern void HelloProc2(unsigned char * pszString);

#define PURPOSE \
"This Microsoft RPC Version 2.0 sample program demonstrates how\n\
to supply multiple implementations of the remote procedure\n\
specified in the interface. It also demonstrates how the client\n\
selects among the implementations by providing a client object uuid.\n\n"

#define NULL_UUID_STRING "00000000-0000-0000-0000-000000000000"

void Usage(char * pszProgramName)
{
    fprintf(stderr, "%s", PURPOSE);
    fprintf(stderr, "Usage:  %s\n", pszProgramName);
    fprintf(stderr, " -p protocol_sequence\n");
    fprintf(stderr, " -e endpoint\n");
    fprintf(stderr, " -m maxcalls\n");
    fprintf(stderr, " -n mincalls\n");
```

```
    fprintf(stderr, " -f flag_wait_op\n");
    fprintf(stderr, " -1 client uuid\n");
    fprintf(stderr, " -2 manager uuid\n");
    exit(1);
}

/* main:  register the interface, start listening for clients */
void _CRTAPI1 main(int argc, char * argv[])
{
    RPC_STATUS status;
    UUID MgrTypeUuid, ClientUuid;
    unsigned char * pszProtocolSequence = "ncacn_np";
    unsigned char * pszSecurity         = NULL;
    unsigned char * pszClientUuid       = NULL_UUID_STRING;
    unsigned char * pszMgrTypeUuid      = "11111111-1111-1111-1111-
111111111111";
    unsigned char * pszEndpoint         = "\\pipe\\cluuid";
    unsigned int    cMinCalls           = 1;
    unsigned int    cMaxCalls           = 20;
    unsigned int    fDontWait           = FALSE;
    int i;

    cluuid_SERVER_EPV epv2;    // the mgr_epv for the 2nd implementation

    /* allow the user to override settings with command line switches */
    for (i = 1; i < argc; i++) {
        if ((*argv[i] == '-') || (*argv[i] == '/')) {
            switch (tolower(*(argv[i]+1))) {
            case 'p':  // protocol sequence
                pszProtocolSequence = argv[++i];
                break;
            case 'e':
                pszEndpoint = argv[++i];
                break;
            case 'm':
                cMaxCalls = (unsigned int) atoi(argv[++i]);
                break;
            case 'n':
                cMinCalls = (unsigned int) atoi(argv[++i]);
                break;
            case 'f':
                fDontWait = (unsigned int) atoi(argv[++i]);
                break;
            case '1':
                pszMgrTypeUuid = argv[++i];
                break;
            case '2':
                pszClientUuid = argv[++i];
                break;
            case 'h':
            case '?':
            default:
                Usage(argv[0]);
            }
        }
```

```
        else
            Usage(argv[0]);
    }

    status = RpcServerUseProtseqEp(pszProtocolSequence,
                                   cMaxCalls,
                                   pszEndpoint,
                                   pszSecurity);  // Security descriptor
    printf("RpcServerUseProtseqEp returned 0x%x\n", status);
    if (status) {
        exit(status);
    }

    status = UuidFromString(pszClientUuid, &ClientUuid);
    printf("UuidFromString returned 0x%x = %d\n", status, status);
    if (status) {
        exit(status);
    }

    status = UuidFromString(pszMgrTypeUuid, &MgrTypeUuid);
    printf("UuidFromString returned 0x%x = %d\n", status, status);
    if (status) {
        exit(status);
    }
    if (strcmp (pszMgrTypeUuid, NULL_UUID_STRING) == 0) {
        printf("Register object using non-null uuid %s\n", pszMgrTypeUuid);
        exit(1);
    }

    if (strcmp (pszClientUuid, NULL_UUID_STRING) == 0) {
        printf("Register object using non-null uuid %s\n", pszMgrTypeUuid);
        ClientUuid = MgrTypeUuid;
    }

    RpcObjectSetType(&ClientUuid, &MgrTypeUuid);  // associate type UUID
with nil UUID
    printf("RpcObjectSetType returned 0x%x\n", status);
    if (status) {
        exit(status);
    }

    status = RpcServerRegisterIf(cluuid_ServerIfHandle, // interface to
register
                                 NULL,   // MgrTypeUuid
                                 NULL);  // MgrEpv; null means use default
    printf("RpcServerRegisterIf returned 0x%x\n", status);
    if (status) {
        exit(status);
    }

    /* register the second manager epv and associate it with the
       specified uuid.  the second uuid must be non-null so that
       it will not conflict with the NULL uuid already registered
       for this interface
```

```
    */
    epv2.HelloProc = HelloProc2;
    epv2.Shutdown = Shutdown;
    status = RpcServerRegisterIf(cluuid_ServerIfHandle, // interface to
register
                                 &MgrTypeUuid,  // MgrTypeUuid
                                 &epv2);        // 2nd manager epv
    printf("RpcServerRegisterIf returned 0x%x\n", status);
    if (status) {
        exit(status);
    }

    printf("Calling RpcServerListen\n");
    status = RpcServerListen(cMinCalls,
                             cMaxCalls,
                             fDontWait);
    printf("RpcServerListen returned: 0x%x\n", status);
    if (status) {
        exit(status);
    }

    if (fDontWait) {
        printf("Calling RpcMgmtWaitServerListen\n");
        status = RpcMgmtWaitServerListen();  //  wait operation
        printf("RpcMgmtWaitServerListen returned: 0x%x\n", status);
        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);
}


/* end file cluuids.c */
```

INTEROP

/***/ /**
/ / Microsoft RPC Examples **/ /** OSF DCE
Interop Sample Application **/ /** Copyright(c) Microsoft Corp. 1993-1994
/ / **/
/***/

Overview: ---------

This is a small demo RPC application. It is designed to be portable between OSF DCE RPC and Microsoft RPC platforms.

The most important part of the demo is the header file dceport.h. This header file maps OSF DCE RPC APIs and data structures to the Microsoft RPC equivalents.

The program just sends simple messages (strings) from the client to the server.

Building on Windows/NT ----------

The following environment variables should be set for you already.

```
set CPU=i386
set INCLUDE=c:\mstools\h
set LIB=c:\mstools\lib
set PATH=c:c:\winnt\system32;\mstools\bin;
```

For mips, set CPU=mips For alpha, set CPU=alpha

Build the sample distributed application:

```
nmake cleanall
nmake
```

Building on MS-DOS Systems ----------

After installing the Microsoft Visual C/C++ version 1.50 development environment and the Microsoft RPC version 2.0 toolkit on a Windows NT computer, you can build the sample client application from Windows NT.

```
nmake -f makefile.dos cleanall
nmake -f makefile.dos
```

This builds the client application callc.exe.

You may also execute the Microsoft Visual C/C++ compiler under MS-DOS. This requires a two step build process.

Step One: Compile the .IDL files under Windows NT

```
nmake -a -f makefile.dos msg.h
```

Step Two: Compile the C sources (stub and application)

```
nmake -f makefile.dos
```

Building on DCE Systems _____

You need to copy the following files to the DCE machine: client.c server.c manager.c msg.idl msg.acf makefile.dce

make -f makefile.dce cleanall all

Note: You will probably need to change the CFLAGS and LIBS variables in makefile.dce to match your platform.

Using the program: ----------

The basic example:

Run server
on the server machine.

Run: client -n <server name> -s "Hi, I'm a client"
on the client machine to send the message.

Run: client -n <server name> -s "Okay, stop this example" -x
on the client to send the message and cause the server to stop.

You can use fixed endpoints by adding the -e switch:

server -e 3452 client -e 3452 -n <server name> -s "Hi, I'm a client" client -e 3452 -n <server name> -s "Okay, stop this example" -x

You can run the demo over a different protocol by adding a -t switch to both the client and server:

server -t ncacn_np client -t ncacn_np -n <server name> -s "Hi, I'm a client" client -t ncacn_np -n <server name> -s "Okay, stop this example" -x

Options: --------

The -h switch displays a usage message.

The -s <message> switch is used to change with message sent from the client

to the server. Without it the message "Hello World" is sent.

The -n <server_name> switch is used for specifying a server machine.

Without it the server is assumed to run on the same machine.

The -e <endpoint> switch is used to specify a fixed endpoint to be used.

Without it a dynamic endpoint will be used and registered with the endpoint mapper.

The -t <protseq> switch is used to specify which protocol to use. Without

it the protocol sequence "ncacn_ip_tcp" will be used.

MAKEFILE (INTEROP RPC Sample)

```
#************************************************************#
#**                                                        **#
#**                 Microsoft RPC Examples                 **#
#**                 OSF DCE Interop Application            **#
#**           Copyright(c) Microsoft Corp. 1993            **#
#**                                                        **#
#************************************************************#

!include <ntwin32.mak>

!if "$(CPU)" == "i386"
cflags = $(cflags:G3=Gz)
cflags = $(cflags:Zi=Z7)
!endif


all : client.exe server.exe

# Make the client
client : client.exe
client.exe : client.obj msg_c.obj midluser.obj
    $(link) $(linkdebug) $(conflags) -out:client.exe -map:client.map \
      client.obj msg_c.obj midluser.obj \
      rpcrt4.lib $(conlibs)

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

# client stub
msg_c.obj : msg_c.c msg.h
   $(cc) $(cdebug) $(cflags) $(cvars) $*.c


# Make the server executable
server : server.exe
server.exe : server.obj manager.obj msg_s.obj midluser.obj
    $(link) $(linkdebug) $(conflags) -out:server.exe -map:server.map \
      server.obj manager.obj msg_s.obj midluser.obj \
      rpcrt4.lib $(conlibsmt)

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

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

# server stub
msg_s.obj : msg_s.c msg.h
```

```
   $(cc) $(cdebug) $(cflags) $(cvarsmt) $*.c

# midl_user* routines
midluser.obj : midluser.c
   $(cc) $(cdebug) $(cflags) $(cvarsmt) $*.c


# Stubs and header file from the IDL file
msg.h msg_s.c msg_c.c : msg.idl msg.acf
    midl -cpp_cmd $(cc) -cpp_opt "-E" msg.idl

# Clean up everything
cleanall : clean
    -del *.exe

# Clean up everything but the .EXEs
clean :
    -del *.obj
    -del *.map
    -del msg_?.c
    -del msg.h
```

DCEPORT.H (INTEROP RPC Sample)

```
/*++

Copyright (c) 1993-1994 Microsoft Corporation

Module Name:

    dceport.h

Abstract:

    Include file defining types and macros which map DCE RPC APIs to
    Microsoft RPC APIs.  Useful when porting DCE RPC applications to MS RPC.

--*/

#ifndef DCEPORT_H
#define DCEPORT_H

#ifdef __cplusplus
extern "C" {
#endif

/*
** Define various idl types
*/
#define idl_char                unsigned char
#define idl_boolean             unsigned char
#define idl_byte                unsigned char
#define idl_small_int           char
#define idl_usmall_int          unsigned char
#define idl_short_int           signed short
#define idl_ushort_int          unsigned short
#define idl_long_int            long
#define idl_ulong_int           unsigned long
#define boolean32               unsigned long
#define unsigned32              unsigned long
#define unsigned16              unsigned short
#define idl_true                1
#define idl_false               0
#define unsigned_char_t         unsigned char
typedef unsigned char __RPC_FAR *unsigned_char_p_t;
typedef void __RPC_FAR          *idl_void_p_t;

#ifndef _ERROR_STATUS_T_DEFINED
typedef unsigned long error_status_t;
#define _ERROR_STATUS_T_DEFINED
#endif

/*
** Define various DCE RPC types
*/
#define rpc_if_handle_t             RPC_IF_HANDLE
```

```
#define rpc_ns_handle_t             RPC_NS_HANDLE
#define rpc_authz_handle_t          RPC_AUTHZ_HANDLE
#define rpc_auth_identity_handle_t  RPC_AUTH_IDENTITY_HANDLE
#define rpc_sm_thread_handle_t      RPC_SS_THREAD_HANDLE
#define rpc_mgr_epv_t               RPC_MGR_EPV __RPC_FAR *
#define rpc_object_inq_fn_t         RPC_OBJECT_INQ_FN __RPC_FAR *
#define rpc_auth_key_retrieval_fn_t RPC_AUTH_KEY_RETRIEVAL_FN
#define rpc_mgmt_authorization_fn_t RPC_MGMT_AUTHORIZATION_FN

/*
** Define rpc_binding_vector_t to match DCE
*/
#ifdef rpc_binding_vector_t
#undef rpc_binding_vector_t
#endif

typedef struct
{
     unsigned long           count;
     handle_t                binding_h[1];
} rpc_binding_vector_t, __RPC_FAR *rpc_binding_vector_p_t;

/*
** Define rpc_protseq_vector_t to match DCE
*/

typedef struct
{
    unsigned long           count;
    unsigned char *         protseq[1];
} rpc_protseq_vector_t, __RPC_FAR *rpc_protseq_vector_p_t;

/*
** Define rpc_stats_vector_t to match DCE
*/

typedef struct
{
    unsigned long           count;
    unsigned long           stats[1];
} rpc_stats_vector_t, __RPC_FAR *rpc_stats_vector_p_t;

/*
** Define uuid_t to match DCE
*/
#ifdef uuid_t
#undef uuid_t
#endif

typedef struct
{
    unsigned long            time_low;
    unsigned short           time_mid;
    unsigned short           time_hi_and_version;
    unsigned char            clock_seq_hi_and_reserved;
```

```
    unsigned char            clock_seq_low;
    unsigned char            node[6];
} uuid_t, __RPC_FAR *uuid_p_t;

/*
** Define uuid_vector_t to match DCE
*/
#ifdef uuid_vector_t
#undef uuid_vector_t
#endif

typedef struct
{
     unsigned long           count;
     uuid_p_t                uuid[1];
} uuid_vector_t, __RPC_FAR *uuid_vector_p_t;

/*
** Define rpc_if_id_t and rpc_if_id_p_t to match DCE
*/

typedef struct
{
    uuid_t                   uuid;
    unsigned short           vers_major;
    unsigned short           vers_minor;
} rpc_if_id_t, __RPC_FAR *rpc_if_id_p_t;

/*
** Define rpc_if_id_vector_t to match DCE
*/

typedef struct
{
    unsigned long           count;
    rpc_if_id_p_t           if_id[1];
} rpc_if_id_vector_t, __RPC_FAR *rpc_if_id_vector_p_t;

/*
** The MinThreads parameters to RpcServerListen()
** is not part of the DCE API rpc_server_listen().
** This is the default value.
*/

#define rpc_c_listen_min_threads_default 1

/*
** Define various constants
*/
#define rpc_c_ns_syntax_default            RPC_C_NS_SYNTAX_DEFAULT
#define rpc_c_ns_syntax_dce                RPC_C_SYNTAX_DCE
#define rpc_c_ns_default_exp_age           RPC_C_DEFAULT_EXP_AGE
#define rpc_c_protseq_max_reqs_default     RPC_C_PROTSEQ_MAX_REQS_DEFAULT
#define rpc_c_protseq_max_calls_default    RPC_C_PROTSEQ_MAX_REQS_DEFAULT
```

```
#define rpc_c_listen_max_calls_default     RPC_C_LISTEN_MAX_CALLS_DEFAULT
#define rpc_c_ep_all_elts                  RPC_C_EP_ALL_ELTS
#define rpc_c_ep_match_by_if               RPC_C_EP_MATCH_BY_IF
#define rpc_c_ep_match_by_obj              RPC_C_EP_MATCH_BY_OBJ
#define rpc_c_ep_match_by_both             RPC_C_EP_MATCH_BY_BOTH
#define rpc_c_vers_all                     RPC_C_VERS_ALL
#define rpc_c_vers_compatible              RPC_C_VERS_COMPATIBLE
#define rpc_c_vers_exact                   RPC_C_VERS_EXACT
#define rpc_c_vers_major_only              RPC_C_VERS_MAJOR_ONLY
#define rpc_c_vers_upto                    RPC_C_VERS_UPTO
#define rpc_c_profile_default_elt          RPC_C_PROFILE_DEFAULT_ELT
#define rpc_c_profile_all_elts             RPC_C_PROFILE_ALL_ELTS
#define rpc_c_profile_match_by_if          RPC_C_PROFILE_MATCH_BY_IF
#define rpc_c_profile_match_by_mbr         RPC_C_PROFILE_MATCH_BY_MBR
#define rpc_c_profile_match_by_both        RPC_C_PROFILE_MATCH_BY_BOTH
#define rpc_c_binding_min_timeout          RPC_C_BINDING_MIN_TIMEOUT
#define rpc_c_binding_default_timeout      RPC_C_BINDING_DEFAULT_TIMEOUT
#define rpc_c_binding_max_timeout          RPC_C_BINDING_MAX_TIMEOUT
#define rpc_c_binding_infinite_timeout     RPC_C_BINDING_INFINITE_TIMEOUT
#define rpc_c_stats_calls_in               RPC_C_STATS_CALLS_IN
#define rpc_c_stats_calls_out              RPC_C_STATS_CALLS_OUT
#define rpc_c_stats_pkts_in                RPC_C_STATS_PKTS_IN
#define rpc_c_stats_pkts_out               RPC_C_STATS_PKTS_OUT
#define rpc_c_mgmt_inq_if_ids              RPC_C_MGMT_INQ_IF_IDS
#define rpc_c_mgmt_inq_princ_name          RPC_C_MGMT_INQ_PRINC_NAME
#define rpc_c_mgmt_inq_stats               RPC_C_MGMT_INQ_STATS
#define rpc_c_mgmt_inq_server_listen       RPC_C_MGMT_INQ_SERVER_LISTEN
#define rpc_c_mgmt_stop_server_listen      RPC_C_MGMT_STOP_SERVER_LISTEN
#define rpc_c_mgmt_cancel_infinite_timeout RPC_C_CANCEL_INFINITE_TIMEOUT

/*
** Define DCE API equivalents
*/
#define rpc_binding_copy(source,dest,status) \
                *(status) = RpcBindingCopy(source,dest)

#define rpc_binding_free(binding,status) *(status) = RpcBindingFree(binding)

#define rpc_binding_from_string_binding(string_binding,binding,status) \
                *(status) =
RpcBindingFromStringBinding(string_binding,binding)

#define rpc_binding_inq_auth_client(binding,privs,princ_name,protect_level,
\
                authn_svc,authz_svc,status) \
                *(status) =
RpcBindingInqAuthClient(binding,privs,princ_name, \
                protect_level,authn_svc,authz_svc)

#define rpc_binding_inq_auth_info(binding,princ_name,protect_level,\
                authn_svc,auth_identity,authz_svc,status) \
                *(status) = RpcBindingInqAuthInfo(binding,princ_name, \
                protect_level,authn_svc,auth_identity,authz_svc)

#define rpc_binding_inq_object(binding,object_uuid,status) \
```

```
                *(status) = RpcBindingInqObject(binding,\
                (UUID __RPC_FAR *)object_uuid)

#define rpc_binding_reset(binding,status) *(status) =
RpcBindingReset(binding)

#define rpc_binding_server_from_client(cbinding,sbinding,status) \
                *(status) = RpcBindingServerFromClient(cbinding,sbinding)

#define rpc_binding_set_auth_info(binding,princ_name,protect_level,\
                authn_svc,auth_identity,authz_svc,status) \
                *(status) = RpcBindingSetAuthInfo(binding,princ_name,\
                protect_level,authn_svc,auth_identity,authz_svc)

#define rpc_binding_set_object(binding,object_uuid,status) \
                *(status) = RpcBindingSetObject(binding,\
                (UUID __RPC_FAR *)object_uuid)


#define rpc_binding_to_string_binding(binding,string_binding,status) \
                *(status) =
RpcBindingToStringBinding(binding,string_binding)

#define rpc_binding_vector_free(binding_vector,status) \
                *(status) = RpcBindingVectorFree(\
                (RPC_BINDING_VECTOR __RPC_FAR * __RPC_FAR *)binding_vector)

#define rpc_ep_register(if_spec,binding_vec,object_uuid_vec,annotation,\
                status)\
                *(status) = RpcEpRegister(if_spec,\
                (RPC_BINDING_VECTOR __RPC_FAR *)binding_vec, \
                (UUID_VECTOR __RPC_FAR *)object_uuid_vec, annotation)

#define rpc_ep_register_no_replace(if_spec,binding_vec,object_uuid_vec,\
                annotation,status) \
                *(status) = RpcEpRegisterNoReplace(if_spec,\
                (RPC_BINDING_VECTOR __RPC_FAR *)binding_vec,\
                (UUID_VECTOR __RPC_FAR *)object_uuid_vec,annotation)

#define rpc_ep_resolve_binding(binding_h,if_spec,status) \
                *(status) = RpcEpResolveBinding(binding_h,if_spec)

#define rpc_ep_unregister(if_spec,binding_vec,object_uuid_vec,status) \
                *(status) = RpcEpUnregister(if_spec,\
                (RPC_BINDING_VECTOR __RPC_FAR *)binding_vec,\
                (UUID_VECTOR __RPC_FAR *)object_uuid_vec)

#define rpc_if_id_vector_free(if_id_vector,status) \
                *(status) = RpcIfIdVectorFree(\
                (RPC_IF_ID_VECTOR __RPC_FAR * __RPC_FAR *)if_id_vector)

#define rpc_if_inq_id(if_spec,if_id,status) \
                *(status) = RpcIfInqId(if_spec,(RPC_IF_ID __RPC_FAR *)if_id)

#define rpc_if_register_auth_info(if_spec,princ_name,protect_level,\
```

```
                authn_svc,auth_identity,authz_svc,status) \
                *(status) = RpcIfRegisterAuthInfo(if_spec,princ_name,\
                protect_level,authn_svc,auth_identity,authz_svc)

#define rpc_mgmt_ep_elt_inq_begin(ep_binding,inquiry_type,if_id,vers_option,
\
                object_uuid,inquiry_context,status) \
                *(status) =
RpcMgmtEpEltInqBegin(ep_binding,inquiry_type,if_id,\
                vers_option,object_uuid,inquiry_context)

#define rpc_mgmt_ep_elt_inq_done(inquiry_context,status) \
                *(status) = RpcMgmtEpEltInqDone(inquiry_context)

#define rpc_mgmt_ep_elt_inq_next(inquiry_context,if_id,binding,object_uuid,\
                annotation,status) \
                *(status) =
RpcMgmtEpEltInqNext(inquiry_context,if_id,binding,\
                object_uuid,annotation)

#define rpc_mgmt_ep_unregister(ep_binding,if_id,binding,object_uuid,status)
\
                *(status) = RpcMgmtEpUnregister(ep_binding,if_id,binding,\
                object_uuid)

#define rpc_mgmt_inq_com_timeout(binding,timeout,status) \
                *(status) = RpcMgmtInqComTimeout(binding,timeout)

#define rpc_mgmt_inq_dflt_protect_level(authn_svc,level,status) \
                *(status) = RpcMgmtInqDefaultProtectLevel(authn_svc,level)

#define rpc_mgmt_inq_if_ids(binding,if_id_vector,status) \
                *(status) = RpcMgmtInqIfIds((bindings),\
                (RPC_IF_ID_VECTOR __RPC_FAR * __RPC_FAR *)(if_id_vector))

#define rpc_mgmt_inq_server_princ_name(binding,authn_svc,princ_name,status)
\
                *(status) = RpcMgmtInqServerPrincName(binding,authn_svc,\
                princ_name)

#define rpc_mgmt_inq_stats(binding,statistics,status) \
                *(status) = RpcMgmtInqStats(binding,\
                (RPC_STATS_VECTOR __RPC_FAR * __RPC_FAR *)statistics)

#define rpc_mgmt_is_server_listening(binding,status) \
                ( ((*(status) = RpcMgmtIsServerListening(binding)) ==
RPC_S_OK) \
                ? (1) : (*(status) == RPC_S_NOT_LISTENING) \
                ? (*(status) = RPS_S_OK, 0) : (0) )

#define rpc_mgmt_set_authorization_fn(authz_fn,status) \
                *(status) = RpcMgmtSetAuthorizathionFn(authz_fn)

#define rpc_mgmt_set_cancel_timeout(seconds,status) \
                *(status) = RpcMgmtSetCancelTimeout(seconds)
```

```
#define rpc_mgmt_set_com_timeout(binding,timeout,status) \
                *(status) = RpcMgmtSetComTimeout(binding,timeout)

#define rpc_mgmt_set_server_stack_size(size,status) \
                *(status) = RpcMgmtSetServerStackSize(size)

#define rpc_mgmt_stats_vector_free(stats,status) \
                *(status) = RpcMgmtStatsVectorFree(\
                (RPC_STATS_VECTOR __RPC_FAR * __RPC_FAR *)stats)


#define rpc_mgmt_stop_server_listening(binding,status) \
                *(status) = RpcMgmtStopServerListening(binding)

#define rpc_network_inq_protseqs(prots,status) \
                *(status) = RpcNetworkInqProtseqs(\
                (RPC_PROTSEQ_VECTOR __RPC_FAR * __RPC_FAR *)prots)

#define rpc_network_is_protseq_valid(protseq,status) \
                *(status) = RpcNetworkIsProtseqValid(protseq)

/*
** Define NSI equivalents
*/
#define rpc_ns_binding_export(name_syntax,entry_name,if_spec,\
                binding_vector, uuid_vector,status) \
                *(status) = RpcNsBindingExport(name_syntax,entry_name,\
                if_spec, (RPC_BINDING_VECTOR *)binding_vector,\
                (UUID_VECTOR __RPC_FAR *)uuid_vector)

#define rpc_ns_binding_import_begin(name_syntax,entry_name,if_spec,\
                object_uuid,import_context,status) \
                *(status) = RpcNsBindingImportBegin(name_syntax,entry_name,\
                if_spec,(UUID __RPC_FAR *)object_uuid,import_context)

#define rpc_ns_binding_import_done(import_context,status) \
                *(status) = RpcNsBindingImportDone(import_context)

#define rpc_ns_binding_import_next(import_context,binding,status) \
                *(status) = RpcNsBindingImportNext(import_context,binding)

#define rpc_ns_binding_inq_entry_name(binding,name_syntax,entry_name,status)
\
                *(status) = RpcNsBindingInqEntryName(binding,name_syntax,\
                entry_name)

#define rpc_ns_binding_lookup_begin(name_syntax,entry_name,if_spec,\
                object_uuid,max_count,lookup_context,status) \
                *(status) = RpcNsBindingLookupBegin(name_syntax,entry_name,\
                if_spec,(UUID __RPC_FAR
*)object_uuid,max_count,lookup_context)

#define rpc_ns_binding_lookup_done(lookup_context,status) \
                *(status) = RpcNsBindingLookupDone(lookup_context)
```

```
#define rpc_ns_binding_lookup_next(lookup_context,binding_vector,status) \
                *(status) = RpcNsBindingLookupNext(lookup_context, \
                (RPC_BINDING_VECTOR __RPC_FAR * __RPC_FAR *)binding_vector)

#define rpc_ns_binding_select(binding_vector,binding,status) \
                *(status) = RpcNsBindingSelect(\
                (RPC_BINDING_VECTOR __RPC_FAR *)binding_vector,binding)

#define rpc_ns_binding_unexport(name_syntax,entry_name,if_spec,\
                uuid_vector,status) \
                *(status) = RpcNsBindingUnexport(name_syntax,entry_name,\
                if_spec, (UUID_VECTOR __RPC_FAR *)uuid_vector)

#define rpc_ns_entry_expand_name(name_syntax,entry_name,expanded_name,\
                status)\
                *(status) = RpcNsEntryExpandName(name_syntax,entry_name,\
                expanded_name)

#define rpc_ns_entry_object_inq_begin(name_syntax,entry_name,\
                inquiry_context,status)\
                *(status) = RpcNsEntryObjectInqBegin(name_syntax,\
                entry_name,inquiry_context)

#define rpc_ns_entry_object_inq_done(inquiry_context,status) \
                *(status) = RpcNsEntryObjectInqDone(inquiry_context)

#define rpc_ns_entry_object_inq_next(inquiry_context,object_uuid,status) \
                *(status) = RpcNsEntryObjectInqNext(inquiry_context,\
                (UUID __RPC_FAR *)object_uuid)

#define rpc_ns_group_delete(name_syntax,group_name,status) \
                *(status) = RpcNsGroupDelete(name_syntax,group_name)

#define rpc_ns_group_mbr_add(name_syntax,group_name,member_name_syntax,\
                member_name,status) \
                *(status) = RpcNsGroupMbrAdd(name_syntax,group_name,\
                member_name_syntax,member_name)

#define
rpc_ns_group_mbr_inq_begin(name_syntax,group_name,member_name_syntax,\
                inquiry_context,status) \
                *(status) = RpcNsGroupMbrInqBegin(name_syntax,group_name,\
                member_name_syntax,inquiry_context)

#define rpc_ns_group_mbr_inq_done(inquiry_context,status) \
                *(status) = RpcNsGroupMbrInqDone(inquiry_context)

#define rpc_ns_group_mbr_inq_next(inquiry_context,member_name,status) \
                *(status) =
RpcNsGroupMbrInqNext(inquiry_context,member_name)

#define rpc_ns_group_mbr_remove(name_syntax,group_name,member_name_syntax,\
                member_name,status) \
```

```
                *(status) = RpcNsGroupMbrRemove(name_syntax,group_name,\
                member_name_syntax,member_name)

#define
rpc_ns_mgmt_binding_unexport(name_syntax,entry_name,if_id,vers_option,\
                uuid_vector,status) \
                *(status) = RpcNsMgmtBindingUnexport(name_syntax,entry_name,
\
                (RPC_IF_ID __RPC_FAR *)if_id,vers_option,\
                (UUID_VECTOR __RPC_FAR *)uuid_vector)

#define rpc_ns_mgmt_entry_create(name_syntax,entry_name,status) \
                *(status) = RpcNsMgmtEntryCreate(name_syntax,entry_name)

#define rpc_ns_mgmt_entry_delete(name_syntax,entry_name,status) \
                *(status) = RpcNsMgmtEntryDelete(name_syntax,entry_name)

#define rpc_ns_mgmt_entry_inq_if_ids(name_syntax,entry_name,if_id_vector,\
                status) \
                *(status) = RpcNsMgmtEntryInqIfIds(name_syntax,entry_name,\
                (RPC_IF_ID_VECTOR __RPC_FAR * __RPC_FAR *)if_id_vector)

#define rpc_ns_mgmt_handle_set_exp_age(ns_handle,expiration_age,status) \
                *(status) =
RpcNsMgmtHandleSetExpAge(ns_handle,expiration_age)

#define rpc_ns_mgmt_inq_exp_age(expiration_age,status) \
                *(status) = RpcNsMgmtInqExpAge(expiration_age)

#define rpc_ns_mgmt_set_exp_age(expiration_age,status) \
                *(status) = RpcNsMgmtSetExpAge(expiration_age)

#define rpc_ns_profile_delete(name_syntax,profile_name,status) \
                *(status) = RpcNsProfileDelete(name_syntax,profile_name)

#define rpc_ns_profile_elt_add(name_syntax,profile_name,if_id,\
                member_name_syntax,member_name,priority,annotation,status) \
                *(status) = RpcNsProfileEltAdd(name_syntax,profile_name,\
                (RPC_IF_ID __RPC_FAR *)if_id,member_name_syntax,member_name,
\
                priority,annotation)

#define rpc_ns_profile_elt_inq_begin(name_syntax,profile_name,inquiry_type,\
                if_id,if_vers_option,member_name_syntax,\
                member_name,inquiry_context,status) \
                *(status) =
RpcNsProfileEltInqBegin(name_syntax,profile_name,\
                inquiry_type,(RPC_IF_ID __RPC_FAR *)if_id,if_vers_option,\
                member_name_syntax,member_name,inquiry_context)

#define rpc_ns_profile_elt_inq_done(inquiry_context,status) \
                *(status) = RpcNsProfileEltInqDone(inquiry_context)

#define rpc_ns_profile_elt_inq_next(inquiry_context,if_id,member_name,\
                priority,annotation,status) \
```

```
                *(status) = RpcNsProfileEltInqNext(inquiry_context,\
                (RPC_IF_ID __RPC_FAR
*)if_id,member_name,priority,annotation)

#define rpc_ns_profile_elt_remove(name_syntax,profile_name,if_id,\
               member_name_syntax,member_name,status) \
                *(status) = RpcNsProfileEltRemove(name_syntax,profile_name,\
                (RPC_IF_ID __RPC_FAR *)if_id,member_name_syntax,member_name)

#define rpc_object_inq_type(object_uuid,type_uuid,status) \
                *(status) = RpcObjectInqType((UUID __RPC_FAR *)object_uuid,\
                (UUID __RPC_FAR *)type_uuid)

#define rpc_object_set_inq_fn(inq_fn,status) \
                *(status) = RpcObjectSetInqFn(inq_fn)

#define rpc_object_set_type(object_uuid,type_uuid,status) \
                *(status) = RpcObjectSetType((UUID __RPC_FAR *)object_uuid,\
                (UUID __RPC_FAR *)type_uuid)

#define rpc_protseq_vector_free(protseq_vector,status) \
                *(status) = RpcProtseqVectorFree( \
                (RPC_PROTSEQ_VECTOR __RPC_FAR * __RPC_FAR *)protseq_vector)

#define rpc_server_inq_bindings(binding_vector,status) \
                *(status) = RpcServerInqBindings(\
                (RPC_BINDING_VECTOR __RPC_FAR * __RPC_FAR *)binding_vector)

#define rpc_server_inq_if(if_spec,type_uuid,mgr_epv,status) \
                *(status) = RpcServerInqIf(if_spec,(UUID __RPC_FAR
*)type_uuid,\
                (RPC_MGR_EPV __RPC_FAR *)mgr_epv)

#define rpc_server_listen(max_calls,status) \
                *(status) =
RpcServerListen(rpc_c_listen_min_threads_default,\
                max_calls,0)

#define rpc_server_register_auth_info(princ_name,auth_svc,get_key_func,arg,\
                status) \
                *(status) = RpcServerRegisterAuthInfo(princ_name,auth_svc,\
                get_key_func,arg)


#define rpc_server_register_if(if_spec,type_uuid,mgr_epv,status) \
                *(status) = RpcServerRegisterIf(if_spec,\
                (UUID __RPC_FAR *)type_uuid,(RPC_MGR_EPV __RPC_FAR
*)mgr_epv)

#define rpc_server_unregister_if(if_spec,type_uuid,status) \
                *(status) = RpcServerUnregisterIf(if_spec,(UUID
*)type_uuid,0)

#define rpc_server_use_all_protseqs(max_call_requests,status) \
                *(status) = RpcServerUseAllProtseqs(max_call_requests,0)
```

```
#define rpc_server_use_all_protseqs_if(max_call_requests,if_spec,status) \
                *(status) = RpcServerUseAllProtseqsIf(max_call_requests,\
                if_spec,0)

#define rpc_server_use_protseq(protseq,max_call_requests,status) \
                *(status) = RpcServerUseProtseq(protseq,max_call_requests,0)

#define rpc_server_use_protseq_ep(protseq,max_call_requests,endpoint,status)
\
                *(status) = RpcServerUseProtseqEp(protseq,max_call_requests,
\
                endpoint,0)

#define rpc_server_use_protseq_if(protseq,max_call_requests,if_spec,status)
\
                *(status) = RpcServerUseProtseqIf(protseq,max_call_requests,
\
                if_spec,0)

#define rpc_sm_alloce(size,status) *(status) = RpcSmAllocate(size)

#define rpc_sm_client_free(ptr,status) *(status) = RpcSmClientFree(ptr)

#define rpc_sm_destroy_client_context(context,status) \
                *(status) = RpcSmDestroyClientContext(context)

#define rpc_sm_disable_allocate(status) *(status) = RpcSmDisableAllocate()

#define rpc_sm_enable_allocate(status) *(status) = RpcSmEnableAllocate()

#define rpc_sm_free(ptr,status) *(status) = RpcSmFree(ptr)

#define rpc_sm_get_thread_handle(status) RpcSmGetThreadHandle(status)

#define rpc_sm_set_client_alloc_free(alloc,free,status) \
                *(status) = RpcSmSetClientAllocFree(alloc,free)

#define rpc_sm_set_thread_handle(id,status) \
                *(status) = RpcSmSetThreadHandle(id)

#define rpc_sm_swap_client_alloc_free(alloc,free,old_alloc,old_free,status)
\
                *(status) = RpcSmSwapClientAllocFree(alloc,free \
                old_alloc, old_free)

#define rpc_string_binding_compose(object_uuid,protseq,netaddr,endpoint,\
                options,binding,status) \
                *(status) = RpcStringBindingCompose(object_uuid,protseq,\
                netaddr,endpoint,options,binding)

#define rpc_string_binding_parse(string_binding,object_uuid,protseq,netaddr,
\
                endpoint,options,status) \
                *(status) = RpcStringBindingParse(string_binding,\
```

```
                object_uuid,protseq,netaddr,endpoint,options)

#define rpc_string_free(string,status) *(status) = RpcStringFree(string)

#define uuid_compare(uuid1,uuid2,status) \
                UuidCompare((UUID __RPC_FAR *)(uuid1),\
                            (UUID __RPC_FAR *)(uuid2),(status))

#define uuid_create(uuid,status) \
                *(status) = UuidCreate((UUID __RPC_FAR *)uuid)

#define uuid_create_nil(uuid,status) \
                *(status) = UuidCreateNil((UUID __RPC_FAR *)uuid)

#define uuid_equal(uuid1,uuid2,status) \
                UuidEqual((UUID __RPC_FAR *)(uuid1), \
                          (UUID __RPC_FAR *)(uuid2), (status))

#define uuid_from_string(string,uuid,status) \
                *(status) = UuidFromString(string,(UUID __RPC_FAR *)uuid)

#define uuid_hash(uuid,status) \
                UuidHash((UUID __RPC_FAR *)(uuid),(status))

#define uuid_is_nil(uuid,status) \
                UuidIsNil((UUID __RPC_FAR *)(uuid), (status))

#define uuid_to_string(uuid,string,status)\
                *(status) = UuidToString((UUID __RPC_FAR *)uuid,string)


#define true  1
#define false 0

/*
** Define exception handling equivalents
**
*/
#if defined (__RPC_WIN16__) || defined (__RPC_DOS__)

#define TRY                                          \
    {                                                  \
    int _exception_mode_finally;                       \
    int _exception_code;                               \
    ExceptionBuff exception;                           \
    _exception_code = RpcSetException(&exception);     \
    if (!_exception_code)                              \
    {

#define CATCH_ALL                                    \
        _exception_mode_finally = false;               \
        RpcLeaveException();                           \
        }                                              \
        else                                           \
```

```
        {
/*
 * #define CATCH(X)                                     \
 *   }else if ((unsigned long)RpcExceptionCode()==(unsigned long)X) {
 */
#define FINALLY                                         \
        _exception_mode_finally = true;                    \
        RpcLeaveException();                               \
        } {
#define ENDTRY                                          \
          }                                                  \
        if (_exception_mode_finally && _exception_code)  \
            RpcRaiseException(_exception_code);             \
        }

#endif /* WIN16 or DOS */


#if defined (__RPC_WIN32__)
#define TRY             try {
/*
 * #define CATCH(X)        \
 *                       } except (GetExceptionCode() == X ? \
 *                         EXCEPTION_EXECUTE_HANDLER : \
 *                         EXCEPTION_CONTINUE_SEARCH) {
 */
#define CATCH_ALL       } except (EXCEPTION_EXECUTE_HANDLER) {
#define FINALLY         } finally {
#define ENDTRY          }
#endif /* WIN32 */

#define RAISE(v)   RpcRaiseException(v)
#define RERAISE    RpcRaiseException(RpcExceptionCode())
#define THIS_CATCH RpcExceptionCode()

/*
** DCE Status code mappings
*/
#ifndef rpc_s_ok
#define rpc_s_ok                        RPC_S_OK
#endif
#ifndef error_status_ok
#define error_status_ok                 RPC_S_OK
#endif
#define ept_s_cant_perform_op           EPT_S_CANT_PERFORM_OP
#define ept_s_invalid_entry             EPT_S_INVALID_ENTRY
#define ept_s_not_registered            EPT_S_NOT_REGISTERED
#define rpc_s_already_listening         RPC_S_ALREADY_LISTENING
#define rpc_s_already_registered        RPC_S_ALREADY_REGISTERED
#define rpc_s_binding_has_no_auth       RPC_S_BINDING_HAS_NO_AUTH
#define rpc_s_binding_imcomplete        RPC_S_BINDING_IMCOMPLETE
#define rpc_s_call_cancelled            RPC_S_CALL_CANCELLED
#define rpc_s_call_failed               RPC_S_CALL_FAILED
#define rpc_s_cant_bind_socket          RPC_S_CANNOT_BIND
#define rpc_s_cant_create_socket        RPC_S_CANT_CREATE_ENDPOINT
```

```
#define rpc_s_comm_failure              RPC_S_COMM_FAILURE
#define rpc_s_connect_no_resources      RPC_S_OUT_OF_RESOURCES
#define rpc_s_cthread_create_failed     RPC_S_OUT_OF_THREADS
#define rpc_s_endpoint_not_found        RPC_S_NO_ENDPOINT_FOUND
#define rpc_s_entry_already_exists      RPC_S_ENTRY_ALREADY_EXISTS
#define rpc_s_entry_not_found           RPC_S_ENTRY_NOT_FOUND
#define rpc_s_fault_addr_error          RPC_S_ADDRESS_ERROR
#define rpc_s_fault_fp_div_by_zero      RPC_S_FP_DIV_ZERO
#define rpc_s_fault_fp_overflow         RPC_S_FP_OVERFLOW
#define rpc_s_fault_fp_underflow        RPC_S_FP_UNDERFLOW
#define rpc_s_fault_int_div_by_zero     RPC_S_ZERO_DIVIDE
#define rpc_s_fault_invalid_bound       RPC_S_INVALID_BOUND
#define rpc_s_fault_invalid_tag         RPC_S_INVALID_TAG
#define rpc_s_fault_remote_no_memory    RPC_S_SERVER_OUT_OF_MEMORY
#define rpc_s_fault_unspec              RPC_S_CALL_FAILED
#define rpc_s_incomplete_name           RPC_S_INCOMPLETE_NAME
#define rpc_s_interface_not_found       RPC_S_INTERFACE_NOT_FOUND
#define rpc_s_internal_error            RPC_S_INTERNAL_ERROR
#define rpc_s_inval_net_addr            RPC_S_INVALID_NET_ADDR
#define rpc_s_invalid_arg               RPC_S_INVALID_ARG
#define rpc_s_invalid_binding           RPC_S_INVALID_BINDING
#define rpc_s_invalid_endpoint_format   RPC_S_INVALID_ENDPOINT_FORMAT
#define rpc_s_invalid_naf_id            RPC_S_INVALID_NAF_IF
#define rpc_s_invalid_name_syntax       RPC_S_INVALID_NAME_SYNTAX
#define rpc_s_invalid_rpc_protseq       RPC_S_INVALID_RPC_PROTSEQ
#define rpc_s_invalid_string_binding    RPC_S_INVALID_STRING_BINDING
#define rpc_s_invalid_timeout           RPC_S_INVALID_TIMEOUT
#define rpc_s_invalid_vers_option       RPC_S_INVALID_VERS_OPTION
#define rpc_s_max_calls_too_small       RPC_S_MAX_CALLS_TOO_SMALL
#define rpc_s_mgmt_op_disallowed        RPC_S_ACCESS_DENIED
#define rpc_s_name_service_unavailable  RPC_S_NAME_SERVICE_UNAVAILABLE
#define rpc_s_no_bindings               RPC_S_NO_BINDINGS
#define rpc_s_no_entry_name             RPC_S_NO_ENTRY_NAME
#define rpc_s_no_interfaces             RPC_S_NO_INTERFACES
#define rpc_s_no_interfaces_exported    RPC_S_NO_INTERFACES_EXPORTED
#define rpc_s_no_memory                 RPC_S_OUT_OF_MEMORY
#define rpc_s_no_more_elements          RPC_X_NO_MORE_ENTRIES
#define rpc_s_no_more_bindings          RPC_S_NO_MORE_BINDINGS
#define rpc_s_no_more_members           RPC_S_NO_MORE_MEMBERS
#define rpc_s_no_ns_permission          RPC_S_ACCESS_DENIED
#define rpc_s_no_princ_name             RPC_S_NO_PRINC_NAME
#define rpc_s_no_protseqs               RPC_S_NO_PROTSEQS
#define rpc_s_no_protseqs_registered    RPC_S_NO_PROTSEQS_REGISTERED
#define rpc_s_not_rpc_tower             RPC_S_CANNOT_SUPPORT
#define rpc_s_not_supported             RPC_S_CANNOT_SUPPORT
#define rpc_s_not_authorized            RPC_S_ACCESS_DENIED
#define rpc_s_nothing_to_unexport       RPC_S_NOTHING_TO_UNEXPORT
#define rpc_s_object_not_found          RPC_S_OBJECT_NOT_FOUND
#define rpc_s_protocol_error            RPC_S_PROTOCOL_ERROR
#define rpc_s_protseq_not_supported     RPC_S_PROTSEQ_NOT_SUPPORTED
#define rpc_s_server_too_busy           RPC_S_SERVER_TOO_BUSY
#define rpc_s_string_too_long           RPC_S_STRING_TOO_LONG
#define rpc_s_type_already_registered   RPC_S_TYPE_ALREADY_REGISTERED
#define rpc_s_unknown_authn_service     RPC_S_UNKNOWN_AUTHN_SERVICE
```

```
#define rpc_s_unknown_authz_service     RPC_S_UNKNOWN_AUTHZ_SERVICE
#define rpc_s_unknown_if                RPC_S_UNKNOWN_IF
#define rpc_s_unknown_mgr_type          RPC_S_UNKNOWN_MGR_TYPE
#define rpc_s_unknown_reject            RPC_S_CALL_FAILED_DNE
#define rpc_s_unsupported_name_syntax   RPC_S_UNSUPPORTED_NAME_SYNTAX
#define rpc_s_unsupported_type          RPC_S_UNSUPPORTED_TYPE
#define rpc_s_wrong_boot_time           RPC_S_CALL_FAILED_DNE
#define rpc_s_wrong_kind_of_binding     RPC_S_WRONG_KIND_OF_BINDING
#define uuid_s_ok                       RPC_S_OK
#define uuid_s_internal_error           RPC_S_INTERNAL_ERROR
#define uuid_s_invalid_string_uuid      RPC_S_INVALID_STRING_UUID
#define uuid_s_no_address               RPC_S_UUID_NO_ADDRESS

/*
** DCE Exception mappings
*/

#define rpc_x_comm_failure              RPC_S_COMM_FAILURE
#define rpc_x_connect_no_resources      RPC_S_OUT_OF_RESOURCES
#define rpc_x_entry_not_found           RPC_S_ENTRY_NOT_FOUND
#define rpc_x_incomplete_name           RPC_S_INCOMPLETE_NAME
#define rpc_x_invalid_arg               RPC_S_INVALID_ARG
#define rpc_x_invalid_binding           RPC_S_INVALID_BINDING
#define rpc_x_invalid_bound             RPC_X_INVALID_BOUND
#define rpc_x_invalid_endpoint_format   RPC_S_INVALID_ENDPOINT_FORMAT
#define rpc_x_invalid_naf_id            RPC_S_INVALID_NAF_IF
#define rpc_x_invalid_name_syntax       RPC_S_INVALID_NAME_SYNTAX
#define rpc_x_invalid_rpc_protseq       RPC_S_INVALID_RPC_PROTSEQ
#define rpc_x_invalid_tag               RPC_X_INVALID_TAG
#define rpc_x_invalid_timeout           RPC_S_INVALID_TIMEOUT
#define rpc_x_no_memory                 RPC_X_NO_MEMORY
#define rpc_x_object_not_found          RPC_S_OBJECT_NOT_FOUND
#define rpc_x_protocol_error            RPC_S_PROTOCOL_ERROR
#define rpc_x_protseq_not_supported     RPC_S_PROTSEQ_NOT_SUPPORTED
#define rpc_x_server_too_busy           RPC_S_SERVER_TOO_BUSY
#define rpc_x_ss_char_trans_open_fail   RPC_X_SS_CHAR_TRANS_OPEN_FAIL
#define rpc_x_ss_char_trans_short_file  RPC_X_SS_CHAR_TRANS_SHORT_FILE
#define rpc_x_ss_context_damaged        RPC_X_SS_CONTEXT_DAMAGED
#define rpc_x_ss_context_mismatch       RPC_X_SS_CONTEXT_MISMATCH
#define rpc_x_ss_in_null_context        RPC_X_SS_IN_NULL_CONTEXT
#define rpc_x_string_too_long           RPC_S_STRING_TOO_LONG
#define rpc_x_unknown_if                RPC_S_UNKNOWN_IF
#define rpc_x_unknown_mgr_type          RPC_S_UNKNOWN_MGR_TYPE
#define rpc_x_unsupported_name_syntax   RPC_S_UNSUPPORTED_NAME_SYNTAX
#define rpc_x_unsupported_type          RPC_S_UNSUPPORTED_TYPE
#define rpc_x_wrong_boot_time           RPC_S_CALL_FAILED_DNE
#define rpc_x_wrong_kind_of_binding     RPC_S_WRONG_KIND_OF_BINDING
#define uuid_x_internal_error           RPC_S_INTERNAL_ERROR

#ifdef __cplusplus
}
#endif

#endif /* DCEPORT_H */
```

CLIENT.C (INTEROP RPC Sample)

```
/*********************************************************************/
/**                                                                 **/
/**                 Microsoft RPC Examples                          **/
/**                 OSF DCE Sample Application                      **/
/**             Copyright(c) Microsoft Corp. 1992                   **/
/**                                                                 **/
/*********************************************************************/

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <ctype.h>

#include "msg.h"        /* header file generated by M/IDL compiler */

#if defined(__RPC_WIN32__) || defined(__RPC_DOS__)

/*
 On MS platforms we must include the dceport.h header file
 which maps OSF DCE style APIs to MS style APIs.
*/

#include "dceport.h"

#else
#include <pthread.h>
#endif

#ifndef _CRTAPI1
#define _CRTAPI1
#endif

void Usage()
{
  printf("Usage : client -n <server_name>\n");
  printf("               -t <transport>   - optional, default
ncacn_ip_tcp\n");
  printf("               -e <endpoint>    - optional, should match
server\n");
  printf("               -s <message>     - optional, send a different
message\n");
  printf("               -x               - use to stop the server\n");
  exit(1);
}

int _CRTAPI1
main(int argc, char *argv[])
{
    unsigned32 status;
    unsigned char * pszProtocolSequence = (unsigned char *)"ncacn_ip_tcp";
    unsigned char * pszNetworkAddress   = NULL;
    unsigned char * pszEndpoint         = NULL;
```

```
unsigned char * pszStringBinding    = NULL;
unsigned char * pszMessage          = (unsigned char *)"Hello World";
int fStopServer = 0;
int i;

printf ("Microsoft RPC Demo - OSF DCE Interop Message Client\n");

for (i = 1; i < argc; i++) {
    if ((*argv[i] == '-') || (*argv[i] == '/')) {
        switch (tolower(*(argv[i]+1))) {
        case 'n':  /* network address */
            pszNetworkAddress = (unsigned char *)argv[++i];
            break;
        case 't':  /* protocol sequence */
            pszProtocolSequence = (unsigned char *)argv[++i];
            break;
        case 'e':  /* network endpoint */
            pszEndpoint = (unsigned char *)argv[++i];
            break;
        case 's':  /* update message */
            pszMessage = (unsigned char *)argv[++i];
            break;
        case 'x':  /* stop the server */
            fStopServer = 1;
            break;
        case 'h':
        case '?':
        default:
            Usage();
        }
    }
    else
        Usage();
}

rpc_string_binding_compose(0,   /* no object uuid */
                           pszProtocolSequence,
                           pszNetworkAddress,
                           pszEndpoint,
                           0,  /* no options */
                           &pszStringBinding,
                           &status);
if (status) {
    printf("rpc_string_binding_compose returned 0x%x\n", status);
    return(status);
}

rpc_binding_from_string_binding(pszStringBinding,
                                &interop_binding_handle,
                                &status);
if (status) {
    printf("rpc_binding_from_string_binding returned 0x%x\n", status);
    return(status);
}
```

```
    rpc_string_free(&pszStringBinding,&status);
    if (status) {
        printf("rpc_string_free returned 0x%x\n", status);
        return(status);
    }

    TRY {
       ClientMessage(pszMessage);

       printf("Message sent okay\n");
       if (fStopServer)
           {
           ShutdownServer();
           printf("Server shutdown\n");
           }
    }
    CATCH_ALL {
       printf("RPC raised exception 0x%x\n", THIS_CATCH);
    }
    ENDTRY

    rpc_binding_free(&interop_binding_handle, &status);

    if (status) {
        printf("rpc_binding_free returned 0x%x\n", status);
        return(status);
    }

    return(0);
}
```

MANAGER.C (INTEROP RPC Sample)

```
/*****************************************************************/
/**                                                             **/
/**                 Microsoft RPC Examples                      **/
/**                 OSF DCE Interop Application                 **/
/**             Copyright(c) Microsoft Corp. 1993               **/
/**                                                             **/
/*****************************************************************/

#include <stdio.h>
#include "msg.h"    /* header file generated by M/IDL compiler */

#if defined(__RPC_WIN32__) || defined(__RPC_DOS__)
#include "dceport.h"
#endif

/*
 * Print out client messages
 */

void ClientMessage(unsigned char *message)
{
    printf("%s\n", message);
    return;
}

/*
 * The client can stop the server by calling this operation.
 */
void ShutdownServer()
{
    unsigned32 status;
    rpc_mgmt_stop_server_listening(0, &status);
    if (status)
        {
        printf("rpc_mgmt_stop_server_listening returns 0x%x\n", status);
        }
    return;
}
```

MIDLUSER.C (INTEROP RPC Sample)

```
/*****************************************************************/
/**                                                             **/
/**                  Microsoft RPC Examples                     **/
/**            OSF DCE Interop Sample Application               **/
/**             Copyright(c) Microsoft Corp. 1993               **/
/**                                                             **/
/*****************************************************************/

#include <stdlib.h>
#include "msg.h"

/*
 * These are used in Microsoft clients and servers for stub memory
allocation
 */

void __RPC_FAR * __RPC_USER MIDL_user_allocate(size_t size)
{
    return malloc(size);
}

void __RPC_USER MIDL_user_free(void __RPC_FAR *p)
{
    free(p);
}
```

SERVER.C (INTEROP RPC Sample)

```
/*****************************************************************/
/**                                                             **/
/**                  Microsoft RPC Examples                     **/
/**                  OSF DCE Interop Application                **/
/**             Copyright(c) Microsoft Corp. 1993               **/
/**                                                             **/
/*****************************************************************/

#include <stdlib.h>
#include <stdio.h>
#include <ctype.h>

#include "msg.h"    /* header file generated by M/IDL compiler */

#if defined(__RPC_WIN32__) || defined(__RPC_DOS__)

/*
 On MS platforms we must include the dceport.h header file
 which maps OSF DCE style APIs to MS style APIs.

 Currently, we must also define a common interface handle name.
*/

#include "dceport.h"

#else
#include <pthread.h>
#endif

#ifndef _CRTAPI1
#define _CRTAPI1
#endif

void Usage()
{
  printf("Usage : server -e <endpoint>   - optional endpoint\n");
  printf("               -t <transport>  - optional, default
ncacn_ip_tcp\n");
  exit(1);
}

int _CRTAPI1
main(int argc, char *argv[])
{
    unsigned32 status;
    unsigned char * pszProtocolSequence = (unsigned char *)"ncacn_ip_tcp";
    unsigned char * pszEndpoint         = NULL;
    unsigned int    cMaxCalls           = 20;
    rpc_binding_vector_p_t pbvBindings  = NULL;
    int i;

    printf ("Microsoft RPC Demo - OSF DCE Interop Message Server\n");
```

```
for (i = 1; i < argc; i++) {
    if ((*argv[i] == '-') || (*argv[i] == '/')) {
        switch (tolower(*(argv[i]+1))) {
        case 'e':
            pszEndpoint = (unsigned char *)argv[++i];
            break;
        case 't':
            pszProtocolSequence = (unsigned char *)argv[++i];
            break;
        case 'h':
        case '?':
        default:
            Usage();
        }
    }
    else
        Usage();
}

if (pszEndpoint != NULL)
    {
    /*
     Since we have an explict endpoint, use it and
     wait for client requests.
    */
    rpc_server_use_protseq_ep(pszProtocolSequence,
                              cMaxCalls,
                              pszEndpoint,
                              &status);
    if (status) {
        printf("rpc_server_use_protseq_ep returned 0x%x\n", status);
        return status;
        }
    }
else
    {
    /*
     No explict endpoint, use the protocol sequence and register
     the endpoint with the endpoint mapper.
    */
    rpc_server_use_protseq(pszProtocolSequence,
                           cMaxCalls,
                           &status);
    if (status) {
        printf("rpc_server_use_protseq returned 0x%x\n", status);
        return status;
        }

    rpc_server_inq_bindings(&pbvBindings, &status);
    if (status) {
        printf("rpc_server_inq_bindings returned 0x%x\n", status);
        return status;
        }
```

```
    rpc_ep_register(interop_v1_0_s_ifspec,
                    pbvBindings,
                    0,
                    0,
                    &status);
    if (status) {
        printf("rpc_ep_register returned 0x%x\n", status);
        return status;
        }
    }

rpc_server_register_if(interop_v1_0_s_ifspec,
                       0,
                       0,
                       &status);
if (status) {
    printf("rpc_server_register_if returned 0x%x\n", status);
    return status;
}

printf("RPC server ready\n");
rpc_server_listen(cMaxCalls,&status);


if (status) {
    printf("rpc_server_listen returned: 0x%x\n", status);
    return status;
}

rpc_server_unregister_if(interop_v1_0_s_ifspec,
                         0,
                         &status);
if (status) {
    printf("rpc_server_unregister_if returned 0x%x\n", status);
    return status;
}

if (pszEndpoint == NULL)
    {
    /*
     Unregister from endpoint mapper
     */
    rpc_ep_unregister(interop_v1_0_s_ifspec,
                      pbvBindings,
                      0,
                      &status);
    if (status) {
        printf("rpc_ep_unregister returned 0x%x\n", status);
        return status;
        }

    rpc_binding_vector_free(&pbvBindings, &status);
    if (status) {
        printf("rpc_binding_vector_free returned 0x%x\n", status);
        return status;
```

```
                }
            }
        }
```

MANDEL

This directory contains the files for the sample distributed application "mandel":

File	Description
README.TXT	Readme file for the MANDEL sample
MDLRPC.IDL	Interface definition language file
MDLRPC.ACF	Attribute configuration file
MANDEL.C	Client main program
MANDEL.H	Client global data
REMOTE.C	Client code that calls remote procedures
RPC.ICO	Client icon
MANDEL.DEF	Client module definition file
MANDEL.RC	Client resource script file
SERVER.C	Server main program
CALC.C	Remote procedures
MAKEFILE	nmake utility for Windows NT
MAKEFILE.WIN	nmake utility for Win 3.x

BUILDING CLIENT AND SERVER APPLICATIONS FOR MICROSOFT WINDOWS NT:

The following environment variables should be set for you already.

```
set CPU=i386
set INCLUDE=c:\mstools\h
set LIB=c:\mstools\lib
set PATH=c:\winnt\system32;c:\mstools\bin;
```

For mips, set CPU=mips For alpha, set CPU=alpha

Build the sample distributed application:

```
nmake cleanall
nmake
```

This builds the executable programs client.exe and server.exe for Microsoft Windows NT.

BUILDING THE CLIENT APPLICATION FOR WINDOWS 3.*x*

After installing the Microsoft Visual C/C++ version 1.50 development environment and the Microsoft RPC version 2.0 toolkit on a Windows NT computer, you can build the sample client application from Windows NT.

```
nmake -f makefile.win cleanall
nmake -f makefile.win
```

This builds the client application client.exe.

You can also execute the Microsoft Visual C/C++ compiler under MS-DOS. This requires a two-step build process.

Step One: Compile the .IDL files under Windows NT

```
nmake -a -f makefile.win yield.h
```

Step Two: Compile the C sources (stub and application) under MS-DOS.

```
nmake -f makefile.win
```

RUNNING THE CLIENT AND SERVER APPLICATIONS

On the server, enter

```
server
```

On the client, enter

```
net start workstation
client
```

Note The client and server applications can run on the same Microsoft Windows NT computer when you use different screen groups. If you run the client on the Microsoft MS-DOS and Windows computer, choose the Run command from the File menu in the Microsoft Windows 3.x Program Manager and enter client.exe.

Several command line switches are available to change settings for the server program. For a listing of switches available from the server program, enter

```
server -?
```

MAKEFILE (MANDEL RPC Sample)

```
#************************************************************#
#**                                                        **#
#**                  Microsoft RPC Examples                **#
#**                Mandelbrot RPC Application              **#
#**            Copyright(c) Microsoft Corp. 1991           **#
#**                                                        **#
#************************************************************#
# The same source code is used to build either a standalone
# or an RPC version of the Microsoft Windows (R) Mandelbrot
# sample application.  The flag RPC determines which version
# is built.  To build a standalone version, use the commands:
#     >nmake cleanall
#     >set NOTRPC=1
#     >nmake
# To build the RPC version, use the commands:
#     >nmake cleanall
#     >set NOTRPC=
#     >nmake
!include <ntwin32.mak>

!ifdef NOTRPC
RPCFLAG =
!else
RPCFLAG = -DRPC
!endif

.c.obj:
   $(cc) $(cdebug) $(cflags) $(cvars) $(RPCFLAG) $<

# Targets
# The RPC version produces client and server executables.
# The standalone version produces a single exe file, "mandel".

!ifndef NOTRPC
all: client.exe server.exe
!else
all: mandel.exe
!endif

mandel.exe: mandel.obj remote.obj mandel.def mandel.rbj calc.obj
    $(link) $(linkdebug) $(guiflags) -out:mandel.exe -map:mandel.map \
      mandel.obj remote.obj calc.obj mandel.rbj $(guilibs)

client.exe: mandel.obj remote.obj mandel.def mandel.rbj mdlrpc_c.obj
    $(link) $(linkdebug) $(guiflags) -out:client.exe -map:client.map \
      mandel.obj remote.obj mdlrpc_c.obj \
      mandel.rbj rpcrt4.lib $(guilibs)

server.exe: server.obj calc.obj mdlrpc_s.obj
    $(link) $(linkdebug) $(conflags) -out:server.exe -map:server.map \
      server.obj calc.obj mdlrpc_s.obj \
      rpcrt4.lib $(conlibs)
```

```
# Update the resource if necessary
mandel.rbj: mandel.rc mandel.h
    rc -r mandel.rc
    cvtres -$(CPU) mandel.res -o mandel.rbj

# Object file dependencies

# server only built for RPC version; always needs mdlrpc.h
server.obj: server.c mandel.h mdlrpc.h

# Compile differently for RPC, standalone versions
!ifndef NOTRPC
mandel.obj: mandel.c mandel.h mdlrpc.h
remote.obj: remote.c mandel.h mdlrpc.h
calc.obj  : calc.c mandel.h mdlrpc.h
!else
mandel.obj: mandel.c mandel.h
remote.obj: remote.c mandel.h
calc.obj  : calc.c mandel.h
!endif

# client stub
mdlrpc_c.obj : mdlrpc_c.c mdlrpc.h

# server stub file
mdlrpc_s.obj : mdlrpc_s.c mdlrpc.h

# Stubs and header file from the IDL file
mdlrpc.h mdlrpc_c.c mdlrpc_s.c: mdlrpc.idl mdlrpc.acf
    midl -oldnames -cpp_cmd $(cc) -cpp_opt "-E" mdlrpc.idl

clean:
    -del client.exe
    -del server.exe
    -del mandel.exe

cleanall:  clean
    -del *.obj
    -del *.map
    -del *.res
    -del *.rbj
    -del mdlrpc_*.c
    -del mdlrpc.h
```

MANDEL.H (MANDEL RPC Sample)

```
/
******************************************************************************

    MANDEL.H -- Constants and function definitions for MANDEL.C

    Copyright (C) 1990, 1992 Microsoft Corporation

******************************************************************************
/

/* Constants */

#ifdef WIN16
#define APIENTRY        PASCAL
#define UNREFERENCED_PARAMETER
#endif

#define IDM_ABOUT        100
#define IDM_ZOOMIN       101
#define IDM_ZOOMOUT      105
#define IDM_TOP          106
#define IDM_REDRAW       107
#define IDM_EXIT         108
#define IDM_CONTINUOUS   109
#define IDM_PROTSEQ      110
#define IDD_PROTSEQNAME  111
#define IDM_SERVER       112
#define IDD_SERVERNAME   113
#define IDM_ENDPOINT     114
#define IDD_ENDPOINTNAME 115
#define IDM_BIND         116
#define IDM_GO           117
#define IDM_1LINE        200
#define IDM_2LINES       201
#define IDM_4LINES       202

#define WIDTH            300
#define HEIGHT           300
#define LINES              4
#define BUFSIZE         1200     // (HEIGHT * LINES)
#define MAX_BUFSIZE     4800     // (BUFSIZE * sizeof(short))

#define POLL_TIME        2000

#define CNLEN              25     // computer name length
#define UNCLEN       CNLEN+2      // \\computername
#define PATHLEN           260     // Path
#define MSGLEN            300     // arbitrary large number for message size
#define MAXPROTSEQ         20     // protocol sequence

#define NCOLORS           11
```

```
#define SVR_TABLE_SZ      20

// Status of connection to server
#define SS_DISCONN         0
#define SS_IDLE            1
#define SS_READPENDING     2
#define SS_PAINTING        3
#define SS_LOCAL           4

#define MINPREC       5.0E-9
#define MAXPREC       5.0E-3

#define WM_DOSOMEWORK   (WM_USER+0)
#define WM_PAINTLINE    (WM_USER+1)

#define EXCEPT_MSG      "The remote procedure raised an exception.\n\
Check your connection settings."

/* Data Structures */

typedef struct _svr_table {
    char    name[UNCLEN];       // name of remote server
    int     hfPipe;             // RPC handle
    int     iStatus;            // status of connection
    int     cPicture;           // picture id for this line
    DWORD   dwLine;             // line we're drawing
    int     cLines;             // lines in this chunk
} svr_table;

#ifndef RPC                     // If RPC, the following data would be
                                // defined in the IDL file
typedef struct _cpoint {
    double    real;
    double    imag;
} CPOINT;

typedef CPOINT * PCPOINT;

typedef struct _LONGRECT {
    long    xLeft;
    long    yBottom;
    long    xRight;
    long    yTop;
} LONGRECT;

typedef LONGRECT *PLONGRECT;

typedef unsigned short LINEBUF[BUFSIZE];

#endif

typedef struct _calcbuf {
    LONGRECT  rclDraw;
    double    dblPrecision;
    DWORD     dwThreshold;
```

```
    CPOINT    cptLL;
} CALCBUF;


/* Function Prototypes */

int WINAPI WinMain(HINSTANCE, HINSTANCE, LPSTR, int);
BOOL InitApplication(HANDLE);
BOOL InitInstance(HANDLE, int);

LONG APIENTRY MainWndProc(HWND, UINT, UINT, LONG);
BOOL APIENTRY About(HWND, UINT, UINT, LONG);
BOOL APIENTRY Protseq(HWND, UINT, UINT, LONG);
BOOL APIENTRY Server(HWND, UINT, UINT, LONG);
BOOL APIENTRY Endpoint(HWND, UINT, UINT, LONG);

#ifdef RPC
RPC_STATUS Bind(HWND);
#endif

void CountHistogram(void);

BOOL InitRemote(HWND);
BOOL CheckDrawStatus(HWND);
void SetNewCalc(CPOINT, double, RECT);
void IncPictureID(void);
void ResetPictureID(void);
BOOL CheckDrawingID(int);
DWORD QueryThreshold(void);

// buffer routines
BOOL TakeDrawBuffer(void);
LPVOID LockDrawBuffer(void);
void UnlockDrawBuffer(void);
void ReturnDrawBuffer(void);
void FreeDrawBuffer(void);



#ifndef RPC                     // If RPC, MandelCalc() would be
                                // defined in the IDL file
void MandelCalc(PCPOINT   pcptLL,
                PLONGRECT prcDraw,
                double    precision,
                DWORD     ulThreshold,
                LINEBUF * pbBuf);
#endif
```

CALC.C (MANDEL RPC Sample)

```
/
*****************************************************************************
                    Microsoft RPC Version 2.0
            Copyright Microsoft Corp. 1992, 1993, 1994
                        mandel Example

    FILE:       calc.c

    PURPOSE:    Server side of the RPC distributed application Mandel

    FUNCTIONS:  MandelCalc() - Do the calculations for the Windows
                               Mandelbrot Set distributed drawing program.

*****************************************************************************
/

#include <stdlib.h>
#include <stdio.h>
#include <windows.h>

#ifdef RPC
#include "mdlrpc.h"
#endif
#include "mandel.h"


short calcmand(double, double, short);


void MandelCalc(PCPOINT    pcptLL,
                PLONGRECT  prcDraw,
                double     precision,
                DWORD      ulThreshold,
                LINEBUF *  pbBuf)
{
    DWORD   h, height, width;
    double  dreal, dimag, dimag2;
    short   maxit = 0;
    short * pbPtr;

    pbPtr = *pbBuf;   // LINEBUF is an array of shorts

    dreal = pcptLL->real + ((double)prcDraw->xLeft * precision);
    dimag = pcptLL->imag + ((double)prcDraw->yBottom * precision);

    maxit = (short) ulThreshold;

    height = (prcDraw->yTop - prcDraw->yBottom) + 1;
    width = (prcDraw->xRight - prcDraw->xLeft) + 1;

    for ( ; width > 0; --width, dreal += precision) {
```

```
        for (dimag2 = dimag, h = height; h > 0; --h, dimag2 += precision) {
            if ((dreal > 4.0) || (dreal < -4.0) ||
                (dimag2 > 4.0) || (dimag2 < -4.0))
                *(pbPtr++) = 0;
            else
                *(pbPtr++) = calcmand(dreal, dimag2, maxit);
        }
    }
}


/* C version of the assembly language program */
short calcmand(double dreal,
               double dimag,
               short  maxit)
{
    double x, y, xsq, ysq;
    short k;

    k = maxit;
    x = dreal;
    y = dimag;

    while (1) {
        xsq = x * x;
        ysq = y * y;
        y = 2.0 * x * y + dimag;
        x = (xsq - ysq) + dreal;
        if (--k == 0)
            return((short) (maxit - k));
        if ((xsq + ysq) > 4.0)
            return((short) (maxit - k));
    }
}
```

MANDEL.C (MANDEL RPC Sample)

```
/
*****************************************************************************
                    Microsoft RPC Version 2.0
            Copyright Microsoft Corp. 1992, 1993, 1994
                        mandel Example

    FILE:       mandel.c

    PURPOSE:    Client side of the RPC distributed application

    COMMENTS:   Main code for the Windows Mandelbrot Set distributed
                drawing program.

*****************************************************************************
/

#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <string.h>
#include <time.h>
#include <windows.h>      // Required for all Windows applications
#include <windowsx.h>     // Allow portability from Win16, Win32

#ifdef RPC
#include "mdlrpc.h"       // header file generated by the MIDL compiler
#endif
#include "mandel.h"


/* data structures */

#ifdef RPC
char szTitle[] = "Mandelbrot RPC";
#else
char szTitle[] = "Mandelbrot Standalone";
#endif

CPOINT      cptUL = { (double) -2.05, (double) 1.4 };
double      dPrec = (double) .01;

HANDLE      hInst;          // current instance
HWND        hWND;           // Main window handle

svr_table   SvrTable;
int         iLines = LINES;

int         fContinueZoom = TRUE;
int         fZoomIn       = TRUE;

// split current picture into 16 regions
```

```
// zoom on most complex region;  region with most colors represented
int         Histogram[4][4][NCOLORS+1] = {0};
int         ColorCount[4][4] = {0};
int         Max[4][4] = {0};

int         iHistMaxI = 2;
int         iHistMaxJ = 3;

RECT        rcZoom;
BOOL        fRectDefined = FALSE;

#ifdef RPC
int             fBound = FALSE;     // flag indicates whether bound to svr
unsigned char * pszUuid             = NULL;
unsigned char pszProtocolSequence[MAXPROTSEQ] = "ncacn_np";
unsigned char pszEndpoint[PATHLEN]            = "\\pipe\\mandel";
unsigned char * pszOptions          = NULL;
unsigned char * pszStringBinding;
unsigned char   pszNetworkAddress[UNCLEN+1] = {'\0'};
#endif

/* function prototypes */

void     DoSomeWork(HWND, BOOL);
void     InitHistogram(void);
void     CalcHistogram(int, int, DWORD, DWORD);
void     PaintLine(HWND, svr_table *, HDC, int);
void     DrawRect(HWND, PRECT, BOOL, HDC);
COLORREF MapColor(DWORD, DWORD);


/*
 *  FUNCTION: WinMain(HANDLE, HANDLE, LPSTR, int)
 *
 *  PURPOSE: Calls initialization function, processes message loop
 *
 *  COMMENTS:
 *
 *      Windows recognizes this function by name as the initial entry point
 *      for the program.  This function calls the application initialization
 *      routine, if no other instance of the program is running, and always
 *      calls the instance initialization routine.  It then executes a
message
 *      retrieval and dispatch loop that is the top-level control structure
 *      for the remainder of execution.  The loop is terminated when a
WM_QUIT
 *      message is received, at which time this function exits the
application
 *      instance by returning the value passed by PostQuitMessage().
 *
 *      If this function must abort before entering the message loop, it
 *      returns the conventional value NULL.
 */

int WINAPI WinMain(
```

```
    HINSTANCE hInstance,            /* current instance         */
    HINSTANCE hPrevInstance,        /* previous instance        */
    LPSTR lpCmdLine,                /* command line             */
    int nCmdShow)                   /* show-window type (open/icon) */
{

    MSG msg;

    UNREFERENCED_PARAMETER(lpCmdLine);

    if (!hPrevInstance)  /* Other instances of app running? */
        if (!InitApplication(hInstance))  /* Initialize shared things */
            return(FALSE);  /* Exits if unable to initialize */

    /* Perform initializations that apply to a specific instance */
    if (!InitInstance(hInstance, nCmdShow))
        return(FALSE);


    /* Acquire and dispatch messages until a WM_QUIT message is received. */
    while (GetMessage(&msg,         /* message structure              */
                      (HWND)NULL,  /* handle of window receiving the message
*/
                      0,           /* lowest message to examine      */
                      0))          /* highest message to examine     */
    {
        TranslateMessage(&msg);    /* Translates virtual key codes   */
        DispatchMessage(&msg);     /* Dispatches message to window   */
    }

    return(msg.wParam);  /* Returns the value from PostQuitMessage */

}


/*
 *  FUNCTION: InitApplication(HANDLE)
 *
 *  PURPOSE: Initializes window data and registers window class
 *
 *  COMMENTS:
 *
 *      This function is called at initialization time only if no other
 *      instances of the application are running.  This function performs
 *      initialization tasks that can be done once for any number of running
 *      instances.
 *
 *      In this case, we initialize a window class by filling out a data
 *      structure of type WNDCLASS and calling the Windows RegisterClass()
 *      function.  Since all instances of this application use the same
window
 *      class, we only need to do this when the first instance is
initialized.
 */
```

```
BOOL InitApplication(HANDLE hInstance)    /* current instance */
{

    WNDCLASS  wc;

    /* Fill in window class structure with parameters that describe the
*/
    /* main window.
*/
    wc.style = 0;                         /* Class style(s).
*/
    wc.lpfnWndProc = (WNDPROC)MainWndProc;
                                          /* Function to retrieve messages for
*/
                                          /* windows of this class.
*/
    wc.cbClsExtra = 0;                    /* No per-class extra data.
*/
    wc.cbWndExtra = 0;                    /* No per-window extra data.
*/
    wc.hInstance = hInstance;             /* Application that owns the class.
*/
    wc.hIcon = LoadIcon(hInstance, "RPC_ICON");
    wc.hCursor = LoadCursor(0, IDC_ARROW);
    wc.hbrBackground = GetStockObject(WHITE_BRUSH);
    wc.lpszMenuName =  "MandelMenu";    /* Name of menu resource in .RC
file. */
    wc.lpszClassName = "MandelClass";   /* Name used in call to
CreateWindow. */

    /* Register the window class and return success/failure code. */
    return(RegisterClass(&wc));

}


/*
 *  FUNCTION:  InitInstance(HANDLE, int)
 *
 *  PURPOSE:  Saves instance handle and creates main window.
 *
 *  COMMENTS:
 *
 *      This function is called at initialization time for every instance of
 *      this application.  This function performs initialization tasks that
 *      cannot be shared by multiple instances.
 *
 *      In this case, we save the instance handle in a static variable and
 *      create and display the main program window.
 */

BOOL InitInstance(HANDLE   hInstance,   /* Current instance identifier.
*/
                  int      nCmdShow)    /* Param for first ShowWindow()
call. */
```

```
{
    HMENU           hMenu;
    RECT            rc;

    /* Save the instance handle in static variable, which will be used in
*/
    /* many subsequence calls from this application to Windows.
*/
    hInst = hInstance;

    /* Create a main window for this application instance.  */
    hWND = CreateWindow(
               "MandelClass",        /* See RegisterClass() call.
*/
               szTitle,              /* Text for window title bar.
*/
               WS_OVERLAPPED | WS_CAPTION | WS_SYSMENU | WS_BORDER |
WS_MINIMIZEBOX,
               CW_USEDEFAULT,        /* Default horizontal position.
*/
               CW_USEDEFAULT,        /* Default vertical position.
*/
               WIDTH,                /* Default width.
*/
               HEIGHT,               /* Default height.
*/
               (HWND) NULL,          /* Overlapped windows have no parent.
*/
               (HMENU) NULL,         /* Use the window class menu.
*/
               hInstance,            /* This instance owns this window.
*/
               (LPVOID) NULL         /* Pointer not needed.
*/
           );

    /* If window could not be created, return "failure" */
    if (!hWND)
        return(FALSE);

    /* Make the window visible; update its client area; and return "success"
*/
    ShowWindow(hWND, nCmdShow);  /* Show the window                 */
    UpdateWindow(hWND);          /* Sends WM_PAINT message          */
    rc.top = rc.left = 0;
    rc.bottom = HEIGHT-1;
    rc.right = WIDTH-1;

    SetNewCalc(cptUL, dPrec, rc);
    hMenu = GetMenu(hWND);

#ifndef RPC
    EnableMenuItem(hMenu, IDM_SERVER, MF_GRAYED);  /* disable option */
#endif
```

```
    return(TRUE);               /* Returns the value from PostQuitMessage */
}


/*
 *  FUNCTION: MainWndProc(HWND, unsigned, WORD, LONG)
 *
 *  PURPOSE:  Processes messages
 *
 *  MESSAGES:
 *      WM_COMMAND    - application menu
 *      WM_DESTROY    - destroy window
 *
 *  COMMENTS:
 */

LONG APIENTRY MainWndProc(
    HWND hWnd,                /* window handle               */
    UINT message,             /* type of message             */
    UINT wParam,              /* additional information      */
    LONG lParam)              /* additional information      */
{
    DLGPROC lpProc;           /* pointer to the dialog box function */
    PAINTSTRUCT ps;
    HDC hdc;
    static HDC     hdcMem;
    static HBITMAP hbmMem;
    static int     width;
    static int     height;
    RECT           rc;
    static BOOL    fButtonDown = FALSE;
    static POINT   pSelected;
    POINT          pMove;
    int            iWidthNew;
    int            iHeightNew;
    static int     miOldLines;
    double         scaling;

    switch (message) {

    case WM_CREATE:

#ifdef WIN16
        RpcWinSetYieldInfo (hWnd, FALSE, 0, 0L); // To make TCP/IP happy
#else
        PostMessage(hWnd, WM_COMMAND, IDM_BIND, 0L);    // bind to server
#endif

        if (!InitRemote(hWnd))
            return(FALSE);

        InitHistogram();

        hdc = GetDC(hWnd);
```

```
        hdcMem = CreateCompatibleDC(hdc);
        GetWindowRect(hWnd, &rc);
        width = rc.right - rc.left;
        height = rc.bottom - rc.top;
        hbmMem = CreateCompatibleBitmap(hdc, width, height);
        SelectObject(hdcMem, hbmMem);

        ReleaseDC(hWnd,hdc);

        rc.left = rc.top = 0;
        rc.right = width+1;
        rc.bottom = height + 1;
        FillRect(hdcMem, &rc, GetStockObject(WHITE_BRUSH));

        CheckMenuItem(GetMenu(hWnd), IDM_4LINES, MF_CHECKED);
        CheckMenuItem(GetMenu(hWnd), IDM_CONTINUOUS, MF_CHECKED);
        miOldLines = IDM_4LINES;  // save to uncheck
        break;

    case WM_PAINT:
        hdc = BeginPaint(hWnd, &ps);
        BitBlt(hdc,
               ps.rcPaint.left,
               ps.rcPaint.top,
               ps.rcPaint.right - ps.rcPaint.left,
               ps.rcPaint.bottom - ps.rcPaint.top,
               hdcMem,
               ps.rcPaint.left,
               ps.rcPaint.top,
               SRCCOPY);
        EndPaint(hWnd, &ps);
        break;

    case WM_COMMAND:  // message: command from application menu
        switch(wParam) {

        case IDM_BIND:

#ifdef RPC
            if (Bind(hWnd) != RPC_S_OK)
                PostMessage(hWnd, WM_DESTROY, 0, 0L);
#endif
            break;

        case IDM_ABOUT:
            lpProc = MakeProcInstance(About, hInst);

            DialogBox(hInst,       // current instance
                      "AboutBox",  // resource to use
                      hWnd,        // parent handle
                      lpProc);     // About() instance address

            FreeProcInstance(lpProc);
            break;
```

```
        case IDM_ZOOMOUT:
            if (dPrec > (double)MAXPREC)  // don't allow the zoom out
                break;

            rcZoom.left = WIDTH/4 + (WIDTH/8);  // center square
            rcZoom.top   = HEIGHT/4 + (HEIGHT/8);
            rcZoom.right = rcZoom.left + (WIDTH/4);
            rcZoom.bottom = rcZoom.top + (HEIGHT/4);

            cptUL.real -= (rcZoom.left * dPrec); // inverse of zoom in
            cptUL.imag += (rcZoom.top * dPrec);
            iWidthNew = (rcZoom.right - rcZoom.left + 1);
            iHeightNew = (rcZoom.bottom - rcZoom.top + 1);
            scaling = ((double) ((iWidthNew > iHeightNew) ? iWidthNew :
iHeightNew) / (double) width);
            dPrec /= scaling;

            rc.left = rc.top = 0;
            rc.bottom = height - 1;
            rc.right = width - 1;

            SetNewCalc(cptUL, dPrec, rc);
            fRectDefined = FALSE;
            DoSomeWork(hWnd, FALSE);
            break;

        case IDM_ZOOMIN:  // zoom in on selected rectangle
            // if no rectangle, don't zoom in
            if (!fRectDefined)
                break;

            if (dPrec < (double)MINPREC)  // don't allow zoom in
                break;

            DrawRect(hWnd, &rcZoom, TRUE, hdcMem);  // draw new rect

            // calculate new upper-left
            cptUL.real += (rcZoom.left * dPrec);
            cptUL.imag -= (rcZoom.top * dPrec);

            iWidthNew = (rcZoom.right - rcZoom.left + 1);
            iHeightNew = (rcZoom.bottom - rcZoom.top + 1);
            scaling = ((double) ((iWidthNew > iHeightNew) ? iWidthNew :
iHeightNew) / (double) width);

            dPrec *= scaling;

            rc.left = rc.top = 0;
            rc.bottom = height - 1;
            rc.right = width - 1;

            SetNewCalc(cptUL, dPrec, rc);
            IncPictureID();
```

```
    fRectDefined = FALSE;
    DoSomeWork(hWnd, FALSE);
    break;

case IDM_CONTINUOUS:  // continuous zoom in
    if (fContinueZoom == TRUE) {
        CheckMenuItem(GetMenu(hWnd), IDM_CONTINUOUS, MF_UNCHECKED);
        fContinueZoom = FALSE;
    }
    else {
        CheckMenuItem(GetMenu(hWnd), IDM_CONTINUOUS, MF_CHECKED);
        fContinueZoom = TRUE;
    }
    break;

case IDM_REDRAW:
    if (fContinueZoom == TRUE)
        InitHistogram();

    rc.left = rc.top = 0;
    rc.right = width+1;
    rc.bottom = height + 1;
    FillRect(hdcMem, &rc, GetStockObject(WHITE_BRUSH));
    InvalidateRect(hWnd, NULL, TRUE);

    rc.left = rc.top = 0;
    rc.bottom = height - 1;
    rc.right = width - 1;
    SetNewCalc( cptUL, dPrec, rc);

    fRectDefined = FALSE;
    DoSomeWork(hWnd, FALSE);
    break;

case IDM_EXIT:
    DestroyWindow(hWnd);
    FreeDrawBuffer();
    break;

case IDM_TOP:
    cptUL.real = (double) -2.05;
    cptUL.imag = (double) 1.4;
    dPrec = .01;

    rc.left = rc.top = 0;
    rc.bottom = height - 1;
    rc.right = width - 1;

    SetNewCalc(cptUL, dPrec, rc);
    ResetPictureID();  // incremented past original

    fRectDefined = FALSE;
    DoSomeWork(hWnd, FALSE);
    break;
```

```
        case IDM_1LINE:

        case IDM_2LINES:

        case IDM_4LINES:

            CheckMenuItem(GetMenu(hWnd), miOldLines, MF_UNCHECKED);
            miOldLines = wParam;
            switch(wParam) {

            case IDM_1LINE:
                iLines = 1;
                break;
            case IDM_2LINES:
                iLines = 2;
                break;
            case IDM_4LINES:
                iLines = 4;
                break;
            }

            CheckMenuItem(GetMenu(hWnd), miOldLines, MF_CHECKED);
            break;

#ifdef RPC
        case IDM_PROTSEQ:
            lpProc = MakeProcInstance(Protseq, hInst);
            DialogBox(hInst,        // current instance
                      "ProtseqBox", // resource to use
                      hWnd,         // parent handle
                      lpProc);      // Server instance address

            FreeProcInstance(lpProc);
            break;

        case IDM_SERVER:
            lpProc = MakeProcInstance(Server, hInst);
            DialogBox(hInst,         // current instance
                      "ServerBox",   // resource to use
                      hWnd,          // parent handle
                      lpProc);       // Server  instance address
            FreeProcInstance(lpProc);
            break;

        case IDM_ENDPOINT:
            lpProc = MakeProcInstance(Endpoint, hInst);
            DialogBox(hInst,        // current instance
                      "EndpointBox",// resource to use
                      hWnd,         // parent handle
                      lpProc);      // Server instance address
            FreeProcInstance(lpProc);
            break;
#endif
        case IDM_GO:
             SetTimer(hWnd, 1, POLL_TIME, NULL);  // set timer for polls
```

```
            EnableMenuItem(GetMenu(hWnd), IDM_GO, MF_GRAYED);    // disable
GO
            break;


        default:  // Lets Windows process it
            return(DefWindowProc(hWnd, message, wParam, lParam));

        }

        break;

    case WM_DESTROY:  // message: window being destroyed
        PostQuitMessage(0);
        DeleteDC(hdcMem);
        DeleteObject(hbmMem);
        break;

    case WM_DOSOMEWORK:  // do another slice of calculation work
        DoSomeWork(hWnd, FALSE);
        break;

    case WM_PAINTLINE:  // The shared buffer contains a line of data; draw
it
        PaintLine(hWnd,
                  &SvrTable,
                  hdcMem,
                  height);
        break;

    case WM_TIMER:  // timer means we should do another slice of work
        DoSomeWork(hWnd, TRUE);
        break;

    case WM_LBUTTONDOWN:  // left button down; start to define a zoom
rectangle
        if (fRectDefined)
            DrawRect(hWnd, &rcZoom, FALSE, hdcMem);  // undraw old rectangle

        // initialize rectangle
        rcZoom.left = rcZoom.right = pSelected.x = LOWORD(lParam);
        rcZoom.top = rcZoom.bottom = pSelected.y = HIWORD(lParam);

        // draw the new rectangle
        DrawRect(hWnd, &rcZoom, TRUE, hdcMem);

        fRectDefined = TRUE;
        fButtonDown = TRUE;
        SetCapture(hWnd);  // capture all mouse events
        break;

    case WM_MOUSEMOVE:  // mouse move
        // if the button is down, change the rect
        if (!fButtonDown)
            break;
```

```
        DrawRect(hWnd, &rcZoom, FALSE, hdcMem);  // undraw old rect

        pMove.x = LOWORD(lParam);
        pMove.y = HIWORD(lParam);

        // update the selection rectangle
        if (pMove.x <= pSelected.x)
            rcZoom.left = pMove.x;
        if (pMove.x >= pSelected.x)
            rcZoom.right = pMove.x;
        if (pMove.y <= pSelected.y)
            rcZoom.top = pMove.y;
        if (pMove.y >= pSelected.y)
            rcZoom.bottom = pMove.y;

        DrawRect(hWnd, &rcZoom, TRUE, hdcMem);  // draw new rect
        break;

    case WM_LBUTTONUP:  // button up; end selection
        fButtonDown = FALSE;
        ReleaseCapture();
        break;

    default:  // Passes it on if unproccessed
        return(DefWindowProc(hWnd, message, wParam, lParam));

    }

    return(0L);
}


/*
 *  FUNCTION: About(HWND, unsigned, WORD, LONG)
 *
 *  PURPOSE:  Processes messages for "About" dialog box
 *
 *  MESSAGES:
 *
 *      WM_INITDIALOG - initialize dialog box
 *      WM_COMMAND    - Input received
 *
 *  COMMENTS:
 *
 *      No initialization is needed for this particular dialog box, but TRUE
 *      must be returned to Windows.
 *
 *      Wait for user to click on "Ok" button, then close the dialog box.
 */

BOOL APIENTRY About(
    HWND hDlg,              /* window handle of the dialog box */
    UINT message,           /* type of message                 */
    UINT wParam,            /* message-specific information    */
```

```
    LONG lParam)
{
    UNREFERENCED_PARAMETER(lParam);

    switch (message) {

    case WM_INITDIALOG:     /* message: initialize dialog box  */
        return(TRUE);

    case WM_COMMAND:        /* message: received a command     */
        if (wParam == IDOK || wParam == IDCANCEL)
        {
            EndDialog(hDlg, TRUE);    /* Exits the dialog box  */
            return(TRUE);
        }
        break;
    }

    return(FALSE);          /* Didn't process a message        */
}



/*
 *  FUNCTION: Protseq(HWND, unsigned, WORD, LONG)
 *
 *  PURPOSE:  Processes messages for "Protseq" dialog box
 *
 *  MESSAGES:
 *
 *      WM_INITDIALOG - initialize dialog box
 *      WM_COMMAND    - Input received
 *
 *  COMMENTS:
 *
 *      No initialization is needed for this particular dialog box, but TRUE
 *      must be returned to Windows.
 *
 *      Wait for user to click on "Ok" button, then close the dialog box.
 */
BOOL APIENTRY Protseq(
    HWND hDlg,               /* window handle of the dialog box */
    UINT message,            /* type of message             */
    UINT wParam,             /* message-specific information    */
    LONG lParam)
{

    UNREFERENCED_PARAMETER(lParam);

#ifdef RPC

    switch (message) {

    case WM_INITDIALOG:    // message: initialize dialog box
```

```
        SetDlgItemText((HANDLE)hDlg, IDD_PROTSEQNAME, pszProtocolSequence);
        return(TRUE);

    case WM_COMMAND:       // message: received a command
        switch(wParam) {

        case IDCANCEL:     // System menu close command?
            EndDialog(hDlg, FALSE);
            return(TRUE);

        case IDOK:         // "OK" box selected?
            GetDlgItemText(hDlg, IDD_PROTSEQNAME, pszProtocolSequence,
MAXPROTSEQ);

            if (Bind(hDlg) != RPC_S_OK) {
                EndDialog(hDlg, FALSE);
                return(FALSE);
            }
             KillTimer(hWND, 1);  // stop timer for polls
            EnableMenuItem(GetMenu(hWND), IDM_GO, MF_ENABLED);   // enable
GO

            EndDialog(hDlg, TRUE);
            return(TRUE);

        }

    }

#endif

    return(FALSE);  // Didn't process a message
}


/*
 *  FUNCTION: Server(HWND, unsigned, WORD, LONG)
 *
 *  PURPOSE:  Processes messages for "Server" dialog box
 *
 *  MESSAGES:
 *
 *      WM_INITDIALOG - initialize dialog box
 *      WM_COMMAND    - Input received
 *
 *  COMMENTS:
 *
 *      No initialization is needed for this particular dialog box, but TRUE
 *      must be returned to Windows.
 *
 *      Wait for user to click on "Ok" button, then close the dialog box.
 */

BOOL APIENTRY Server(
    HWND hDlg,               /* window handle of the dialog box */
```

```
    UINT message,            /* type of message             */
    UINT wParam,             /* message-specific information    */
    LONG lParam)
{

    UNREFERENCED_PARAMETER(lParam);

#ifdef RPC

    switch (message) {

    case WM_INITDIALOG:      /* message: initialize dialog box */
        SetDlgItemText(hDlg, IDD_SERVERNAME, pszNetworkAddress);
        return(TRUE);

    case WM_COMMAND:         /* message: received a command    */
        switch(wParam) {

        case IDCANCEL:       /* System menu close command?     */
            EndDialog(hDlg, FALSE);
            return(TRUE);

        case IDOK:            /* "OK" box selected?           */
            GetDlgItemText( hDlg, IDD_SERVERNAME, pszNetworkAddress,
UNCLEN);

            if (Bind(hDlg) != RPC_S_OK) {
                EndDialog(hDlg, FALSE);
                return(FALSE);
            }
             KillTimer(hWND, 1);  // stop timer for polls
            EnableMenuItem(GetMenu(hWND), IDM_GO, MF_ENABLED);   // enable
GO

            EndDialog(hDlg, TRUE);
            return(TRUE);
        }

    }

#endif

    return(FALSE);           /* Didn't process a message      */

}

/*
 *  FUNCTION: Endpoint(HWND, unsigned, WORD, LONG)
 *
 *  PURPOSE:  Processes messages for "Endpoint" dialog box
 *
 *  MESSAGES:
 *
 *      WM_INITDIALOG - initialize dialog box
```

```
 *      WM_COMMAND    - Input received
 *
 *  COMMENTS:
 *
 *      No initialization is needed for this particular dialog box, but TRUE
 *      must be returned to Windows.
 *
 *      Wait for user to click on "Ok" button, then close the dialog box.
 */

BOOL APIENTRY Endpoint(
    HWND hDlg,                /* window handle of the dialog box */
    UINT message,             /* type of message             */
    UINT wParam,              /* message-specific information    */
    LONG lParam)
{

    UNREFERENCED_PARAMETER(lParam);

#ifdef RPC

    switch (message) {

    case WM_INITDIALOG:    // message: initialize dialog box
        SetDlgItemText(hDlg, IDD_ENDPOINTNAME, pszEndpoint);
        return(TRUE);

    case WM_COMMAND:       // message: received a command
        switch(wParam) {

        case IDCANCEL:
            EndDialog(hDlg, FALSE);
            return(TRUE);

        case IDOK:
            GetDlgItemText(hDlg, IDD_ENDPOINTNAME, pszEndpoint, PATHLEN);

            if (Bind(hDlg) != RPC_S_OK) {
                EndDialog(hDlg, FALSE);
                return(FALSE);
            }
             KillTimer(hWND, 1);  // stop timer for polls
            EnableMenuItem(GetMenu(hWND), IDM_GO, MF_ENABLED);   // enable
GO

            EndDialog(hDlg, TRUE);
            return(TRUE);

        }

    }

#endif

    return(FALSE);  // Didn't process a message
```

```
}


/*
 *  DoSomeWork --
 *
 *  This function does our work for us. It does it in little pieces, and
 *  will schedule itself as it sees fit.
 */

void
DoSomeWork(HWND    hwnd,
           BOOL    fTimer)
{
    static WORD   wIteration = 0;

    if (fTimer) {
        wIteration++;

        // on every nth tick, we send out a poll
        if (wIteration == 120) {  // tune this?
            wIteration = 0;
            return;
        }

        // on the half-poll, we check for responses
        if ((wIteration == 2) || (wIteration == 10)) {
            return;
        }
    }

    if (CheckDrawStatus(hwnd))
        SendMessage(hwnd, WM_DOSOMEWORK, 0, 0L);

    return;
}


/*
 *  DrawRect --
 *
 *  This function draws (or undraws) the zoom rectangle.
 */

void
DrawRect(HWND      hwnd,
         PRECT     prc,
         BOOL      fDrawIt,
         HDC       hdcBM)
{
    HDC     hdc;
    DWORD   dwRop;

    hdc = GetDC(hwnd);
```

```
    if (fDrawIt)
        dwRop = NOTSRCCOPY;
    else
        dwRop = SRCCOPY;

    // top side
    BitBlt(hdc, prc->left, prc->top, (prc->right - prc->left) + 1,
           1, hdcBM, prc->left, prc->top, dwRop);

    // bottom side
    BitBlt(hdc, prc->left, prc->bottom, (prc->right - prc->left) + 1,
           1, hdcBM, prc->left, prc->bottom, dwRop);

    // left side
    BitBlt(hdc,prc->left, prc->top, 1, (prc->bottom - prc->top) + 1,
           hdcBM, prc->left, prc->top, dwRop);

    // right side
    BitBlt(hdc,prc->right, prc->top, 1, (prc->bottom - prc->top) + 1,
           hdcBM, prc->right, prc->top, dwRop);

    ReleaseDC(hwnd, hdc);
}


/*
 *  PaintLine --
 *
 *  This function paints a buffer of data into the bitmap.
 */

void
PaintLine(HWND        hwnd,
          svr_table * pst,
          HDC         hdcBM,
          int         cHeight)
{
    LPWORD  pwDrawData;
    int     y;
    int     x;
    DWORD   dwThreshold;
    RECT    rc;
    WORD    lines;

    lines  = (WORD) pst->cLines;

    // picture ID had better match, or else we skip it
    if (CheckDrawingID(pst->cPicture))
    {
        // figure out our threshold
        dwThreshold = QueryThreshold();

        // get a pointer to the draw buffer
        pwDrawData = (LPWORD) LockDrawBuffer();
```

```
        if (pwDrawData == NULL) {
            ReturnDrawBuffer();
            return;
        }

        // starting x coordinate
        x = (int) pst->dwLine;

        // now loop through the rectangle
        while (lines-- > 0)
        {
            // bottom to top, since that's the order of the data in the
buffer
            y = (int) cHeight-1;

            while (y >= 0)
            {
                // draw a pixel
                SetPixel(hdcBM, x,y, MapColor(*pwDrawData, dwThreshold));

                if (fContinueZoom == TRUE)
                    CalcHistogram(x, y, *pwDrawData, dwThreshold);

                // now increment buffer pointer and y coord
                y--;
                pwDrawData++;
            }

            // increment X coordinate
            x++;
        }

        // figure out the rectangle to invalidate
        rc.top = 0;
        rc.bottom = cHeight;
        rc.left = (int)(pst->dwLine);
        rc.right = (int)(pst->dwLine) + pst->cLines;

        UnlockDrawBuffer();

        // and invalidate it on the screen so we redraw it
        InvalidateRect(hwnd, &rc, FALSE);
    }

    // free this for someone else to use
    ReturnDrawBuffer();

    // and change the pipe state, if necessary
    if (pst->iStatus == SS_PAINTING)
        pst->iStatus = SS_IDLE;

}


#define CLR_BLACK       RGB(0,0,0)
```

```
#define CLR_DARKBLUE    RGB(0,0,127)
#define CLR_BLUE        RGB(0,0,255)
#define CLR_CYAN        RGB(0,255,255)
#define CLR_DARKGREEN   RGB(0,127,0)
#define CLR_GREEN       RGB(0,255,0)
#define CLR_YELLOW      RGB(255,255,0)
#define CLR_RED         RGB(255,0,0)
#define CLR_DARKRED     RGB(127,0,0)
#define CLR_WHITE       RGB(255,255,255)
#define CLR_PALEGRAY    RGB(194,194,194)
#define CLR_DARKGRAY    RGB(127,127,127)


static COLORREF ColorMapTable[] = {     // size = NCOLORS
    CLR_DARKBLUE,
    CLR_BLUE,
    CLR_CYAN,
    CLR_DARKGREEN,
    CLR_GREEN,
    CLR_YELLOW,
    CLR_RED,
    CLR_DARKRED,
    CLR_WHITE,
    CLR_PALEGRAY,
    CLR_DARKGRAY};


/*
 *  MapColor --
 *
 *  This function maps an iteration count into a corresponding RGB color.
 */

COLORREF
MapColor(DWORD  dwIter,
         DWORD  dwThreshold)
{

    /* if it's beyond the threshold, call it black */
    if (dwIter >= dwThreshold) {
        return(CLR_BLACK);
    }

    /* get a modulus based on the number of colors */
    dwIter = (dwIter / 3) % NCOLORS; // 11;

    /* and return the appropriate color */
    return(ColorMapTable[dwIter]);

}


/*
 * CalcHistogram --
```

```
 *
 * This function is used to select the region that is the
 * most complex and will be used to zoom in for the next picture;
 * it contains the most colors.  The number of colors are counted.
 */

void
CalcHistogram(int    x,
              int    y,
              DWORD  dwIter,
              DWORD  dwThreshold)
{

    /* if it's beyond the threshold, call it black */
    if (dwIter >= dwThreshold) {
        Histogram[x/(WIDTH/4)][y/(HEIGHT/4)][NCOLORS]++;
        return;
    }

    /* get a modulus based on the number of colors */
    dwIter = (dwIter / 3) % NCOLORS; // 11;

    /* and bump the count for the appropriate color */
    Histogram[x/(WIDTH/4)][y/(HEIGHT/4)][dwIter]++;  // region of map

    return;

}


/*
 * InitHistogram --
 *
 * This function initializes the histogram data structures.
 */

void InitHistogram(void)
{
    int i, j, k;

    for (i = 0; i < 4; i++)
        for (j = 0; j < 4; j++)
            for (k = 0; k <= NCOLORS; k++)
                Histogram[i][j][k] = 0;  // count of colors
}


/*
 * CountHistogram --
 *
 * This function determines the number of colors represented
 * within a region.  The region with the most colors is
 * selected using the maxi and maxj values.  X and Y coordinates
 * corresponding to these regions are stored in the HistRegion
 * table and are used for the next picture.
```

```
 */

void CountHistogram(void)
{

    int i, j, k;

    /* count the number of colors in each region */
    /* find the color that dominates each region */
    for (i = 0; i < 4; i++) {
        for (j = 0; j < 4; j++) {
            ColorCount[i][j] = 0;
            Max[i][j] = 0;
            for (k = 0; k <= NCOLORS; k++) {
                if (Histogram[i][j][k] > Max[i][j])
                    Max[i][j] = Histogram[i][j][k];
                if (Histogram[i][j][k] != 0)  // count of colors
                    ColorCount[i][j]++;
            }
        }
    }

    iHistMaxI = 0;
    iHistMaxJ = 0;

    /* if several regions have the same number of colors,        */
    /* select the region with the most variety: the smallest max */
    for (i = 0; i < 4; i++) {
        for (j = 0; j < 4; j++) {
            if (   (ColorCount[i][j] >= ColorCount[iHistMaxI][iHistMaxJ])
                && (Max[i][j] < Max[iHistMaxI][iHistMaxJ]) ) {
                iHistMaxI = i;
                iHistMaxJ = j;
            }
        }
    }

    InitHistogram();  // initialize for next time

}


#ifdef RPC

void __RPC_FAR * __RPC_API midl_user_allocate(size_t len)
{
    UNREFERENCED_PARAMETER(len);
    return(NULL);
}


void __RPC_API midl_user_free(void __RPC_FAR * ptr)
{
    UNREFERENCED_PARAMETER(ptr);
```

```
    return;
}


/*
 *  FUNCTION: Bind(HWND)
 *
 *  PURPOSE:  Make RPC API calls to bind to the server application
 *
 *  COMMENTS:
 *
 *      The binding calls are made from InitInstance() and whenever
 *      the user changes the server name or endpoint. If the bind
 *      operation is successful, the global flag fBound is set to TRUE.
 *
 *      The global flag fBound is used to determine whether to call
 *      the RPC API function RpcBindingFree.
 */

RPC_STATUS Bind(HWND hWnd)
{
    RPC_STATUS status;
    char pszFail[MSGLEN];

    if (fBound == TRUE) {  // unbind only if bound
        status = RpcStringFree(&pszStringBinding);
        if (status) {
            sprintf(pszFail, "RpcStringFree failed 0x%x", status);
            MessageBox(hWnd,
                       pszFail,
                       "RPC Sample Application",
                       MB_ICONSTOP);
            return(status);
        }

        status = RpcBindingFree(&hMandel);
        if (status) {
            sprintf(pszFail, "RpcBindingFree failed 0x%x", status);
            MessageBox(hWnd,
                       pszFail,
                       "RPC Sample Application",
                       MB_ICONSTOP);
            return(status);
        }

        fBound = FALSE;  // unbind successful; reset flag
    }

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

```
        sprintf(pszFail, "RpcStringBindingCompose returned: (0x%x)\nNetwork
Address = %s\n",
              status, pszNetworkAddress);
        MessageBox(hWnd, pszFail, "RPC Sample Application",
MB_ICONINFORMATION);
        return(status);
    }

    status = RpcBindingFromStringBinding(pszStringBinding,
                                         &hMandel);
    if (status) {
        sprintf(pszFail, "RpcBindingFromStringBinding returned: (0x%x)
\nString = %s\n",
             status, pszStringBinding);
        MessageBox(hWnd, pszFail, "RPC Sample Application",
MB_ICONINFORMATION);
        return(status);
    }

    fBound = TRUE;  // bind successful; reset flag

    return(status);
}

#endif


/* end mandel.c */
```

REMOTE.C (MANDEL RPC Sample)

```
/
******************************************************************************
                  Microsoft RPC Version 2.0
          Copyright Microsoft Corp. 1992, 1993, 1994
                     mandel Example

    FILE:       remote.c

    PURPOSE:    Client side of the RPC distributed application Mandel

    COMMENTS:   Code to do the remote calculations for the Windows
                Mandelbrot Set distributed drawing program.

                Information coming into this module (via API calls) is
                based on upper-left being (0,0) (the Windows standard).
                We translate that to lower-left is (0,0) before we ship
                it out onto the net, and we do reverse translations
                accordingly.

                The iteration data is passed back to the main window
                procedure (by means of a WM_PAINTLINE message) which
                draws the picture.

                A word about the shared buffer: multiple buffers could
                be used, but a single one is used. The buffer is requested
                in this code, and then released after the data has been
                drawn (in PaintLine() in mandel.c). So long as the painting
                is done quickly, this is efficient.

******************************************************************************
/

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <windows.h>

#ifdef RPC
#include "mdlrpc.h"
#endif
#include "mandel.h"


/*
 * External variables
 */
extern int          fBound;
extern svr_table    SvrTable;    // the server table
extern int          iLines;
extern double       dPrec;
extern int          fContinueZoom;
extern int          fZoomIn;
```

```
extern int          iHistMaxI;
extern int          iHistMaxJ;
extern RECT         rcZoom;
extern BOOL         fRectDefined;


/*
 * Picture information
 */
int             cPictureID = 0;    // picture id, in case we reset in the
middle
static CPOINT   cptLL;             // upper-left
static double   dPrecision;        // precision of draw
static LONGRECT rclPicture;        // rectangle defining client window
static DWORD    dwCurrentLine;     // next line to be drawn
static DWORD    dwThreshold;       // threshold for iterations

/*
 * Function prototypes for local procs
 */
DWORD CalcThreshold(double);


/*
 *  InitRemote --
 *
 *  This function initializes everything for our remote connections.
 *  It gets the local wksta name (making sure the wksta is started)
 *  and it creates the mailslot with which to collect replies to our poll.
 *
 *  RETURNS
 *      TRUE    - initialization succeeded
 *      FALSE   - initialization failed, can't go on
 */

BOOL InitRemote(HWND hWnd)
{

#ifndef RPC
    UNREFERENCED_PARAMETER(hWnd);
#endif

    // set up our local entry
    strcpy(SvrTable.name, "Local machine");
    SvrTable.iStatus = SS_LOCAL;

    // good, we succeeded
    return(TRUE);
}


/*
 *  CheckDrawStatus --
 *
 *  This function does a check of all buffers being drawn.
```

```
 *
 *  If it finds an idle pipe, and there is work to be done, it assigns
 *      a line, and writes out the request.
 *  If it finds a read-pending pipe, it checks if the read has completed.
 *      If it has, it is read and a message is sent so the read data can
 *      be processed.
 *
 *  RETURNS
 *      TRUE    - we did a piece of work
 *      FALSE   - we could not find any work to do.
 */

BOOL CheckDrawStatus(HWND hwnd)
{
    CALCBUF     cb;
    LPVOID      pbBuf;

    while(TRUE) {

        // Check the status

        switch(SvrTable.iStatus) {

        case SS_PAINTING:
            break;

        case SS_IDLE:
            break;

        case SS_LOCAL:
            // Do a chunk of work locally

#ifdef RPC
            if (fBound == FALSE)
                break;
#endif

            if ((long) dwCurrentLine > rclPicture.xRight) {
                if (fContinueZoom == TRUE) {
                    if ((fZoomIn == TRUE) && (dPrec < (double)MINPREC))
                        fZoomIn = FALSE;  // start zooming out
                    if ((fZoomIn == FALSE) && (dPrec > (double)MAXPREC))
                        fZoomIn = TRUE;
                    if (fZoomIn) {
                        CountHistogram();
                        rcZoom.top    = iHistMaxJ * (WIDTH/4);
                        rcZoom.bottom = rcZoom.top + (WIDTH/4) - 1;
                        rcZoom.left   = iHistMaxI * (HEIGHT/4);
                        rcZoom.right  = rcZoom.left + (HEIGHT/4) - 1;
                        fRectDefined = TRUE;
                        PostMessage(hwnd, WM_COMMAND, IDM_ZOOMIN, 0L);
                    }
                    else
                        PostMessage(hwnd, WM_COMMAND, IDM_ZOOMOUT, 0L);
                }
```

```
                break;
            }

            if (TakeDrawBuffer() == FALSE)
                break;

            pbBuf = LockDrawBuffer();

            cb.rclDraw.xLeft = dwCurrentLine;
            cb.rclDraw.xRight = dwCurrentLine + iLines - 1;
            cb.rclDraw.yTop = rclPicture.yTop;
            cb.rclDraw.yBottom = rclPicture.yBottom;

            RpcTryExcept {
                MandelCalc(&cptLL,
                           &(cb.rclDraw),
                           dPrecision,
                           dwThreshold,
                           (LINEBUF *) pbBuf);
            }
            RpcExcept(1) {
                char szFail[MSGLEN];

                 sprintf (szFail, "%s (0x%x)\n", EXCEPT_MSG,
RpcExceptionCode());
                MessageBox(hwnd,
                           szFail,
                           "Remote Procedure Call",
                           MB_ICONINFORMATION);
                  KillTimer(hwnd, 1);  // stop timer for polls
                 EnableMenuItem(GetMenu(hwnd), IDM_GO, MF_ENABLED);    //
enable GO
                 UnlockDrawBuffer();

                     ReturnDrawBuffer();
                   return(FALSE);
            }
            RpcEndExcept

            UnlockDrawBuffer();

            SvrTable.cPicture = cPictureID;
            SvrTable.dwLine = dwCurrentLine;
            SvrTable.cLines = iLines;

            PostMessage(hwnd, WM_PAINTLINE, 0, 0L);
            dwCurrentLine += iLines;

            return(TRUE);
        }

        return(FALSE);
    }
}
```

```
/*
 *  SetNewCalc --
 *
 *  This sets up new information for a drawing and
 *  updates the drawing ID so any calculations in progress will not
 *  be mixed in.
 */

void SetNewCalc(CPOINT cptUL, double dPrec, RECT rc)
{
    // First, translate from upper left to lower left
    cptLL.real = cptUL.real;
    cptLL.imag = cptUL.imag - (dPrec * (rc.bottom - rc.top));

    // Now the precision
    dPrecision = dPrec;

    // The rectangle. Once again, translate.
    rclPicture.xLeft = (long) rc.left;
    rclPicture.xRight = (long) rc.right;
    rclPicture.yBottom = (long) rc.top;
    rclPicture.yTop = (long) rc.bottom;

    // Current line, start of drawing
    dwCurrentLine = rclPicture.xLeft;

    dwThreshold = CalcThreshold(dPrecision);
}


void IncPictureID(void)
{
    cPictureID++;
}


void ResetPictureID(void)
{
    cPictureID = 0;
}


/*
 *  CheckDrawing --
 *
 *  Just a sanity check here -- a function to check to make sure that we're
 *  on the right drawing
 */

BOOL CheckDrawingID(int id)
{
    return((id == cPictureID) ? TRUE : FALSE);
}
```

```
/*
 *  TakeDrawBuffer ensures only one pipe read at a time.
 *  LockDrawBuffer locks the handle and returns a pointer.
 *  UnlockDrawBuffer unlocks the handle.
 *  ReturnDrawBuffer lets another pipe read go.
 *  FreeDrawBuffer ensures the allocated buffer is freed upon exit.
 */

static BOOL fBufferTaken = FALSE;
static HANDLE hSharedBuf = (HANDLE) NULL;


BOOL TakeDrawBuffer(void)
{
    if (fBufferTaken) {
        return(FALSE);
    }

    if (hSharedBuf == (HANDLE) NULL) {
        hSharedBuf = LocalAlloc(LMEM_MOVEABLE, MAX_BUFSIZE);
        if (hSharedBuf == (HANDLE) NULL)
            return(FALSE);
    }

    fBufferTaken = TRUE;
    return(TRUE);
}


LPVOID LockDrawBuffer(void)
{
    if (hSharedBuf == (HANDLE) NULL)
        return(NULL);

    return(LocalLock(hSharedBuf));
}


void UnlockDrawBuffer(void)
{
    LocalUnlock(hSharedBuf);
}


void ReturnDrawBuffer(void)
{
    fBufferTaken = FALSE;
}


void FreeDrawBuffer(void)
{
    if (hSharedBuf != (HANDLE) NULL)
```

```
        LocalFree(hSharedBuf);
}


/*
 *  CalcThreshold --
 *
 *  We need an iteration threshold beyond which we give up. We want it to
 *  increase the farther we zoom in. This code generates a threshold value
 *  based on the precision of drawing.
 *
 *  RETURNS
 *      threshold calculated based on precision
 */

DWORD CalcThreshold(double precision)
{
    DWORD   thres = 25;
    double  multiplier = (double) 100.0;

    /* for every 100, multiply by 2 */
    while ((precision *= multiplier) < (double)1.0)
        thres *= 2;

    return(thres);
}


/*
 *  QueryThreshold --
 *
 *  Callback for finding out what the current drawing's threshold is.
 */

DWORD QueryThreshold(void)
{
    return(dwThreshold);
}
```

SERVER.C (MANDEL RPC Sample)

```
/
*********************************************************************
                  Microsoft RPC Version 2.0
          Copyright Microsoft Corp. 1992, 1993, 1994
                       mandel Example

    FILE:       server.c

    USAGE:      server  -p protocol_sequence
                        -e endpoint
                        -m max calls
                        -n min calls
                        -f flag for RpcServerListen

    PURPOSE:    Server side of RPC distributed application mandel

    FUNCTIONS:  main() - registers interface and listen for clients

*********************************************************************
/

#include <stdlib.h>
#include <stdio.h>
#include <ctype.h>
#include "mdlrpc.h"    // header file generated by MIDL compiler


void Usage(char * pszProgramName)
{
    fprintf(stderr, "Usage:  %s\n", pszProgramName);
    fprintf(stderr, " -p protocol_sequence\n");
    fprintf(stderr, " -e endpoint\n");
    fprintf(stderr, " -m maxcalls\n");
    fprintf(stderr, " -n mincalls\n");
    fprintf(stderr, " -f flag_wait_op\n");
    exit(1);
}

void _CRTAPI1 main(int argc, char * argv[])
{
    RPC_STATUS status;
    unsigned char * pszProtocolSequence = "ncacn_np";
    unsigned char * pszSecurity         = NULL;
    unsigned char * pszEndpoint         = "\\pipe\\mandel";
    unsigned int    cMinCalls           = 1;
    unsigned int    cMaxCalls           = 20;
    unsigned int    fDontWait           = FALSE;
    int i;

    /* allow the user to override settings with command line switches */
    for (i = 1; i < argc; i++) {
        if ((*argv[i] == '-') || (*argv[i] == '/')) {
```

```
            switch (tolower(*(argv[i]+1))) {
            case 'p':  // protocol sequence
                pszProtocolSequence = argv[++i];
                break;
            case 'e':
                pszEndpoint = argv[++i];
                break;
            case 'm':
                cMaxCalls = (unsigned int) atoi(argv[++i]);
                break;
            case 'n':
                cMinCalls = (unsigned int) atoi(argv[++i]);
                break;
            case 'f':
                fDontWait = (unsigned int) atoi(argv[++i]);
                break;
            case 'h':
            case '?':
            default:
                Usage(argv[0]);
            }
        }
        else
            Usage(argv[0]);
    }

    status = RpcServerUseProtseqEp(pszProtocolSequence,
                                   cMaxCalls,
                                   pszEndpoint,
                                   pszSecurity);  // Security descriptor
    printf("RpcServerUseProtseqEp returned 0x%x\n", status);
    if (status) {
        exit(status);
    }

    status = RpcServerRegisterIf(mdlrpc_ServerIfHandle, // interface to
register
                                 NULL,   // MgrTypeUuid
                                 NULL);  // MgrEpv; null means use default
    printf("RpcServerRegisterIf returned 0x%x\n", status);
    if (status) {
        exit(status);
    }

    printf("Calling RpcServerListen\n");
    status = RpcServerListen(cMinCalls,
                             cMaxCalls,
                             fDontWait);
    printf("RpcServerListen returned: 0x%x\n", status);
    if (status) {
        exit(status);
    }

    if (fDontWait) {
        printf("Calling RpcMgmtWaitServerListen\n");
```

```
        status = RpcMgmtWaitServerListen();  //  wait operation
        printf("RpcMgmtWaitServerListen returned: 0x%x\n", status);
        if (status) {
            exit(status);
        }
    }

} // end main()


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

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);
}

/* end server.c */
```

CountClipboardFormats

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	user32.lib
Header File	winuser.h
Unicode	No
Platform Notes	None

CreateAcceleratorTable

Windows NT	Yes
Win95	Yes
Win32s	No
Import Library	user32.lib
Header File	winuser.h
Unicode	WinNT
Platform Notes	None

CreateAntiMoniker

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	ole32.lib
Header File	objbase.h
Unicode	No
Platform Notes	None

CreateBindCtx

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	ole32.lib
Header File	objbase.h
Unicode	No
Platform Notes	None

CreateBitmap

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	gdi32.lib
Header File	wingdi.h
Unicode	No
Platform Notes	Windows 95: int == 16 bits

CreateBitmapIndirect

Windows NT	Yes

Win95	Yes
Win32s	Yes
Import Library	gdi32.lib
Header File	wingdi.h
Unicode	No
Platform Notes	Windows 95: int == 16 bits

CreateBrushIndirect

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	gdi32.lib
Header File	wingdi.h
Unicode	No
Platform Notes	Windows 95: int == 16 bits

CreateCaret

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	user32.lib
Header File	winuser.h
Unicode	No
Platform Notes	None

CreateColorSpace

Windows NT	Stub
Win95	Yes
Win32s	No
Import Library	gdi32.lib
Header File	wingdi.h
Unicode	WinNT
Platform Notes	None

CreateCompatibleBitmap

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	gdi32.lib
Header File	wingdi.h
Unicode	No
Platform Notes	Windows 95: int == 16 bits

CreateCompatibleDC

Windows NT	Yes
Win95	Yes
Win32s	Yes

Import Library	gdi32.lib
Header File	wingdi.h
Unicode	No
Platform Notes	Windows 95: int == 16 bits

CreateConsoleScreenBuffer

Windows NT	Yes
Win95	Yes
Win32s	No
Import Library	kernel32.lib
Header File	wincon.h
Unicode	No
Platform Notes	None

CreateCursor

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	user32.lib
Header File	winuser.h
Unicode	No
Platform Notes	None

CreateDataAdviseHolder

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	ole32.lib
Header File	objbase.h
Unicode	No
Platform Notes	None

CreateDataCache

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	ole32.lib
Header File	objbase.h
Unicode	No
Platform Notes	None

CreateDC

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	gdi32.lib
Header File	wingdi.h

Unicode	WinNT
Platform Notes	Windows 95: int == 16 bits

CreateDesktop

Windows NT	Yes
Win95	No
Win32s	No
Import Library	user32.lib
Header File	winuser.h
Unicode	WinNT
Platform Notes	None

CreateDialog

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	
Header File	winuser.h
Unicode	No
Platform Notes	None

CreateDialogIndirect

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	
Header File	winuser.h
Unicode	No
Platform Notes	None

CreateDialogIndirectParam

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	user32.lib
Header File	winuser.h
Unicode	WinNT
Platform Notes	None

CreateDialogParam

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	user32.lib
Header File	winuser.h
Unicode	WinNT
Platform Notes	None

CreateDIBitmap

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	gdi32.lib
Header File	wingdi.h
Unicode	No
Platform Notes	Windows 95: int == 16 bits

CreateDIBPatternBrush

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	gdi32.lib
Header File	wingdi.h
Unicode	No
Platform Notes	Windows 95: int == 16 bits

CreateDIBPatternBrushPt

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	gdi32.lib
Header File	wingdi.h
Unicode	No
Platform Notes	None

CreateDIBSection

Windows NT	New
Win95	Yes
Win32s	No
Import Library	gdi32.lib
Header File	wingdi.h
Unicode	No
Platform Notes	None

CreateDirectory

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	kernel32.lib
Header File	winbase.h
Unicode	WinNT
Platform Notes	None

CreateDirectoryEx

Windows NT	Yes

Win95	Yes
Win32s	Yes
Import Library	kernel32.lib
Header File	winbase.h
Unicode	WinNT
Platform Notes	None

CreateDiscardableBitmap

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	gdi32.lib
Header File	wingdi.h
Unicode	No
Platform Notes	Windows 95: int == 16 bits

CreateDispTypeInfo

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	oleaut32.lib
Header File	oleauto.h
Unicode	WinNT; Win95; Win32s
Platform Notes	All 32-bit OLE Apis are UNICODE only

CreateEditableStream

Windows NT	Yes
Win95	Yes
Win32s	No
Import Library	vfw32.lib
Header File	vfw.h
Unicode	No
Platform Notes	None

CreateEllipticRgn

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	gdi32.lib
Header File	wingdi.h
Unicode	No
Platform Notes	Windows 95: 16-bit coordinates only

CreateEllipticRgnIndirect

Windows NT	Yes
Win95	Yes

Win32s	Yes
Import Library	gdi32.lib
Header File	wingdi.h
Unicode	No
Platform Notes	Windows 95: 16-bit coordinates only

CreateEnhMetaFile

Windows NT	Yes
Win95	Yes
Win32s	No
Import Library	gdi32.lib
Header File	wingdi.h
Unicode	WinNT
Platform Notes	None

CreateErrorInfo

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	oleaut32.lib
Header File	oleauto.h
Unicode	No
Platform Notes	None

CreateEvent

Windows NT	Yes
Win95	Yes
Win32s	No
Import Library	kernel32.lib
Header File	winbase.h
Unicode	WinNT
Platform Notes	None

CreateFile

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	kernel32.lib
Header File	winbase.h
Unicode	WinNT
Platform Notes	None

CreateFileMapping

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	kernel32.lib

Header File	winbase.h
Unicode	WinNT
Platform Notes	None

CreateFileMoniker

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	ole32.lib
Header File	objbase.h
Unicode	WinNT; Win95; Win32s
Platform Notes	All 32-bit OLE Apis are UNICODE only

CreateFont

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	gdi32.lib
Header File	wingdi.h
Unicode	WinNT
Platform Notes	Windows 95: int == 16 bits

CreateFontIndirect

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	gdi32.lib
Header File	wingdi.h
Unicode	WinNT
Platform Notes	Windows 95: int == 16 bits

CreateGenericComposite

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	ole32.lib
Header File	objbase.h
Unicode	No
Platform Notes	None

CreateHalftonePalette

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	gdi32.lib
Header File	wingdi.h

Unicode	No
Platform Notes	None

CreateHatchBrush

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	gdi32.lib
Header File	wingdi.h
Unicode	No
Platform Notes	Windows 95: int == 16 bits

CreateIC

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	gdi32.lib
Header File	wingdi.h
Unicode	WinNT
Platform Notes	Windows 95: int == 16 bits

CreateIcon

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	user32.lib
Header File	winuser.h
Unicode	No
Platform Notes	None

CreateIconFromResource

Windows NT	Yes
Win95	Yes
Win32s	No
Import Library	user32.lib
Header File	winuser.h
Unicode	No
Platform Notes	None

CreateIconFromResourceEx

Windows NT	Stub
Win95	Yes
Win32s	No
Import Library	user32.lib
Header File	winuser.h
Unicode	No

Platform Notes	None

CreateIconIndirect

Windows NT	Yes
Win95	Yes
Win32s	No
Import Library	user32.lib
Header File	winuser.h
Unicode	No
Platform Notes	None

CreateILockBytesOnHGlobal

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	ole32.lib
Header File	ole2.h
Unicode	No
Platform Notes	None

CreateIoCompletionPort

Windows NT	New
Win95	No
Win32s	No
Import Library	kernel32.lib
Header File	winbase.h
Unicode	No
Platform Notes	None

CreateItemMoniker

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	ole32.lib
Header File	objbase.h
Unicode	WinNT; Win95; Win32s
Platform Notes	All 32-bit OLE Apis are UNICODE only

CreateMailslot

Windows NT	Yes
Win95	Yes
Win32s	No
Import Library	kernel32.lib
Header File	winbase.h
Unicode	WinNT
Platform Notes	None

CreateMappedBitmap

Windows NT	Yes
Win95	Yes
Win32s	
Import Library	
Header File	commctrl.h
Unicode	No
Platform Notes	None

CreateMDIWindow

Windows NT	Yes
Win95	Yes
Win32s	No
Import Library	user32.lib
Header File	winuser.h
Unicode	WinNT
Platform Notes	None

CreateMenu

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	user32.lib
Header File	winuser.h
Unicode	No
Platform Notes	None

CreateMetaFile

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	gdi32.lib
Header File	wingdi.h
Unicode	WinNT
Platform Notes	Windows 95: int == 16 bits

CreateMutex

Windows NT	Yes
Win95	Yes
Win32s	No
Import Library	kernel32.lib
Header File	winbase.h
Unicode	WinNT
Platform Notes	None

CreateNamedPipe

Windows NT	Yes

Win95	No
Win32s	No
Import Library	kernel32.lib
Header File	winbase.h
Unicode	WinNT
Platform Notes	None

CreateOleAdviseHolder

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	ole32.lib
Header File	ole2.h
Unicode	No
Platform Notes	None

CreatePalette

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	gdi32.lib
Header File	wingdi.h
Unicode	No
Platform Notes	Windows 95: int == 16 bits

CreatePatternBrush

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	gdi32.lib
Header File	wingdi.h
Unicode	No
Platform Notes	Windows 95: int == 16 bits

CreatePen

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	gdi32.lib
Header File	wingdi.h
Unicode	No
Platform Notes	Windows 95: int == 16 bits

CreatePenIndirect

Windows NT	Yes
Win95	Yes
Win32s	Yes

Import Library	gdi32.lib
Header File	wingdi.h
Unicode	No
Platform Notes	Windows 95: int == 16 bits

CreatePipe

Windows NT	Yes
Win95	Yes
Win32s	No
Import Library	kernel32.lib
Header File	winbase.h
Unicode	No
Platform Notes	None

CreatePointerMoniker

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	ole32.lib
Header File	objbase.h
Unicode	No
Platform Notes	None

CreatePolygonRgn

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	gdi32.lib
Header File	wingdi.h
Unicode	No
Platform Notes	Windows 95: 16-bit coordinates only

CreatePolyPolygonRgn

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	gdi32.lib
Header File	wingdi.h
Unicode	No
Platform Notes	Windows 95: 16-bit coordinates only

CreatePopupMenu

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	user32.lib

Header File	winuser.h
Unicode	No
Platform Notes	None

CreatePrivateObjectSecurity

Windows NT	Yes
Win95	No
Win32s	No
Import Library	advapi32.lib
Header File	winbase.h
Unicode	No
Platform Notes	None

CreateProcess

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	kernel32.lib
Header File	winbase.h
Unicode	WinNT
Platform Notes	None

CreateProcessAsUser

Windows NT	New
Win95	No
Win32s	No
Import Library	
Header File	winbase.h
Unicode	WinNT
Platform Notes	None

CreatePropertySheetPage

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	
Header File	prsht.h
Unicode	WinNT
Platform Notes	None

CreateRectRgn

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	gdi32.lib
Header File	wingdi.h
Unicode	No

Platform Notes	Windows 95: 16-bit coordinates only

CreateRectRgnIndirect

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	gdi32.lib
Header File	wingdi.h
Unicode	No
Platform Notes	Windows 95: 16-bit coordinates only

CreateRemoteThread

Windows NT	Yes
Win95	No
Win32s	No
Import Library	kernel32.lib
Header File	winbase.h
Unicode	No
Platform Notes	None

CreateRoundRectRgn

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	gdi32.lib
Header File	wingdi.h
Unicode	No
Platform Notes	Windows 95: 16-bit coordinates only

CreateScalableFontResource

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	gdi32.lib
Header File	wingdi.h
Unicode	WinNT
Platform Notes	Windows 95: int == 16 bits

CreateSemaphore

Windows NT	Yes
Win95	Yes
Win32s	No
Import Library	kernel32.lib
Header File	winbase.h
Unicode	WinNT

Platform Notes	None

CreateService

Windows NT	Yes
Win95	No
Win32s	No
Import Library	advapi32.lib
Header File	winsvc.h
Unicode	WinNT
Platform Notes	None

CreateSolidBrush

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	gdi32.lib
Header File	wingdi.h
Unicode	No
Platform Notes	Windows 95: int == 16 bits

CreateStatusWindow

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	
Header File	commctrl.h
Unicode	WinNT
Platform Notes	None

CreateStdDispatch

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	oleaut32.lib
Header File	oleauto.h
Unicode	No
Platform Notes	None

CreateStreamOnHGlobal

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	ole32.lib
Header File	ole2.h
Unicode	No
Platform Notes	None

CreateTapePartition

Windows NT	Yes
Win95	No
Win32s	No
Import Library	kernel32.lib
Header File	winbase.h
Unicode	No
Platform Notes	None

CreateThread

Windows NT	Yes
Win95	Yes
Win32s	No
Import Library	kernel32.lib
Header File	winbase.h
Unicode	No
Platform Notes	None

CreateToolbarEx

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	
Header File	commctrl.h
Unicode	No
Platform Notes	None

CreateTypeLib

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	oleaut32.lib
Header File	oleauto.h
Unicode	WinNT; Win95; Win32s
Platform Notes	All 32-bit OLE Apis are UNICODE only

CreateUpDownControl

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	
Header File	commctrl.h
Unicode	No
Platform Notes	None

CreateWindow

Windows NT	Yes
Win95	Yes

Win32s	Yes
Import Library	
Header File	winuser.h
Unicode	No
Platform Notes	None

CreateWindowEx

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	user32.lib
Header File	winuser.h
Unicode	WinNT
Platform Notes	hMenu limited to WORD on Win32s

CreateWindowStation

Windows NT	Yes
Win95	No
Win32s	No
Import Library	user32.lib
Header File	winuser.h
Unicode	WinNT
Platform Notes	None

data_from_ndr

Windows NT	Yes
Win95	No
Win32s	No
Import Library	rpcrt4.lib
Header File	rpcndr.h
Unicode	
Platform Notes	None

data_into_ndr

Windows NT	Yes
Win95	No
Win32s	No
Import Library	rpcrt4.lib
Header File	rpcndr.h
Unicode	
Platform Notes	None

data_size_ndr

Windows NT	Yes
Win95	No
Win32s	No

Import Library	rpcrt4.lib
Header File	rpcndr.h
Unicode	
Platform Notes	None

DceErrorInqText

Windows NT	New
Win95	No
Win32s	No
Import Library	rpcrt4.lib
Header File	rpcdce.h
Unicode	WinNT
Platform Notes	None

DdeAbandonTransaction

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	user32.lib
Header File	ddeml.h
Unicode	No
Platform Notes	None

DdeAccessData

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	user32.lib
Header File	ddeml.h
Unicode	No
Platform Notes	None

DdeAddData

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	user32.lib
Header File	ddeml.h
Unicode	No
Platform Notes	None

DdeClientTransaction

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	user32.lib

Header File	ddeml.h
Unicode	No
Platform Notes	None

DdeCmpStringHandles

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	user32.lib
Header File	ddeml.h
Unicode	No
Platform Notes	None

DdeConnect

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	user32.lib
Header File	ddeml.h
Unicode	No
Platform Notes	None

DdeConnectList

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	user32.lib
Header File	ddeml.h
Unicode	No
Platform Notes	None

DdeCreateDataHandle

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	user32.lib
Header File	ddeml.h
Unicode	No
Platform Notes	None

DdeCreateStringHandle

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	user32.lib
Header File	ddeml.h
Unicode	WinNT

Platform Notes	None

DdeDisconnect

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	user32.lib
Header File	ddeml.h
Unicode	No
Platform Notes	None

DdeDisconnectList

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	user32.lib
Header File	ddeml.h
Unicode	No
Platform Notes	None

DdeEnableCallback

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	user32.lib
Header File	ddeml.h
Unicode	No
Platform Notes	None

DdeFreeDataHandle

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	user32.lib
Header File	ddeml.h
Unicode	No
Platform Notes	None

DdeFreeStringHandle

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	user32.lib
Header File	ddeml.h
Unicode	No
Platform Notes	None

DdeGetData

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	user32.lib
Header File	ddeml.h
Unicode	No
Platform Notes	None

DdeGetLastError

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	user32.lib
Header File	ddeml.h
Unicode	No
Platform Notes	None

DdeImpersonateClient

Windows NT	Yes
Win95	No
Win32s	No
Import Library	user32.lib
Header File	ddeml.h
Unicode	No
Platform Notes	None

DdeInitialize

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	user32.lib
Header File	ddeml.h
Unicode	WinNT
Platform Notes	None

DdeKeepStringHandle

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	user32.lib
Header File	ddeml.h
Unicode	No
Platform Notes	None

DdeNameService

Windows NT	Yes
Win95	Yes

Win32s	Yes
Import Library	user32.lib
Header File	ddeml.h
Unicode	No
Platform Notes	None

DdePostAdvise

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	user32.lib
Header File	ddeml.h
Unicode	No
Platform Notes	None

DdeQueryConvInfo

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	user32.lib
Header File	ddeml.h
Unicode	No
Platform Notes	None

DdeQueryNextServer

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	user32.lib
Header File	ddeml.h
Unicode	No
Platform Notes	None

DdeQueryString

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	user32.lib
Header File	ddeml.h
Unicode	WinNT
Platform Notes	None

DdeReconnect

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	user32.lib

Header File	ddeml.h
Unicode	No
Platform Notes	None

DdeSetQualityOfService

Windows NT	Yes
Win95	No
Win32s	No
Import Library	user32.lib
Header File	dde.h
Unicode	No
Platform Notes	None

RegEnumValue

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	advapi32.lib
Header File	winreg.h
Unicode	WinNT
Platform Notes	None

RegFlushKey

Windows NT	Yes
Win95	Yes
Win32s	No
Import Library	advapi32.lib
Header File	winreg.h
Unicode	No
Platform Notes	None

RegGetKeySecurity

Windows NT	Yes
Win95	No
Win32s	No
Import Library	advapi32.lib
Header File	winreg.h
Unicode	No
Platform Notes	None

RegisterActiveObject

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	oleaut32.lib
Header File	oleauto.h
Unicode	No
Platform Notes	None

RegisterClass

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	user32.lib
Header File	winuser.h
Unicode	WinNT
Platform Notes	None

RegisterClassEx

Windows NT	Stub

Win95	Yes
Win32s	Yes
Import Library	user32.lib
Header File	winuser.h
Unicode	WinNT
Platform Notes	None

RegisterClipboardFormat

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	user32.lib
Header File	winuser.h
Unicode	WinNT
Platform Notes	None

RegisterDialogClasses

Windows NT	Yes
Win95	Yes
Win32s	No
Import Library	scrnsave.lib
Header File	scrnsave.h
Unicode	No
Platform Notes	None

RegisterDragDrop

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	ole32.lib
Header File	ole2.h
Unicode	No
Platform Notes	None

RegisterEventSource

Windows NT	Yes
Win95	No
Win32s	No
Import Library	advapi32.lib
Header File	winbase.h
Unicode	WinNT
Platform Notes	None

RegisterHotKey

Windows NT	Yes
Win95	Yes
Win32s	No

Import Library	user32.lib
Header File	winuser.h
Unicode	No
Platform Notes	None

RegisterServiceCtrlHandler

Windows NT	Yes
Win95	No
Win32s	No
Import Library	advapi32.lib
Header File	winsvc.h
Unicode	WinNT
Platform Notes	None

RegisterTypeLib

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	oleaut32.lib
Header File	oleauto.h
Unicode	WinNT; Win95; Win32s
Platform Notes	All 32-bit OLE Apis are UNICODE only

RegisterWindowMessage

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	user32.lib
Header File	winuser.h
Unicode	WinNT
Platform Notes	None

RegLoadKey

Windows NT	Yes
Win95	Yes
Win32s	No
Import Library	advapi32.lib
Header File	winreg.h
Unicode	WinNT
Platform Notes	None

RegNotifyChangeKeyValue

Windows NT	Yes
Win95	No
Win32s	No
Import Library	advapi32.lib
Header File	winreg.h

Unicode	No
Platform Notes	None

RegOpenKey

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	advapi32.lib
Header File	winreg.h
Unicode	WinNT
Platform Notes	None

RegOpenKeyEx

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	advapi32.lib
Header File	winreg.h
Unicode	WinNT
Platform Notes	None

RegQueryInfoKey

Windows NT	Yes
Win95	Yes
Win32s	No
Import Library	advapi32.lib
Header File	winreg.h
Unicode	WinNT
Platform Notes	None

RegQueryMultipleValues

Windows NT	No
Win95	Yes
Win32s	No
Import Library	kernel32.lib
Header File	winbase.h
Unicode	
Platform Notes	None

RegQueryValue

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	advapi32.lib
Header File	winreg.h
Unicode	WinNT

Platform Notes	None

RegQueryValueEx

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	advapi32.lib
Header File	winreg.h
Unicode	WinNT
Platform Notes	None

RegReplaceKey

Windows NT	Yes
Win95	Yes
Win32s	No
Import Library	advapi32.lib
Header File	winreg.h
Unicode	WinNT
Platform Notes	None

RegRestoreKey

Windows NT	Yes
Win95	No
Win32s	No
Import Library	advapi32.lib
Header File	winreg.h
Unicode	WinNT
Platform Notes	None

RegSaveKey

Windows NT	Yes
Win95	Yes
Win32s	No
Import Library	advapi32.lib
Header File	winreg.h
Unicode	WinNT
Platform Notes	None

RegSetKeySecurity

Windows NT	Yes
Win95	No
Win32s	No
Import Library	advapi32.lib
Header File	winreg.h
Unicode	No
Platform Notes	None

RegSetValue

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	advapi32.lib
Header File	winreg.h
Unicode	WinNT
Platform Notes	None

RegSetValueEx

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	advapi32.lib
Header File	winreg.h
Unicode	WinNT
Platform Notes	None

RegUnLoadKey

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	advapi32.lib
Header File	winreg.h
Unicode	WinNT
Platform Notes	None

ReleaseCapture

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	user32.lib
Header File	winuser.h
Unicode	No
Platform Notes	None

ReleaseDC

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	user32.lib
Header File	winuser.h
Unicode	No
Platform Notes	None

ReleaseMutex

Windows NT	Yes
Win95	Yes

Win32s	No
Import Library	kernel32.lib
Header File	winbase.h
Unicode	No
Platform Notes	None

ReleaseSemaphore

Windows NT	Yes
Win95	Yes
Win32s	No
Import Library	kernel32.lib
Header File	winbase.h
Unicode	No
Platform Notes	None

ReleaseStgMedium

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	ole32.lib
Header File	ole2.h
Unicode	No
Platform Notes	None

RemoveDirectory

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	kernel32.lib
Header File	winbase.h
Unicode	WinNT
Platform Notes	None

RemoveFontResource

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	gdi32.lib
Header File	wingdi.h
Unicode	WinNT
Platform Notes	Windows 95: int == 16 bits

RemoveMenu

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	user32.lib

Header File	winuser.h
Unicode	No
Platform Notes	None

RemoveProp

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	user32.lib
Header File	winuser.h
Unicode	WinNT
Platform Notes	None

ReplaceText

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	comdlg32.lib
Header File	commdlg.h
Unicode	WinNT
Platform Notes	None

ReplyMessage

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	user32.lib
Header File	winuser.h
Unicode	No
Platform Notes	None

ReportEvent

Windows NT	Yes
Win95	No
Win32s	No
Import Library	advapi32.lib
Header File	winbase.h
Unicode	WinNT
Platform Notes	None

ResetDC

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	gdi32.lib
Header File	wingdi.h
Unicode	WinNT

Platform Notes	Windows 95: int == 16 bits

ResetDisplay

Windows NT	Stub
Win95	Yes
Win32s	No
Import Library	user32.lib
Header File	winuser.h
Unicode	No
Platform Notes	None

ResetEvent

Windows NT	Yes
Win95	Yes
Win32s	No
Import Library	kernel32.lib
Header File	winbase.h
Unicode	No
Platform Notes	None

ResetPrinter

Windows NT	Yes
Win95	No
Win32s	No
Import Library	winspool.lib
Header File	winspool.h
Unicode	WinNT
Platform Notes	None

ResizePalette

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	gdi32.lib
Header File	wingdi.h
Unicode	No
Platform Notes	Windows 95: int == 16 bits

RestoreDC

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	gdi32.lib
Header File	wingdi.h
Unicode	No
Platform Notes	Windows 95: int == 16 bits

ResumeThread

Windows NT	Yes
Win95	Yes
Win32s	No
Import Library	kernel32.lib
Header File	winbase.h
Unicode	No
Platform Notes	None

ReuseDDElParam

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	user32.lib
Header File	dde.h
Unicode	No
Platform Notes	None

RevertToSelf

Windows NT	Yes
Win95	No
Win32s	No
Import Library	advapi32.lib
Header File	winbase.h
Unicode	No
Platform Notes	None

RevokeActiveObject

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	oleaut32.lib
Header File	oleauto.h
Unicode	No
Platform Notes	None

RevokeDragDrop

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	ole32.lib
Header File	ole2.h
Unicode	No
Platform Notes	None

RoundRect

Windows NT	Yes
Win95	Yes

Win32s	Yes
Import Library	gdi32.lib
Header File	wingdi.h
Unicode	No
Platform Notes	Windows 95: 16-bit coordinates only

RpcAbnormalTermination

Windows NT	New
Win95	Yes
Win32s	No
Import Library	
Header File	rpc.h
Unicode	No
Platform Notes	None

RpcBindingCopy

Windows NT	Yes
Win95	Yes
Win32s	No
Import Library	rpcrt4.lib
Header File	rpcdce.h
Unicode	No
Platform Notes	None

RpcBindingFree

Windows NT	Yes
Win95	Yes
Win32s	No
Import Library	rpcrt4.lib
Header File	rpcdce.h
Unicode	No
Platform Notes	None

RpcBindingFromStringBinding

Windows NT	Yes
Win95	Yes
Win32s	No
Import Library	rpcrt4.lib
Header File	rpcdce.h
Unicode	WinNT
Platform Notes	None

RpcBindingInqAuthClient

Windows NT	Yes
Win95	Yes
Win32s	No
Import Library	rpcrt4.lib

Header File	rpcdce.h
Unicode	
Platform Notes	None

RpcBindingInqAuthInfo

Windows NT	Yes
Win95	Yes
Win32s	No
Import Library	rpcrt4.lib
Header File	rpcdce.h
Unicode	WinNT
Platform Notes	None

RpcBindingInqObject

Windows NT	Yes
Win95	Yes
Win32s	No
Import Library	rpcrt4.lib
Header File	rpcdce.h
Unicode	No
Platform Notes	None

RpcBindingReset

Windows NT	Yes
Win95	Yes
Win32s	No
Import Library	rpcrt4.lib
Header File	rpcdce.h
Unicode	No
Platform Notes	None

RpcBindingServerFromClient

Windows NT	Yes
Win95	Yes
Win32s	
Import Library	rpcrt4.lib
Header File	rpcdce.h
Unicode	No
Platform Notes	None

RpcBindingSetAuthInfo

Windows NT	Yes
Win95	Yes
Win32s	No
Import Library	rpcrt4.lib
Header File	rpcdce.h

Unicode	
Platform Notes	None

RpcBindingSetObject

Windows NT	Yes
Win95	Yes
Win32s	No
Import Library	rpcrt4.lib
Header File	rpcdce.h
Unicode	No
Platform Notes	None

RpcBindingToStringBinding

Windows NT	Yes
Win95	Yes
Win32s	No
Import Library	rpcrt4.lib
Header File	rpcdce.h
Unicode	WinNT
Platform Notes	None

RpcBindingVectorFree

Windows NT	Yes
Win95	Yes
Win32s	No
Import Library	rpcrt4.lib
Header File	rpcdce.h
Unicode	No
Platform Notes	None

RpcCancelThread

Windows NT	New
Win95	No
Win32s	No
Import Library	rpcrt4.lib
Header File	rpcdce.h
Unicode	No
Platform Notes	None

RpcEndExcept

Windows NT	New
Win95	Yes
Win32s	No
Import Library	
Header File	rpc.h
Unicode	No
Platform Notes	None

RpcEndFinally

Windows NT	New
Win95	Yes
Win32s	No
Import Library	
Header File	rpc.h
Unicode	No
Platform Notes	None

RpcEpRegister

Windows NT	Yes
Win95	Yes
Win32s	No
Import Library	rpcrt4.lib
Header File	rpcdce.h
Unicode	WinNT
Platform Notes	None

RpcEpRegisterNoReplace

Windows NT	Yes
Win95	Yes
Win32s	No
Import Library	rpcrt4.lib
Header File	rpcdce.h
Unicode	WinNT
Platform Notes	None

RpcEpResolveBinding

Windows NT	Yes
Win95	Yes
Win32s	No
Import Library	rpcrt4.lib
Header File	rpcdce.h
Unicode	No
Platform Notes	None

RpcEpUnregister

Windows NT	Yes
Win95	Yes
Win32s	No
Import Library	rpcrt4.lib
Header File	rpcdce.h
Unicode	No
Platform Notes	None

RpcExcept

Windows NT	New

Win95	Yes
Win32s	No
Import Library	
Header File	rpc.h
Unicode	No
Platform Notes	None

RpcExceptionCode

Windows NT	New
Win95	Yes
Win32s	No
Import Library	
Header File	rpc.h
Unicode	No
Platform Notes	None

RpcFinally

Windows NT	New
Win95	Yes
Win32s	No
Import Library	
Header File	rpc.h
Unicode	No
Platform Notes	None

RpcIfIdVectorFree

Windows NT	Yes
Win95	Yes
Win32s	No
Import Library	rpcrt4.lib
Header File	rpcdce.h
Unicode	No
Platform Notes	None

RpcIfInqId

Windows NT	Yes
Win95	Yes
Win32s	No
Import Library	rpcrt4.lib
Header File	rpcdce.h
Unicode	No
Platform Notes	None

RpcImpersonateClient

Windows NT	Yes
Win95	Yes
Win32s	No

Import Library	rpcrt4.lib
Header File	rpc.h
Unicode	
Platform Notes	None

RpcMgmtEnableIdleCleanup

Windows NT	Yes
Win95	Yes
Win32s	No
Import Library	rpcrt4.lib
Header File	rpcdce.h
Unicode	No
Platform Notes	None

RpcMgmtEpEltInqBegin

Windows NT	New
Win95	Yes
Win32s	No
Import Library	rpcrt4.lib
Header File	rpcdce.h
Unicode	No
Platform Notes	None

RpcMgmtEpEltInqDone

Windows NT	New
Win95	Yes
Win32s	No
Import Library	rpcrt4.lib
Header File	rpcdce.h
Unicode	No
Platform Notes	None

RpcMgmtEpEltInqNext

Windows NT	New
Win95	Yes
Win32s	No
Import Library	rpcrt4.lib
Header File	rpcdce.h
Unicode	WinNT
Platform Notes	None

RpcMgmtEpUnregister

Windows NT	New
Win95	Yes
Win32s	No
Import Library	rpcrt4.lib
Header File	rpcdce.h

Unicode	No
Platform Notes	None

RpcMgmtInqComTimeout

Windows NT	Yes
Win95	Yes
Win32s	No
Import Library	rpcrt4.lib
Header File	rpcdce.h
Unicode	No
Platform Notes	None

RpcMgmtInqDefaultProtectLevel

Windows NT	Yes
Win95	Yes
Win32s	
Import Library	rpcrt4.lib
Header File	rpcdce.h
Unicode	No
Platform Notes	None

RpcMgmtInqIfIds

Windows NT	Yes
Win95	Yes
Win32s	
Import Library	rpcrt4.lib
Header File	rpcdce.h
Unicode	No
Platform Notes	None

RpcMgmtInqServerPrincName

Windows NT	New
Win95	
Win32s	No
Import Library	rpcrt4.lib
Header File	rpcdce.h
Unicode	WinNT
Platform Notes	None

RpcMgmtInqStats

Windows NT	Yes
Win95	Yes
Win32s	No
Import Library	rpcrt4.lib
Header File	rpcdce.h
Unicode	No
Platform Notes	None

RpcMgmtIsServerListening

Windows NT	Yes
Win95	Yes
Win32s	No
Import Library	rpcrt4.lib
Header File	rpcdce.h
Unicode	No
Platform Notes	None

RpcMgmtSetAuthorizationFn

Windows NT	New
Win95	Yes
Win32s	No
Import Library	rpcrt4.lib
Header File	rpcdce.h
Unicode	No
Platform Notes	None

RpcMgmtSetCancelTimeout

Windows NT	New
Win95	No
Win32s	No
Import Library	rpcrt4.lib
Header File	rpcdce.h
Unicode	No
Platform Notes	None

RpcMgmtSetComTimeout

Windows NT	Yes
Win95	Yes
Win32s	No
Import Library	rpcrt4.lib
Header File	rpcdce.h
Unicode	No
Platform Notes	None

RpcMgmtSetServerStackSize

Windows NT	Yes
Win95	Yes
Win32s	No
Import Library	rpcrt4.lib
Header File	rpcdce.h
Unicode	No
Platform Notes	None

RpcMgmtStatsVectorFree

Windows NT	Yes

Win95	Yes
Win32s	No
Import Library	rpcrt4.lib
Header File	rpcdce.h
Unicode	No
Platform Notes	None

RpcMgmtStopServerListening

Windows NT	Yes
Win95	Yes
Win32s	No
Import Library	rpcrt4.lib
Header File	rpcdce.h
Unicode	No
Platform Notes	None

RpcMgmtWaitServerListen

Windows NT	Yes
Win95	Yes
Win32s	No
Import Library	rpcrt4.lib
Header File	rpcdce.h
Unicode	No
Platform Notes	None

RpcNetworkInqProtseqs

Windows NT	Yes
Win95	Yes
Win32s	No
Import Library	rpcrt4.lib
Header File	rpcdce.h
Unicode	WinNT
Platform Notes	None

RpcNetworkIsProtseqValid

Windows NT	Yes
Win95	Yes
Win32s	No
Import Library	rpcrt4.lib
Header File	rpcdce.h
Unicode	WinNT
Platform Notes	None

RpcNsBindingExport

Windows NT	Yes
Win95	Yes
Win32s	No

Import Library	rpcns4.lib
Header File	rpcnsi.h
Unicode	WinNT
Platform Notes	None

RpcNsBindingImportBegin

Windows NT	Yes
Win95	Yes
Win32s	No
Import Library	rpcns4.lib
Header File	rpcnsi.h
Unicode	WinNT
Platform Notes	None

RpcNsBindingImportDone

Windows NT	Yes
Win95	Yes
Win32s	No
Import Library	rpcns4.lib
Header File	rpcnsi.h
Unicode	No
Platform Notes	None

RpcNsBindingImportNext

Windows NT	Yes
Win95	Yes
Win32s	No
Import Library	rpcns4.lib
Header File	rpcnsi.h
Unicode	No
Platform Notes	None

RpcNsBindingInqEntryName

Windows NT	Yes
Win95	Yes
Win32s	No
Import Library	rpcrt4.lib
Header File	rpcdce.h
Unicode	WinNT
Platform Notes	None

RpcNsBindingLookupBegin

Windows NT	Yes
Win95	Yes
Win32s	No
Import Library	rpcns4.lib
Header File	rpcnsi.h

Unicode	WinNT
Platform Notes	None

RpcNsBindingLookupDone

Windows NT	Yes
Win95	Yes
Win32s	No
Import Library	rpcns4.lib
Header File	rpcnsi.h
Unicode	No
Platform Notes	None

RpcNsBindingLookupNext

Windows NT	Yes
Win95	Yes
Win32s	No
Import Library	rpcns4.lib
Header File	rpcnsi.h
Unicode	No
Platform Notes	None

RpcNsBindingSelect

Windows NT	Yes
Win95	Yes
Win32s	No
Import Library	rpcns4.lib
Header File	rpcnsi.h
Unicode	No
Platform Notes	None

RpcNsBindingUnexport

Windows NT	Yes
Win95	Yes
Win32s	No
Import Library	rpcns4.lib
Header File	rpcnsi.h
Unicode	WinNT
Platform Notes	None

RpcNsEntryExpandName

Windows NT	Yes
Win95	Yes
Win32s	No
Import Library	rpcns4.lib
Header File	rpcnsi.h
Unicode	WinNT
Platform Notes	None

RpcNsEntryObjectInqBegin

Windows NT	Yes
Win95	Yes
Win32s	No
Import Library	rpcns4.lib
Header File	rpcnsi.h
Unicode	WinNT
Platform Notes	None

RpcNsEntryObjectInqDone

Windows NT	Yes
Win95	Yes
Win32s	No
Import Library	rpcns4.lib
Header File	rpcnsi.h
Unicode	No
Platform Notes	None

RpcNsEntryObjectInqNext

Windows NT	Yes
Win95	Yes
Win32s	No
Import Library	rpcns4.lib
Header File	rpcnsi.h
Unicode	No
Platform Notes	None

RpcNsGroupDelete

Windows NT	Yes
Win95	Yes
Win32s	No
Import Library	rpcns4.lib
Header File	rpcnsi.h
Unicode	WinNT
Platform Notes	None

RpcNsGroupMbrAdd

Windows NT	Yes
Win95	Yes
Win32s	No
Import Library	rpcns4.lib
Header File	rpcnsi.h
Unicode	WinNT
Platform Notes	None

RpcNsGroupMbrInqBegin

Windows NT	Yes

Win95	Yes
Win32s	No
Import Library	rpcns4.lib
Header File	rpcnsi.h
Unicode	WinNT
Platform Notes	None

RpcNsGroupMbrInqDone

Windows NT	Yes
Win95	Yes
Win32s	No
Import Library	rpcns4.lib
Header File	rpcnsi.h
Unicode	No
Platform Notes	None

RpcNsGroupMbrInqNext

Windows NT	Yes
Win95	Yes
Win32s	No
Import Library	rpcns4.lib
Header File	rpcnsi.h
Unicode	WinNT
Platform Notes	None

RpcNsGroupMbrRemove

Windows NT	Yes
Win95	Yes
Win32s	No
Import Library	rpcns4.lib
Header File	rpcnsi.h
Unicode	WinNT
Platform Notes	None

RpcNsMgmtBindingUnexport

Windows NT	Yes
Win95	Yes
Win32s	No
Import Library	rpcns4.lib
Header File	rpcnsi.h
Unicode	WinNT
Platform Notes	None

RpcNsMgmtEntryCreate

Windows NT	Yes
Win95	Yes
Win32s	No

Import Library	rpcns4.lib
Header File	rpcnsi.h
Unicode	WinNT
Platform Notes	None

RpcNsMgmtEntryDelete

Windows NT	Yes
Win95	Yes
Win32s	No
Import Library	rpcns4.lib
Header File	rpcnsi.h
Unicode	WinNT
Platform Notes	None

RpcNsMgmtEntryInqIfIds

Windows NT	Yes
Win95	Yes
Win32s	No
Import Library	rpcns4.lib
Header File	rpcnsi.h
Unicode	WinNT
Platform Notes	None

RpcNsMgmtHandleSetExpAge

Windows NT	Yes
Win95	Yes
Win32s	No
Import Library	rpcns4.lib
Header File	rpcnsi.h
Unicode	No
Platform Notes	None

RpcNsMgmtInqExpAge

Windows NT	Yes
Win95	Yes
Win32s	No
Import Library	rpcns4.lib
Header File	rpcnsi.h
Unicode	No
Platform Notes	None

RpcNsMgmtSetExpAge

Windows NT	Yes
Win95	Yes
Win32s	No
Import Library	rpcns4.lib
Header File	rpcnsi.h
Unicode	No
Platform Notes	None

RpcNsProfileDelete

Windows NT	Yes

Win95	Yes
Win32s	No
Import Library	rpcns4.lib
Header File	rpcnsi.h
Unicode	WinNT
Platform Notes	None

RpcNsProfileEltAdd

Windows NT	Yes
Win95	Yes
Win32s	No
Import Library	rpcns4.lib
Header File	rpcnsi.h
Unicode	WinNT
Platform Notes	None

RpcNsProfileEltInqBegin

Windows NT	Yes
Win95	Yes
Win32s	No
Import Library	rpcns4.lib
Header File	rpcnsi.h
Unicode	WinNT
Platform Notes	None

RpcNsProfileEltInqDone

Windows NT	Yes
Win95	Yes
Win32s	No
Import Library	rpcns4.lib
Header File	rpcnsi.h
Unicode	No
Platform Notes	None

RpcNsProfileEltInqNext

Windows NT	Yes
Win95	Yes
Win32s	No
Import Library	rpcns4.lib
Header File	rpcnsi.h
Unicode	WinNT
Platform Notes	None

RpcNsProfileEltRemove

Windows NT	Yes
Win95	Yes
Win32s	No

Import Library	rpcns4.lib
Header File	rpcnsi.h
Unicode	WinNT
Platform Notes	None

RpcObjectInqType

Windows NT	Yes
Win95	Yes
Win32s	No
Import Library	rpcrt4.lib
Header File	rpcdce.h
Unicode	No
Platform Notes	None

RpcObjectSetInqFn

Windows NT	Yes
Win95	Yes
Win32s	No
Import Library	rpcrt4.lib
Header File	rpcdce.h
Unicode	No
Platform Notes	None

RpcObjectSetType

Windows NT	Yes
Win95	Yes
Win32s	No
Import Library	rpcrt4.lib
Header File	rpcdce.h
Unicode	No
Platform Notes	None

RpcProtseqVectorFree

Windows NT	Yes
Win95	Yes
Win32s	No
Import Library	rpcrt4.lib
Header File	rpcdce.h
Unicode	
Platform Notes	None

RpcRaiseException

Windows NT	Yes
Win95	Yes
Win32s	No
Import Library	rpcrt4.lib
Header File	rpcdce.h

Unicode	
Platform Notes	None

RpcRevertToSelf

Windows NT	Yes
Win95	Yes
Win32s	No
Import Library	rpcrt4.lib
Header File	rpc.h
Unicode	
Platform Notes	None

RpcServerInqBindings

Windows NT	Yes
Win95	Yes
Win32s	No
Import Library	rpcrt4.lib
Header File	rpcdce.h
Unicode	
Platform Notes	None

RpcServerInqIf

Windows NT	Yes
Win95	Yes
Win32s	No
Import Library	rpcrt4.lib
Header File	rpcdce.h
Unicode	
Platform Notes	None

RpcServerListen

Windows NT	Yes
Win95	Yes
Win32s	No
Import Library	rpcrt4.lib
Header File	rpcdce.h
Unicode	
Platform Notes	None

RpcServerRegisterAuthInfo

Windows NT	Yes
Win95	Yes
Win32s	No
Import Library	rpcrt4.lib
Header File	rpcdce.h
Unicode	
Platform Notes	None

RpcServerRegisterIf

Windows NT	Yes
Win95	Yes
Win32s	No
Import Library	rpcrt4.lib
Header File	rpcdce.h
Unicode	
Platform Notes	None

RpcServerUnregisterIf

Windows NT	Yes
Win95	Yes
Win32s	No
Import Library	rpcrt4.lib
Header File	rpcdce.h
Unicode	
Platform Notes	None

RpcServerUseAllProtseqs

Windows NT	Yes
Win95	Yes
Win32s	No
Import Library	rpcrt4.lib
Header File	rpcdce.h
Unicode	
Platform Notes	None

RpcServerUseAllProtseqsIf

Windows NT	Yes
Win95	Yes
Win32s	No
Import Library	rpcrt4.lib
Header File	rpcdce.h
Unicode	
Platform Notes	None

RpcServerUseProtseq

Windows NT	Yes
Win95	Yes
Win32s	No
Import Library	rpcrt4.lib
Header File	rpcdce.h
Unicode	
Platform Notes	None

RpcServerUseProtseqEp

Windows NT	Yes

Win95	Yes
Win32s	No
Import Library	rpcrt4.lib
Header File	rpcdce.h
Unicode	
Platform Notes	None

RpcServerUseProtseqIf

Windows NT	Yes
Win95	Yes
Win32s	No
Import Library	rpcrt4.lib
Header File	rpcdce.h
Unicode	
Platform Notes	None

RpcSmAllocate

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	rpcrt4.lib
Header File	rpcndr.h
Unicode	No
Platform Notes	None

RpcSmClientFree

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	rpcrt4.lib
Header File	rpcndr.h
Unicode	No
Platform Notes	None

RpcSmDestroyClientContext

Windows NT	New
Win95	Yes
Win32s	No
Import Library	rpcrt4.lib
Header File	rpcndr.h
Unicode	No
Platform Notes	None

RpcSmDisableAllocate

Windows NT	New
Win95	Yes
Win32s	No

Import Library	rpcrt4.lib
Header File	rpcndr.h
Unicode	No
Platform Notes	None

RpcSmEnableAllocate

Windows NT	New
Win95	Yes
Win32s	No
Import Library	rpcrt4.lib
Header File	rpcndr.h
Unicode	No
Platform Notes	None

RpcSmFree

Windows NT	New
Win95	Yes
Win32s	No
Import Library	rpcrt4.lib
Header File	rpcndr.h
Unicode	No
Platform Notes	None

RpcSmGetThreadHandle

Windows NT	Yes
Win95	Yes
Win32s	
Import Library	rpcrt4.lib
Header File	rpcndr.h
Unicode	No
Platform Notes	None

RpcSmSetClientAllocFree

Windows NT	New
Win95	Yes
Win32s	No
Import Library	rpcrt4.lib
Header File	rpcndr.h
Unicode	No
Platform Notes	None

RpcSmSetThreadHandle

Windows NT	Yes
Win95	Yes
Win32s	
Import Library	rpcrt4.lib
Header File	rpcndr.h

Unicode	No
Platform Notes	None

RpcSmSwapClientAllocFree

Windows NT	New
Win95	Yes
Win32s	No
Import Library	rpcrt4.lib
Header File	rpcndr.h
Unicode	No
Platform Notes	None

RpcSsAllocate

Windows NT	Yes
Win95	Yes
Win32s	
Import Library	rpcrt4.lib
Header File	rpcndr.h
Unicode	No
Platform Notes	None

RpcSsDestroyClientContext

Windows NT	Yes
Win95	Yes
Win32s	No
Import Library	rpcrt4.lib
Header File	rpcndr.h
Unicode	No
Platform Notes	None

RpcSsDisableAllocate

Windows NT	Yes
Win95	Yes
Win32s	
Import Library	rpcrt4.lib
Header File	rpcndr.h
Unicode	No
Platform Notes	None

RpcSsEnableAllocate

Windows NT	Yes
Win95	Yes
Win32s	
Import Library	rpcrt4.lib
Header File	rpcndr.h
Unicode	No
Platform Notes	None

RpcSsFree

Windows NT	Yes
Win95	Yes
Win32s	
Import Library	rpcrt4.lib
Header File	rpcndr.h
Unicode	No
Platform Notes	None

RpcSsGetThreadHandle

Windows NT	Yes
Win95	Yes
Win32s	
Import Library	rpcrt4.lib
Header File	rpcndr.h
Unicode	No
Platform Notes	None

RpcSsSetClientAllocFree

Windows NT	New
Win95	Yes
Win32s	No
Import Library	rpcrt4.lib
Header File	rpcndr.h
Unicode	No
Platform Notes	None

RpcSsSetThreadHandle

Windows NT	Yes
Win95	Yes
Win32s	
Import Library	rpcrt4.lib
Header File	rpcndr.h
Unicode	No
Platform Notes	None

RpcSsSwapClientAllocFree

Windows NT	New
Win95	Yes
Win32s	No
Import Library	rpcrt4.lib
Header File	rpcndr.h
Unicode	No
Platform Notes	None

RpcStringBindingCompose

Windows NT	Yes

Win95	Yes
Win32s	No
Import Library	rpcrt4.lib
Header File	rpcdce.h
Unicode	WinNT
Platform Notes	None

RpcStringBindingParse

Windows NT	Yes
Win95	Yes
Win32s	No
Import Library	rpcrt4.lib
Header File	rpcdce.h
Unicode	WinNT
Platform Notes	None

RpcStringFree

Windows NT	Yes
Win95	Yes
Win32s	No
Import Library	rpcrt4.lib
Header File	rpcdce.h
Unicode	WinNT
Platform Notes	None

RpcTestCancel

Windows NT	Yes
Win95	No
Win32s	No
Import Library	rpcrt4.lib
Header File	rpcdce.h
Unicode	WinNT
Platform Notes	None

RpcTryExcept

Windows NT	New
Win95	Yes
Win32s	No
Import Library	
Header File	rpc.h
Unicode	No
Platform Notes	None

RpcTryFinally

Windows NT	New
Win95	Yes
Win32s	No

Import Library	
Header File	rpc.h
Unicode	No
Platform Notes	None

RxNetAccessAdd

Windows NT	Yes
Win95	No
Win32s	No
Import Library	netapi32.lib
Header File	lmaccess.h
Unicode	WinNT
Platform Notes	All LanMan APIs are UNICODE only

RxNetAccessDel

Windows NT	Yes
Win95	No
Win32s	No
Import Library	netapi32.lib
Header File	lmaccess.h
Unicode	WinNT
Platform Notes	All LanMan APIs are UNICODE only

RxNetAccessEnum

Windows NT	Yes
Win95	No
Win32s	No
Import Library	netapi32.lib
Header File	lmaccess.h
Unicode	WinNT
Platform Notes	All LanMan APIs are UNICODE only

RxNetAccessGetInfo

Windows NT	Yes
Win95	No
Win32s	No
Import Library	netapi32.lib
Header File	lmaccess.h
Unicode	WinNT
Platform Notes	All LanMan APIs are UNICODE only

RxNetAccessGetUserPerms

Windows NT	Yes
Win95	No

Win32s	No
Import Library	netapi32.lib
Header File	lmaccess.h
Unicode	WinNT
Platform Notes	All LanMan APIs are UNICODE only

RxNetAccessSetInfo

Windows NT	Yes
Win95	No
Win32s	No
Import Library	netapi32.lib
Header File	lmaccess.h
Unicode	WinNT
Platform Notes	All LanMan APIs are UNICODE only

RxRemoteApi

Windows NT	Yes
Win95	No
Win32s	No
Import Library	netapi32.lib
Header File	lmremutl.h
Unicode	WinNT
Platform Notes	All LanMan APIs are UNICODE only

SafeArrayAccessData

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	oleaut32.lib
Header File	oleauto.h
Unicode	No
Platform Notes	None

SafeArrayAllocData

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	oleaut32.lib
Header File	oleauto.h
Unicode	No
Platform Notes	None

SafeArrayAllocDescriptor

Windows NT	Yes
Win95	Yes

Win32s	Yes
Import Library	oleaut32.lib
Header File	oleauto.h
Unicode	No
Platform Notes	None

SafeArrayCopy

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	oleaut32.lib
Header File	oleauto.h
Unicode	No
Platform Notes	None

SafeArrayCreate

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	oleaut32.lib
Header File	oleauto.h
Unicode	No
Platform Notes	None

SafeArrayDestroy

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	oleaut32.lib
Header File	oleauto.h
Unicode	No
Platform Notes	None

SafeArrayDestroyData

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	oleaut32.lib
Header File	oleauto.h
Unicode	No
Platform Notes	None

SafeArrayDestroyDescriptor

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	oleaut32.lib

Header File	oleauto.h
Unicode	No
Platform Notes	None

SafeArrayGetDim

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	oleaut32.lib
Header File	oleauto.h
Unicode	No
Platform Notes	None

SafeArrayGetElement

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	oleaut32.lib
Header File	oleauto.h
Unicode	No
Platform Notes	None

SafeArrayGetElemsize

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	oleaut32.lib
Header File	oleauto.h
Unicode	No
Platform Notes	None

SafeArrayGetLBound

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	oleaut32.lib
Header File	oleauto.h
Unicode	No
Platform Notes	None

SafeArrayGetUBound

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	oleaut32.lib
Header File	oleauto.h
Unicode	No

Platform Notes	None

SafeArrayLock

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	oleaut32.lib
Header File	oleauto.h
Unicode	No
Platform Notes	None

SafeArrayPtrOfIndex

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	oleaut32.lib
Header File	oleauto.h
Unicode	No
Platform Notes	None

SafeArrayPutElement

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	oleaut32.lib
Header File	oleauto.h
Unicode	No
Platform Notes	None

SafeArrayRedim

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	oleaut32.lib
Header File	oleauto.h
Unicode	No
Platform Notes	None

SafeArrayUnaccessData

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	oleaut32.lib
Header File	oleauto.h
Unicode	No
Platform Notes	None

SafeArrayUnlock

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	oleaut32.lib
Header File	oleauto.h
Unicode	No
Platform Notes	None

SaveDC

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	gdi32.lib
Header File	wingdi.h
Unicode	No
Platform Notes	Windows 95: int == 16 bits

ScaleViewportExtEx

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	gdi32.lib
Header File	wingdi.h
Unicode	No
Platform Notes	Windows 95: 16-bit coordinates only

ScaleWindowExtEx

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	gdi32.lib
Header File	wingdi.h
Unicode	No
Platform Notes	Windows 95: 16-bit coordinates only

ScheduleJob

Windows NT	Yes
Win95	Yes
Win32s	No
Import Library	winspool.lib
Header File	winspool.h
Unicode	No
Platform Notes	None

ScreenSaverConfigureDialog

Windows NT	Yes

Win95	Yes
Win32s	No
Import Library	scrnsave.lib
Header File	scrnsave.h
Unicode	No
Platform Notes	None

ScreenSaverProc

Windows NT	Yes
Win95	Yes
Win32s	No
Import Library	scrnsave.lib
Header File	scrnsave.h
Unicode	No
Platform Notes	None

ScreenToClient

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	user32.lib
Header File	winuser.h
Unicode	No
Platform Notes	None

ScrollConsoleScreenBuffer

Windows NT	Yes
Win95	Yes
Win32s	No
Import Library	kernel32.lib
Header File	wincon.h
Unicode	WinNT
Platform Notes	None

ScrollDC

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	user32.lib
Header File	winuser.h
Unicode	No
Platform Notes	None

ScrollWindow

Windows NT	Yes
Win95	Yes
Win32s	Yes

Import Library	user32.lib
Header File	winuser.h
Unicode	No
Platform Notes	None

ScrollWindowEx

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	user32.lib
Header File	winuser.h
Unicode	No
Platform Notes	None

SearchPath

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	kernel32.lib
Header File	winbase.h
Unicode	WinNT
Platform Notes	None

select

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	wsock32.lib
Header File	winsock.h
Unicode	No
Platform Notes	None

SelectClipPath

Windows NT	Yes
Win95	Yes
Win32s	No
Import Library	gdi32.lib
Header File	wingdi.h
Unicode	No
Platform Notes	Windows 95: 16-bit coordinates only

SelectClipRgn

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	gdi32.lib
Header File	wingdi.h

Unicode	No
Platform Notes	Windows 95: 16-bit coordinates only

SelectObject

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	gdi32.lib
Header File	wingdi.h
Unicode	No
Platform Notes	

SelectPalette

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	gdi32.lib
Header File	wingdi.h
Unicode	No
Platform Notes	Windows 95: int == 16 bits

send

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	wsock32.lib
Header File	winsock.h
Unicode	No
Platform Notes	None

SendDlgItemMessage

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	user32.lib
Header File	winuser.h
Unicode	WinNT
Platform Notes	None

SendMessage

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	user32.lib
Header File	winuser.h
Unicode	WinNT

Platform Notes	None

SendMessageCallback

Windows NT	Yes
Win95	Yes
Win32s	No
Import Library	user32.lib
Header File	winuser.h
Unicode	WinNT
Platform Notes	None

SendMessageTimeout

Windows NT	Yes
Win95	Yes
Win32s	No
Import Library	user32.lib
Header File	winuser.h
Unicode	WinNT
Platform Notes	None

SendNotifyMessage

Windows NT	Yes
Win95	Yes
Win32s	No
Import Library	user32.lib
Header File	winuser.h
Unicode	WinNT
Platform Notes	None

sendto

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	wsock32.lib
Header File	winsock.h
Unicode	No
Platform Notes	None

ServiceMain

Windows NT	Yes
Win95	No
Win32s	No
Import Library	
Header File	
Unicode	No
Platform Notes	None

SetAbortProc

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	gdi32.lib
Header File	wingdi.h
Unicode	No
Platform Notes	Windows 95: int == 16 bits

SetAclInformation

Windows NT	Yes
Win95	No
Win32s	No
Import Library	advapi32.lib
Header File	winbase.h
Unicode	No
Platform Notes	None

SetActiveWindow

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	user32.lib
Header File	winuser.h
Unicode	No
Platform Notes	None

SetArcDirection

Windows NT	Yes
Win95	No
Win32s	No
Import Library	gdi32.lib
Header File	wingdi.h
Unicode	No
Platform Notes	Windows 95: 16-bit coordinates only

SetBitmapBits

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	gdi32.lib
Header File	wingdi.h
Unicode	No
Platform Notes	Windows 95: int == 16 bits

SetBitmapDimensionEx

Windows NT	Yes

Win95	Yes
Win32s	Yes
Import Library	gdi32.lib
Header File	wingdi.h
Unicode	No
Platform Notes	Windows 95: int == 16 bits

SetBkColor

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	gdi32.lib
Header File	wingdi.h
Unicode	No
Platform Notes	Windows 95: int == 16 bits

SetBkMode

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	gdi32.lib
Header File	wingdi.h
Unicode	No
Platform Notes	Windows 95: int == 16 bits

SetBoundsRect

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	gdi32.lib
Header File	wingdi.h
Unicode	No
Platform Notes	Windows 95: 16-bit coordinates only

SetBrushOrgEx

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	gdi32.lib
Header File	wingdi.h
Unicode	No
Platform Notes	Windows 95: 16-bit coordinates only

SetCapture

Windows NT	Yes

Win95	Yes
Win32s	Yes
Import Library	user32.lib
Header File	winuser.h
Unicode	No
Platform Notes	None

SetCaretBlinkTime

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	user32.lib
Header File	winuser.h
Unicode	No
Platform Notes	None

SetCaretPos

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	user32.lib
Header File	winuser.h
Unicode	No
Platform Notes	None

SetClassLong

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	user32.lib
Header File	winuser.h
Unicode	WinNT
Platform Notes	None

SetClassWord

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	user32.lib
Header File	winuser.h
Unicode	No
Platform Notes	None

SetClipboardData

Windows NT	Yes
Win95	Yes
Win32s	Yes

Import Library	user32.lib
Header File	winuser.h
Unicode	No
Platform Notes	None

SetClipboardViewer

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	user32.lib
Header File	winuser.h
Unicode	No
Platform Notes	None

TranslateMessage

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	user32.lib
Header File	winuser.h
Unicode	No
Platform Notes	None

TransmitCommChar

Windows NT	Yes
Win95	Yes
Win32s	No
Import Library	kernel32.lib
Header File	winbase.h
Unicode	No
Platform Notes	None

TransmitFile

Windows NT	Yes
Win95	No
Win32s	No
Import Library	wsock32.lib
Header File	winsock.h
Unicode	No
Platform Notes	None

tree_into_ndr

Windows NT	Yes
Win95	No
Win32s	No
Import Library	rpcrt4.lib
Header File	rpcndr.h
Unicode	
Platform Notes	None

tree_peek_ndr

Windows NT	Yes
Win95	No
Win32s	No
Import Library	rpcrt4.lib
Header File	rpcndr.h
Unicode	
Platform Notes	None

tree_size_ndr

Windows NT	Yes

Win95	No
Win32s	No
Import Library	rpcrt4.lib
Header File	rpcndr.h
Unicode	
Platform Notes	None

UInt32x32To64

Windows NT	Yes
Win95	Yes
Win32s	No
Import Library	
Header File	winnt.h
Unicode	No
Platform Notes	None

UnhandledExceptionFilter

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	kernel32.lib
Header File	winbase.h
Unicode	No
Platform Notes	None

UnhookWindowsHook

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	user32.lib
Header File	winuser.h
Unicode	No
Platform Notes	None

UnhookWindowsHookEx

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	user32.lib
Header File	winuser.h
Unicode	No
Platform Notes	None

UnionRect

Windows NT	Yes
Win95	Yes
Win32s	Yes

Import Library	user32.lib
Header File	winuser.h
Unicode	No
Platform Notes	None

UnloadKeyboardLayout

Windows NT	Yes
Win95	Yes
Win32s	No
Import Library	user32.lib
Header File	winuser.h
Unicode	No
Platform Notes	None

UnlockFile

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	kernel32.lib
Header File	winbase.h
Unicode	No
Platform Notes	None

UnlockFileEx

Windows NT	Yes
Win95	No
Win32s	No
Import Library	kernel32.lib
Header File	winbase.h
Unicode	No
Platform Notes	None

UnlockResource

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	
Header File	winbase.h
Unicode	No
Platform Notes	None

UnlockSegment

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	
Header File	winbase.h

Unicode	No
Platform Notes	None

UnlockServiceDatabase

Windows NT	Yes
Win95	No
Win32s	No
Import Library	advapi32.lib
Header File	winsvc.h
Unicode	No
Platform Notes	None

UnmapViewOfFile

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	kernel32.lib
Header File	winbase.h
Unicode	No
Platform Notes	None

UnpackDDElParam

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	user32.lib
Header File	dde.h
Unicode	No
Platform Notes	None

UnrealizeObject

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	gdi32.lib
Header File	wingdi.h
Unicode	No
Platform Notes	Windows 95: int == 16 bits

UnregisterClass

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	user32.lib
Header File	winuser.h
Unicode	WinNT
Platform Notes	None

UnregisterHotKey

Windows NT	Yes
Win95	Yes
Win32s	No
Import Library	user32.lib
Header File	winuser.h
Unicode	No
Platform Notes	None

UpdateColors

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	gdi32.lib
Header File	wingdi.h
Unicode	No
Platform Notes	Windows 95: int == 16 bits

UpdateResource

Windows NT	Yes
Win95	No
Win32s	No
Import Library	kernel32.lib
Header File	winbase.h
Unicode	WinNT
Platform Notes	None

UpdateWindow

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	user32.lib
Header File	winuser.h
Unicode	No
Platform Notes	None

UuidCompare

Windows NT	New
Win95	Yes
Win32s	No
Import Library	rpcrt4.lib
Header File	rpcdce.h
Unicode	No
Platform Notes	None

UuidCreate

Windows NT	Yes

Win95	No
Win32s	No
Import Library	rpcrt4.lib
Header File	rpcdce.h
Unicode	No
Platform Notes	None

UuidCreateNil

Windows NT	New
Win95	Yes
Win32s	No
Import Library	rpcrt4.lib
Header File	rpcdce.h
Unicode	No
Platform Notes	None

UuidEqual

Windows NT	New
Win95	Yes
Win32s	No
Import Library	rpcrt4.lib
Header File	rpcdce.h
Unicode	No
Platform Notes	None

UuidFromString

Windows NT	Yes
Win95	No
Win32s	No
Import Library	rpcrt4.lib
Header File	rpcdce.h
Unicode	WinNT
Platform Notes	None

UuidHash

Windows NT	New
Win95	Yes
Win32s	No
Import Library	rpcrt4.lib
Header File	rpcdce.h
Unicode	No
Platform Notes	None

UuidIsNil

Windows NT	New
Win95	Yes
Win32s	No

Import Library	rpcrt4.lib
Header File	rpcdce.h
Unicode	No
Platform Notes	None

UuidToString

Windows NT	Yes
Win95	No
Win32s	No
Import Library	rpcrt4.lib
Header File	rpcdce.h
Unicode	WinNT
Platform Notes	None

ValidateRect

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	user32.lib
Header File	winuser.h
Unicode	No
Platform Notes	None

ValidateRgn

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	user32.lib
Header File	winuser.h
Unicode	No
Platform Notes	None

VarBoolFromCy

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	oleaut32.lib
Header File	oleauto.h
Unicode	No
Platform Notes	None

VarBoolFromDate

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	oleaut32.lib
Header File	oleauto.h

Unicode	No
Platform Notes	None

VarBoolFromDisp

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	oleaut32.lib
Header File	oleauto.h
Unicode	No
Platform Notes	None

VarBoolFromI2

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	oleaut32.lib
Header File	oleauto.h
Unicode	No
Platform Notes	None

VarBoolFromI4

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	oleaut32.lib
Header File	oleauto.h
Unicode	No
Platform Notes	None

VarBoolFromR4

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	oleaut32.lib
Header File	oleauto.h
Unicode	No
Platform Notes	None

VarBoolFromR8

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	oleaut32.lib
Header File	oleauto.h
Unicode	No
Platform Notes	None

VarBoolFromStr

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	oleaut32.lib
Header File	oleauto.h
Unicode	WinNT; Win95; Win32s
Platform Notes	All 32-bit OLE Apis are UNICODE only

VarBoolFromUI1

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	oleaut32.lib
Header File	oleauto.h
Unicode	No
Platform Notes	None

VarBstrFromBool

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	oleaut32.lib
Header File	oleauto.h
Unicode	WinNT; Win95; Win32s
Platform Notes	All 32-bit OLE Apis are UNICODE only

VarBstrFromCy

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	oleaut32.lib
Header File	oleauto.h
Unicode	WinNT; Win95; Win32s
Platform Notes	All 32-bit OLE Apis are UNICODE only

VarBstrFromDate

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	oleaut32.lib
Header File	oleauto.h
Unicode	WinNT; Win95; Win32s
Platform Notes	All 32-bit OLE Apis are

	UNICODE only

VarBstrFromDisp

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	oleaut32.lib
Header File	oleauto.h
Unicode	WinNT; Win95; Win32s
Platform Notes	All 32-bit OLE Apis are UNICODE only

VarBstrFromI2

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	oleaut32.lib
Header File	oleauto.h
Unicode	WinNT; Win95; Win32s
Platform Notes	All 32-bit OLE Apis are UNICODE only

VarBstrFromI4

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	oleaut32.lib
Header File	oleauto.h
Unicode	WinNT; Win95; Win32s
Platform Notes	All 32-bit OLE Apis are UNICODE only

VarBstrFromR4

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	oleaut32.lib
Header File	oleauto.h
Unicode	WinNT; Win95; Win32s
Platform Notes	All 32-bit OLE Apis are UNICODE only

VarBstrFromR8

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	oleaut32.lib
Header File	oleauto.h

Unicode	WinNT; Win95; Win32s
Platform Notes	All 32-bit OLE Apis are UNICODE only

VarBstrFromUI1

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	oleaut32.lib
Header File	oleauto.h
Unicode	WinNT; Win95; Win32s
Platform Notes	All 32-bit OLE Apis are UNICODE only

VarCyFromBool

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	oleaut32.lib
Header File	oleauto.h
Unicode	No
Platform Notes	None

VarCyFromDate

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	oleaut32.lib
Header File	oleauto.h
Unicode	No
Platform Notes	None

VarCyFromDisp

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	oleaut32.lib
Header File	oleauto.h
Unicode	No
Platform Notes	None

VarCyFromI2

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	oleaut32.lib
Header File	oleauto.h
Unicode	No

Platform Notes	None

VarCyFromI4

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	oleaut32.lib
Header File	oleauto.h
Unicode	No
Platform Notes	None

VarCyFromR4

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	oleaut32.lib
Header File	oleauto.h
Unicode	No
Platform Notes	None

VarCyFromR8

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	oleaut32.lib
Header File	oleauto.h
Unicode	No
Platform Notes	None

VarCyFromStr

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	oleaut32.lib
Header File	oleauto.h
Unicode	WinNT; Win95; Win32s
Platform Notes	All 32-bit OLE Apis are UNICODE only

VarCyFromUI1

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	oleaut32.lib
Header File	oleauto.h
Unicode	No
Platform Notes	None

VarDateFromBool

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	oleaut32.lib
Header File	oleauto.h
Unicode	No
Platform Notes	None

VarDateFromCy

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	oleaut32.lib
Header File	oleauto.h
Unicode	No
Platform Notes	None

VarDateFromDisp

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	oleaut32.lib
Header File	oleauto.h
Unicode	No
Platform Notes	None

VarDateFromI2

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	oleaut32.lib
Header File	oleauto.h
Unicode	No
Platform Notes	None

VarDateFromI4

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	oleaut32.lib
Header File	oleauto.h
Unicode	No
Platform Notes	None

VarDateFromR4

Windows NT	Yes

Win95	Yes
Win32s	Yes
Import Library	oleaut32.lib
Header File	oleauto.h
Unicode	No
Platform Notes	None

VarDateFromR8

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	oleaut32.lib
Header File	oleauto.h
Unicode	No
Platform Notes	None

VarDateFromStr

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	oleaut32.lib
Header File	oleauto.h
Unicode	WinNT; Win95; Win32s
Platform Notes	All 32-bit OLE Apis are UNICODE only

VarDateFromUI1

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	oleaut32.lib
Header File	oleauto.h
Unicode	No
Platform Notes	None

VarI2FromBool

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	oleaut32.lib
Header File	oleauto.h
Unicode	No
Platform Notes	None

VarI2FromCy

Windows NT	Yes
Win95	Yes
Win32s	Yes

Import Library	oleaut32.lib
Header File	oleauto.h
Unicode	No
Platform Notes	None

VarI2FromDate

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	oleaut32.lib
Header File	oleauto.h
Unicode	No
Platform Notes	None

VarI2FromDisp

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	oleaut32.lib
Header File	oleauto.h
Unicode	No
Platform Notes	None

VarI2FromI4

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	oleaut32.lib
Header File	oleauto.h
Unicode	No
Platform Notes	None

VarI2FromR4

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	oleaut32.lib
Header File	oleauto.h
Unicode	No
Platform Notes	None

VarI2FromR8

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	oleaut32.lib

Header File	oleauto.h
Unicode	No
Platform Notes	None

VarI2FromStr

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	oleaut32.lib
Header File	oleauto.h
Unicode	WinNT; Win95; Win32s
Platform Notes	All 32-bit OLE Apis are UNICODE only

VarI2FromUI1

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	oleaut32.lib
Header File	oleauto.h
Unicode	No
Platform Notes	None

VarI4FromBool

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	oleaut32.lib
Header File	oleauto.h
Unicode	No
Platform Notes	None

VarI4FromCy

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	oleaut32.lib
Header File	oleauto.h
Unicode	No
Platform Notes	None

VarI4FromDate

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	oleaut32.lib
Header File	oleauto.h
Unicode	No

Platform Notes	None

VarI4FromDisp

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	oleaut32.lib
Header File	oleauto.h
Unicode	No
Platform Notes	None

VarI4FromI2

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	oleaut32.lib
Header File	oleauto.h
Unicode	No
Platform Notes	None

VarI4FromR4

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	oleaut32.lib
Header File	oleauto.h
Unicode	No
Platform Notes	None

VarI4FromR8

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	oleaut32.lib
Header File	oleauto.h
Unicode	No
Platform Notes	None

VarI4FromStr

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	oleaut32.lib
Header File	oleauto.h
Unicode	WinNT; Win95; Win32s
Platform Notes	All 32-bit OLE Apis are

	UNICODE only

VarI4FromUI1

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	oleaut32.lib
Header File	oleauto.h
Unicode	No
Platform Notes	None

VariantChangeType

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	oleaut32.lib
Header File	oleauto.h
Unicode	No
Platform Notes	None

VariantChangeTypeEx

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	oleaut32.lib
Header File	oleauto.h
Unicode	No
Platform Notes	None

VariantClear

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	oleaut32.lib
Header File	oleauto.h
Unicode	No
Platform Notes	None

VariantCopy

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	oleaut32.lib
Header File	oleauto.h
Unicode	No
Platform Notes	None

VariantCopyInd

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	oleaut32.lib
Header File	oleauto.h
Unicode	No
Platform Notes	None

VariantInit

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	oleaut32.lib
Header File	oleauto.h
Unicode	No
Platform Notes	None

VariantTimeToDosDateTime

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	oleaut32.lib
Header File	oleauto.h
Unicode	No
Platform Notes	None

VarR4FromBool

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	oleaut32.lib
Header File	oleauto.h
Unicode	No
Platform Notes	None

VarR4FromCy

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	oleaut32.lib
Header File	oleauto.h
Unicode	No
Platform Notes	None

VarR4FromDate

Windows NT	Yes
Win95	Yes

Win32s	Yes
Import Library	oleaut32.lib
Header File	oleauto.h
Unicode	No
Platform Notes	None

VarR4FromDisp

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	oleaut32.lib
Header File	oleauto.h
Unicode	No
Platform Notes	None

VarR4FromI2

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	oleaut32.lib
Header File	oleauto.h
Unicode	No
Platform Notes	None

VarR4FromI4

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	oleaut32.lib
Header File	oleauto.h
Unicode	No
Platform Notes	None

VarR4FromR8

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	oleaut32.lib
Header File	oleauto.h
Unicode	No
Platform Notes	None

VarR4FromStr

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	oleaut32.lib

Header File	oleauto.h
Unicode	WinNT; Win95; Win32s
Platform Notes	All 32-bit OLE Apis are UNICODE only

VarR4FromUI1

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	oleaut32.lib
Header File	oleauto.h
Unicode	No
Platform Notes	None

VarR8FromBool

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	oleaut32.lib
Header File	oleauto.h
Unicode	No
Platform Notes	None

VarR8FromCy

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	oleaut32.lib
Header File	oleauto.h
Unicode	No
Platform Notes	None

VarR8FromDate

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	oleaut32.lib
Header File	oleauto.h
Unicode	No
Platform Notes	None

VarR8FromDisp

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	oleaut32.lib
Header File	oleauto.h
Unicode	No

Platform Notes	None

VarR8FromI2

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	oleaut32.lib
Header File	oleauto.h
Unicode	No
Platform Notes	None

VarR8FromI4

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	oleaut32.lib
Header File	oleauto.h
Unicode	No
Platform Notes	None

VarR8FromR4

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	oleaut32.lib
Header File	oleauto.h
Unicode	No
Platform Notes	None

VarR8FromStr

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	oleaut32.lib
Header File	oleauto.h
Unicode	No
Platform Notes	None

VarR8FromUI1

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	oleaut32.lib
Header File	oleauto.h
Unicode	No
Platform Notes	None

VarUI1FromBool

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	oleaut32.lib
Header File	oleauto.h
Unicode	No
Platform Notes	None

VarUI1FromCy

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	oleaut32.lib
Header File	oleauto.h
Unicode	No
Platform Notes	None

VarUI1FromDate

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	oleaut32.lib
Header File	oleauto.h
Unicode	No
Platform Notes	None

VarUI1FromDisp

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	oleaut32.lib
Header File	oleauto.h
Unicode	No
Platform Notes	None

VarUI1FromI2

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	oleaut32.lib
Header File	oleauto.h
Unicode	No
Platform Notes	None

VarUI1FromI4

Windows NT	Yes

Win95	Yes
Win32s	Yes
Import Library	oleaut32.lib
Header File	oleauto.h
Unicode	No
Platform Notes	None

VarUI1FromR4

Windows NT	Yes
Win95	Yes
Win32s	Yes
Import Library	oleaut32.lib
Header File	oleauto.h
Unicode	No
Platform Notes	None

Microsoft Win32 Developer's Reference

You have requested information from the **Microsoft Win32 Developer's Reference**. One or more of these help files is not available on your system.