0Part IV: Type Information

1Interface Definition Language

As was described previously in this specification, the COM infrastructure is completely divorced from the source-level tools used to create and use COM components. COM is completely a *binary* specification, and thus source-level specifications and standards have no role to play in the fundamental architecture of the system.

Specifically, and somewhat different than other environments, this includes any and all forms of interface definition language (IDL). Having an interoperable standard for an appropriate IDL (or any other source level tool for that matter) is still incredibly valuable and useful, it's just important to understand that this is a *tool* standard and not a fundamental *system* standard. Contrast this, for example, with the DCE RPC API specification, where, if only because the fundamental SendReceive API is not part of the public standard runtime infrastructure, one *must* use IDL to interoperate with the system.[1] People can (and have, out of necessity) built COM components with custom COM interfaces without using any interface definition language at all. This clear separation of system standards from tools standards is an important point, for without it COM tools vendors cannot innovate without centralizing their innovations through some central standards body. Innovation is stifled, and the customers suffer a loss of valuable tools in the marketplace.

That all being said, as was just mentioned, source-level standards are still useful, and DCE IDL is one such standard. The following enhancements to DCE IDL enable it to specify COM interfaces in addition to DCE RPC interfaces.[2]

1.1Object RPC IDL Extensions

1.1.1'Object' interface attribute

This page intentionally left blank.

COM interfaces are signified by a new interface attribute, 'object'. See [CAE RPC], page 238.

```
<interface_attributes> ::= <interface_attribute> [ , <interface_attribute> ] ...
<interface_attribute>  ::= uuid ( <Uuid_rep> )
    | version ( <Integer_literal>[.Integer_literal>])
    | endpoint ( <port_spec> [ ,<port_spec> ] ... )
    | local
    | pointer_default ( <ptr_attr> )
    | object
<port_spec> ::= <Family_string> : <[> <Port_string> <]>
```

The object interface attribute attributed may not be specified with the version attribute. However, it may be specified with any of the others, though the uuid attribute is virtually always used and the local attribute is used but rarely.

If this keyword is present, the following extensions are enabled in the interface.

1.1.2Interface name as a type

The interface name becomes the name of a type, which can then be used as a parameter in methods. For example:

```
[object, uuid(b5483f00-4f6c-101b-a1c7-00aa00389acb)]
    interface IFoo {
        };
```

causes a typed named "IFoo" to be declared, such that a method

```
[object, uuid(6A874340-57EB-11ce-A964-00AA006C3706)]
    interface IBar {
        HRESULT M1([in] short i, [in] IFoo* pfoo);
        };
```

[1] By definition one cannot, for example, write a source-portable DCE IDL compiler, for the code that calls SendReceive in the proxies is implementation-specific.

[2] Microsoft has in its MIDL specification language defined additional extensions to DCE IDL; however, these are orthogonal to the subject of COM interface, and thus are not dealt with here.

is a legal declaration.

1.1.3No handle_t required

In methods, no handle_t argument is needed, and it's absence does not indicate auto-binding. Instead, a "this" pointer is used in the C++ binding to indicate the remote object being referenced, and an implicit extra first argument is used in C. For example:

```
[object, uuid(b5483f00-4f6c-101b-a1c7-00aa00389acb)]
    interface IBar {
        HRESULT Bar([in] short i, [in] IFoo * pIF);
        };
```

can be invoked from C++ with:

```
IFoo * pIF;
IBar * pIB;
pIB->Bar(3, pIF);
```

or from C with the equivalent

```
pIB->lpVtbl->Bar(pIB, 3, pIF);
```

1.1.4Interface inheritance

Single inheritance of interfaces is supported, using the C++ notation for same. Referring again to [CAE RPC], page 238:

```
<interface_header> ::=
    <[> <interface_attributes> <]> interface <Identifier> [ <:> <Identifier> ]
```

For example:

```
[object, uuid(b5483f00-4f6c-101b-a1c7-00aa00389acb)]
    interface IBar : IWazoo {
        HRESULT Bar([in] short i, [in] IFoo * pIF);
        };
```

cases the first methods in IBar to be the methods of IWazoo.

1.1.5iid_is and use of void*

The use of "void*" pointers are permitted, as long as such pointers are qualified with an "iid_is" pointer attribute. See [CAE RPC], page 253.

```
<ptr_attr> ::= ref
    | ptr
    | iid_is ( <attr_var_list> )
    | unique[3]
```

The iid_is construct says that the void* parameter is an interface pointer whose type is only known at run time, but whose interface ID is the parameter named in the iid_is attribute. For example:

```
[object, uuid(b5483f00-4f6c-101b-a1c7-00aa00389acb)]
    interface IBar : IWazoo {
        Bar([in] short i, [in, ref] uuid_t *piid, [out, iid_is(piid)] void ** ppvoid);
        };
```

This can be invoked in C++ as:

```
IFoo* pIF;
pIB->Bar(i, &IID_IFoo, (void*)&pIF);
```

where "IID_IFoo" is the name of a constant whose value is the interface ID for IFoo.

1.1.6All methods must return void or HRESULT

Asynchronous methods (and only asynchronous methods) must return void, all others must return HRESULT.

1.1.7The wire_marshal attribute

```
    typedef  [wire_marshal( transmissible_type)]   type_specifier  user_type;
```

[3] This is a non-COM-related Microsoft extension, shown here for completeness.

This attribute is a type attribute used in the IDL file and is somewhat similar in syntax and semantic to the transmit_as attribute. Each user-specific type has a corresponding transmissible type that defines the wire representation.

The user can define his specific type quite freely, (simple types, pointer types and composite types may be used) although some restrictions apply. The main one is that the type object needs to have well defined (fixed) memory size. If the changeable size needs to be accommodated, the type should have a pointer field as opposed to a conformant array; or, it can be a pointer to the interesting type. General restrictions apply as usual. Specific restrictions related to embedding affect the way types can be specified. For more information see the "User type vs. wire type" section.

The [wire_marshal] attribute cannot be used with [allocate()] attribute (directly or indirectly) as the engine doesn't control the memory allocation for the type. Also the wire type cannot be an interface pointer (these may be marshaled directly) or a full pointer (we cannot take care of the aliasing).

The following is a list of additional points regarding wire_marshal:

- The wire type cannot be an interface pointer.
- The wire type cannot be a full pointer.
- The wire type cannot have allocate attribute on it (like [allocate(all_nodes)]).
- The wire type has to have a well defined memory size (cannot be a conformant structure etc.) as we allocate the top level object for the user as usual.
- When the engine delegates responsibility for a wire_marshalable type to the user supplied routines, everything is up to the user including servicing of the possible embedded types that are defined with wire_marshal, user_marshal, transmit_as etc.
- wire_marshal is mutually exclusive with user_marshal, transmit_as or represent_as when applied to the same type.
- Two different user types cannot resolve to the same wire type and vice versa.
- The user type may or may not be rpc-able.
- The user type must be known to MIDL.

1.1.8The user_marshal attribute

```
typedef  [user_marshal( user_type)]   transmissible_type;
```

This attribute is a type attribute used in the ACF file and is somewhat similar in syntax and semantic to the represent_as attribute. Each user-specific type has a corresponding transmissible type that defines the wire representation. Similar to represent_as, in the generated files, each usage of the trasmissible_type name is substituted by the user_type name.

The user can define his specific type quite freely, (simple types, pointer types and composite types may be used) although some restrictions apply. The main one is that the type object needs to have well defined (fixed) memory size. If the changeable size needs to be accommodated, the type should have a pointer field as opposed to a conformant array; or, it can be a pointer to the interesting type. General restrictions apply as usual. Specific restrictions related to embedding affect the way types can be specified. For more information see the "User type vs. wire type" section.

The [user_marshal] attribute cannot be used with [allocate()] attribute (directly or indirectly) as the engine doesn't control the memory allocation for the type. Also the wire type cannot be an interface pointer (these may be marshaled directly) or a full pointer (we cannot take care of the aliasing).

Additional points regarding user_marshal:

- The wire type cannot be an interface pointer.
- The wire type cannot be a full pointer.
- The wire type cannot have allocate attribute on it (like [allocate(all_nodes)]).
- The wire type has to have a well defined memory size (cannot be a conformant structure etc.) as we allocate the top level object for the user as usual.

- When the engine delegates responsibility for a user_marshalable type to the user supplied routines, everything is up to the user including servicing of the possible embedded types that are defined with user_marshal, transmit_as etc.
- user_marshal is mutually exclusive with wire_marshal, transmit_as or represent_as when applied to the same type.
- Two different wire types cannot resolve to the same user type and vice versa.
- The user type may or may not be rpc-able.
- The user type may or may not be known to MIDL.

1.1.9User supplied routines

The routines required by user_marshall have the following prototypes.

<type_name> means a user specific type name. This may be non-rpcable type or even, when used with user_marshal, a type unknown to MIDL at all. The wire type name (the name of transmissible type) is not used here.

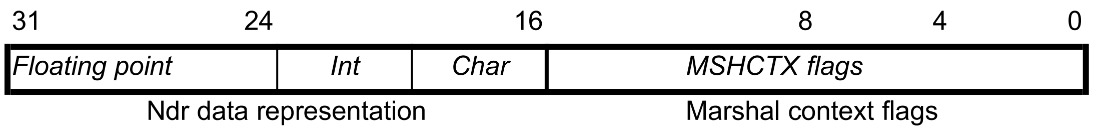
```
unsigned long __RPC_USER  <type_name>_UserSize(
      unsigned long __RPC_FAR *   pFlags,
      unsigned long               StartingSize,
      <type_name>  __RPC_FAR *  pFoo);

unsigned char __RPC_FAR * __RPC_USER  <type_name>_UserMarshal(
      unsigned long   __RPC_FAR * pFlags,
      unsigned char __RPC_FAR *    Buffer,
      <type_name>  __RPC_FAR *  pFoo);

unsigned char __RPC_FAR * __RPC_USER  <type_name>_UserUnmarshal(
      unsigned long   __RPC_FAR * pFlags,
      unsigned char __RPC_FAR *         Buffer,
      <type_name>  __RPC_FAR *  pFoo);

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

The meaning of the arguments is as follows:

pFlags	- pointer to a flag ulong. Flags: local call flag, data rep flag.
pBuffer	- the current buffer pointer,
pFoo	- pointer to a user type object
StartingSize	- the buffer size (offset) before the object

The return value when sizing, marshaling or unmarshaling is the new offset or buffer position. See the function description below for details.

The flags pointed to by the first argument have the following layout.



- Upper word: NDR representation flags as defined by DCE: floating point, endianess and character representations.
- Lower word: marshaling context flags as defined by the COM channel. The flags are defined in the public wtypes.h file (and in wtypes.idl file). Currently the following flags are defined:

```
typedef
enum tagMSHCTX
  {  MSHCTX_LOCAL          = 0,
        MSHCTX_NOSHAREDMEM   = 1,
        MSHCTX_DIFFERENTMACHINE = 2,
        MSHCTX_INPROC         = 3
  } MSHCTX;
```

The flags make it possible to differ the behavior of the routines depending on the context for the RPC call. For example when a handle is remoted in-process it could be sent as a handle (a long), while sending it remotely would mean sending the data related to the handle.

_UserSize

The *_UserSize routine is called when sizing the RPC data buffer before the marshaling on the client or server side. The routine should work in terms of cumulative size. The StartingSize argument is the current buffer offset . The routine should return the cumulative size that includes the possible padding and then the data size. The starting size indicates the buffer offset for the user object and it may or may not be aligned properly. User's routine should account for all padding as necessary. In other words, the routine should return a new offset, after the user object. The sizing routine is not called if the wire size can be computed at the compile time. Note that for most unions, even if there are no pointers, the actual size of the wire representation may be determined only at the runtime.

This routine actually can return an overestimate as long as the marshaling routine does not use more than the sizing routine promised and so the marshaling buffer is not overwritten then or later (by subsequent objects).

_UserMarsahal

The *_UserMarshal routine is called when marshaling the data on the client or server side. The buffer pointer may or may not be aligned upon the entry. The routine should align the buffer pointer appropriately, marshal the data and then return the new buffer pointer position which is at the first "free" byte after the marshaled object. For the complications related to pointees see the next chapter.

Please note that the wire type specification is a contract that determines the actual layout of the data in the buffer. For example, if the conversion is needed and done by the NDR engine, it follows from the wire type definitions how much data would be processed in the buffer for the type.

_UserUnmarshal

The *_UserUnmarshal routine is called when unmarshaling the data on the client or server side. The flags indicate if data conversion is needed (if needed, it has been performed by the NDR engine before the call to the routine). The buffer pointer may or may not be aligned upon the entry. The routine should align the buffer as appropriate, unmarshal the data and then return the new buffer pointer position, which is at the first "free" byte after the unmarshaled object. For the complications related to pointees see the next chapter

_UserFree

The *_UserFree routine is called when freeing the data on the server side. The object itself doesn't get freed as the engine takes care of it. The user shall free the pointees of the top level objects.

1.1.10The library keyword

```
[attributes] library libname {definitions};
```

The library keyword indicates that a type library (See Chapter 14) should be generated.[4] Below is an example library section.

```
[
    uuid(3C591B22-1F13-101B-B826-00DD01103DE1),  // IID_ISome
    object
]
```

[4] Historically the library statement was supported only in a variant of IDL called ODL that was central to OLE Automation.

```
interface ISome : IUnknown
{
    HRESULT DoSomething(void);
}

[
    uuid(3C591B20-1F13-101B-B826-00DD01103DE1),      // LIBID_Lines
    helpstring("Lines 1.0 Type Library"),
    lcid(0x0409),
    version(1.0)
]
library Lines
{
    importlib("stdole.tlb");
    [
        uuid(3C591B21-1F13-101B-B826-00DD01103DE1),  // CLSID_Lines
        helpstring("Lines Class"),
        appobject
    ]
    coclass Lines
    {
        [default] interface ISome;
        interface IDispatch;
    }
}
```

1.2Mapping from ORPC IDL to DCE RPC IDL.

From the above extensions, and the wire representation definitions, one can conclude the following rules for converting ORPC IDL files to DCE IDL files:

1. Remove the [object] attribute from the interface definition.
2. Insert "[in] handle_t h" as the first argument of each method, "[in] ORPCTHIS *_orpcthis" as the second, and "[out] ORPCTHAT *_orpcthat" as the third.
3. Manually insert declarations for the operations that were inherited, if any. You may want to make the method names unique, unless the EPV invocation style is always going to be used. One way to do this is to prefix each method with the name of the interface. (Note that the IUnknown methods will never be called, as the IRemUnknown interface is used instead.)
4. Replace each occurrence of a type name derived from an interface name, or an [iid_is] qualified void* with OBJREF. Remove [iid_is] attributes.

1.2.1An Example

Object RPC Style

```
[object, uuid(b5483f00-4f6c-101b-a1c7-00aa00389acb)]
interface IFoo: IUnknown
    {
    HRESULT Bar([in] short i, [in] IBozo* pIB, [out] IWaz** ppIW);
    HRESULT Zork([in, ref] UUID* iid,  [out, iid_is(iid)] void** ppvoid);
    };
```

DCE style

```
[uuid(b5483f00-4f6c-101b-a1c7-00aa00389acb)]
interface IFoo
    {
    HRESULT IFoo_QueryInterface([in] handle_t h, [in] ORPCTHIS* _orpcthis, [out] ORPCTHAT* _orpcthat, [in, ref] UUID* iid,
        [out] OBJREF** ppOR);
    ULONG IFoo_AddRef([in] handle_t, [in] ORPCTHIS* _orpcthis, [out] ORPCTHAT* _orpcthat);
    ULONG IFoo_Release([in] handle_t, [in] ORPCTHIS* _orpcthis, [out] ORPCTHAT* _orpcthat);
    HRESULT IFoo_Bar([in] handle_t h, [in] ORPCTHIS* _orpcthis, [out] ORPCTHAT* _orpcthat, [in, ref] OBJREF* porIB, [out,
        ref] OBJREF** pporIW);
    HRESULT IFoo_Zork([in]handle_t h, [in] ORPCTHIS* _orpcthis, [out] ORPCTHAT* _orpcthat, [in, ref] UUID* iid, [out]
        OBJREF** ppvoid);
    };
```

See Chapter 15 Network Protocol for information on the ORPCTHIS and ORPCTHAT structures and the IRemUnknown interface.

This page intentionally left blank.