Legal Information

MIDL Programmer's Guide and Reference

About This Guide

This book is a reference for the Microsoft Interface Definition Language (MIDL) and a user's guide for the MIDL compiler, which generates C language files from your IDL files.

Part I, Using the MIDL Compiler, lists the requirements for the C-compiler and pre-processor that MIDL must interoperate with. This section also describes the files that the MIDL compiler generates for RPC stubs, OLE interfaces, and OLE type libraries.

Part II, the MIDL Command-Line Reference, contains reference information for each command-line switch and switch option recognized by the MIDL Compiler.

Part III, the MIDL Language Reference, contains a reference entry for each keyword in the Microsoft Interface Definition Language.

Part IV, MIDL Compiler Errors and Warnings, lists the error and warning messages that the MIDL compiler can generate.

Using The MIDL Compiler

The MIDL compiler is automatically installed as part of the Win32 SDK or Win16 SDK setup. The following topics describe MIDL's C compiler and preprocessor requirements, the link libraries that are part of the RPC product, and the files that MIDL generates for OLE interfaces, RPC interfaces, and type libraries. For more information about the files that make up the RPC product, see Installing the RPC Programming Environment.

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:

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. You cannot use the backslash character (\) 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 following topics describe the requirements for the C compiler and preprocessor.

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.

These are the four workarounds that 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 also 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.

Incorrect usage of the MIDL command-line **/cpp_opt** switch can produce error messages related to the IDL file. These 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 shown:

```
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.

As an 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 16-bit Visual C/C++ 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, 16-bit 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 16-bit Visual C/C++, use the compiler switch that is equivalent to **/GA**.

Previous versions of the Microsoft C compiler and 16-bit Windows used calls to the Windows 3.*x* function **MakeProcInstance**, and the **/Gw** switch to generate this code. The **/GA** switch is a more efficient way to accomplish the same tasks.

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

These are the static libraries that are included in MS-DOS for the RPC client application:

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 included 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
RPC16DG3.RPC	Datagram UPD transport
RPC16DG6.RPC	Datagram IPX 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 included for Microsoft 16-bit Windows 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 included for Microsoft 16-bit Windows clients:

Dynamic-link library	Description
RPCNS1.DLL	Name service
RPCRT1.DLL	16-bit Windows run-time library
RPC16C1.DLL	Client named-pipe transport
RPC16C3.DLL	Client TCP/IP transport
RPC16C5.DLL	Client NetBIOS transport
RPC16C6.DLL	SPX transport
RPC16DG3.DLL	Datagram UDP transport
RPC16DG6.DLL	Datagram IPX transport

Link Libraries for Microsoft Windows NT and Windows 95

The following RPC import libraries are included for Microsoft 32-bit Windows clients and servers:

Import library	Description
RPCNDR.LIB	Helper functions
RPCNS4.LIB	Name-service functions
RPCRT4.LIB	32-bit Windows run-time functions

The following RPC libraries are included for Microsoft 32-bit Windows clients and servers:

Dynamic-link library	Description
RPCLTC1.DLL	Client named-pipe transport
RPCLTS1.DLL	Server named-pipe transport (NT only)
RPCLTC3.DLL	Client TCP/IP transport
RPCLTS3.DLL	Server TCP/IP transport
RPCLTC5.DLL	Client NetBIOS transport
RPCLTS5.DLL	Server NetBIOS transport
RPCLTC6.DLL	Client SPX transport
RPCLTS6.DLL	Server SPX transport
RPCDGC6.DLL	Client IPX transport (NT only)
RPCDGS6.DLL	Server IPX transport (NT only)
RPCDGC3.DLL	Client UDP transport (NT only)
RPCDGS3.DLL	Server UDP transport (NT only)
RPCNS4.DLL	Name service
RPCRT4.DLL	32-bit 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 and 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.

If you choose, you can override this default by specifying the following on the command line:

```
-U__midl
```

See also

/U

Files Generated for an RPC Interface

The MIDL compiler generates the C-language stub and header files necessary to create the interface between the client application and the server application. The following topics describe each of the files generated for an RPC interface. For more information on defining and implementing an RPC interface see *The Microsoft RPC Programmer's Guide and Reference*.

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.

Targetting Stubs for Specific 32-Bit Platforms

Some of the features of Microsoft RPC and the MIDL 3.0 compiler are platform-specific and are intended for implementation in distributed applications that run only on Windows NT 4.0 or a later version. Some features are supported in Windows NT 3.51 and Windows 95, as well as in later versions, but are not supported on older 32-bit, or 16-bit platforms.

As a safeguard, the MIDL 3.0 compiler generates macros that facilitate compatibility checking during the C compilation phase. If the interface uses features supported only on Windows NT 4.0, MIDL generates a TARGET_IS_NT40_OR_LATER macro. If supported features require Windows NT 3.51 or Windows95, a TARGET_IS_NT351_OR_WIN95_OR_LATER macro is generated. These macros, defined in rpcndr.h, depend on the environment variables WINVER and _WIN32_WINNT and are evaluated by the C/C++ compiler.

If, at compile time, you get an error message indicating that you need a specific platform to run this stub, first check to make sure you have not used a feature not available on this platform. For example, the **pipe** type constructor, the /**Oif** compiler option, and the **user_marshal** and **wire_marshal** attributes are only available on Windows NT 4.0. Stubs using these features will not run on earlier systems. The /**Oic** compiler switch is available on NT. 3.51 and Windows 95, but is not available in earlier versions.

Also, the technology needed to use OLE and OLE Automation data types (for example, BSTR or STGMEDIUM) in remote operations is present only on NT 4.0. Therefore, you cannot develop a custom interface that uses these data types in a remote procedure call to run on earlier platforms.

If you know that your target platform is correct for the features you are using, you need to explicitly set the environment variables in your makefile.

To build for Windows NT 3.51 or Windows 95

Add this line to your makefile:

```
CFLAGS = $(CFLAGS) -DWINVER=0x400
```

To build for Windows NT 4.0

Add this line to your makefile:

```
CFLAGS = $(CFLAGS) -D_WIN32_WINNT=0x400
```

Note that this target control is not in effect when you are building for MS-DOS, 16-bit Windows, or Macintosh platforms.

See Also

/Oi, pipe, wire_marshal, user_marshal, Marshaling OLE Data Types

Files Generated for an OLE Interface

The following topics describe 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.

See Also

ACF, **/app_config**, **IDL**, Building a Proxy/Stub DLL

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, and 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.

Marshaling OLE Data Types

To make it easier to use certain OLE Automation data types and OLE handles in remote operations, **wire_marshal** typedefs for these data types, and their related helper functions, are available by importing Win32 IDL files and linking to the OLE and OLE AUTOMATION DLL files. These files are automatically installed on your system during installation of Microsoft® Windows NT® or Windows® 95.

- To use the BSTR data type in remote procedure calls, import the wtypes.idl file into your interface definition (IDL) file and link to oleaut32.lib when building your distributed application. This will let your stubs use the ready-made helper functions BSTR_UserSize, BSTR_UserMarshal, BSTR_UserUnmarshal, and BSTR_UserFree.
- To use other OLE Automation data types, such as VARIANT and SAFEARRAY, or types that use those types (for example, DISPPARAM and EXCEPINFO), import the objidl.idl file into your IDL file and link to the oleaut32.lib at build time. This will allow you to use the appropriate helper routines.
- To use OLE data types and handles (such as CLIPFORMAT, SNB, STGMEDIUM, ASYNC_STGMEDIUM, HMETAFILE_PICT, HENHMETAFILE, HMETAFILE, HBITMAP, HPALETTE, and HGLOBAL), import the objidl.idl file into your interface definition file and link to the ole32.lib at build time.
- The following OLE handles are also defined with the **wire_marshal** attribute, but only as handles within a machine since they cannot be used in remote procedure calls to other machines at this time: HWND, HMENU, HACCEL, HDC, HFONT, HICON, HBRUSH. Import the objidl.idl file into your IDL file and link to ole32.lib at build time to use these handles in interprocess communication on a single machine. The helper routines for these data types may change to include support for remote marshaling in a future release.

Note The technology needed to support the above-described data types in proxy/stub code is part of Windows NT 4.0 only; this support is not present in Windows NT 3.51, Windows 95, or earlier versions of Windows. Therefore any application you develop on Windows NT 4.0, that uses these data types in remote procedure calls, **will not run** on an earlier platform. (The upcoming DCOM release for Windows 95 will include this support.) If your application needs to run on older platforms, and it uses these data types in remote procedure calls, you will need to create your own marshaling routines, using the **transmit_as** or **represent_as** attributes and their related helper functions. This restriction does not apply to type libraries generated with MIDL 3.0 for Automation/dual interfaces.

See Also

The wire_marshal Attribute, The type_UserSize Function, The type_UserMarshal Function, The type_UserUnmarshal Function, The type_UserFree Function, The transmit_as Attribute, The represent_as Attribute, Targetting Stubs for Specific 32-Bit Platforms

Generating a Type Library With MIDL

Microsoft's Interface Definition Language (IDL) now includes the complete Object Definition Language (ODL) syntax. This allows you to use the 32-bit MIDL compiler instead of MKTYPLIB.EXE to generate a type library and optional header files for an OLE application.

> **Note** When the documentation refers to an ODL file, this means a file that MKTYPLIB can parse. When it refers to an IDL file, this means a file that MIDL parses. This is strictly a naming convention. The MIDL compiler will parse an input file regardless of its filename extension.

The top-level element of the ODL syntax is the library statement (library block). Every other ODL statement, with the exception of the attributes that are applied to the library statement, must be defined within the library block.When the MIDL compiler sees a library block it generates a type library in much the same way as MKTYPLIB does. With a few exceptions, described in Differences Between MIDL and MKTYPLIB, the statements within the library block should follow the same syntax as in the ODL language and MKTYPLIB.

You can apply ODL attributes to elements that are defined either inside or outside the library block. These attributes have no effect outside the library block unless the element they are applied to is referenced from within the library block. Statements inside the library block can reference an outside element either by using it as a base type, inheriting from it, or by referencing it on a line as shown:

```
<some IDL definitions including definitions for interface IFoo and struct
bar>
[<some attributes>]
library a
{
interface IFoo;
struct bar;
...
};
```

If an element defined outside the library block is referenced within the library block, then its definition will be put into the generated type library.

The MIDL compiler treats the statements outside of a library block as a typical IDL file and parses those statements as it has always done. Normally, this means generating C-language stubs for an RPC application.

For more information about the general syntax for an ODL file see ODL File Syntax.

Additional Files Required To Generate a Type Library

In order to compile an IDL file that contains a library statement, the OLE and OLE AUTOMATION DLL files must be on your system. These files are automatically installed during installation of Microsoft® Windows NT™ or Windows® 95.

Effective with Windows NT 4.0, there is a new version of OLEAUT32.DLL that supports a richer format for 32-bit type libraries. MIDL looks for this DLL on the build machine; if the new version is present, MIDL generates a new-format type library, otherwise it generates an old-format type library.

Note for 16-bit developers

If your application must interoperate with 16-bit applications, you must use the old-format type library for compatibility. The MIDL command-line option **/old** overrides this default and directs the MIDL compiler to generate old-format type libraries even if the newer version of OLEAUT32.DLL is present.

Some of the base types that MKTYPLIB supports are not directly supported in MIDL. MIDL obtains definitions for these base types by automatically importing oaidl.idl whenever it sees a library statement. You need to ensure that this file is somewhere in your include path. The oaidl.idl file, and the files that it imports (objidl.idl, unkwn.idl, and wtypes.idl) are automatically installed when you install the Win32 SDK.

See Also

Marshaling OLE Data Types, **/old**, **/new**

Differences Between MIDL and MKTYPLIB

There are a few key areas in which the MIDL compiler differs from MKTYPLIB. Most of these differences arise because MIDL is oriented more toward C-syntax than MKTYPLIB.

In general, you will want to use the MIDL syntax in your IDL files. However, if you need to compile an existing ODL file, or otherwise maintain compatibility with MKTYPLIB, use the **/mktyplib203** MIDL compiler option to force MIDL to behave like MKTYPLIB.EXE, version 2.03. (This is the last release of the MKTYPLIB tool.) Specifically, the /**mktyplib203** option resolves these differences:

- **typedef syntax for complex data types**

  In MKTYPLIB, both of the following definitions generate a TKIND_RECORD for "bar" in the type library. The tag "foo" is optional and, if used, will not show up in the type library.

  ```
  typedef struct foo { ... } bar;
  typedef struct { ... } bar;
  ```

  If an optional tag is missing, MIDL will generate it, effectively adding a tag to the definition supplied by the user. Since the first definition has a tag, MIDL will generate a TKIND_RECORD for "foo" and a TKIND_ALIAS for "bar" (defining "bar" as an alias for "foo"). Because the tag is missing in the second definition, MIDL will generate a TKIND_RECORD for a mangled name, internal to MIDL, that is not meaningful to the user and a TKIND_ALIAS for "bar". This has potential implications for type library browsers that simply show the name of a record in their user interface. If you expect a TKIND_RECORD to have a real name, unrecognizable names could appear in the user interface.This behavior also applies to **union** and **enum** definitions, with the MIDL compiler generating TKIND_UNIONs and TKIND_ENUMs, respectively.

  MIDL also allows C-style **struct**, **union** and **enum** definitions. For example, the following definition is legal in MIDL:

  ```
  struct foo { ... };
  typedef struct foo bar;
  ```

- **boolean data types**

  In MKTYPLIB, the **boolean** base type and the MKTYPLIB data type BOOL equate to VT_BOOL, which maps to VARIANT_BOOL, and which is defined as a **short**. In MIDL, the **boolean** base type is equivalent to VT_UI1, which is defined as an **unsigned char**, and the BOOL data type is defined as a **long**. This leads to difficulties if you mix IDL syntax and ODL syntax in the same file while still trying to maintain compatibility with MKTYPLIB. Because the data types are different sizes, the marshaling code will not match what is described in the type information. If you want a VT_BOOL in your type library, you should use the VARIANT_BOOL data type.

- **GUID definitions in header files**

  In MKTYPLIB, GUIDs are defined in the header file with a macro that can be conditionally compiled to generate either a GUID predefinition or an instantiated GUID. MIDL normally puts GUID predefinitions in its generated header files and GUID instantiations only in the file generated by the /**iid** switch.

The following differences in behavior can not be resolved by using the /mktyplib203 switch:

- **Scope of symbols in an enum declaration**

  In MKTYPLIB the scope of symbols in an enum is local. In MIDL, the scope of symbols in an enum is global, as it is in C. For example, the following code will compile in MKTYPLIB, but will generate a duplicate name error in MIDL:

  ```
  typedef struct { ... } a;
  enum {a=1, b=2, c=3};
  ```

- **Scope of public attribute**

  If you apply the **public** attribute to an interface block, MKTYPLIB treats every typdef inside that interface block as public. MIDL requires that you explicitly apply the **public** attribute to those typedefs that you want public.
- **Importlib search order**

  If you import more than one type library, and if these libraries contain duplicate references, MKTYPLIB resolves this by using the first reference that it finds. MIDL will use the last reference that it finds. For example, given the following ODL syntax, library C will use the FOO typedef from library A if you compile with MKTYPLIB, and the FOO typedef from library B if you compile with MIDL:

```
[...]library A
{
    typedef struct tagFOO
    {...}FOO
}

[...]library B
{
    typedef struct tagFOO
    {...} FOO
}

[...]library C
{
    importlib (A.TLB)
    importlib (B.TLB)
    typedef struct tagBAR
    {FOO y;}BAR
}
```

  The appropriate workaround for this is to qualify each such reference with the correct import library name, like this:

```
typedef struct tagBAR
    {A.FOO y;}BAR
```

- **VOID data type not recognized**

  MIDL recognizes the C-language **void** data type and does not recognize the OLE Automation VOID data type. If you have an ODL file that uses VOID, place this definition at the top of the file:

```
#define VOID void
```

- **Exponential notation**

  MIDL requires that values expressed in exponential notation be contained within quotation marks. For example, "-2.5E+3".
- **LCID values and constants**

  Normally MIDL does not consider the LCID when parsing files. To force this behavior for a value, or if you need to use locale-specific notation when defining a constant, enclose the value or constant in quotation marks.

See Also

/mktyplib203, /iid, Marshaling OLE Data Types

ODL Language Features in MIDL

The following topics list the Object Description Language (ODL) attributes, keywords, statements, and directives that are now part of the Microsoft Interface Definition Language (MIDL).

ODL Attributes

appobject
bindable
control
default
defaultvalue
displaybind
dllname
dual
entry
helpcontext
helpstring
helpfile
hidden
id
immediatebind
in
lcid
licensed
nonextensible
odl
oleautomation
optional
out
propget
propput
propputref
public
readonly
requestedit
restricted
retval
source
uuid
vararg
version

ODL Keywords, Statements, and Directives

coclass
dispinterface
enum
importlib
interface
library
module
struct
typedef
union

For information on how to marshal OLE Automation types, such as BSTR, VARIANT, and SAFEARRAY, see Marshaling OLE Data Types.

Generating a Proxy DLL and a Type Library From a Single IDL File

You can use a single IDL file to generate both the proxy stubs and header files for marshaling code, and a type library. You do this by defining an interface outside the library block and then referencing that interface from inside the library block, as shown in this example:

```
//file: AllKnown.idl

[object, uuid(. . .), <other interface attributes>]
interface IKnown : IUnknown {
import "unknwn.idl"
<declarations, etc. for IKnown interface go here>
};

[<library attributes>]library KnownLibrary {

//reference interface IKnown:
interface IKnown;

//or create a new class:
  [<coclass attributes>] coclass KnowMore {
   interface IKnown;
  };
};
```

See Also

Marshaling OLE Data Types, Additional Files Required To Generate a Type Library

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 General MIDL Command-line Syntax describes the general command-line syntax.

General MIDL Command-line Syntax

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, 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.

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.

@ Response File Command

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 ("\") cannot 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 does not support nested response files.

See Also

General MIDL Command-line Syntax

/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 filename.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 relating to **/acf**, see **IDL**.

See Also

General MIDL Command-line Syntax

/align

midl /align:*alignment*

alignment

Specifies the alignment for types in the library. The *alignment* value can be 1, 2, 4, or 8. The value 1 indicates natural alignment; *n* indicates alignment on byte *n*. When you do not specify the /**align** switch, the default is 8.

Example

```
midl /align:4 filename.idl
```

Remarks

The **/align** switch is functionally the same as the MIDL /**Zp** option and is recognized by the MIDL compiler solely for backward compatibility with MKTYPLIB. If you are generating a new makefile, use the /**Zp** switch.

The alignment value corresponds to the **/Zp** option value used by the Microsoft C/C++ compiler.Be sure that you specify the same alignment when you invoke the C compiler as when you invoke the MIDL compiler. For more information, see your Microsoft C/C++ programming documentation.

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

See Also

General MIDL Command-line Syntax, **/Zp**

/app_config

midl /app_config

Examples

```
midl /app_config filename.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

General MIDL Command-line Syntax

/c_ext

midl /c_ext

This switch is obsolete with the current version (3.0) of the MIDL compiler. However, using the switch will not generate a compiler error, so you do not have to remove references to /**ms_ext** or /**c_ext** from an existing makefile.

The following features are now available by default:

- Many existing header files define types with 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 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.
- You can omit directional attributes (**in**, **out**).

The following C-language extensions are supported in default mode:

- Bit fields in structures and unions
- Comments that start with two slash characters ("//")
- External declarations
- Procedures with ellipses in the parameter list
- On 32-bit platforms, **int** is a native 32-bit base type. On 16-bit platforms, **int** is recognized but is not a remotable type
- 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**.

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

See Also

/app_config, **/osf**, General MIDL Command-line Syntax

/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 filename.idl
midl /char unsigned filename.idl
midl /char ascii7 filename.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. Note that the use of explicitly signed **char** types in your IDL file is not supported by DCE IDL. Therefore, this feature is not available when you compile with the MIDL /**osf** switch.

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

See Also

char, General MIDL Command-line Syntax, /**osf**, **small**

/client

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

stub

Generates the client-side files.

none

Does not generate any client-side files.

Examples

```
midl /client none filename.idl
midl /client stub filename.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, **/server**, General MIDL Command-line Syntax

/confirm

midl /confirm

Examples

```
midl /confirm
midl /confirm @response.rsp filename.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, General MIDL Command-line Syntax

/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" filename.idl
midl /cpp_cmd "mycpp" /DFLAG=TRUE /Ic:\tmp filename.idl
midl /cpp_opt "/E /DFLAG=TRUE" filename.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, General MIDL Command-line Syntax, **/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" filename.idl
midl /cpp_cmd "mycpp" /DFLAG=TRUE /Ic:\tmp filename.idl
midl /cpp_opt "/E /DFLAG=TRUE" filename.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, General MIDL Command-line Syntax, **/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 filename.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, General MIDL Command-line Syntax, **/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 filename.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**, General MIDL Command-line Syntax, **/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

General MIDL Command-line Syntax

/env

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

dos

Directs the MIDL compiler to generate stub files, or a type library file, for an MS-DOS environment.

win16

Directs the MIDL compiler to generate stub files, or a type library file, for the 16-bit Microsoft Windows environment such as Microsoft Windows 3.*x* or Microsoft Windows for Workgroups 3.11.

mac

Directs the MIDL compiler to generate stub files, or a type library file, for the Apple® Macintosh® (680x0) environment.

Note The MIDL compiler does not generate a server-stub file when you use the **/env** switch with the **dos**, **win16**, or **mac** options.

win32

Directs the MIDL compiler to generate stub files, or a type library file, for a 32-bit Microsoft Windows environment—either Microsoft® Windows® 95 or Microsoft® Windows NT™.

Examples

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

Be sure you 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:

- When **dos** is specified, **_ _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 specified, **_ _far** precedes pointer declarations in the generated files, stub files use 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 specified, generated stubs use C-compiler packing-level 8 for all types involved in remote operations.
- When **mac** is specified, the stub files use packing level 2 for all types involved in remote operations. The **mac** environment does not support object interfaces.

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

See Also

General MIDL Command-line Syntax, **/pack**, **/Zp**

/error

midl /error { allocation | stub_data | ref | bounds_check | **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.

ref

Generates code that checks at run time to ensure that no NULL [ref] pointers are being passed to the client stubs and raises an RPC_X_NULL_REF_POINTER exception if it finds any.

bounds_check

Checks size of conformant-varying and varying arrays against transmission length specification.

none

Performs no error checking.

all

Performs all error checking.

Examples

```
midl /error allocation filename.idl
midl /error none filename.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 certain memory-access errors. The enum errors that are checked are truncation errors caused by conversion between **long enum** types (32-bit integers) and **short enum** types (the network-data representation of **enum**) and the number of identifiers in an enumeration exceeding 32,767. The memory-access error checking is for pointers that exceed the end of the buffer in marshalling code and for conformant arrrays whose size is less than zero. Use the /**error bounds_check** flag to check for other invalid array bounds.

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

General MIDL Command-line Syntax

/h

midl /h *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 special characters.

Examples

```
midl /h tlibhead.h filename.idl
midl /h "midl.h" filename.idl
```

Remarks

The **/h** option is functionally equivalent to the /**header** option. The /**h** switch specifies *filename* as the name for a header file that contains all the definitions contained in the IDL file, without the IDL syntax. This file can be used as a C or C++ header file.

See Also

General MIDL Command-line Syntax, **/header**

/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 special characters.

Example

```
midl /header "bar.h" filename.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

General MIDL Command-line Syntax, **/h**, **/cstub**, **/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

General MIDL Command-line Syntax, **/confirm**

/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 filename.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 (32-bit Windows, 16-bit Windows, 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

General MIDL Command-line Syntax, **/acf**, **/cpp_cmd**, **/cpp_opt**, **/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" filename.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

General MIDL Command-line Syntax, **/header**, **/out**, **/proxy**

/import

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

/mktyplib203

midl /mktyplib203

Examples

```
midl /mktyplib203 myoldodl.odl
midl /mktyplib203 oldhabit.idl
```

Remarks

The /**mktyplib203** switch forces the MIDL compiler to parse the input file in much the same manner as MKTYPLIB.EXE, version 2.03, would handle the file. In order to use this switch, the input file must follow the ODL syntax exactly – you cannot place any statements outside of the library block. Specifically, using the /**mktyplib203** switch resolves the following discrepancies between MKTYPLIB and MIDL:

- MKTYPLIB typdef syntax is required for **struct**, **union** and **enum** data types.
- GUIDs in the header files are defined with a macro that can be conditionally compiled to generate either a GUID predefinition or an instantiated GUID.
- The base type BOOL is represented as a VARIANT_BOOL, which is defined as a short.

See Differences Between MIDL and MKTYPLIB for a more detailed description of these differences.

There is only one discrepancy betweey MKTYPLIB.EXE and MIDL 3.0 that using the /**mktyplib203** switch will not resolve and that is the scope of symbols in an **enum** declaration. In MKTYPLIB, these symbols have local scope; in MIDL 3.0 they have global scope. For example, the following code will generate a duplicate name error in MIDL:

```
typedef struct {. . . } a;
enum { a=1, b=2, c=3};
```

See Also

General MIDL Command-line Syntax, Generating a Type Library With MIDL, Differences Between MIDL and MKTYPLIB

/ms_ext

midl /ms_ext

Effective with MIDL version 3.0, the features enabled by this switch are now the default mode for the MIDL compiler. Using the switch will not generate a compiler error, so you do not have to remove references to /**ms_ext** or /**c_ext** from an existing makefile.

The following Microsoft extensions to OSF DCE are now available by default:

- Interface definitions for OLE objects. For more information on the files generated for OLE interfaces, see Files Generated for an OLE Interface.
- 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.
- You can omit directional attributes (**in**, **out**).

See Also

General MIDL Command-line Syntax, Pointer-Attribute Type Inheritance, **/app_config**, /**osf**

/ms_union

midl /ms_union

Example

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

Remarks

The **/ms_union** switch controls the NDR alignment of nonencapsulated unions.

The MIDL compiler mirrors the behavior of the OSF-DCE IDL compiler for nonencapsulated 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

General MIDL Command-line Syntax, **IDL**, **ms_union**

/new

midl /new

Examples

```
midl /new filename.idl
midl filename.idl
```

Remarks

This is the default setting for choosing a type library format. Effective with Windows NT 4.0, there is a new version of OLEAUT32.DLL that supports a richer format for 32-bit type libraries. MIDL looks for this DLL on the build machine; if the new version is present, MIDL generates a new-format type library, otherwise it generates an old-format type library. Thus, if the new OLEAUT32.DLL is present, this switch does nothing. If the new OLEAUT32.DLL is not present, specifying this switch in the MIDL command line will generate an error.

See Also

General MIDL Command-line Syntax, **/old**

/no_cpp

midl /no_cpp

Example

```
midl /no_cpp filename.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

General MIDL Command-line Syntax, **/cpp_cmd**, **/cpp_opt**, **/D**, **/I**, **/U**

/no_default_epv

midl /no_default_epv

Example

```
midl /no_default_epv filename.idl
```

Remarks

The **/no_default_epv** switch directs the MIDL compiler not to generate a default entry-point vector (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

General MIDL Command-line Syntax, **IDL**, **/use_epv**, **RpcServerRegisterIf**

/no_def_idir

midl /no_def_idir

Examples

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

General MIDL Command-line Syntax, **/acf**, **/I**

/nologo

midl /nologo

Remarks

Disables the display of the copyright banner.

See Also

General MIDL Command-line Syntax

/no_warn

midl /no_warn

Examples

```
midl /no_warn filename.idl
midl /W0 filename.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

General MIDL Command-line Syntax, **/W**, **/WX**

/o

midl /o *outputfile*

outputfile

Specifies a filename for the MIDL compiler to redirect output (error messages and warnings) to.

See Also

General MIDL Command-line Syntax

/Oi

midl /{Oi | Oic | Oif | Oicf}

/Oi

Specifies the fully-interpreted method for marshalling stub code passed between client and server.

/Oic

Specifies the codeless proxy method of marshaling that provides all the features of **/Oi** and also further reduces the size of the client stub code for object interfaces.

/Oif or **/Oicf**

Specifies the codeless proxy method of marshaling that includes all the features provided by **/Oi** and **/Oic** but uses a new interpreter ("fast format strings") that provides better performance than **/Oi** or **/Oic**.

Please note the restrictions related to supporting platforms, below.

Examples

```
midl /Oi filename.idl
midl /Oic filename.idl
midl /Oif filename.idl
```

Remarks

The MIDL 3.0 compiler provides two methods for marshalling code: fully-interpreted (**/Oi**, **/Oic** and **/Oif**) and mixed-mode (**/Os**). Mixed-mode is the default. Some language features are not supported in some modes. In this case, the compiler automatically switches to the appropriate mode and issues a warning.

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.

The fully-interpreted method marshals data completely offline. This considerably reduces the size of the stub code, but results in decreased performance. Also, with the fully-interpreted method, there is a limit of 16 parameters for each procedure. Any procedure containing more than 16 parameters will automatically be processed in **/Os** mode. Among the interpreted modes, **/Oif** offers the best performance and **/Oi** offers the best backward compatibility.

If your application uses OLE object interfaces and if it will never run on a version of Microsoft Windows NT earlier than 3.51, you can reduce the size of your client stub code for object interfaces by using the **/Oic** option. With this option the MIDL compiler does not generate any client-side stub code. Since object interface calls are through vtable pointers, the compiler generates the proper vtable structures and the application calls the stubs through them.

If your application will be run *only* on Microsoft Windows NT 4.0 or later, you can use the faster **/Oif** option. Specifically, you may want to use the **/Oif** option if your application uses MIDL features that were introduced with MIDL 3.0, such as the **wire_marshal** and **user_marshal** attributes. If your application uses pipes you *must* use the /**Oif** option; if you specify another mode, the MIDL compiler will switch to /**Oif**.

To fine-tune the way your stub code is marshalled, Microsoft RPC provides an ACF **optimize** attribute. This attribute is used as an interface attribute or operation attribute to select the marshalling mode for individual interfaces or for individual operations.

Calling Conventions

Stubs generated by the MIDL compiler in the interpreted method using the **/Oi**, /**Oic**, or **/Oif** switches 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.

Supporting Platforms

/Oi is supported on Windows NT 3.5 or later and Windows 95
/Oic is supported on Windows NT 3.51 or later and Windows 95
/Oif is supported on Windows NT 4.0.

See Also

General MIDL Command-line Syntax, **/Os**, **optimize**

/old

midl /old

Examples

```
midl /old filename.idl
midl /old myoldodl.odl
```

Remarks

The /**old** switch determines the format of MIDL-generated type libraries.

Effective with Windows NT 4.0, there is a new version of OLEAUT32.DLL that supports a richer format for 32-bit type libraries. MIDL looks for this DLL on the build machine; if the new version is present, MIDL generates a new-format type library. Otherwise, it generates an old-format type library. The **/old** switch overrides this default and directs the MIDL compiler to generate old-format type libraries even if the newer version of OLEAUT32.DLL is present.

See Also

General MIDL Command-line Syntax

/oldnames

midl /oldnames

Example

```
midl /oldnames filename.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

General MIDL Command-line Syntax, **IDL**

/Os

midl /Os

Examples

```
midl /Os filename.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, but 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 do not need to 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

General MIDL Command-line Syntax, **/Oi**, **optimize**

/osf

midl /osf

Examples

```
midl /osf filename.idl
midl /osf /app_config filename.idl
```

Remarks

The **/osf** switch forces strict compatibility with OSF DCE. Use this switch if your application requires strict compatibility with OSF DCE for portability reasons.

In /**osf** mode, the Rpcss package is automatically enabled when you use full pointers, the arguments require memory allocation, or when you use the **enable_allocate** attribute. This means that you do not have to supply the **midl_user_allocate** and **midl_user_free** functions in your client and server application.

The following Microsoft-extended features are *not* available when you compile with the **/osf** switch:

- Abstract declarators (unnamed parameters) in the IDL file.
- Interface definitions for OLE objects.
- MIDL-only attributes, such as **wire_marshal**, **user_marshal**, and the typelib (ODL)extensions.
- Using ACF keywords in an IDL file.
- Static callback functions on the client.
- **cpp_quote**(*quoted_string*) and **#pragma midl_echo**.
- **wchar_t** wide-character types, constants, and strings.
- **enum** initialization (sparse enumerators).
- **[out]** -only size specification.
- Mixed sized-pointers and sized arrays.
- Expressions used for size and discriminator specifiers.
- Explicit handle parameters in any position in the argument list. In **/osf** mode, the MIDL compiler looks 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**.
- Pointer-attribute type inheritance. OSF DCE does not allow unattributed pointers. Therefore, in /**osf** mode each IDL file must define attributes for its pointers. If any pointer does not have an explicit attribute, the IDL file must have a **pointer_default** specification to set the pointer type.
- Multiple interfaces in an IDL file.
- Definitions outside of the interface block.
- Type qualifiers such as **far** and **stdcall**.
- Omitting directional attributes.

The following C/C++ language extensions are *not* available when you compile with the **/osf** switch:

- Bit fields in structures and unions.
- Single line comments delimited with two slash characters (//).
- External declarations.
- Procedures with ellipses in the parameter list.

- Type **int**.
- Type **void *** (except with the **context_handle** attribute).
- 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**.

See Also

General MIDL Command-line Syntax, **/app_config**, **/ms_ext**, Rpcss Memory Management Model

/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 filename.idl
midl /out c:  filename.idl
midl /out \mydir\mysubdir\another filename.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

General MIDL Command-line Syntax, **/cstub**, **/header**, **/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 filename.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 nonstandard packing levels, see the **/Zp** help topic.

See Also

General MIDL Command-line Syntax, **/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

General MIDL Command-line Syntax

/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 filename.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 General MIDL Command-line Syntax.

See Also

General MIDL Command-line Syntax, **/header**, **/iid**, **/out**

/rpcss

midl /rpcss

Example

```
midl /rpcss filename.idl
```

Remarks

The **/rpcss** switch, when specified, acts as though the **enable_allocate** attribute was specified on all operations of the interface. In the default 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. In /**osf** mode, the server side stub enables the rpcss allocation package.

See Also

General MIDL Command-line Syntax, **IDL**, **enable_allocate**, /**osf**

/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

General MIDL Command-line Syntax, **/client**, **/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 filename.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

General MIDL Command-line Syntax, **/cstub**, **/header**, **/out**

/syntax_check

midl /syntax_check
midl /Zs

Examples

```
midl /Zs filename.idl
midl /syntax_check filename.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

General MIDL Command-line Syntax, **/Zs**

/<system>

midl /<system> where **<system>** is one of: **/win16 /win32 /mac /mips /alpha /ppc /ppc32**

Example

```
midl /alpha filename.idl
```

Remarks

Directs the MIDL compiler to generate a type library for the specified system.The default is the current operating system. The /**mac** switch directs MIDL to generate a type library for a 680*x*0-based Apple Macintosh system.

The **/<system>** switch is functionally the same as the MIDL /**env** option and is recognized by the MIDL compiler solely for backward compatibility with MKTYPLIB. If you are generating a new makefile, use the /**env** switch.

See Also

General MIDL Command-line Syntax, **/env**

/tlb

midl /tlb *filename*

filename

Specifies *filename* as the name of the output type library (TLB) file. By default, if the **/tlb** switch is not used, the TLB file has the same name as the IDL file, with the extension .TLB.

Example

```
midl /tlb newname.tlb
```

See Also

General MIDL Command-line Syntax

/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 filename.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

General MIDL Command-line Syntax, **/cpp_cmd**, **/cpp_opt**, **/D**, **/I**, **/no_cpp**

/use_epv

midl /use_epv

Example

```
midl /use_epv filename.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

General MIDL Command-line Syntax, **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 filename.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

General MIDL Command-line Syntax, **/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 filename.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

General MIDL Command-line Syntax

/WX

midl /WX

Examples

```
midl /WX filename.idl
midl /W3 /WX filename.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

General MIDL Command-line Syntax, **/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 filename.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 run time, 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

General MIDL Command-line Syntax, **/pack**

/Zs

midl /Zs
midl /syntax_check

Examples

```
midl /Zs filename.idl
midl /syntax_check filename.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

General MIDL Command-line Syntax, **/syntax_check**

MIDL Language Reference

This section provides a reference entry for each keyword in the Microsoft® Interface Definition Language (MIDL). Reference entries are also included for important language productions and concepts.

The reference entries are arranged in alphabetical order and each entry includes syntax, examples, descriptions, and cross-references. 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 do not need to 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.

As with the IDL file, the ACF consists of a header portion and a body portion and, except in /**osf** mode, can contain multiple interfaces.

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 the options with commas.

type-attribute-list

Specifies other optional ACF type attributes. When you specify more than one type attribute, separate the 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** lets you customize memory allocation and deallocation for a type defined in the IDL file. These are the valid options:

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	Does not 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 or a memory leak will occur.

See Also

ACF, **midl_user_allocate**, **midl_user_free**, **typedef**

appobject

[uuid(. . .)**, appobject** [, *coclass-attribute-list*]**]**
coclass *classname* { [*coclass definition*]}

coclass-attribute-list

Specifies zero or more attributes that apply to the **coclass** statement. Allowable **coclass** attributes are **helpstring, helpcontext, licensed, version, control,** and **hidden.**

classname

Specifies the name by which the component object is known in the type library.

Example

```
[uuid(. . .),helpstring("Hello Class"),appobject] coclass Hello
    {[default]    interface IHello : IUnknown;
                  interface IDispatch;
    }
```

Remarks

The **appobject** attribute identifies the **coclass** as an application object, which is associated with a full EXE application, and indicates that the functions and properties of the **coclass** are globally available in this type library.

Flags

TYPEFLAG_FAPPOBJECT

See Also

ODL File Syntax, ODL File Example, Generating a Type Library With MIDL, **coclass**, TYPEFLAGS

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. 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 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 zero. 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 in this way: 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, and 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 determined at run time. 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 dependent on run time.
- 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.

Note You cannot use automatic binding (either with the **auto_handle** attribute, or by default) if you are processing data through pipes.

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: **int**, **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 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
int	**signed**	32-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. The following restrictions apply to data types in interfaces:

- On 16-bit platforms, 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**. Therefore, this feature is not available when you use the MIDL compiler /**osf** switch.

See Also

byte, **char**, **handle_t**, **long**, **/osf**, **short**, **small**, **wchar_t**

bindable

[*interface-attribute-list*] **interface | dispinterface** *interface-name*
{
[bindable[*, attribute-list*]**]** *returntype function-name(params)*
}

attribute-list

Specifies zero or more attributes that apply to the function prototype for a property or a method in an **interface** or **dispinterface**. The following attributes are accepted: **helpstring**, **helpcontext**, **string**, **defaultbind**, **displaybind**, **immediatebind**, **propget**, **propput**, **propputref**, and **vararg**. If **vararg** is specified, the last parameter must be a safe array of VARIANT type. Separate multiple attributes with commas.

Example

```
[uuid(. . .)]dispinterface MyObject
{
properties:
methods:
 [id(1), propget, bindable, defaultbind, displaybind]
        long x();
 [id(1), propput, bindable, defaultbind, displaybind]
        void x(long rhs);
}
```

Remarks

The **bindable** attribute indicates that the property supports data binding. This allows the client to be notified whenever a property has changed value. (If you want the client to be notified of *impending* changes to a property, use the **requestedit** attribute.)

Because the **bindable** attribute refers to the property as a whole, it must be specified wherever the property is defined. Therefore, you need to specify the attribute on both the property-accessing function and the property-setting function.

Flags

FUNCFLAG_FBINDABLE, VARFLAG_FBINDABLE

See Also

defaultbind, **displaybind**, ODL File Syntax, ODL File Example, Generating a Type Library With MIDL, TYPEFLAGS, **dispinterface**

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 broadcasted to all the servers on the network. The client receives output from the first reply to return successfully, while 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 is not completed. 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 a **char** type can.

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. Therefore, this attribute is not available when you use the MIDL compiler switch /**osf**.

The **byte_count** attribute is a parameter attribute that associates a size, in bytes, with the memory area indicated by the pointer.

Memory referenced by the pointer parameter is contiguous and is not allocated or freed by the client stubs. This feature of the **byte_count** attribute lets you 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 lets you 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, while 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, the alignment padding 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 nonremotable function to be mapped to a remote function. This is particularly helpful in interfaces that have numerous nonremotable 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 nonremotable interface to be remoted transparently. The **call_as** attribute cannot be used with /**osf** 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, these are the prototypes for the bond routines.

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 nonobject interfaces, these are the prototypes for the bond routines.

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 and lets the server 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.

Only the connection-oriented and local protocol sequences support the callback attribute. If an RPC interface uses a connectionless (datagram) protocol sequence, calls to procedures with the callback attribute will fail.

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, **/osf**

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. Therefore, this feature is not available when you use the MIDL compiler /**osf** switch.

See Also

base_types, **byte**, **/char**, **/osf**, **signed**, **string**, **wchar_t**

coclass

[*attribute-list***]coclass** *classname* {[*attributes2*] [**interface** | **dispinterface**] *interfacename* **{**. . . **};**

attribute-list

The **uuid** attribute is required on a **coclass**. This is the same **uuid** that is registered as a CLSID in the system registration database. The **helpstring**, **helpcontext**, **licensed**, **version**, **control**, **hidden**, and **appobject** attributes are accepted, but not required, before a **coclass** definition.

classname

Name by which the common object is known in the type library.

attributes2

Optional attributes for the interface or dispinterface. The **source**, **default**, and **restricted** attributes are accepted on an interface or dispinterface within a coclass.

interfacename

Either an interface declared with the **interface** keyword, or a dispinterface declared with the **dispinterface** keyword.

Examples

```
[uuid(. . .), version(1.0), helpstring("A class"), helpcontext(2481),
appobject] coclass myapp
{
    [source] interface IMydocfuncs : IUnknown;
    dispinterface DMydocfuncs;
};

[uuid(. . .)]
coclass foo
{
    [restricted] interface bar;
    interface baz;
}
```

Remarks

The **coclass** statement provides a listing of the supported interfaces for a component object.

The Microsoft® Component Object Model defines a class as an implementation that allows **QueryInterface** between a set of interfaces.

See Also

ODL File Syntax, ODL File Example, Generating a Type Library With MIDL, TYPEFLAGS

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 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, one, 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 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, but the one you choose can appear only 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 zero, one, 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 zero, one, 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 is completed 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. Therefore this feature is not available when you use the MIDL compiler /**osf** switch.

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**, **/osf**

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.

When you use the MIDL 3.0 compiler in default 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 should only be manipulated by the server application. DCE IDL compilers restrict context handles to pointers of type **void ***. Therefore this feature is not available when you use the MIDL compiler /**osf** switch.

As with other handle types, the context handle is opaque to the client application and any data associated with it 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.

See Also

auto_handle, handle, handles, Context Handles, Server Context Rundown Routine, Client Context Reset

control

[uuid, **control** [, attribute-list]**] library** | **coclass** *lib-or-coclassname*
{ *definitions* **}**

attribute-list

Specifies zero or more attributes that apply to the **library** or **coclass** statement. Separate multiple attributes with commas.

Example

```
[uuid(. . .),helpstring("Hello 2.1 OLE Control Library"),
   control,version(2.1)]
library Hello
{ /* library definitions */}
```

Remarks

The **control** attribute identifies a **coclass** or **library** as an OLE control, from which a container site will derive additional type libraries or component object classes. This attribute allows you to mark type libraries that describe controls so they will not be displayed in type browsers intended for nonvisual objects.

Flags

TYPEFLAG_FCONTROL, LIBFLAG_FCONTROL

See Also

ODL File Syntax, ODL File Example, Generating a Type Library With MIDL, TYPEFLAGS, **coclass**, **library**

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 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 into 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 Encoding Services and **encode**.

See Also

encode

default

[uuid, *attribute-list***] coclass** *coclass-name*
{
[default [*, optional-interface-attribute]***] interface | dispinterface** *interface-name*
}

attribute-list

Specifies additional **coclass** attributes. Separate multiple attributes with commas.

optional-interface-attribute

The **source** attribute, which specifies that an interface or dispinterface is outgoing, is the only other attribute that can be used here.

Example

```
[ uuid(. . .), helpstring("Hello Class"),appobject]  coclass Hello
   {[default] interface IHello;
              interface IDispatch;
   };
```

Remarks

The **default** attribute Indicates that the interface or dispinterface, defined within a **coclass**, represents the default programmability interface. This attribute is intended for use by macro languages.

A **coclass** may have at most two **default** members. One represents the outgoing (source) interface or dispinterface, and the other represents the incoming (sink) interface or dispinterface. If the **default** attribute is not specified for any member of the **coclass** or **cotype**, the first outgoing and incoming members that do not have the **restricted** attribute are treated as the defaults.

Flags

IMPLTYPEFLAG_FDEFAULT

See Also

TYPEFLAGS, ODL File Syntax, ODL File Example, Generating a Type Library With MIDL

defaultbind

[*interface-attribute-list*] **interface | dispinterface** *interface-name*
{
[bindable, **defaultbind**[*, attribute-list*]**]** *returntype function-name(params)*
}

Example

```
[uuid(. . .)] interface MyObject : IUnknown
{
    properties:
    methods:
        [id(1), propget, bindable, defaultbind, displaybind]
        long Size(void);

        [id(1), propput, bindable, defaultbind, displaybind]
        void Size([in]long lSize);
}
```

Remarks

The **defaultbind** attribute indicates the single, bindable property that best represents the object. Properties that have the **defaultbind** attribute must also have the **bindable** attribute. Only one property in an interface or dispinterface can have the **defaultbind** attribute.

This attribute is used by containers that have a user model involving binding to an object rather than binding to a property of an object. An object can support data binding but not have this attribute.

Flags

FUNCFLAG_FDEFAULTBIND, VARFLAG_FDEFAULTBIND

See Also

TYPEFLAGS, ODL File Syntax, ODL File Example, Generating a Type Library With MIDL

defaultvalue

interface *interface-name*
{

return-type func-name(*mandatory-param-list,*
[[*attribute-list*,**] defaultvalue(value)]** *param-type param-name*[, *optional-param-list*]*);*

}

Examples

```
interface IFoo : IUnknown
{
    HRESULT Ex1([defaultvalue(44)] LONG       i);
    HRESULT Ex2([defaultvalue(44)] SHORT      i);
    HRESULT Ex3([defaultvalue("Hello")] BSTR  i);
...
}
interface QueryDef : IUnknown
{
    HRESULT    OpenRecordset(  [in, defaultvalue(DBOPENTABLE)]
    LONG Type,
    [out,retval] Recordset **pprst);
}
 //  Type is now known to be a LONG type (good for browser in VBA and
//  good for a C/C++ programmer) and has a default value of
//  DBOPENTABLE
```

Remarks

The **defaultvalue** attribute allows specification of a default value for a typed optional parameter. The value can be any constant, or an expression that resolves to a constant, that can be represented by a VARIANT. Specifically, you cannot apply the **defaultvalue** attribute to a parameter that is a structure, an array, or a SAFEARRAY.type.

The MIDL compiler accepts the following parameter ordering (from left-to-right):

1. Required parameters (parameters that do not have the **defaultvalue** or **optional** attributes),
2. optional parameters with or without the **defaultvalue** attribute,
3. parameters with the **optional** attribute and without the **defaultvalue** attribute,
4. **lcid** parameter, if any,
5. **retval** parameter

See Also

interface, **dispinterface**, TYPEFLAGS, ODL File Syntax, ODL File Example, Generating a Type Library With MIDL

dispinterface

[*attributes***]dispinterface** *intfname* **{ properties:***proplist* **methods:** *methlist***};**

[*attributes***]dispinterface** *intfname* **{ interface** *interfacename***};**

attributes

Specifies attributes that apply to the entire **dispinterface**. The following attributes are accepted: **helpstring**, **helpcontext**, **helpfile**, **hidden**, **nonextensible**, **oleautomation**, **restricted**, **uuid**, **version**.

intfname

The name by which the **dispinterface** is known in the type library. This name must be unique within the type library.

interfacename

(Syntax 2) The name of the interface to declare as an **IDispatch** interface.

proplist

(Syntax 1) An optional list of properties supported by the object, declared in the form of variables. This is the short form for declaring the property functions in the methods list. See the comments section for details.

methlist

(Syntax 1) A list comprising a function prototype for each method and property in the **dispinterface**. Any number of function definitions can appear in *methlist*. A function in *methlist* has the following form:

[*attributes***]** *returntype methname***(***params***);**

The following attributes are accepted on a method in a **dispinterface**: **helpstring**, **helpcontext**, **string**, **bindable**, **defaultbind**, **displaybind**, **propget**, **propput**, **propputref**, and **vararg**. If **vararg** is specified, the last parameter must be a safe array of VARIANT type.

The parameter list is a comma-delimited list, each element of which has the following form:

[*attributes***]** *type paramname*

The *type* can be any declared or built-in type, or a pointer to any type. Attributes on parameters are:

in, **out**, **optional**, **string**

The MIDL compiler accepts the following parameter ordering (from left-to-right):

1. Required parameters (parameters that do not have the **defaultvalue** or **optional** attributes),
2. optional parameters with or without the **defaultvalue** attribute,
3. parameters with the **optional** attribute and without the **defaultvalue** attribute,
4. **lcid** parameter, if any,
5. **retval** parameter

Examples

```
[ uuid(. . .), version(1.0), helpstring("Useful help string."),
helpcontext(2480)]
dispinterface MyDispatchObject {
    properties:
        [id(1)] int x;    //An integer property named x
```

```
        [id(2)] BSTR y;   //A string property named y
    methods:
        [id(3)] void show();    //No arguments, no result
        [id(11)] int computeit(int inarg, double *outarg);
};

[uuid(. . .)]
dispinterface MyObject
{
    properties:
    methods:
        [id(1), propget, bindable, defaultbind, displaybind]
        long x();

        [id(1), propput, bindable, defaultbind, displaybind]
        void x(long rhs);
}
```

Remarks

The **dispinterface** statement defines a set of properties and methods on which you can call **IDispatch::Invoke**. A dispinterface may be defined by explicitly listing the set of supported methods and properties (Syntax 1), or by listing a single interface (Syntax 2).

Method functions are specified exactly as described in the reference page for **module** except that the **entry** attribute is not allowed. Note that STDOLE32.TLB (STDOLE.TLB on 16-bit systems) must be imported, because a **dispinterface** inherits from **IDispatch**.

You can declare properties in either the properties or methods lists. Declaring properties in the properties list does not indicate the type of access the property supports (that is, get, put, or putref). Specify the **readonly** attribute for properties that don't support put or putref. If you declare the property functions in the methods list, functions for one property all have the same ID.

Using the first syntax, the **properties:** and **methods:** tags are required. The **id** attribute is also required on each member. For example:

```
properties:
    [id(0)] int Value;    // Default property.
methods:
    [id(1)] void Show();
```

Unlike **interface** members, **dispinterface** members cannot use the **retval** attribute to return a value in addition to an HRESULT error code. The **lcid** attribute is likewise invalid for dispinterfaces, because **IDispatch::Invoke** passes an LCID. However, it is possible to redeclare an interface that uses these attributes.

Using the second syntax, interfaces that support **IDispatch** and are declared earlier in an ODL script can be redeclared as **IDispatch** interfaces as follows:

```
dispinterface helloPro {
    interface hello;
};
```

The preceding example declares all the members of hello and all the members that hello inherits as supporting **IDispatch**. In this case, if hello were declared earlier with **lcid** and **retval** members that returned HRESULTs, MkTypLib would remove each **lcid** parameter and HRESULT return type, and instead mark the return type as that of the **retval** parameter.

The properties and methods of a dispinterface are not part of the VTBL of the dispinterface. Consequently, **CreateStdDispatch** and **DispInvoke** cannot be used to implement **IDispatch::Invoke**. The dispinterface is used when an application needs to expose existing non-VTBL functions through OLE Automation. These applications can implement **IDispatch::Invoke** by examining the *dispidMember* parameter and directly calling the corresponding function.

See Also

interface, TYPEFLAGS, ODL File Syntax, ODL File Example, Generating a Type Library With MIDL

displaybind

[*interface-attribute-list*] **interface | dispinterface** *interface-name*
{
[bindable, **displaybind**[, *attribute-list*]**]** *returntype function-name(params)*
}

Example

```
[uuid(. . .)] interface MyObject : IUnknown
{
    properties:
    methods:
        [id(1), propget, bindable, defaultbind, displaybind]
        long Size(void);

        [id(1), propput, bindable, defaultbind, displaybind]
        void Size([in]long lSize);
}
```

Remarks

The **displaybind** attribute indicates a property that should be displayed to the user as bindable. Properties that have the **displaybind** attribute must also have the **bindable** attribute. An object can support data binding but not have this attribute.

Flags

FUNCFLAG_FDISPLAYBIND, VARFLAG_FDISPLAYBIND

See Also

TYPEFLAGS, ODL File Syntax, ODL File Example, Generating a Type Library With MIDL

dllname(*str*)

[uuid, dllname("*filename***")**[, *optional-attribute-list*]**]**
module *modulename* **{**
elementlist
};

Example

```
[uuid(. . .),helpstring("A meaningful comment"),      dllname("HANDY.DLL")]
module HandyStuff{
    . . .
};
```

Remarks

The **dllname** attribute defines the name of the DLL that contains the entry points for a module. This attribute is required on a module.

See Also

module,**entry**, ODL File Syntax, ODL File Example, Generating a Type Library With MIDL

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). This 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**

dual

[uuid, **oleautomation**, **dual** [, *optional-attribute-list*]**]**
interface *interfacename*
{. . .};

Example

```
[uuid(. . .), oleautomation, dual]
    interface IHello : Idispatch
    {
     //diverse and sundry properties and methods defined here
    };
```

Remarks

The **dual** attribute identifies an interface that exposes properties and methods through **IDispatch** and directly through the VTBL. The interface must be compatible with OLE Automation and be derived from **IDispatch**. The attribute is not allowed on dispinterfaces.

The **dual** attribute creates an interface that is both a **Dispatch** interface and a Component Object Model (COM) interface. The first seven entries of the VTBL for a dual interface are the seven members of **IDispatch**, and the remaining entries are OLE COM entries for direct access to members of the dual interface. All the parameters and return types specified for members of a dual interface must be OLE Automation-compatible types.

All members of a dual interface must pass an HRESULT as the function return value. Members, such as property accessor functions, that need to return other values, such as should specify the last parameter as [**out,** **retval**], indicating an output parameter that returns the value of the function. In addition, members that need to support multiple locales should pass an **lcid** parameter.

A dual interface provides for both the speed of direct VTBL binding and the flexibility of **IDispatch** binding. For this reason, dual interfaces are recommended whenever possible.

Note If your application accesses object data by casting the **this** pointer within the interface call, you should check the VTBL pointers in the object against your own VTBL pointers to ensure that you are connected to the appropriate proxy.

Specifying **dual** on an interface implies that the interface is compatible with OLE Automation, and therefore causes both the TYPEFLAG_FDUAL and TYPEFLAG_FOLEAUTOMATION flags to be set.

Flags

TYPEFLAG_FDUAL, TYPEFLAG_FOLEAUTOMATION

See Also

interface, **oleautomation**, ODL File Syntax, ODL File Example, Generating a Type Library With MIDL

enable_allocate

Remarks

The keyword **enable_allocate** specifies that the server stub code should enable the stub memory management environment. In default 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**. In **osf** mode (when you compile using the /**osf** switch), 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.

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 sensitive only when the **enable_allocate** attribute is used. Typically, the client side stub operates in the disabled environment. In **osf** mode (when you compile using the /**osf** switch), the client is always sensitive to the **Rpcss** memory management environment and, therefore, the **enable_allocate** attribute will not affect the client stubs.

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 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. 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 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 tnonencapsulated 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. Except when you use the MIDL compiler switch **/osf**, 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 character string that represents a valid combination of an RPC protocol (such as "ncacn"), a transport protocol (such as "tcp"), and a network protocol (such as "ip"). Microsoft RPC supports the following protocol sequences:

Protocol sequence	Description	Supporting Platforms
ncacn_nb_tcp	Connection-oriented NetBIOS over TCP	Client and server: Windows NT Client only: MS-DOS, Windows 3.*x*™
ncacn_nb_ipx	Connection-oriented NetBIOS over IPX	Client and server: Windows NT Client : MS-DOS, Windows 3.*x*
ncacn_nb_nb	Connection-oriented NetBEUI	Client and server: Windows NT, Windows® 95 Client : MS-DOS, Windows 3.*x*
ncacn_ip_tcp	Connection-oriented TCP/IP	Client and server: Windows 95 and Windows NT Client: MS-DOS,Windows 3.*x,* and Apple® Macintosh®
ncacn_np	Connection-oriented named pipes	Client and server: Windows NT Client: MS-DOS, Windows 3.*x*, Windows 95
ncacn_spx	Connection-oriented SPX	Client and server: Windows NT, Windows 95 Client: MS-DOS, Windows 3.*x*
ncacn_dnet_nsp	Connection-oriented DECnet	Client only: MS-DOS, Windows 3.*x*
ncacn_at_dsp	Connection-oriented AppleTalk DSP	Server: Windows NT Client: Apple Macintosh
ncacn_vns_spp	Connection-oriented Vines SPP	Client and server: Windows NT Client: MS-DOS, Windows 3.*x*
ncadg_ip_udp	Datagram (connectionless) UDP/IP	Client and server: Windows NT Client: MS-DOS, Windows 3.*x*
ncadg_ipx	Datagram (connectionless) IPX	Client and server: Windows NT Client: MS-DOS, Windows 3.*x*
ncalrpc	Local procedure call	Client and server: Windows NT and Windows 95

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**.

Make sure that you correctly quote backslash characters in endpoints. This error 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

entry

[uuid, entry(*entry-id***)**[, *optional-attribute-list*]**]**
  module *modulename* **{**
    elementlist
  };

Example

```
[dllname("MyAppsFirst.dll")] module MyModule
    {
    [entry(20), bindable, requestedit, propputref, defaultbind]
        void Func1([in]IUnknown * Param1, [out] MyType * Param2);
    [entry("TwentyOne"), hidden, vararg]
         SAFEARRAY (int) Func2 ([in, out] SAFEARRAY (variant) *varP) ;
    [entry(22)] Float Func3 ([in] lpstr pName, [in] double dLevel,
                          [out] short * sByte) ;
    } ;
```

Remarks

The **entry** attribute specifies an exported function or constant in a module by identifying the entry point in the DLL. If *entryid* is a string, this is a named entry point. If *entryid* is a number, the entry point is defined by an ordinal. This attribute provides a way to obtain the address of a function in a module.

See Also

dllname, **module**, ODL File Syntax, ODL File Example, Generating a Type Library With MIDL

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**.

In the MIDL compiler's default mode, you can assign integer values to enumerators. (This feature is not available when you compile with the **/osf** switch.) As with C-language enumerators, enumerator names must be unique, but the enumerator values need not be.

When assignment operators are not provided, identifiers are mapped to consecutive integers from left to right, starting with zero. When assignment operators are provided, assigned values start from the most recently assigned value.

The maximum number of identifiers is 65,535.

Objects of type **enum** are **int** types, and their size is system-dependent. By default, objects of **enum** types are treated as 16-bit objects of type **unsigned short** when transmitted over a network. Values outside the range 0 - 32,767 cause the run-time exception RPC_X_ENUM_VALUE_OUT_OF_RANGE. To transmit objects as 32-bit entities, apply the **v1_enum** attribute to the **enum** typedef.

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 this, a prototype with an argument containing a handle type is supplied to the IDL file. Note that in default mode, an argument that does not appear first can also be used as a handle directing the call. Therefore, while the **explicit_handle** attribute is a way of giving the IDL prototype a primitive **explicit_handle** attribute, it does not necessarily require a change to the IDL file. In /**osf** mode only the first argument can be used as an explicit handle type.

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. Also 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. Parameter attributes are enclosed in square brackets. Separate multiple parameter attributes with commas. 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 be applied only 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 is successfully completed, 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, while 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. Each expression evaluates to an integer that represents the array index of the first array element to be transmitted. 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 increment and decrement operators. Separate multiple expressions with commas.

Example

```
void Proc1(
    [in] short First,
    [first_is(First)] Arr[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 zero 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:

```
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.

Using a constant expression with the **first_is** attribute is an inappropriate use of the attribute. It is legal, but inefficient, and will result in slower marshalling code.

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. If unsuccessful, the routine should return 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.

helpcontext

[uuid, helpcontext(*helpcontext-value*)[,*attribute-list*]**]** *statement* | *directive statement-name* **{***definitions***}**

Example

```
[uuid(. . .),helpcontext(7035943), helpstring("Hello Class"),appobject]
coclass Hello
    {[default, helpcontext(3914972)]    interface IHello : IUnknown;
                        interface IDispatch;
    }
```

Remarks

The **helpcontext** attribute specifies a context ID that lets the user view information about this element in the Help file. This attribute can be applied to the following elements: **library**, **importlib**, **interface**, **dispinterface**, **module**, **typedef**, **method**, **property**, **coclass**.

The *helpcontext-value* is a 32-bit context identifier within the Help file that can be retrieved with the **GetDocumentation** functions in the **ITypeLib** and **ITypeInfo** interfaces.

See Also

ODL File Syntax, ODL File Example, Generating a Type Library With MIDL

helpfile

[uuid, helpfile("*filename"*) [, *optional-attribute-list*]**] library {***library statements***};**

Example

```
[uuid(. . .),helpfile("filename.hlp"),lcid(0x0409), version(2.0)]
library Hello
{ . . .};
```

Remarks

The **helpfile** attribute sets the name of the Help file for a type library. All types in a library share the same Help file.

Use the **GetDocumentation** functions in the **ITypeLib** and **ITypeInfo** interfaces to retrieve the filename.

See Also

library, ODL File Syntax, ODL File Example, Generating a Type Library With MIDL

helpstring

[helpstring("*string"*)[, *optional-attribute-list*]**]** *odl-statement*

Example

```
  [uuid(. . .),helpstring("Lines 1.0 Type Library"),version(1.0)]
library Lines
{
     [uuid(. . .), helpstring("Line object."),oleautomation,dual]
    interface ILine : IDispatch
    {
      [propget, helpstring("Returns and sets RGB color.")]
      HRESULT Color([out, retval] long* retval);
      [propput, helpstring("Returns and sets RGB color.")]
      HRESULT Color([in] long rgb);
     }
};
```

Remarks

The **helpstring** attribute specifies a character string that is used to describe the element to which it applies. You can apply the **helpstring** attribute to library, importlib, interface, dispinterface, module, or coclass statements, typedefs, properties, and methods.

Use the **GetDocumentation** functions in the **ITypeLib** and **ITypeInfo** interfaces to retrieve the help string.

See Also

library, **importlib**, **interface**, **dispinterface**, **module**, **coclass**, **typedef**, ODL File Syntax, ODL File Example, Generating a Type Library With MIDL

hidden

[*other-attributes*, **hidden]** *statement-or-function-type statement-or-function-name*

Examples

```
[hidden, vararg]
  SAFEARRAY (int) SecretFunc ([in, out] SAFEARRAY (variant) *varP) ;
[uuid(. . .), hidden, version (3.0)] library HiddenLib {. . .};
```

Remarks

The **hidden** attribute indicates that the item exists but should not be displayed in a user-oriented browser. This attribute allows you to remove members from your interface (by shielding them from further use) while maintaining compatibility with existing code. You can use the **hidden** attribute on properties, methods, and the **coclass**, **dispinterface**, **interface**, and **library** statements.

When specified for a library, the **hidden** attribute prevents the entire library from being displayed. This usage is intended for use with controls. Hosts need to create a new type library that wraps the control with extended properties.

Flags

VARFLAG_FHIDDEN, FUNCFLAG_FHIDDEN, TYPEFLAG_FHIDDEN

See Also

TYPEFLAGS, **dispinterface**, **coclass**, ODL File Syntax, ODL File Example, Generating a Type Library With MIDL

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

id

[id(*id-num*) [,*optional-attribute-list*]**]** *return-type function-name*

Example

```
Interface IKnown : IUnknown
{
properties:
[id(90), propget, helpstring("A meaningful comment."]
        long Func1(void);
. . .
}
```

Remarks

The **id** attribute specifies a DISPID for a member function (either a property or a method, in an interface or dispinterface). You use the **id** attribute when you want to assign a standard DISPID (like DISPID_VALUE, DISPID_NEWENUM etc.) to a method or property, or when you implement your own IDispatch::Invoke instead of delegating to DispInvoke/ITypeInfo::Invoke.

If you do not use the **id** attribute, the MIDL compiler will assign a DISPID for you.

The *id-num* is a 32-bit integral value in the following format:

Bits	Value
0-15	Offset. Any value is permissible.
16-21	The nesting level of this typeinfo in the inheritance hierarchy. For example: `interface mydisp : IDispatch` The nesting level of **IUnknown** is 0, **IDispatch** is 1, and mydisp is 2.
22-25	Reserved; must be zero
26-28	DISPID value.
29	True if this is the member ID for a FuncDesc; otherwise False.
30-31	Must be 01.

Negative IDs are reserved for use by OLE Automation.

See Also

interface, **dispinterface**, ODL File Syntax, ODL File Example, Generating a Type Library With MIDL

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 and is 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 also contains zero or more import lists, constant declarations, general declarations, and function declarators.

In Microsoft RPC, an IDL file can contain multiple interfaces and these interfaces can be forward declared (within the IDL file that defines them). For example:

```
interface ITwo; //forward declaration
interface IOne {
...uses ITwo...
}
interface ITwo {
...uses IOne...
}
```

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 through 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 only need to be unique within an interface. Note that multiple interfaces are not permitted when you use the /**osf** switch.

The syntax for declarative constructs in the IDL file is similar to that for C. MIDL supports all Microsoft C/C++ 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).

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. Except in /**osf** 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 Language Reference**, **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 C-language expression. 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 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 in default mode, that is not using the /**osf** switch.

See Also

object, **uuid**

immediatebind

[*interface-attribute-list*] **interface | dispinterface** *interface-name*
{
[bindable, immediatebind[*, optional-attribute-list*]**]** *returntype function-name(params)*
}

Example

```
[uuid(. . .)] interface MyObject : IUnknown
{
    properties:
    methods:
        [id(1), propget, bindable, immediatebind]
        long Size(void);

        [id(1), propput, bindable, immediatebind]
        void Size([in]long lSize);
}
```

Remarks

The **immediatebind** attribute indicates that the database will be notified immediately of all changes to a property of a data-bound object.

This attribute allows controls to differentiate between properties that need to notify the database of every change, and those that do not. For example, every change to a checkbox control should be sent to the underlying database immediately, even if the control has not lost the focus. However, for a listbox control, a change occurs whenever a different selection is highlighted. Notifying the database of a change before the control loses focus would be inefficient and unnecessary. The **immediatebind** attribute allows individual properties on a form to specify, by setting the ImmediateBind bit, that changes should be reported immediately.

Properties that have the **immediatebind** attribute must also have the **bindable** attribute.

Flags

FUNCFLAG_FIMMEDIATEBIND, VARFLAG_FIMMEDIATEBIND

See Also

bindable, TYPEFLAGS, **interface**, **dispinterface**, ODL File Syntax, ODL File Example, Generating a Type Library With MIDL

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 header or IDL file to import.

Examples

```
import "recycled.idl";
import "system.h";
import "unknwn.idl";
import "part1.idl", "part2.idl", "part3.idl";
```

Remarks

The import directive specifies another .IDL file containing definitions you wish to reference from the main .IDL file. The imported file is processed separately from the main IDL file and the CPP preprocessor is invoked independently on this file. All resultant type, constant, and interface definitions are available to the main .IDL file. This implies that CPP directives like #define do not carry over from an imported IDL file to the main IDL file, and vice versa

IDL statements, such as const declarations, typedefs, and interfaces become available to the importing file.

Similar to the C-language preprocessor macro **#include**, the **import** directive directs the compiler to include data types defined in the imported IDL files. Unlike the **#include** directive, the **import** directive ignores procedure prototypes, since no stubs are generated for anything in the imported file.

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 file name. Separate multiple filenames with commas. You must enclose the file name within quotation marks and end the import statement with a semicolon (;). 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 no effect.

The behavior of the **import** directive is independent of the MIDL compiler mode switches **/osf**, and **/app_config**. However, pointer attribute decoration across imports may depend on the compiler mode (/**osv** vs. the default, /**ms_ext**). For details see Pointer-Attribute Type Inheritance.

See Also

IDL, **Importing System Header Files**, **Importing Other IDL Files**, **/ms_ext**, **/osf**

importlib

library (*library-name)***{ importlib**(*file-to-import*) **}**

file-to-import

The name and location of the imported file at MIDL compile-time.

Example

```
library BrowseHelper
{
    importlib("stdole.tlb");
    importlib("mydisp.tlb");
//remainder of library definition
};
```

Remarks

The **importlib** directive makes types that have already been compiled into another type library available to the library being created. All **importlib** directives must precede the other type descriptions in the library.

Note Because the **importlib** directive makes any type defined in the imported library accessible from within the library being compiled, ambiguity is resolved as follows. If the imported libraries contain duplicate references, MIDL will use the last reference that it finds. This is in contrast to MKTYPLIB, which uses the first reference that it finds.

The imported type library should be distributed with the library being compiled.

See Also

library, ODL File Syntax, ODL File Example, Generating a Type Library With MIDL

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, 16-bit Windows, and 32-bit Windows 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. On 16-bit platforms, 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;
int j = i;
typedef struct {
    small int          i1;
    short              i2;
    unsigned long int i3;
} INTSIZETYPE;

void MyFunc([in] long int lCount);
```

Remarks

On 32-bit platforms, the keyword **int** specifies a 32-bit signed integer. On 16-bit platforms, 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**.

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. Therefore, this feature is not available when you use the MIDL compiler /**osf** switch.

See Also

base_types, **long**, **/osf**, **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. Each expression evaluates to an integer that represents the array index of the last array element to be transmitted. 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 increment and decrement operators. 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 cannot be used as a field attribute at the same time as the **length_is** attribute or the **string** attribute.

Using a constant expression with the **last_is** attribute is an inappropriate use of the attribute. It is legal, but inefficient, and will result in slower marshalling code.

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
```

See Also

field_attributes, **first_is**, **IDL**, **length_is**, **max_is**, **size_is**

lcid

[uuid, lcid(*numid*)[, *optional-attribute-list*]**] library { }**

Examples

```
[  uuid(. . .),lcid(0x09),version(1.0)]
library MyLibrary
{. . .};

interface IMyFace : IDispatch
{
[propget] HRESULT MyFunc([in, lcid] long LocaleID,
                         [out, retval] BSTR * retval);
. . .
}
```

Remarks

The **lcid** attribute identifies the locale for a type library or for a function argument.

The *numid* is a 32-bit locale ID as used in Win32 National Language Support. The locale ID is typically entered in hexadecimal.

The MIDL compiler accepts the following parameter ordering (from left-to-right):

1. Required parameters (parameters that do not have the **defaultvalue** or **optional** attributes),
2. optional parameters with or without the **defaultvalue** attribute,
3. parameters with the **optional** attribute and without the **defaultvalue** attribute,
4. **lcid** parameter, if any,
5. **retval** parameter

See Also

library, ODL File Syntax, ODL File Example, Generating a Type Library With MIDL

length_is

[length_is(*limited-expression-list* **)]**

limited-expression-list

Specifies one or more C-language expressions. Each expression evaluates to an integer that represents the number of array elements to be transmitted. 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 increment and decrement operators. 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.

Using a constant expression with the **length_is** attribute is an inappropriate use of the attribute. It is legal, but inefficient, and will result in slower marshalling code.

See Also

field_attributes, **first_is**, **IDL**, **last_is**, **max_is**, **min_is**, **size_is**

library

[uuid [, *optional-attribute-list*]**] library** *libname* **{***definitions***};**

optional-attribute-list

Specifies additional attributes that apply to the entire **library** statement. Allowable attributes include **control, helpcontext, helpfile, helpstring**, **hidden**, **lcid**, **restricted**, and **version**.

libname

The name by which the type library is known.

definitions

Descriptions of any imported libraries, data types, modules, interfaces, dispinterfaces, and coclasses relevant to the object being exposed.

Example

```
[uuid(. . .), helpstring("Hello 2.0 Type Library"),
    lcid(0x0409), version(2.0)]
library Hello
{. . .};
```

Remarks

The library statement contains all the information that the MIDL compiler uses to generate a type library. In addition to elements defined inside of the library block, statements inside the library block can use elements that are declared outside of the library block by using those elements as base types, inheriting from those elements, or by simply referencing them on a line, as follows:

```
interface MyFace {. . .};
[library attributes] library
{
interface My Face;
};
```

The MIDL compiler will create a type library that includes definitions for every element inside the library block, plus definitions for any elements defined outside and referenced from within the library block.

For information on generating both a type library and proxy stubs and headers from a single IDL file see Generating a Proxy DLL and a Type Library From a Single IDL File.

See Also

Generating a Type Library With MIDL, Contents of a Type Library, ODL File Syntax

licensed

[*attribute-list***]coclass** *classname* {*coclass-definitions*];

Example

```
[uuid(. . .), licensed, helpstring("A meaningfulcomment"]   coclass MyClass
{. . .};
```

Remarks

The **licensed** attribute indicates that the **coclass** to which it applies is licensed, and must be instantiated using **IClassFactory2.**

Licensing is a feature of COM that provides control over object creation. Licensed objects can be created only by clients that are authorized to use them. Licensing is implemented in COM through the **IClassFactory2** interface and by support for a license key that can be passed at run time.

Flags

TYPEFLAG_FLICENSED

See Also

coclass, Generating a Type Library With MIDL, Contents of a Type Library, ODL File Syntax, TYPEFLAGS

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. If you are not using the MIDL compiler switch /**osf**, you can enclose the UUID string in quotes.

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. Therefore this attribute is not available when you compile using the MIDL /**osf** switch.

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, **/osf**, **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**. On 32-bit platforms, **long** is synonymous with **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. Each expression evaluates to an integer that represents the highest valid array index. 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 increment and decrement operators. Separate multiple expressions with commas.

Examples

```
/* if m = 10, there are 11 transmitted elements (a[0]...a[10])*/
void Proc1(
    [in] short m,
    [in, max_is(m)] short a[]);

/* if m = 10, the valid range for b is b[0...10][20] */
void Proc2(
    [in] short m,
    [in, max_is(m)] short b[][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 zero, 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.

While it is legal to use the **max_is** attribute with a constant expression, doing so is inefficient and unnecessary. For example, use a fixed size array:

```
/* transmits values of a[0]... a[MAX_SIZE-1] */
void Proc3([in] short Arr[MAX_SIZE]);
```

instead of:

```
/* legal but marshalling code is much slower */
void Proc3([in max_is(MAX_SIZE-1)] short Arr[] );
```

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

Both client applications and server applications must implement the **midl_user_allocate** function, unless you are compiling in OSF-compatibility (/**osf**) mode. Applications and generated stubs call **midl_user_allocate** 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

Both client application and server application must implement the **midl_user_free** function, unless you are compiling in OSF-compatibility (/**osf**) mode. The **midl_user_free** function must be able to free all storage allocated by **midl_user_allocate**.

Applications and stubs call **midl_user_free** when dealing with 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**

module

[*attributes***] module** *modulename* **{***elementlist***};**

attributes

The **uuid**, **version**, **helpstring**, **helpcontext**, **hidden**, and **dllname** attributes are accepted before a **module** statement. See "Attribute Descriptions," in the Ole Automation book, for more information on the attributes accepted before a module definition. The **dllname** attribute is required. If the **uuid** attribute is omitted, the module is not uniquely specified in the system.

modulename

The name of the module.

elementlist

List of constant definitions and function prototypes for each function in the DLL. Any number of function definitions can appear in the function list. A function in the function list has the following form:

[*attributes***]** *returntype* **[***calling convention***]** *funcname***(***params***);**
[*attributes***] const** *constname* **=** *constval***;**

Only the **helpstring** and **helpcontext** attributes are accepted for a **const**.

The following attributes are accepted on a function in a module: **helpstring**, **helpcontext**, **string**, **entry**, **propget**, **propput**, **propputref**, **vararg**. If **vararg** is specified, the last parameter must be a safe array of VARIANT type.

The optional *calling convention* can be one of **__pascal/_pascal/pascal**, **__cdecl/_cdecl/cdecl**, or **__stdcall/_stdcall/stdcall**. The *calling convention* can include up to two leading underscores.

The parameter list is a comma-delimited list of:

[*attributes***]** *type paramname*

The *type* can be any previously declared type or built-in type, a pointer to any type, or a pointer to a built-in type. Attributes on parameters are:

in, **out**, **optional**

If **optional** is used, the types of those parameters must be VARIANT or VARIANT *.

Example

```
[uuid(. . .),
    helpstring("This is not GDI.EXE"), helpcontext(190),
    dllname("MATH.DLL")]
module somemodule{
    [helpstring("Color for the frame")] unsigned long const COLOR_FRAME
        = 0xH80000006;
    [helpstring("Not a rectangle but a square"), entry(1)] pascal double
square([in] double x);
};
```

Remarks

The **module** statement defines a group of functions, typically a set of DLL entry points. The header file (.H) output for modules is a series of function prototypes. The **module** keyword and surrounding brackets are stripped from the header (.H) file output, but a comment (// **module** *modulename*) is inserted before the prototypes. The keyword **extern** is inserted before the declarations.

See Also

Generating a Type Library With MIDL, Contents of a Type Library, ODL File Syntax, TYPEFLAGS

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 nonencapsulated unions.

The MIDL compiler in this version of Microsoft RPC mirrors the behavior of the OSF DCE IDL compiler for nonencapsulated 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_at_dsp

endpoint("ncacn_at_dsp:[*port-name***]")**

port-name

Specifies a character string up to 22 bytes long.

Example

```
[   uuid(12345678-4000-2006-0000-20000000001a),
    version(1.1),
    endpoint("ncacn_at_dsp:[my_services_endpoint]")
]
```

Remarks

The **ncacn_at_dsp** keyword identifies Appletalk DSP as the protocol family for the endpoint.

The syntax of the Appletalk DSP 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**, **string_binding**

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, **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.

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_ipx

endpoint("ncacn_nb_ipx:[*port-name***]")**

port-name

Specifies an optional 8-bit value ranging from zero 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_ipx:[100]")
]
```

Remarks

The **ncacn_nb_ipx** keyword identifies NetBIOS over IPX 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_tcp**, **ncacn_nb_nb**, **ncacn_at_dsp**, **ncacn_spx**, **ncacn_np**, **ncalrpc**, **ncadg_ip_udp**, **ncadg_ipx**, **string_binding**

ncacn_nb_nb

endpoint("ncacn_nb_nb:[*port-name***]")**

port-name

Specifies an optional 8-bit value ranging from zero 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 zero 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

Optional. Specifies the name of the server. Backslash characters are optional.

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.

Examples

```
[   uuid(12345678-4000-2006-0000-20000000001a),
    version(1.1),
    endpoint("ncacn_np:[\\pipe\\stove\\hat]")
]
[   uuid(12345678-4000-2006-0000-20000000001b),
    version(1.1),
    endpoint("ncacn_np:\\\\myotherserver[\\pipe\\corncob]")
]
```

Remarks

The **ncacn_np** keyword identifies named-pipes as the protocol family for the endpoint.

Note For Windows 95 platforms, **ncacn_np** is supported only for client applications.

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_at_dsp**, **ncacn_dnet_nsp**, **ncacn_ip_tcp**, **ncacn_nb_ipx**, **ncacn_spx**, **ncacn_nb_nb**, **ncacn_nb_tcp**, **ncacn_vns_spp** **ncalrpc**, **ncadg_ipx**, **ncadg_ip_udp**, **string_binding**

ncacn_spx

endpoint("ncacn_spx:*link-address***[***port-name***]")**

link-address

Specifies the host server. This may be either a character string (the server name), or a 20 digit hexadecimal number that consists of the host server's network address (8 digits)concatenated with the node address (12 digits). See **Remarks** for instructions on how to obtain the network address and 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 32-bit Windows NT 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.

16-bit Windows client applications that use the **ncacn_spx** transport require that the file NWIPXSPX.DLL be installed in order to run under the Windows NT Windows on Windows (WOW) subsystem. Contact Novell to obtain this file.

Note To obtain the network and node addresses, use Novell's **comcheck** utility, or the Novell-defined API **IPXGetInternetAddress**. On Windows NT, you can also obtain these addresses with the **ipxroute config**To obtain the network and node addresses, use Novell's **comcheck** utility, or the Novell-defined API **IPXGetInternetAddress**. On Windows NT, you can also obtain these addresses with the **ipxroute config** command.

See Also

endpoint, **IDL**, **ncacn_at_dsp**, **ncacn_dnet_nsp**, **ncacn_ip_tcp**, **ncacn_nb_ipx**, **ncacn_nb_nb**, **ncacn_nb_tcp**, **ncacn_np**, **ncacn_vns_spp**, **ncalrpc**, **ncadg_ipx**, **ncadg_ip_udp**, **string_binding**

ncacn_vns_spp

endpoint("ncacn_vns_spp:*server-name***[***port-address***]")**

server-name

Specifies the StreetTalk name of the server. The name is of the form item@group@organization. The item can be up to 31 characters long and the group and organization can be up to 15 characters.

port-name

Specifies a Banyan Vines SPP port. The valid range for static endpoints is 250 - 511.

Example

```
[   uuid( ... ),
    version(1.1),
    endpoint("ncacn_vns_spp:printer@sdkgrp@company[260]")
```

Remarks

The **ncacn_vns_spp** keyword identifies Banyan Vines SPP as the protocol family for the endpoint.

The syntax of the Banyan Vines SPP 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.

In order to use the **ncacn_vns_spp** transport protocol in distributed applications running on Windows NT, Banyan's *Enterprise Client for Windows NT* must be installed. After installation, open the Control Panel, choose Configuration and Add, then select "Service | Banyan | RPC services for Banyan". Support for 16-bit clients (MS-DOS and Windows 3.*x*) requires appropriate Vines software.

For more information on *Enterprise Client for Windows NT* and 16-bit Vines software, contact http://WWW.BANYAN.COM or call:

In USA	1-800-2-BANYAN
In Canada	(905)-855-2971
In Europe	31-3465-75080
In Asia Pacific	(852) 2821-9700

See Also

endpoint, **IDL**, **ncacn_at_dsp**, **ncacn_dnet_nsp**, **ncacn_ip_tcp**, **ncacn_nb_ipx**, **ncacn_spx**, **ncacn_nb_nb**, **ncacn_nb_tcp**, **ncacn_np**, **ncalrpc**, **ncadg_ipx**, **ncadg_ip_udp**,**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 and may be a fully-qualified domain name. 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 following restrictions apply to the datagram protocols, **ncadg_ipx** and **ncadg_ip_udp**:

- They do not support callbacks. Any functions using the **callback** attribute will fail.
- They do not support use of the **pipe** type constructor.

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_at_dsp**, **ncacn_dnet_nsp**, **ncacn_ip_tcp**, **ncacn_nb_ipx**, **ncacn_spx**, **ncacn_nb_nb**, **ncacn_nb_tcp**, **ncacn_np**, **ncacn_vns_spp**, **ncalrpc**, **ncadg_ipx**, **string_binding**

ncadg_ipx

endpoint("ncadg_ipx:*link-address***[***port-name***]")**

link-address

Specifies the host server. This may be either a character string (the server name), or a 20 digit hexadecimal number that consists of the host server's network address (8 digits)concatenated with the node address (12 digits). See **Remarks** for instructions on how to obtain the network address and 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 following restrictions apply to the datagram protocols, **ncadg_ipx** and **ncadg_ip_udp**:

- They do not support callbacks. Any functions using the **callback** attribute will fail.
- They do not support use of the **pipe** type constructor.

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.

When using the **ncadg_ipx** transport, the server name is exactly the same as the 32-bit Windows 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.

16-bit Windows client applications that use the **ncadg_ipx** transport require that the file NWIPXSPX.DLL be installed in order to run under the Windows NT Windows on Windows (WOW) subsystem. Contact Novell to obtain this file.

To obtain the network and node addresses, use Novell's **comcheck** utility, or the Novell-defined API **IPXGetInternetAddress**. On Windows NT, you can also obtain these addresses with the **ipxroute config** command.

See Also

endpoint, **IDL**, **ncacn_at_dsp**, **ncacn_dnet_nsp**, **ncacn_ip_tcp**, **ncacn_nb_ipx**, **ncacn_spx**,

ncacn_nb_nb, ncacn_nb_tcp, ncacn_np, ncacn_vns_spp, ncalrpc, ncadg_ip_udp, string_binding

ncalrpc

endpoint("ncalrpc:[*port-name***]")**

port-name

A character string that specifies the communication port (an application, a service, or an instance of a service) that a client uses to make interprocess calls to a server. The string can contain up to 250 characters and should not contain any backslash (\) characters. The computer name must not be used with the **ncalrpc** keyword.

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.

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_at_dsp**, **ncacn_dnet_nsp**, **ncacn_ip_tcp**, **ncacn_nb_ipx**, **ncacn_spx**, **ncacn_nb_nb**, **ncacn_nb_tcp**, **ncacn_np**, **ncacn_vns_spp**, **ncadg_ip_udp**, **ncadg_ipx**, **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 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 nonencapsulated 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**

nonextensible

[uuid, **nonextensible** [, *optional-attribute-list*] **interface** | **dispinterface** *interface-name* **{***interface-definitions***}**

Example

```
library Hello
{
[uuid( . . .), helpstring("A helpful description."),
  oleautomation, dual, nonextensible] interface IHello : IDispatch
    {. . .};
```

Remarks

By default, OLE Automation assumes that interfaces may add members at run time; that is, it assumes they are extensible. The **nonextensible** attribute specifies that the **IDispatch** implementation includes only the properties and methods listed in the interface description and cannot be extended with additional members at run time.

You can apply the **nonextensible** attribute to either an interface or a dispinterface. However, an interface must also have the **dual** and **oleautomation** attributes.

Flags

TYPEFLAG_FNONEXTENSIBLE

See Also

TYPEFLAGS, **interface**, **dispinterface**, Generating a Type Library With MIDL, Contents of a Type Library, ODL File Syntax, TYPEFLAGS

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, except when you use the MIDL compiler switch **/osf**.

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 another 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. This DLL 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)]
    interface IStream : IUnknown{
        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.

Non-[**local**] object interface member functions must have a return value of HRESULT or SCODE. (Note that earlier versions of MIDL allowed member functions to return **void**. However, beginning with MIDL version 3.0, returning **void** generates a compiler error.) Having a return value of HRESULT or SCODE means that if an exception is generated during a remote call, the generated proxies catch the exception and return the exception code in the return value. If your application can afford to ignore errors that occur during a remote procedure call, you can specify HRESULT as the return type without checking the return value after the call.

If you are recompiling an old application, changing the return types can introduce backward compatibility

problems when the server sends the newly introduced result to the client. As an alternative to changing the return type, you can label the function that returns **void** with the **[call_as]** attribute, thus making it a local function. Then define a related remote function with the same parameters but with the return type of HRESULT. The local function can raise an exception based on that HRESULT value, if necessary.

The **object** attribute is not available when you compile using the MIDL compiler **/osf** switch.

See Also

IDL, **iid_is**, **local**, **/osf**, **uuid**, **version**

odl

Remarks

MKTYPLIB required this attribute on ODL interfaces. The MIDL compiler does not require the **odl** attribute; it is recognized only for compatibility with older **ODL** files.

oleautomation

[oleautomation, uuid(*string-uuid***)[,** *interface-attribute-list*] **]**
interface *interface-name* **:** *base-interface*

string-uuid

Specifies a UUID string generated by the **uuidgen** utility.

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 Automation interface from which this derived interface inherits member functions, status codes, and interface attributes. All OLE Automation interfaces are derived from **IUnknown** or **IDispatch**.

Example

```
library Hello
{
 importlib("stdole32.tlb");
    [
      uuid( . . .),
      helpstring("Application object for the Hello application."),
      oleautomation,
      dual
    ]
    interface IHello : IDispatch
    {. . .}
```

Remarks

The **oleautomation** attribute indicates that an interface is compatible with OLE Automation. The parameters and return types specified for its members must be OLE Automation-compatible, as listed in the following table.

Type	Description
boolean	Data item that can have the value TRUE or FALSE. In MIDL, the size corresponds to **unsigned char**.
unsigned char	8-bit unsigned data item.
double	64-bit IEEE floating-point number.
float	32-bit IEEE floating-point number.
int	Integer whose size is system dependent. On 32-bit platforms, MIDL treats **int** as a 32-bit signed integer.
long	32-bit signed integer.
short	16-bit signed integer.
BSTR	Length-prefixed string, as described in the OLE

	Automation topic BSTR.
CY	(Formerly CURRENCY) 8-byte fixed-point number.
DATE	64-bit floating-point fractional number of days since December 30, 1899.
SCODE	Built-in error type that corresponds to VT_ERROR.
enum	Signed integer, whose size is system-dependent. In remote operations, **enum** objects are treated as 16-bit unsigned entities. Applying the **v1_enum** attribute to an **enum** type definition allows **enum** objects to be transmitted as 32-bit entities.
IDispatch *	Pointer to **IDispatch** interface (VT_DISPATCH).
IUnknown *	Pointer to interface that is not derived from **IDispatch** (VT_UNKNOWN). (Any OLE interface can be represented by its **IUnknown** interface.)

A parameter is compatible with OLE Automation if its type is an OLE Automation-compatible type, a pointer to an OLE Automation-compatible type, or a SAFEARRAY of an OLE Automation-compatible type.

A return type is compatible with OLE Automation if its type is an HRESULT, SCODE or **void**. However, MIDL requires that interface methods return either HRESULT or SCODE. Returning **void** generates a compiler error.

A member is compatible with OLE Automation if its return type and all its parameters are OLE-Automation compatible.

An interface is compatible with OLE Automation if it is derived from **IDispatch** or **IUnknown**, it has the **oleautomation** attribute, and all of its VTBL entries are OLE-Automation compatible. For 32-bit platforms, the calling convention for all methods in the interface must be STDCALL. For 16-bit systems, all methods must have the CDECL calling convention.

Every **dispinterface** is implicitly OLE Automation-compatible. Therefore you should not use the **oleautomation** attribute on **dispinterfaces.**

The **oleautomation** attribute is not available when you compile using the MIDL compiler **/osf** switch.

Flags

TYPEFLAG_FOLEAUTOMATION

See Also

ODL File Syntax, ODL File Example, Generating a Type Library With MIDL, Differences Between MIDL and MKTYPLIB, **IDL**, **uuid**

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**

optional

return-type func-name(**[optional** [, *other-attributes*]**]** *param-type param-name*)

Example

```
HRESULT MyFunc([in, optional] VARIANT Param1,
               [out, optional] VARIANT Param2)
```

Remarks

The **optional** attribute specifies an optional parameter for a member function. This attribute is valid only if the parameter is of type VARIANT or VARIANT*.

The MIDL compiler accepts the following parameter ordering (from left-to-right):

1. Required parameters (parameters that do not have the **defaultvalue** or **optional** attributes),
2. optional parameters with or without the **defaultvalue** attribute,
3. parameters with the **optional** attribute and without the **defaultvalue** attribute,
4. **lcid** parameter, if any,
5. **retval** parameter

You cannot apply the **optional** attribute to a parameter that also has the **lcid** or **retval** attributes.

See Also

ODL File Syntax, ODL File Example, Generating a Type Library With MIDL, Differences Between MIDL and MKTYPLIB

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

typedef pipe *element-type pipe-declarator*;

element-type

Defines the size of a single element in the transfer buffer. The *element-type* can be a **base_type**, **predefined_type**, **struct**, **enum**, or type identifier. Several restrictions apply to *element-types*, as described in **Remarks**, below.

pipe-declarator

Specifies one or more identifiers or pointers to identifiers. Separate multiple declarators with commas.

Examples

```
typedef pipe unsigned char UCHAR_PIPE1, UCHAR_PIPE2;

//SIMPLE_STRUCT defined elsewhere
typedef pipe SIMPLE_STRUCT SIMPLE_STRUCT_PIPE;
```

Remarks

The **pipe** type constructor makes it possible to transmit an open-ended stream of typed data across a remote procedure call. An **in** pipe parameter allows the server to pull the data stream from the client during a remote procedure call. An **out** pipe parameter allows the server to push the data stream back to the client. You supply the client-side routines to push and pull the data stream and to allocate a global buffer for the data. The client and server stub routines marshal and unmarshal data and pass a reference to the buffer back to the application.

The following restrictions apply to pipes:

- A pipe element cannot be or contain a pointer, a conformant or varying array, a handle, or a context handle. In this (MIDL 3.0) implementation of pipes, a pipe element cannot be or contain a **union**.
- You cannot apply the **transmit_as**, **represent_as**, **wire_marshal**, or **user_marshal** attributes to a pipe type or to the *element-type*.
- A pipe type cannot be a member of a structure or union, the target of a pointer, or the base type of an array.
- A data type declared to be a pipe type can only be used as a parameter of a remote call.
- You can pass a pipe parameter in either direction by value or by reference (**ref** pointer). However you cannot apply the **ptr** attribute to a pipe that is passed by reference. You cannot specify a pipe parameter with a **unique** or a full pointer, regardless of direction.
- You cannot use pipes in **object** interfaces.
- You cannot apply the **idempotent** attribute to a routine that has a pipe parameter.
- You cannot use the serialization attributes, **encode** and **decode** with pipes.
- You cannot use automatic handles, either by default, or with the **auto_handle** attribute, with pipes.

Note The MIDL 3.0 compiler supports pipes in **/Oi2** mode only.

For more information on implementing routines with pipe parameters, see Pipes in the RPC Programmer's Guide and Reference.

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** (**osf** mode); **unique** (Microsoft RPC default mode)

When the IDL file is compiled in default mode, imported files can inherit pointer attributes from importing files. This feature is not available when you compile using the /**osf** switch. 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 default (Microsoft-extensions) mode of the MIDL compiler, this constraint is relaxed.

See Also

allocate, **IDL**, **import**, **/osf**, **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**

propget

[propget [,*optional-property-attributes*]**]** *return-type func-name(parameters);*

Example

```
interface InMyFace : IDispatch
    {
      [propget, helpstring("A meaningful comment.")]
      HRESULT Method1([out, retval] int* retval);
      [propput, helpstring("Another meaningful comment.")]
      HRESULT Method1([in] int Value);

     [propget, helpstring("A meaningful comment."), id(1)]
      HRESULT Method2([out, retval] InYourFace** retval);
      [propputref, helpstring("Another meaningful comment."), id(1)]
      HRESULT Method2([in] InYourFace* Point);
    }
```

Remarks

The **propget** attribute specifies a property accessor function. The property must have the same name as the function.

A function that has the propget attribute must also have, as its last parameter, a pointer type with the **out** and **retval** attributes.

At most, one of **propget**, **propput**, and **propputref** can be specified for a function.

Flags

INVOKE_PROPERTYGET

See Also

TYPEFLAGS, ODL File Syntax, ODL File Example, Generating a Type Library With MIDL, Differences Between MIDL and MKTYPLIB

propput

[propput [,*optional-property-attributes*]**]** *return-type func-name(parameters);*

Example

```
interface InMyFace : IDispatch
    {
      [propget, helpstring("A meaningful comment.")]
      HRESULT Method1([out, retval] int* retval);
      [propput, helpstring("Another meaningful comment.")]
      HRESULT Method1([in] int Value);
    }
```

Remarks

The **propput** attribute specifies a property-setting function. The property must have the same name as the function.

A function that has the **propput** attribute must also have, as its last parameter, a parameter that has the **in** attribute.

At most, one of **propget**, **propput** and **propputref** can be specified for a function.

Flags

INVOKE_PROPERTYPUT

See Also

TYPEFLAGS, ODL File Syntax, ODL File Example, Generating a Type Library With MIDL, Differences Between MIDL and MKTYPLIB

propputref

[propputref [,*optional-property-attributes*]**]** *return-type func-name(parameters);*

Example

```
interface InMyFace : IDispatch
   {
     [propget, helpstring("A meaningful comment."), id(1)]
     HRESULT Method2([out, retval] InYourFace** retval);
     [propputref, helpstring("Another meaningful comment."), id(1)]
     HRESULT Method2([in] InYourFace* Point);
```

Remarks

The **propputref** attribute specifies a property-setting function that uses a reference instead of a value.

A function that has the **propputref** attribute must also have, as its last parameter, a pointer that has the **in** attribute.

The property must have the same name as the function. At most, one of **propget**, **propput** and **propputref** attributes can be specified for a function.

Flags

INVOKE_PROPERTYPUTREF

See Also

TYPEFLAGS, ODL File Syntax, ODL File Example, Generating a Type Library With MIDL, Differences Between MIDL and MKTYPLIB

public

typedef [public] *data-type identifier;*

Example

```
typedef [public] long MEMBERID;
```

Remarks

The **public** attribute includes an alias declared with the **typedef** keyword in the type library.

By default, an alias that is declared with **typedef** and has no other attributes is treated as a **#define** and is not included in the type library. Using the **public** attribute ensures that the alias becomes part of the type library.

> **Note** The MIDL compiler requires that you explicitly apply the **public** attribute to each typedef that you want public. This is in contrast to MKTYPLIB, which treated as public, every typedef inside of a public interface block.

See Also

interface, ODL File Syntax, ODL File Example, Generating a Type Library With MIDL, Differences Between MIDL and MKTYPLIB

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**

readonly

[readonly [, *optional-attributes*]**]** *data-type identifier*

Example

```
HRESULT Method3([in, readonly] int iMmutable);
```

Remarks

The **readonly** attribute prohibits assignment to a variable.

Flags

VARFLAG_FREADONLY

See Also

TYPEFLAGS, ODL File Syntax, ODL File Example, Generating a Type Library With MIDL, Differences Between MIDL and MKTYPLIB

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. 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.

Example

```
//these data types defined in .IDL or elsewhere
typedef struct  _lbox {
    long         data;
    struct _lbox *next;
} lbox;
typedef [ref] lbox *PBOX_LOC;
typedef long LONG4[4];

//in .ACF file :
interfaceFoo

{
typedef  [ represent_as(PBOX_LOC) ]  LONG4;
}
```

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**.

You 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:

- 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**.
- A type cannot have both the **represent_as** attribute and the **handle** attribute.

See Also

ACF, **arrays**, **base_types**, **typedef**

requestedit

[requestedit [,*optional-attributes*]**]** *return-type function-name(params)*

Example

```
properties:
 [propget, helpstring("A useful comment"), bindable, defaultbind,
     displaybind, requestedit] long Func1(void);
```

Remarks

The **requestedit** attribute indicates that the property supports the OnRequestEdit notification. This means that, before a change is made, the object will send the client a request for permission to change a property. An object can support data binding but not have this attribute.

Flags

FUNCFLAG_FREQUESTEDIT, VARFLAG_FREQUESTEDIT

See Also

TYPEFLAGS, ODL File Syntax, ODL File Example, Generating a Type Library With MIDL, Differences Between MIDL and MKTYPLIB

restricted

[restricted[, *other-attributes*]**]**
statement-type statement-name {*definitions*};

Examples

```
[uuid(. . .), version (1.0), restricted] library UCantTouchThis
{. . .};

[propget, restricted] HRESULT DontTouch(void);
```

Remarks

The **restricted** attribute specifies that a library, or member of a module, interface, or dispinterface cannot be called arbitrarily. For example, this prevents a data item from being used by a macro programmer. You can apply this attribute to a member of a **coclass**, independent of whether the member is a dispinterface or interface, and independent of whether the member is a sink (incoming) or source (outgoing). A member of a coclass cannot have both the **restricted** and **default** attributes.

Flags

IMPLTYPEFLAG_FRESTRICTED, FUNCFLAG_FRESTRICTED

See Also

TYPEFLAGS, **library**, **interface**, **dispinterface**, **module**, ODL File Syntax, ODL File Example, Generating a Type Library With MIDL, Differences Between MIDL and MKTYPLIB

retval

return-type function-name(**[out**, **retval**
[, *optional-attributes*]**]** *data-type* * **retval**)

Examples

```
HRESULT MyMethod([out, retval] InMyFace** retval);
HRESULT MyOtherMethod([out, retval] boolean* retval);
```

Remarks

The **retval** attribute designates the parameter that receives the return value of the member. You can use this atttribute on parameters of interface members that describe methods or get properties. (The attribute is required on the last parameter of a method that has the **propget** attribute.) Note that the parameter takes the name of the attribute. The parameter must have the **out** attribute and must be a pointer type.

You cannot apply the **optional** attribute to a **retval** parameter.

The MIDL compiler accepts the following parameter ordering (from left-to-right):

1. Required parameters (parameters that do not have the **defaultvalue** or **optional** attributes),
2. optional parameters with or without the **defaultvalue** attribute,
3. parameters with the **optional** attribute and without the **defaultvalue** attribute,
4. **lcid** parameter, if any,
5. **retval** parameter

Parameters with the **retval** attribute are not displayed in user-oriented browsers.

Flags

IDLFLAG_FRETVAL

See Also

TYPEFLAGS, ODL File Syntax, ODL File Example, Generating a Type Library With MIDL, Differences Between MIDL and MKTYPLIB

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-list***)]**

limited-expression-list

Specifies one or more C-language expressions. Each expression evaluates to a non-negative integer that represents the amount of memory allocated to a sized pointer or an array. In the case of an array, specifies a single expression that represents the maximum allocation size, in elements, of the first dimension of that 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 increment and decrement operators. Use commas as placeholders for implicit parameters, or to separate multiple expressions.

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]
void Proc3(
    [in, short] m;
    [size_is(m)] short * pshort); //specifies a pointer
                      // to an m-sized block of shorts
void Proc4(
    [in, short] m;
    [size_is(,m)] short ** ppshort); /*specifies a pointer
                                to a pointer to an m-sized
                                        block of shorts */
void Proc5(
    [in, short] m;
    [size_is(m,)] short ** ppshort); /*specifies an
            m-sized block of pointers to shorts */
void Proc6(
    [in, short] m;
    [in, short] n;
    [size_is(m,n)] short ** ppshort); /* specifies a
            pointer to an m-sized block of pointers,
                 each of which points to an n-sized
                                block of shorts.*/
```

Remarks

You can use the **size_is** attribute to specify the size of memory allocated for sized pointers, sized pointers to sized pointers, and single- or multi-dimensional arrays. However, if you are using array [ ] notation, only the first dimension of a multi-dimensional array can be determined at run time.

For more information on using the **size_is** attribute with multiple levels of pointers to enable a server to return a dynamically-sized array of data to a client, see Multiple Levels of Pointers.

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.

While it is legal to use the **size_is** attribute with a constant, doing so is inefficient and unnecessary. For example, use a fixed size array:

```
void Proc3([in] short Arr[MAX_SIZE]);
```

instead of:

```
// legal but marshalling code is much slower
void Proc3([in size_is(MAX_SIZE)] short Arr[] );
```

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**

source

[source [, *optional-attributes*]**]** *statement-type statement-name* {*definitions*};

Example

```
[default, source] dispinterface DIMyFaceAdviseSink;
[source]interface IMyFaceAdviseSink;
```

Remarks

The **source** attribute indicates that a member of a **coclass**, property, or method is a source of events. For a member of a **coclass**, this attribute means that the member is called rather than implemented.

On a property or method, the **source** attribute indicates that the member returns an object or VARIANT that is a source of events. The object implements **IConnectionPointContainer**.

Flags

IMPLTYPEFLAG_FSOURCE, VARFLAG_SOURCE, FUNCFLAG_SOURCE

See Also

TYPEFLAGS, ODL File Syntax, ODL File Example, Generating a Type Library With MIDL, Differences Between MIDL and MKTYPLIB

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**, **/osf**, **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 nonencapsulated unions. The encapsulated union is compatible with previous implementations of RPC (NCA version 1). The nonencapsulated 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 nonencapsulated 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**, **int, 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**

user_marshal

typedef [user_marshal(*userm_type***)]** *wire-type***;**

unsigned long __RPC_USER < *userm_type* >**_UserSize**(
    unsigned long __RPC_FAR **pFlags*,
    unsigned long *StartingSize*,
    < *userm_type* > **__RPC_FAR** * *pUser_typeObject*);

unsigned char __RPC_FAR * __RPC_USER **<** *userm_type* **>_UserMarshal**(
    unsigned long __RPC_FAR **pFlags*,
    unsigned char __RPC_FAR * *Buffer*,
    < *userm_type* > **__RPC_FAR** * *pUser_typeObject*);

unsigned char __RPC_FAR * __RPC_USER < *userm_type* >**_UserUnmarshal**(
    unsigned long __RPC_FAR * *pFlags*,
    unsigned char __RPC_FAR * *Buffer*,
    < *userm_type* > **__RPC_FAR** * *pUser_typeObject*);

void __RPC_USER < *userm_type* >**_UserFree**(
    unsigned long __RPC_FAR * *pFlags*,
    < *userm_type* > **__RPC_FAR** * *pUser_typeObject*);

userm-type

Specifies the id of the user data type to be marshaled. The *userm-type* need not be remotable and need not be a type known to the MIDL compiler.

wire-type

Specifies the named transfer data type that is actually transferred between client and server. the *wire-type* must be a MIDL base type, predefined type, or a type identifier of a remotable.

pFlags

Specifies a pointer to a flag field(unsigned long). The high-order word specifies NDR data representation flags as defined by DCE for floating point, big- or little-endian, and character representation. The low-order word specifies a marshaling context flag. The exact layout of the flags is described in The type_UserSize Function.

StartingSize

Specifies the current buffer size (offset) before sizing the object.

Buffer

Specifies the current buffer pointer.

pUser_typeObject

Specifies a pointer to an object of *userm_type.*

Example

```
// Marshal a long as a structure containing two shorts.
typedef unsigned long FOUR_BYTE_DATA;
typedef struct _TWO_X_TWO_BYTE_DATA {
    unsigned short low;
    unsigned short high;
} TWO_X_TWO_BYTE_DATA;
```

```
//in ACFL file:
typedef [user_marshal(FOUR_BYTE_DATA)] TWO_X_TWO_BYTE_DATA;
//Marshaling functions:
unsigned long __RPC_USER FOUR_BYTE_DATA_UserSize(
    ULONG __RPC_FAR * pulFlags,
    char __RPC_FAR * pBufferStart,
    FOUR_BYTE_DATA __RPC_FAR * pul
    );//calculate size that converted data will
      // require in the buffer
unsigned long __RPC_USER FOUR_BYTE_DATA_UserMarshal(
    ULONG __RPC_FAR *pulFlags,
    char __RPC_FAR * pBufferStart,
    FOUR_BYTE_DATA __RPC_FAR * pul
    );//copy FOUR_BYTE_DATA into buffer as
      //TWO_X_TWO_BYTE_DATA
unsigned long __RPC_USER FOUR_BYTE_DATA_UserUnmarshal(
    ULONG __RPC_FAR * pulFlags,
    char __RPC_FAR * pBufferStart,
    FOUR_BYTE_DATA __RPC_FAR * pul
    );//recreate FOUR_BYTE_DATA from TWO_X_TWO_BYTE_DATA
      //in buffer
void __RPC_USER FOUR_BYTE_DATA_UserFree(
    ULONG __RPC_FAR * pulFlags,
    FOUR_BYTE_DATA __RPC_FAR * pul
    );//nothing to do here as the engine frees the
      // top node and FOUR_BYTE_DATA is a flat data
      //type.
```

Remarks

The **user_marshal** attribute associates a named local type in the target language (*userm-type*) with a transfer type (*wire-type*) that is transferred between client and server. Each *userm-type* has a one-to-one correspondence with a *wire-type* that defines the wire representation of the type. You must supply routines to size the data for marshaling, to marshal and unmarshal the data, and to free memory. For more information on these routines, see The user_marshal Attribute. Note that if there are embedded types in your data that are also defined with **user_marshal** or **wire_marshal**, you need to manage the servicing of those embedded types also.

The *wire-type* cannot be an interface pointer or a full pointer. The *wire-type* must have a well-defined memory size. See Marshaling Rules for user_marshal and wire_marshal for details on how to marshal a given *wire-type*.

The *userm-type* should not be an interface pointer as these can be marshaled directly. If the user type is a full pointer, you must manage the aliasing yourself.

See Also

The user_marshal Attribute, **wire_marshal**, **represent_as**, **base_types**

usesgetlasterror

[*module-attributes***] module** *module-name*
{[entry(*entry-id***),** **usesgetlasterror** [, *other-attributes*]**]** *return-typefunction-name(param-list);*
};

Example

```
[dllname("MyOwn.dll")] module MyModule
    {
    [entry("One"), usesgetlasterror, bindable, requestedit,
   propputref, defaultbind]
        void Func1 ([in]IUnknown * iParam1, [out] long * Param2) ;
    [entry("TwentyOne"), usesgetlasterror, hidden, vararg]
         SAFEARRAY (int) Func2 ([in, out] SAFEARRAY (variant) *varP) ;
. . .};
```

Remarks

The **usesgetlasterror** attribute can be set on a module entry point, if that entry point uses the Win32 function **SetLastError** to return error codes. The attribute tells the caller that, if there is an error when calling that function, the caller can then call GetLastError to retrieve the error code.

See Also

ODL File Syntax, ODL File Example, Generating a Type Library With MIDL, Differences Between MIDL and MKTYPLIB

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, except when you use the MIDL compiler switch **/osf**.

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**, **/osf**, **version**

v1_enum

[v1_enum] enum {...}

Example

```
typedef [v1_enum] enum {label1, label2, ...};
```

Remarks

The **v1_enum** attribute directs that the specified enumerated type be transmitted as a 32-bit entity, rather than the 16-bit default. This increases the efficiency of marshalling and unmarshalling data when such an enumerator is embedded in structures or unions.

For improved performance, we recommend applying the **v1_enum** attribute to enumerators in 32-bit applications. Keep in mind, however, that on 16-bit platforms the C compiler treats an enumerated type as a 16-bit **int**. Therefore 16-bit client applications need to convert **enum** types to **long** for remote transmission in order to avoid overwriting data or sending incorrect values.

See Also

enum, **IDL**

vararg

[vararg [, *optional-attributes*]**]** *return-type function-name*([*optional-param-attributes*] *param-list*, SAFEARRAY(VARIANT) *last-param-name*)

optional-attributes

Specifies zero or more attributes to be applied to the function. Separate multiple attributes with commas.

optional-param-attributes

Specifies zero or more attributes to be applied to the function parameter immediately following the attribute listing.

param-list

Specifies all the parameters, save the final, varying, parameters.

Example

```
[vararg] boolean Button([in]SAFEARRAY(VARIANT) psa);
```

Remarks

The **vararg** attribute specifies that the function takes a variable number of arguments. To accomplish this, the last argument must be a safe array of VARIANT type that contains all the remaining arguments.

You cannot apply the **optional** or **defaultvalue** attributes to any parameters in a function that has the **vararg** attribute.

See Also

ODL File Syntax, ODL File Example, Generating a Type Library With MIDL, Differences Between MIDL and MKTYPLIB

version

version (*major-value*[**.** *minor-value*] **)**

major-value

Specifies a short unsigned integer between zero and 65,535, inclusive, that represents the major version number.

minor-value

Specifies a short unsigned integer between zero 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**

wire_marshal

typedef [wire_marshal(*wire_type***)]** *type-specifier userm-type***;**

unsigned long __RPC_USER < *userm_type* >**_UserSize**(
unsigned long __RPC_FAR **pFlags*,
unsigned long *StartingSize*,
< *userm_type* > **__RPC_FAR** * *pUser_typeObject*);

unsigned char __RPC_FAR * __RPC_USER **<** *userm_type* **>_UserMarshal**(
unsigned long __RPC_FAR * *pFlags*,
unsigned char __RPC_FAR * *Buffer*,
< *userm_type* > **__RPC_FAR** * *pUser_typeObject*);

unsigned char __RPC_FAR * __RPC_USER < *userm_type* >**_UserUnmarshal**(
unsigned long __RPC_FAR * *pFlags*,
unsigned char __RPC_FAR * *Buffer*,
< *userm_type* > **__RPC_FAR** * *pUser_typeObject*);

void __RPC_USER < *userm_type* >**_UserFree**(
unsigned long __RPC_FAR * *pFlags*,
< *userm_type* > **__RPC_FAR** * *pUser_typeObject*);

userm-type

Specifies the id of the user data type to be marshaled. It can be any type, as given by the *type-specifier*,.as long as it has a well-defined size. The *userm-type* need not be remotable but must be a type known to the MIDL compiler.

wire-type

Specifies the named transfer data type that is actually transferred between client and server. The *wire-type* must be a MIDL base type, predefined type, or a type identifier of a remotable type.

pFlags

Specifies a pointer to a flag field (unsigned long). The high-order word specifies NDR data representation flags as defined by DCE for floating point, big- or little-endian, and character representation. The low-order word specifies a marshaling context flag. The exact layout of the flags is described in The type_UserSize Function.

StartingSize

Specifies the current buffer size (offset) before sizing the object.

Buffer

Specifies the current buffer pointer.

pUser_typeObject

Specifies a pointer to an object of *userm_type.*

Example

```
typedef unsigned long _FOUR_BYTE_DATA;

    typedef struct _TWO_X_TWO_BYTE_DATA {
        unsigned short low;
        unsigned short high;
```

```
        } TWO_X_TWO_BYTE_DATA;

    typedef [wire_marshal(TWO_X_TWO_BYTE_DATA)] _FOUR_BYTE_DATA
FOUR_BYTE_DATA;
//Marshaling functions:
unsigned long __RPC_USER FOUR_BYTE_DATA_UserSize(
    ULONG __RPC_FAR * pulFlags,
    char __RPC_FAR * pBufferStart,
    FOUR_BYTE_DATA __RPC_FAR * pul
    );//calculate size that converted data will
      // require in the buffer
unsigned long __RPC_USER FOUR_BYTE_DATA_UserMarshal(
    ULONG __RPC_FAR *pulFlags,
    char __RPC_FAR * pBufferStart,
    FOUR_BYTE_DATA __RPC_FAR * pul
    );//copy FOUR_BYTE_DATA into buffer as
      //TWO_X_TWO_BYTE_DATA
unsigned long __RPC_USER FOUR_BYTE_DATA_UserUnmarshal(
    ULONG __RPC_FAR * pulFlags,
    char __RPC_FAR * pBufferStart,
    FOUR_BYTE_DATA __RPC_FAR * pul
    );//recreate FOUR_BYTE_DATA from TWO_X_TWO_BYTE_DATA
      //in buffer
void __RPC_USER FOUR_BYTE_DATA_UserFree(
    ULONG __RPC_FAR * pulFlags,
    FOUR_BYTE_DATA __RPC_FAR * pul
    );//nothing to do here as the engine frees the top
      //node and FOUR_BYTE_DATA is a flat data type.
```

Remarks

The **wire_marshal** attribute specifies a data type that will be used for transmission (the *wire-type*) instead of an application-specific data type (the *userm-type*). Each *userm-type* has a one-to-one correspondence with a *wire-type* that defines the wire representation of the type. You must supply routines to size the data for marshaling, to marshal and unmarshal the data, and to free memory. Note that if there are embedded types in your data that are also defined with **wire_marshal** or **user_marshal**, you need to manage the servicing of those embedded types also. For more information on these routines, see The wire_marshal Attribute.

The *wire-type* cannot be an interface pointer or a full pointer. The *wire-type* must have a well-defined memory size. See Marshaling Rules for user_marshal and wire_marshal for details on how to marshal a given *wire-type*.

The *userm-type* should not be an interface pointer because these can be marshaled directly. If the user type is a full pointer, you must manage the aliasing yourself.

You cannot use the **wire_marshal** attribute with the **allocate** attribute, either directly or indirectly, because the NDR engine does not control memory allocation for the transmitted type.

See Also

The wire_marshal Attribute, **user_marshal**, **transmit_as**, **base_types**

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, you can include ACF attributes in an IDL file when you use the /**app_config** switch, but that IDL file will generate an error if you compile without using the /**app_config** switch.

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. This feature is a Microsoft extension and is not available when you compile in DCE-compatibility (**osf**) mode.

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 slash 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.