

[MS-DTYP]: Windows Data Types

Intellectual Property Rights Notice for Protocol Documentation

- This protocol documentation is covered by Microsoft copyrights. Regardless of any other terms that are contained in the terms of use for the Microsoft website that hosts this documentation, you may make copies of it in order to develop implementations of the protocols, and may distribute portions of it in your implementations of the protocols or your documentation as necessary to properly document the implementation. This permission also applies to any documents that are referenced in the protocol documentation.
- Microsoft does not claim any trade secret rights in this documentation.
- Microsoft has patents that may cover your implementations of the protocols. Neither this notice nor Microsoft's delivery of the documentation grants any licenses under those or any other Microsoft patents. If you are interested in obtaining a patent license, please contact protocol@microsoft.com.
- The names of companies and products contained in this documentation may be covered by trademarks or similar intellectual property rights. This notice does not grant any licenses under those rights.
- All other rights are reserved, and this notice does not grant any rights other than specifically described above, whether by implication, estoppel, or otherwise.

This protocol documentation is intended for use in conjunction with publicly available standard specifications, network programming art, and Microsoft Windows distributed systems concepts, and assumes that the reader either is familiar with the aforementioned material or has immediate access to it.

A protocol specification does not require the use of Microsoft programming tools or programming environments in order for you to develop an implementation. If you have access to Microsoft programming tools and environments you are free to take advantage of them.

Revision Summary

Date	Revision History	Revision Class	Comments
02/14/2008	3.1.2	Editorial	Revised and edited the technical content.

Table of Contents

1	Introduction	5
1.1	Glossary	5
1.2	References	5
1.2.1	Normative References	5
1.2.2	Informative References.....	6
1.3	Relationship to Protocols and Other Structures	6
1.4	Applicability Statement	6
1.5	Versioning and Localization.....	6
1.6	Vendor-Extensible Fields	6
2	Data Types.....	7
2.1	Common Base Types	7
2.1.1	BIT	7
2.1.2	byte.....	7
2.1.3	handle_t.....	7
2.1.4	Integer Types.....	7
2.1.4.1	__int8	8
2.1.4.2	__int16.....	8
2.1.4.3	__int32.....	8
2.1.4.4	__int64.....	8
2.1.4.5	hyper	8
2.1.5	octet.....	8
2.1.6	wchar_t.....	8
2.2	Common Data Types	8
2.2.1	BOOL.....	8
2.2.2	BOOLEAN	9
2.2.3	BSTR	9
2.2.4	BYTE.....	9
2.2.5	CHAR.....	9
2.2.6	DOUBLE	10
2.2.7	DWORD.....	10
2.2.8	DWORD_PTR.....	10
2.2.9	DWORD32	10
2.2.10	DWORD64	11
2.2.11	DWORDLONG.....	11
2.2.12	error_status_t.....	11
2.2.13	FLOAT	11
2.2.14	HANDLE	11
2.2.15	HCALL.....	12
2.2.16	HRESULT	12
2.2.17	INT	12
2.2.18	INT8	13
2.2.19	INT16	13
2.2.20	INT32	13
2.2.21	INT64	13
2.2.22	LMCSTR	14
2.2.23	LMSTR	14
2.2.24	LONG.....	14
2.2.25	LONGLONG	14
2.2.26	LONG_PTR.....	14
2.2.27	LONG32	15
2.2.28	LONG64	15

2.2.29	LPCSTR	15
2.2.30	LPCTSTR	16
2.2.31	LPCWSTR	16
2.2.32	LPSTR	16
2.2.33	LPTSTR	17
2.2.34	LPWSTR	17
2.2.35	NET_API_STATUS	17
2.2.36	NTSTATUS	18
2.2.37	PCONTEXT_HANDLE	18
2.2.38	QWORD	18
2.2.39	RPC_BINDING_HANDLE	19
2.2.40	SHORT	19
2.2.41	STRING	19
2.2.42	TCHAR	20
2.2.43	TIME	20
2.2.44	UCHAR	20
2.2.45	UINT	20
2.2.46	UINT8	21
2.2.47	UINT16	21
2.2.48	UINT32	21
2.2.49	UINT64	21
2.2.50	ULONG	22
2.2.51	ULONG_PTR	22
2.2.52	ULONG32	22
2.2.53	ULONG64	23
2.2.54	ULONGLONG	23
2.2.55	UNICODE	23
2.2.56	USHORT	23
2.2.57	VOID	23
2.2.58	WCHAR	24
2.2.59	WORD	24
2.3	Common Data Structures	24
2.3.1	FILETIME	24
2.3.2	GUID, UUID	25
2.3.2.1	Curly Braced GUID String Syntax	25
2.3.3	LARGE_INTEGER	25
2.3.4	LCID	25
2.3.5	RPC_UNICODE_STRING	26
2.3.6	SYSTEMTIME	26
2.3.7	UINT128	27
2.3.8	ULARGE_INTEGER	27
2.3.9	UNICODE_STRING	27
2.4	Constructed Security Types	28
2.4.1	SID_IDENTIFIER_AUTHORITY	28
2.4.2	SID	28
2.4.3	ACCESS_MASK	29
2.4.4	ACE	30
2.4.4.1	ACE_HEADER	30
2.4.4.2	ACCESS_ALLOWED_ACE	33
2.4.4.3	ACCESS_ALLOWED_OBJECT_ACE	33
2.4.4.4	ACCESS_DENIED_ACE	34
2.4.4.5	ACCESS_ALLOWED_CALLBACK_ACE	35
2.4.4.6	ACCESS_DENIED_CALLBACK_ACE	35
2.4.4.7	ACCESS_ALLOWED_CALLBACK_OBJECT_ACE	36
2.4.4.8	ACCESS_DENIED_CALLBACK_OBJECT_ACE	37

2.4.4.9	SYSTEM_AUDIT_ACE	39
2.4.4.10	SYSTEM_AUDIT_CALLBACK_ACE	39
2.4.4.11	SYSTEM_MANDATORY_LABEL_ACE	40
2.4.4.12	SYSTEM_AUDIT_CALLBACK_OBJECT_ACE	41
2.4.5	ACL	42
2.4.6	SECURITY_DESCRIPTOR	43
2.4.7	SECURITY_INFORMATION	45
2.4.8	RPC_SID	46
2.5	Additional Information for Security Types	46
2.5.1	Security Descriptor Description Language	46
2.5.1.1	Syntax.....	47
2.5.2	Security Descriptor Algorithms.....	54
2.5.2.1	Access Check Algorithm Pseudocode	54
2.5.2.2	Algorithm for Creating a Security Descriptor	57
2.5.2.3	CreateSecurityDescriptor.....	57
2.5.2.4	ComputeACL.....	59
2.5.2.5	ContainsInheritableACEs	61
2.5.2.6	ComputeInheritedACLfromParent.....	62
2.5.2.7	ComputeInheritedACLfromCreator	63
2.5.2.8	PreProcessACLfromCreator	64
2.5.2.9	PostProcessACL.....	64
3	Structure Examples	66
4	Security Considerations	67
5	Appendix A: Full MS-DTYP IDL.....	68
6	Appendix B: Windows Behavior	74
7	Index.....	76

1 Introduction

This document is a collection of commonly used data types. These data types fall into two basic categories: common base types and common data types. The common base types are those types that Microsoft compilers natively support. The common data types are data types that are frequently used by many protocols. These data types are user-defined types.

1.1 Glossary

The following terms are defined in [\[MS-GLOS\]](#):

Globally Unique Identifier (GUID)
Interface Definition Language (IDL)
Interface Identifier (IID)
Little-Endian
Microsoft Interface Definition Language (MIDL)
Remote Procedure Call (RPC)
Unicode
Universally Unique Identifier (UUID)

The following terms are specific to this document:

MAY, SHOULD, MUST, SHOULD NOT, MUST NOT: These terms (in all caps) are used as described in [\[RFC2119\]](#). All statements of optional behavior use either MAY, SHOULD, or SHOULD NOT.

1.2 References

1.2.1 Normative References

We conduct frequent surveys of the normative references to assure their continued availability. If you have any issue with finding a normative reference, please contact dochelp@microsoft.com. We will assist you in finding the relevant information. Please check the archive site, <http://msdn2.microsoft.com/en-us/library/E4BD6494-06AD-4aed-9823-445E921C9624>, as an additional source.

[C706] The Open Group, "DCE 1.1: Remote Procedure Call", C706, August 1997, <http://www.opengroup.org/public/pubs/catalog/c706.htm>

[IEEE754] Institute of Electrical and Electronics Engineers, "Standard for Binary Floating-Point Arithmetic", IEEE 754-1985, October 1985, <http://grouper.ieee.org/groups/754/>

[ISO-9899] International Organization for Standardization, "Programming Languages - C", ISO/IEC 9899:TC2, May 2005, <http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1124.pdf>

[MS-ERREF] Microsoft Corporation, "[Windows Error Codes](#)", January 2007.

[MS-GLOS] Microsoft Corporation, "[Windows Protocols Master Glossary](#)", March 2007.

[MS-SECO] Microsoft Corporation, "[Windows Security Overview](#)", January 2007.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997, <http://www.ietf.org/rfc/rfc2119.txt>

[RFC2781] Hoffman, P. and Yergeau, F., "UTF-16, an encoding of ISO 10646", RFC 2781, February 2000, <http://www.ietf.org/rfc/rfc2781.txt>

[RFC4122] Leach, P., Mealling, M., and Salz, R., "A Universally Unique Identifier (UUID) URN Namespace", RFC 4122, July 2005, <http://www.ietf.org/rfc/rfc4122.txt>

[RFC4234] Crocker, D., Ed. and Overell, P., "Augmented BNF for Syntax Specifications: ABNF", RFC 4234, October 2005, <http://www.ietf.org/rfc/rfc4234.txt>

1.2.2 Informative References

[DALB] Dalbey, J., "Pseudocode Standard", http://www.csc.calpoly.edu/~jdalbey/SWE/pdl_std.html

[MSDN-ACCTOKENS] Microsoft Corporation, "Access Tokens", <http://msdn2.microsoft.com/en-us/library/aa374909.aspx>

[MS-LCID] Microsoft Corporation, "[Windows Language Code Identifier \(LCID\) Reference](#)", July 2007.

1.3 Relationship to Protocols and Other Structures

The data types in this document are used by many protocols.

1.4 Applicability Statement

Not applicable.

1.5 Versioning and Localization

Not applicable.

1.6 Vendor-Extensible Fields

There are no vendor-extensible fields.

2 Data Types

The following sections describe data types that include common base types, data types, and data structures.

Many protocols are intended to be extensions of local programming models. Other protocols have a distinct purpose but share many common elements. This section is a discussion of data types common to many protocols.

There are cases where a component may veer from normal practice. If that is the case, it is specified in the relevant specification.

There are often integer names that may have an alias. These names are interchangeable and there is no difference in using either term.

Where the Windows behavior is unclear, it is described.

2.1 Common Base Types

This section contains commonly used primitive data types that may not have an ANSI/ISO C equivalent (as specified in [\[ISO-9899\]](#) for the standard C definitions) but are supported by the Microsoft C/C++ compiler.

Unless explicitly noted, any integer, either unsigned or signed, is in **little-endian** order.

2.1.1 BIT

A **bit** is a single binary digit, the smallest primitive element of any data structure.

2.1.2 byte

The byte type specifies an 8-bit data item.

A **byte** is a base **IDL** type as specified in [\[C706-Ch4InterfaceDef\]](#) section 4.2.9.5. A byte item is opaque, in that its contents are not interpreted, as a character data type might be.

2.1.3 handle_t

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 declarations (as a function-return-type specifier and a parameter-type specifier).

2.1.4 Integer Types

Microsoft C/C++ supports different sized integer types. You can declare 8-, 16-, 32-, or 64-bit integer variables by using the `__intn` type specifier, where `n` is 8, 16, 32, or 64.

The types `__int8`, `__int16`, and `__int32` are synonyms for the ANSI/ISO C types (as specified in [\[ISO-9899\]](#)) that have the same size, and are useful for writing portable code that behaves identically across multiple platforms. The `__int8` data type is synonymous with type `char`, `__int16` is synonymous with type `short`, and `__int32` is synonymous with type `int`. The `__int64` type has no ANSI equivalent.

2.1.4.1 `__int8`

An 8-bit signed integer (range: -128 through 127 decimal). The first bit (Most Significant Bit (MSB)) is the signing bit. This type MAY be specified as signed or unsigned.

2.1.4.2 `__int16`

A 16-bit signed integer (range: -32768 through 32767 decimal). The first bit (Most Significant Bit (MSB)) is the signing bit. This type MAY be specified as signed or unsigned.

2.1.4.3 `__int32`

A 32-bit signed integer (range: -2147483648 through 2147483647 decimal). The first bit (Most Significant Bit (MSB)) is the signing bit. This type MAY be specified as signed or unsigned.

2.1.4.4 `__int64`

A 64-bit signed integer (range: -9223372036854775808 through 9223372036854775807 decimal). The first bit (Most Significant Bit (MSB)) is the signing bit. This type MAY be specified as signed or unsigned.

2.1.4.5 `hyper`

The keyword **hyper** indicates a 64-bit integer that can be declared as either signed or unsigned.

2.1.5 `octet`

The `octet` type specifies an 8-bit data item.

An **octet** is an 8-bit data type as specified in [\[C706-Ch14TransSyntaxNDR\]](#) section 14.2.

2.1.6 `wchar_t`

A 16-bit, UTF-16 (as specified in [\[RFC2781\]](#)) encoded **UNICODE** character for use with the **Microsoft Interface Definition Language (MIDL)** compiler.

This type is declared as follows:

```
typedef unsigned short wchar_t;
```

2.2 Common Data Types

This section contains simple data types that are defined by either a C/C++ typedef or #define statement. The data types in this section are essentially aliases for C/C++ primitive data types.

2.2.1 `BOOL`

A **BOOL** is a 32-bit field that is set to 1 to indicate **TRUE**, or 0 to indicate **FALSE**.

This type is declared as follows:

```
typedef int BOOL, *PBOOL, *LPBOOL;
```

2.2.2 BOOLEAN

A **BOOLEAN** is an 8-bit field that is set to 1 to indicate **TRUE**, or 0 to indicate **FALSE**.

This type is declared as follows:

```
typedef BYTE BOOLEAN, *PBOOLEAN;
```

2.2.3 BSTR

A **BSTR** is a pointer to a null-terminated character string in which the string length is stored with the string. Because the length is stored with the string, **BSTR** variables can contain embedded null characters. For example:

```
[4 bytes (length prefix)],  
wchar_t[length], [\0]
```

This type is declared as follows:

```
typedef WCHAR* BSTR;
```

2.2.4 BYTE

A **BYTE** is an 8-bit, unsigned value that corresponds to a single octet in a network protocol.

This type is declared as follows:

```
typedef unsigned char BYTE, *PBYTE, *LPBYTE;
```

2.2.5 CHAR

A **CHAR** is an 8-bit block of data that typically contains a Windows (ANSI) character.

This type is declared as follows:

```
typedef char CHAR, *PCHAR;
```

2.2.6 DOUBLE

A **DOUBLE** is an 8-byte, double-precision, floating-point number that represents a double-precision, 64-bit [IEEE754](#) value with the approximate range: $+/-5.0 \times 10^{-324}$ through $+/-1.7 \times 10^{308}$.

The **DOUBLE** type can also represent NAN (Not a Number), positive and negative infinity, or positive and negative 0.

This type is declared as follows:

```
typedef double DOUBLE;
```

2.2.7 DWORD

A **DWORD** is a 32-bit unsigned integer (range: 0 through 4294967295 decimal). Because a **DWORD** is unsigned, its first bit (Most Significant Bit (MSB)) is not reserved for signing.

This type is declared as follows:

```
typedef unsigned long DWORD, *PDWORD, *LPDWORD;
```

2.2.8 DWORD_PTR

A **DWORD_PTR** is an unsigned long type used for pointer precision. It is used when casting a pointer to an unsigned long type to perform pointer arithmetic. **DWORD_PTR** is also commonly used for general 32-bit parameters that have been extended to 64 bits in 64-bit Windows. For more information, see [ULONG_PTR](#).

This type is declared as follows:

```
typedef ULONG_PTR DWORD_PTR;
```

2.2.9 DWORD32

A **DWORD32** is a 32-bit unsigned integer.

This type is declared as follows:

```
typedef unsigned int DWORD32;
```

2.2.10 DWORD64

A **DWORD64** is a 64-bit unsigned integer.

This type is declared as follows:

```
typedef __int64 DWORD64;
```

2.2.11 DWORDLONG

A **DWORDLONG** is a 64-bit unsigned integer (range: 0 through 18446744073709551615 decimal).

This type is declared as follows:

```
typedef ULONGLONG DWORDLONG, *PDWORDLONG;
```

2.2.12 error_status_t

The **error_status_t** return type is used for all methods. This is a Win32 error code.

This type is declared as follows:

```
typedef unsigned long error_status_t;
```

2.2.13 FLOAT

This type is declared as follows:

```
typedef float FLOAT;
```

2.2.14 HANDLE

Handle to an object.

This type is declared as follows:

```
typedef void* HANDLE;
```

2.2.15 HCALL

An **HCALL** is an alias for a [DWORD](#) used to specify a **handle** to a call, typically used in telephony-related applications.

An **HCALL** is a 32-bit unsigned integer used to store a handle to a call.

This type is declared as follows:

```
typedef DWORD HCALL;
```

2.2.16 HRESULT

An **HRESULT** is a 32-bit value that is used to describe an error or warning and contains the following fields:

- A 1-bit code that indicates severity, where 0 represents success and 1 represents failure.
- A 4-bit reserved value.
- An 11-bit code, also known as a facility code, that indicates responsibility for the error or warning.
- A 16-bit code that describes the error or warning.

For details on **HRESULT** values, see [\[MS-ERREF\]](#).

This type is declared as follows:

```
typedef DWORD HRESULT;
```

2.2.17 INT

An **INT** is a 32-bit signed integer (range: -2147483648 through 2147483647 decimal).

This type is declared as follows:

```
typedef int INT, *LPINT;
```

2.2.18 INT8

An **INT8** is an 8-bit signed integer (range: -128 through 127 decimal). The first bit (Most Significant Bit (MSB)) is the signing bit.

This type is declared as follows:

```
typedef signed char INT8;
```

2.2.19 INT16

An **INT16** is a 16-bit signed integer (range: -32768 through 32767 decimal). The first bit (Most Significant Bit (MSB)) is the signing bit.

This type is declared as follows:

```
typedef signed short INT16;
```

2.2.20 INT32

An **INT32** is a 32-bit signed integer (range: -2147483648 through 2147483647 decimal). The first bit (Most Significant Bit (MSB)) is the signing bit.

This type is declared as follows:

```
typedef signed int INT32;
```

2.2.21 INT64

An **INT64** is a 64-bit signed integer (range: -9223372036854775808 through 9223372036854775807 decimal). The first bit (Most Significant Bit (MSB)) is the signing bit.

This type is declared as follows:

```
typedef signed __int64 INT64;
```

2.2.22 LMCSTR

A **LMCSTR** is a 32-bit pointer to a constant null-terminated string of 16-bit [UNICODE](#) characters.

This type is declared as follows:

```
typedef const wchar_t* LMCSTR;
```

2.2.23 LMSTR

A **LMSTR** is a 32-bit pointer to a null-terminated string of 16-bit [UNICODE](#) characters.

This type is declared as follows:

```
typedef WCHAR* LMSTR;
```

2.2.24 LONG

A **LONG** is a 32-bit signed integer, in twos-complement format (range: -2147483648 through 2147483647 decimal). The first bit (Most Significant Bit (MSB)) is the signing bit.

This type is declared as follows:

```
typedef long LONG, *PLONG, *LPLONG;
```

2.2.25 LONGLONG

A **LONGLONG** is a 64-bit signed integer (range: -9223372036854775808 through 9223372036854775807 decimal).

This type is declared as follows:

```
typedef signed __int64 LONGLONG;
```

2.2.26 LONG_PTR

A **LONG_PTR** is a long type used for pointer precision. It is used when casting a pointer to a long type to perform pointer arithmetic.

Note In a 64-bit environment (indicated below by a TRUE value for `_WIN64`), the **LONG_PTR** MUST be defined as an [__int64](#), not as a [LONG](#).

```
#ifdef _WIN64
typedef __int64 LONG_PTR;
#else
typedef LONG LONG_PTR;
#endif
```

2.2.27 LONG32

A **LONG32** is a 32-bit signed integer.

This type is declared as follows:

```
typedef signed int LONG32;
```

2.2.28 LONG64

A **LONG64** is a 64-bit signed integer.

This type is declared as follows:

```
typedef signed __int64 LONG64;
```

2.2.29 LPCSTR

A **LPCSTR** is a 32-bit pointer to a constant null-terminated string of 8-bit Windows (ANSI) characters.

This type is declared as follows:

```
typedef const char* LPCSTR;
```

2.2.30 LPCTSTR

If **Unicode** support is enabled (generally via a #define statement or a compiler option), a **LPCSTR** is a 32-bit pointer to a constant null-terminated string of 16-bit **Unicode** characters.

If **Unicode** support is not enabled, a **LPCSTR** is a 32-bit pointer to a constant null-terminated string of 8-bit Windows (ANSI) characters.

Conceptually, if the existence of Unicode support could be tested by the (hypothetical) conditional directive "#ifdef Unicode", this type would be defined as follows:

```
#ifdef Unicode
typedef const wchar_t* LPCTSTR;
#else
typedef const char* LPCTSTR;
#endif
```

2.2.31 LPCWSTR

A **LPCWSTR** is a 32-bit pointer to a constant string of 16-bit **UNICODE** characters, which MAY be null-terminated.

This type is declared as follows:

```
typedef const wchar_t* LPCWSTR;
```

2.2.32 LPSTR

The **LPSTR** type and its alias **PSTR** specify a pointer to an array of 8-bit characters, which MAY be terminated by a null character.

In some protocols, it may be acceptable to not terminate with a null character, and this option will be indicated in the specification. In this case, the **LPSTR** or **PSTR** type MUST either be tagged with the IDL modifier [string], that indicates string semantics, or be accompanied by an explicit length specifier, for example [size_is()].

The format of the characters MUST be specified by the protocol that uses them. Two common 8-bit formats are ANSI and UTF-8.

A 32-bit pointer to a string of 8-bit characters, which MAY be null-terminated.

This type is declared as follows:

```
typedef char* PSTR, *LPSTR;
```

2.2.33 LPTSTR

If [UNICODE](#) support is enabled (generally via a `#define` statement or a compiler option), a **LPTSTR** is a 32-bit pointer to a null-terminated string of 16-bit **UNICODE** characters.

If **UNICODE** support is not enabled, a **LPTSTR** is a 32-bit pointer to a null-terminated string of 8-bit Windows (ANSI) characters.

Conceptually, if the existence of **UNICODE** support could be tested by the (hypothetical) conditional directive `"#ifdef Unicode"`, this type would be defined as follows:

```
#ifdef Unicode
typedef WCHAR* LPTSTR;
#else
typedef CHAR* LPTSTR;
#endif
```

2.2.34 LPWSTR

The **LPWSTR** type and its alias **PWSTR** specifies a pointer to a sequence of [UNICODE](#) characters, which MAY be terminated by a null character (usually referred to as "null-terminated Unicode").

In some protocols, it may be acceptable to not terminate with a null character, and this option will be indicated in the protocol specification. In this case, the **LPWSTR** or **PWSTR** type MUST either be tagged with the IDL modifier `[string]`, that indicates string semantics, or be accompanied by an explicit length specifier, as specified in the [UNICODE STRING \(section 2.3.9\)](#) structure.

A 32-bit pointer to a string of 16-bit Unicode characters, which MAY be null-terminated.

This type is declared as follows:

```
typedef wchar_t* LPWSTR, *PWSTR;
```

2.2.35 NET_API_STATUS

The **NET_API_STATUS** type is commonly used as the return value of **RPC** methods in Microsoft network protocols. See the Win32 error codes as specified in [\[MS-ERREF\]](#) for details.

This type is declared as follows:

```
typedef DWORD NET_API_STATUS;
```

2.2.36 NTSTATUS

NTSTATUS is a standard 32-bit datatype for system-supplied status code values.

NTSTATUS values are used to communicate system information. They are of four types: success values, information values, warnings, and error values, as specified in [\[MS-ERREF\]](#).

This type is declared as follows:

```
typedef long NTSTATUS;
```

2.2.37 PCONTEXT_HANDLE

The **PCONTEXT_HANDLE** type keeps state information associated with a given client on a server. The state information is called the server's context. Clients can obtain a context handle to identify the server's context for their individual RPC sessions.

A context handle must be of the void * type, or a type that resolves to void *. The server program casts it to the required type.

The IDL attribute [**context_handle**], as specified in [\[C706\]](#), is used to declare **PCONTEXT_HANDLE**.

An interface that uses a context handle must have a binding handle for the initial binding, which has to take place before the server can return a context handle. The [handle_t](#) type is one of the predefined types of the interface definition language (IDL), which is used to create a binding handle.

This type is declared as follows:

```
typedef [context_handle] void* PCONTEXT_HANDLE, *PPCONTEXT_HANDLE;
```

2.2.38 QWORD

A **QWORD** is a 64-bit unsigned integer.

This type is declared as follows:

```
typedef unsigned __int64 QWORD;
```

2.2.39 RPC_BINDING_HANDLE

A **RPC_BINDING_HANDLE** is an untyped 32-bit pointer containing information that the RPC run-time library uses to access binding information.

The **RPC_BINDING_HANDLE** data type declares a binding handle containing information that the RPC run-time library uses to access binding information.

The run-time library uses binding information to establish a client/server relationship that allows the execution of remote procedure calls. Based on the context in which a binding handle is created, it is considered a server-binding handle or a client-binding handle.

A server-binding handle contains the information necessary for a client to establish a relationship with a specific server. Any number of RPC API run-time routines return a server-binding handle that can be used for making a remote procedure call.

A client-binding handle cannot be used to make a remote procedure call. The RPC run-time library creates and provides a client-binding handle to a called-server procedure (also called a server-manager routine) as the **RPC_BINDING_HANDLE** parameter. The client-binding handle contains information about the calling client.

This type is declared as follows:

```
typedef void* RPC_BINDING_HANDLE;
```

2.2.40 SHORT

A **SHORT** is a 16-bit signed integer (range: -32768 through 32767 decimal). The first bit (Most Significant Bit (MSB)) is the signing bit.

This type is declared as follows:

```
typedef short SHORT;
```

2.2.41 STRING

Unless otherwise noted, a **STRING** is a **UCHAR** buffer that represents a null-terminated **ASCII string**.

This type is declared as follows:

```
typedef UCHAR* STRING;
```

2.2.42 TCHAR

If [UNICODE](#) support is enabled (generally via a `#define` statement or a compiler option), a **TCHAR** is a [WCHAR](#); otherwise, it is a [CHAR](#).

Conceptually, if the existence of **UNICODE** support could be tested by the (hypothetical) conditional directive `"#ifdef Unicode"`, this type would be defined as follows:

```
#ifdef Unicode
typedef WCHAR TCHAR;
#else
typedef CHAR TCHAR;
#endif
```

2.2.43 TIME

A **TIME** is a 64-bit integer that represents an absolute time or a time interval. Times are specified in units of 100 milliseconds.

A positive value expresses an absolute time, where the base time is the beginning of the year 1601 A.D. in the Gregorian calendar. A negative value expresses a time interval relative to some base time, typically the current time.

This type is declared as follows:

```
typedef __int64 TIME;
```

2.2.44 UCHAR

A **UCHAR** is an 8-bit integer with the range: 0 through 255 decimal. Because a **UCHAR** is unsigned, its first bit (Most Significant Bit (MSB)) is not reserved for signing.

This type is declared as follows:

```
typedef unsigned char UCHAR, *PUCHAR;
```

2.2.45 UINT

A **UINT** is a 32-bit unsigned integer (range: 0 through 4294967295 decimal). Because a **UINT** is unsigned, its first bit (Most Significant Bit (MSB)) is not reserved for signing.

This type is declared as follows:

```
typedef unsigned int UINT;
```

2.2.46 UINT8

A **UINT8** is an 8-bit unsigned integer (range: 0 through 255 decimal). Because a **UINT8** is unsigned, its first bit (Most Significant Bit (MSB)) is not reserved for signing.

This type is declared as follows:

```
typedef unsigned char UINT8;
```

2.2.47 UINT16

A **UINT16** is a 16-bit unsigned integer (range: 0 through 65535 decimal). Because a **UINT16** is unsigned, its first bit (Most Significant Bit (MSB)) is not reserved for signing.

This type is declared as follows:

```
typedef unsigned short UINT16;
```

2.2.48 UINT32

A **UINT32** is a 32-bit unsigned integer (range: 0 through 4294967295 decimal). Because a **UINT32** is unsigned, its first bit (Most Significant Bit (MSB)) is not reserved for signing.

This type is declared as follows:

```
typedef unsigned int UINT32;
```

2.2.49 UINT64

A **UINT64** is a 64-bit unsigned integer (range: 0 through 18446744073709551615 decimal). Because a **UINT64** is unsigned, its first bit (Most Significant Bit (MSB)) is not reserved for signing.

This type is declared as follows:

```
typedef unsigned __int64 UINT64;
```

2.2.50 ULONG

A **ULONG** is a 32-bit unsigned integer (range: 0 through 4294967295 decimal). Because a **ULONG** is unsigned, its first bit (Most Significant Bit (MSB)) is not reserved for signing.

This type is declared as follows:

```
typedef unsigned long ULONG, *PULONG;
```

2.2.51 ULONG_PTR

A **ULONG_PTR** is an unsigned long type used for pointer precision. It is used when casting a pointer to a long type to perform pointer arithmetic.

Note In a 64-bit environment (indicated below by a TRUE value for `_WIN64`), the **ULONG_PTR** MUST be defined as an unsigned [__int64](#), not as a [ULONG](#).

```
#ifdef _WIN64
typedef unsigned __int64 ULONG_PTR;
#else
typedef ULONG ULONG_PTR;
#endif
```

2.2.52 ULONG32

A **ULONG32** is an unsigned **LONG32**.

This type is declared as follows:

```
typedef unsigned int ULONG32;
```

2.2.53 ULONG64

A **ULONG64** is a 64-bit unsigned integer (range: 0 through 18446744073709551615 decimal). Because a **ULONG64** is unsigned, its first bit (Most Significant Bit (MSB)) is not reserved for signing.

This type is declared as follows:

```
typedef unsigned __int64 ULONG64;
```

2.2.54 ULONGLONG

A **ULONGLONG** is a 64-bit unsigned integer (range: 0 through 18446744073709551615 decimal). Because a **ULONGLONG** is unsigned, its first bit (Most Significant Bit (MSB)) is not reserved for signing.

This type is declared as follows:

```
typedef unsigned __int64 ULONGLONG;
```

2.2.55 UNICODE

A single **UNICODE** character.

This type is declared as follows:

```
typedef wchar_t UNICODE;
```

2.2.56 USHORT

A **USHORT** is a 16-bit unsigned integer (range: 0 through 65535 decimal). Because a **USHORT** is unsigned, its first bit (Most Significant Bit (MSB)) is not reserved for signing.

This type is declared as follows:

```
typedef unsigned short USHORT;
```

2.2.57 VOID

VOID is an alias for **void**.

This type is declared as follows:

```
typedef void VOID, *PVOID;
```

2.2.58 WCHAR

A **WCHAR** is a 16-bit [UNICODE](#) character.

This type is declared as follows:

```
typedef wchar_t WCHAR, *PWCHAR;
```

2.2.59 WORD

A **WORD** is a 16-bit unsigned integer (range: 0 through 65535 decimal). Because a **WORD** is unsigned, its first bit (Most Significant Bit (MSB)) is not reserved for signing.

This type is declared as follows:

```
typedef unsigned short WORD, *PWORD, *LPWORD;
```

2.3 Common Data Structures

This section contains common data structures that are defined in either C, C++, or ABNF.

2.3.1 FILETIME

The **FILETIME** structure is a 64-bit value that represents the number of 100-nanosecond intervals that have elapsed since January 1, 1601, in Coordinated Universal Time (UTC) format.

```
typedef struct {
    DWORD dwLowDateTime;
    DWORD dwHighDateTime;
} FILETIME,
*PFILETIME,
*LPFILETIME;
```

dwLowDateTime: A 32-bit unsigned integer that contains the low-order bits of the file time.

dwHighDateTime: A 32-bit unsigned integer that contains the high-order bits of the file time.

2.3.2 GUID, UUID

A **GUID** is a 16-byte structure, intended to serve as a unique identifier for an object. Compatible with a DCE **UUID** (as specified in [\[C706\]](#) section [\[C706-AppendixAUUID\]](#)) it is generally treated as an opaque series of bytes. A GUID is defined to be compatible with a UUID as follows:

```
typedef struct {
    unsigned long Data1;
    unsigned short Data2;
    unsigned short Data3;
    byte Data4[8];
} GUID,
  UUID,
  *PGUID;
```

2.3.2.1 Curly Braced GUID String Syntax

The curly braced GUID string syntax is a format commonly used for a string representation of the GUID type (as specified in section [2.3.2](#)) is described by the following ABNF syntax, as specified in [\[RFC4234\]](#).

```
CurlyBraceGuidIdString = "{" UUID "}"
```

Where UUID represents the string form of a UUID, as specified in [\[RFC4122\]](#) section 3. The non-terminal symbol CurlyBraceGuidIdString represents (that is, generates) strings that satisfy the definition of **curly braced GUID string**.

By way of illustration, the UUID string specified in [\[RFC4122\]](#) section 3 as an example would have the following representation as a curly braced GUID string.

```
{f81d4fae-7dec-11d0-a765-00a0c91e6bf6}
```

2.3.3 LARGE_INTEGER

The **LARGE_INTEGER** structure is used to represent a 64-bit signed integer value.

```
typedef struct _LARGE_INTEGER {
    signed __int64 QuadPart;
} LARGE_INTEGER,
  *PLARGE_INTEGER;
```

2.3.4 LCID

A language code identifier (**LCID**) structure is stored as a **DWORD**. It contains the language identifier in the lower word and contains the sorting identifier (ID), in addition to a reserved value in the upper word. For additional details about the structure and possible values, see the [Windows Language Code Identifier \(LCID\) Reference](#).

This type is declared as follows:

```
typedef DWORD LCID;
```

2.3.5 RPC_UNICODE_STRING

The **RPC_UNICODE_STRING** structure specifies a [UNICODE_STRING](#). This structure is defined by using the interface definition language (IDL) as follows:

```
typedef struct _RPC_UNICODE_STRING {
    unsigned short Length;
    unsigned short MaximumLength;
    [size_is(MaximumLength/2), length_is(Length/2)]
    WCHAR* Buffer;
} RPC_UNICODE_STRING,
*PRPC_UNICODE_STRING;
```

Length: A 16-bit unsigned integer in little-endian format that contains the length, in bytes, of the string pointed to by the **Buffer** member, not including the terminating NULL character (if any). The length MUST be a multiple of 2. The length MAY equal the entire size of the **Buffer** (in which case there is no terminating NULL character). Any method that accesses the **RPC_UNICODE_STRING** structure MUST use the **Length** specified instead of relying on the presence or absence of a NULL character.

MaximumLength: A 16-bit unsigned integer in little-endian format that contains the maximum size, in bytes, of the string pointed to by **Buffer**. The size MUST be a multiple of 2. If not, the server decrements this value by 1. This value MUST not be less than **Length**.

Buffer: A pointer to a wide-character string buffer. If **MaximumLength** field is greater than 0, the buffer MUST contain a non-NULL value. In some protocols, it may be acceptable to not terminate with a NULL character; in these cases, it will be indicated in the specification.

2.3.6 SYSTEMTIME

The **SYSTEMTIME** structure is a date and time, in Coordinated Universal Time (UTC), represented by using individual [WORD](#)-sized structure members for the month, day, year, day of week, hour, minute, second, and millisecond.

```
typedef struct _SYSTEMTIME {
    WORD wYear;
    WORD wMonth;
    WORD wDayOfWeek;
    WORD wDay;
    WORD wHour;
    WORD wMinute;
    WORD wSecond;
    WORD wMilliseconds;
} SYSTEMTIME,
*PSYSTEMTIME;
```

2.3.7 UINT128

The **UINT128** structure is intended to hold 128-bit unsigned integers, such as an IPv6 destination address.

```
typedef struct _UINT128 {
    UINT64 lower;
    UINT64 upper;
} UINT128,
*PUINT128;
```

2.3.8 ULARGE_INTEGER

The **ULARGE_INTEGER** structure is used to represent a 64-bit unsigned integer value.

```
typedef struct _ULARGE_INTEGER {
    unsigned __int64 QuadPart;
} ULARGE_INTEGER,
*PULARGE_INTEGER;
```

2.3.9 UNICODE_STRING

The **UNICODE_STRING** structure specifies a string, which MAY be null-terminated, of 16-bit UTF-16 **UNICODE** characters.

```
typedef struct _UNICODE_STRING {
    USHORT Length;
    USHORT MaximumLength;
    [size_is(MaximumLength/2), length_is(Length/2)]
    WCHAR* Buffer;
} UNICODE_STRING,
*PUNICODE_STRING;
```

Length: A 16-bit unsigned integer in little-endian format that contains the length, in bytes, of the string pointed to by the **Buffer** member, not including the terminating NULL character (if any). The length MUST be a multiple of 2. The length MAY equal the entire size of the **Buffer** (in which case there is no terminating NULL character.) Any method that accesses the **UNICODE_STRING** structure MUST use the **Length** specified instead of relying on the presence or absence of a NULL character.

MaximumLength: A 16-bit unsigned integer in little-endian format that contains the maximum size, in bytes, of the string pointed to by **Buffer**. The size MUST be a multiple of 2. If not, the server decrements this value by 1. This value MUST not be less than **Length**.

Buffer: A pointer to a wide-character string buffer. If **MaximumLength** field is greater than 0, the buffer MUST contain a non-NULL value. In some protocols, it may be acceptable to not terminate with a NULL character; in these cases, it will be indicated in the specification.

2.4 Constructed Security Types

The following types are used to specify structures that are specific to the Windows security model.

2.4.1 SID_IDENTIFIER_AUTHORITY

The **SID_IDENTIFIER_AUTHORITY** structure represents the top-level authority of a security identifier (SID).

```
typedef struct _SID_IDENTIFIER_AUTHORITY {  
    BYTE Value[6];  
} SID_IDENTIFIER_AUTHORITY;
```

Value: A six-element array of 8-bit unsigned integers that specify the top-level authority of a [SID](#), [RPC SID](#), and [LSAPR SID INFORMATION](#).

This value is generally set to {0, 0, 0, 0, 0, 5} for SIDs that are associated with accounts or groups.

The identifier authority value identifies the domain security authority that issued the SID. The following identifier authorities are predefined.

Identifier Authority	Meaning
NULL_SID_AUTHORITY 0x00	The authority is the NULL SID authority. It defines only the NULL well-known-SID: S-1-0-0.
WORLD_SID_AUTHORITY 0x01	The authority is the World SID authority. It only defines the Everyone well-known-SID: S-1-1-0.
LOCAL_SID_AUTHORITY 0x02	The authority is the Local SID authority. It defines only the Local well-known-SID: S-1-2-0.
CREATOR_SID_AUTHORITY 0x03	The authority is the Creator SID authority. It defines the Creator Owner , Creator Group , and Creator Owner Server well-known-SIDs: S-1-3-0, S-1-3-1, and S-1-3-2. These SIDs are used as placeholders in an access control list (ACL) and are replaced by the user, group, and machine SIDs of the security principal.
NON_UNIQUE_AUTHORITY 0x04	Not used.
NT_AUTHORITY 0x05	The authority is the Windows NT security subsystem SID authority. It defines all other SIDs in the forest.

2.4.2 SID

The **SID** structure defines a security identifier (SID), which is a variable-length byte array that uniquely identifies a security principal. Each security principal has a unique SID that is issued by a security agent. The agent can be a Windows local system or domain. The agent generates the SID when the security principal is created. This structure is defined by using the Interface Definition Language (IDL) as follows:

```
typedef struct _SID {
```

```

    BYTE Revision;
    BYTE SubAuthorityCount;
    SID_IDENTIFIER_AUTHORITY IdentifierAuthority;
    [size_is(SubAuthorityCount)] ULONG SubAuthority[*];
} SID,
*PSID;

```

Revision: An 8-bit unsigned integer that defines the revision level of the **SID** structure. This value **MUST** be set to 0x01.

SubAuthorityCount: The number of elements in the **SubAuthority** array. The maximum number of elements allowed is 15.

IdentifierAuthority: A [SID_IDENTIFIER_AUTHORITY \(section 2.4.1\)](#) structure that contains information that indicates the authority under which the SID was created. It describes the entity that created the SID and manages the account.

SubAuthority: A variable length array of unsigned 32-bit integers that uniquely identifies a principal relative to the **IdentifierAuthority**. Its length is determined by **SubAuthorityCount**.

In abstract, the sequence of identifiers that comprise the SID is a hierarchical series. The first integer in the array contains a value that is unique relative to the **IdentifierAuthority** of the SID. All subsequent integers are unique relative to the previous integer. The **IdentifierAuthority** of the SID, or **SubAuthority**_[-1], "issues" **SubAuthority**_[n], for n>0. Each integer is referred to as a RID (relative identifier), and the last RID in the array uniquely identifies the security principal or group in the domain specified by SubAuthorities 0 through -1. RID values are determined by the creator of the SID. Any SID is considered valid only if it has at least one valid RID value.

When represented as strings, for example in documentation or logs, SIDs are expressed as follows: [<1>](#)

```
S-1-IdentifierAuthority-SubAuthority1-SubAuthority2-...-SubAuthorityn
```

2.4.3 ACCESS_MASK

An **ACCESS_MASK** is a 32-bit set of flags that are used to encode the user rights to an object. An access mask is used both to encode the rights to an object assigned to a principal and to encode the requested access when opening an object.

The bits with a 0 value in the table below are used for object-specific user rights. A file object would encode, for example, Read Access and Write Access. A registry key object would encode Create Subkey and Read Value, for example. **Note** The bits with a 0 value are reserved for use by specific protocols that make use of the **ACCESS_MASK** data type. The nature of this usage differs according to each protocol and is implementation-specific.

The bits with a value different than 0 in the table below are user rights that are common to all objects, or are generic rights that can be mapped to object-specific user rights by the object itself.

```

typedef struct {
    unsigned long ACCESS_MASK;
} ACCESS_MASK,
*PACCESS_MASK;

```

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
G	G	G	G	0	0	M	A	0	0	0	S	W	W	R	D	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
R	W	X	A			A	S				Y	O	D	C	E																

GR (Generic Read): The principal identified by the [SID](#) of the [ACE](#) has been granted read access to the object.

GW (Generic Write): The principal identified by the **SID** of the ACE has been granted write access to the object.

GX (Generic Execute): The principal identified by the **SID** of the ACE has been granted execute access to the object.

GA (Generic All): The principal identified by the **SID** of the ACE has been granted all access to the object.

MA (Maximum Allowed): The principal identified by the **SID** of the ACE has been granted the most access allowed to the object.

AS (Access System Security): Specifies access to the system security portion of the [SECURITY_DESCRIPTOR \(section 2.4.6\)](#).

SY (Synchronize): Specifies access to the object sufficient to synchronize or wait on the object.

WO (Write Owner): Specifies access to change the owner of the object as listed in the security descriptor.

WD (Write DACL): Specifies access to change the discretionary access control list of the security descriptor of an object.

RC (Read Control): Specifies access to read the security descriptor of an object.

DE (Delete): Specifies access to delete an object.

2.4.4 ACE

An access control entry (ACE) is used to encode the user rights afforded to a principal, either a user or group. This is generally done by combining an [ACCESS_MASK](#), and the [SID](#) of the principal. There are some variations to accommodate other groupings, which are specified in the following sections.

2.4.4.1 ACE_HEADER

The **ACE_HEADER** structure defines the type and size of an access control entry (ACE).

```
typedef struct _ACE_HEADER {
    UCHAR AceType;
    UCHAR AceFlags;
    USHORT AceSize;
}
```

```

} ACE_HEADER,
 *PACE_HEADER;

```

AceType: ACE Type. **AceType** MUST be one of the following values.

Value	Meaning
ACCESS_ALLOWED_ACE_TYPE 0x00	Access-allowed ACE that uses the ACCESS_ALLOWED_ACE (section 2.4.4.2) structure.
ACCESS_DENIED_ACE_TYPE 0x01	Access-denied ACE that uses the ACCESS_DENIED_ACE (section 2.4.4.4) structure.
SYSTEM_AUDIT_ACE_TYPE 0x02	System-audit ACE that uses the SYSTEM_AUDIT_ACE (section 2.4.4.9) structure.
SYSTEM_ALARM_ACE_TYPE 0x03	Reserved for future use.
ACCESS_ALLOWED_COMPOUND_ACE_TYPE 0x04	Reserved for future use.
ACCESS_ALLOWED_OBJECT_ACE_TYPE 0x05	Object-specific access-allowed ACE that uses the ACCESS_ALLOWED_ACE (section 2.4.4.2) structure. <2>
ACCESS_DENIED_OBJECT_ACE_TYPE 0x06	Object-specific access-denied ACE that uses the ACCESS_DENIED_ACE (section 2.4.4.4) structure. <3>
SYSTEM_AUDIT_OBJECT_ACE_TYPE 0x07	Object-specific system-audit ACE that uses the SYSTEM_AUDIT_ACE (section 2.4.4.9) structure. <4>
SYSTEM_ALARM_OBJECT_ACE_TYPE 0x08	Reserved for future use.
ACCESS_ALLOWED_CALLBACK_ACE_TYPE 0x09	Access-allowed callback ACE that uses the ACCESS_ALLOWED_CALLBACK_ACE (section 2.4.4.5) structure. <5>
ACCESS_DENIED_CALLBACK_ACE_TYPE 0x0A	Access-denied callback ACE that uses the ACCESS_DENIED_CALLBACK_ACE (section 2.4.4.6) structure. <6>
ACCESS_ALLOWED_CALLBACK_OBJECT_ACE_TYPE 0x0B	Object-specific access-allowed callback ACE that uses the ACCESS_ALLOWED_CALLBACK_OBJECT_ACE (section 2.4.4.7) structure. <7>
ACCESS_DENIED_CALLBACK_OBJECT_ACE_TYPE 0x0C	Object-specific access-denied callback ACE that uses the ACCESS_DENIED_CALLBACK_OBJECT_ACE (section 2.4.4.8) structure. <8>

Value	Meaning
SYSTEM_AUDIT_CALLBACK_ACE_TYPE 0x0D	System-audit callback ACE that uses the SYSTEM_AUDIT_CALLBACK_ACE (section 2.4.4.10) structure.<9>
SYSTEM_ALARM_CALLBACK_ACE_TYPE 0x0E	Reserved for future use.
SYSTEM_AUDIT_CALLBACK_OBJECT_ACE_TYPE 0x0F	Object-specific system-audit callback ACE that uses the SYSTEM_AUDIT_CALLBACK_OBJECT_ACE (section 2.4.4.12) structure.
SYSTEM_ALARM_CALLBACK_OBJECT_ACE_TYPE 0x10	Reserved for future use.
SYSTEM_MANDATORY_LABEL_ACE_TYPE 0x11	Mandatory label ACE that uses the SYSTEM_MANDATORY_LABEL_ACE (section 2.4.4.11) structure.

AceFlags: Specifies a set of ACE type-specific control flags. This member can be a combination of the following values.

Value	Meaning
CONTAINER_INHERIT_ACE 0x02	Child objects that are containers, such as directories, inherit the ACE as an effective ACE. The inherited ACE is inheritable unless the NO_PROPAGATE_INHERIT_ACE bit flag is also set.
FAILED_ACCESS_ACE_FLAG 0x80	Used with system-audit ACEs in a system access control list (SACL) to generate audit messages for failed access attempts.
INHERIT_ONLY_ACE 0x08	Indicates an inherit-only ACE, which does not control access to the object to which it is attached. If this flag is not set, the ACE is an effective ACE that controls access to the object to which it is attached. Both effective and inherit-only ACEs can be inherited depending on the state of the other inheritance flags.
INHERITED_ACE 0x10	Indicates that the ACE was inherited. The system sets this bit when it propagates an inherited ACE to a child object.<10>
NO_PROPAGATE_INHERIT_ACE 0x04	If the ACE is inherited by a child object, the system clears the OBJECT_INHERIT_ACE and CONTAINER_INHERIT_ACE flags in the inherited ACE. This prevents the ACE from being inherited by subsequent generations of objects.
OBJECT_INHERIT_ACE 0x01	Noncontainer child objects inherit the ACE as an effective ACE. For child objects that are containers, the ACE is inherited as an inherit-only ACE unless the NO_PROPAGATE_INHERIT_ACE bit flag is also set.
SUCCESSFUL_ACCESS_ACE_FLAG 0x40	Used with system-audit ACEs in a SACL to generate audit messages for successful access attempts.

AceSize: Size, in bytes, of the ACE.

2.4.4.2 ACCESS_ALLOWED_ACE

The **ACCESS_ALLOWED_ACE** structure defines an ACE for the discretionary access control list (DACL) that controls access to an object. An access-allowed ACE allows access to an object for a specific trustee identified by a security identifier (SID).

```
typedef struct _ACCESS_ALLOWED_ACE {
    ACE_HEADER Header;
    ACCESS_MASK Mask;
    DWORD SidStart;
} ACCESS_ALLOWED_ACE,
*PACCESS_ALLOWED_ACE;
```

Header: An [ACE_HEADER](#) structure that specifies the size and type of ACE. It also contains flags that control inheritance of the ACE by child objects.

Mask: An [ACCESS_MASK](#) structure that specifies the user rights allowed by this ACE.

SidStart: The first [DWORD](#) of a trustee's [SID](#). The remaining bytes of the SID are stored in contiguous memory after the **SidStart** member. This SID can be appended with application data.

2.4.4.3 ACCESS_ALLOWED_OBJECT_ACE

The **ACCESS_ALLOWED_OBJECT_ACE** structure defines an access control entry (ACE) that controls allowed access to an object, a property set, or property. The ACE contains a set of access rights, a GUID that identifies the type of object, and a security identifier (SID) that identifies the trustee to whom the system will grant access. The ACE also contains a GUID and a set of flags that control inheritance of the ACE by child objects.

```
typedef struct _ACCESS_ALLOWED_OBJECT_ACE {
    ACE_HEADER Header;
    ACCESS_MASK Mask ;
    DWORD Flags ;
    GUID ObjectType;
    GUID InheritedObjectType;
    DWORD SidStart;
} ACCESS_ALLOWED_OBJECT_ACE,
*PACCESS_ALLOWED_OBJECT_ACE;
```

Header: An [ACE_HEADER](#) structure that specifies the size and type of ACE. It also contains flags that control inheritance of the ACE by child objects.

Mask : An [ACCESS_MASK](#) structure that specifies the user rights allowed by this ACE.

Flags : A set of bit flags that indicate whether the **ObjectType** and **InheritedObjectType** members are present. This parameter can be one or more of the following values.

Value	Meaning
0x00000000	Neither ObjectType nor InheritedObjectType are present. The SidStart member follows immediately

Value	Meaning
	after the Flags member.
ACE_OBJECT_TYPE_PRESENT 0x00000001	ObjectType is present and contains a GUID. If this value is not specified, the InheritedObjectType member follows immediately after the Flags member.
ACE_INHERITED_OBJECT_TYPE_PRESENT 0x00000002	InheritedObjectType is present and contains a GUID. If this value is not specified, all types of child objects can inherit the ACE.

ObjectType: This member exists only if the ACE_OBJECT_TYPE_PRESENT bit is set in the **Flags** member. Otherwise, the **InheritedObjectType** member follows immediately after the **Flags** member. If this member exists, it is a GUID structure that identifies a property set, property, extended right, or type of child object. The purpose of this GUID depends on the user rights specified in the **Mask** member.

Value	Meaning
ADS_RIGHT_DS_CONTROL_ACCESS 0X0000100	The ObjectTypeGUID identifies an extended access right.
ADS_RIGHT_DS_CREATE_CHILD 0X0000001	The ObjectTypeGUID identifies a type of child object. The ACE controls the trustee's right to create this type of child object.
ADS_RIGHT_DS_READ_PROP 0x0000010	The ObjectTypeGUID identifies a property set or property of the object. The ACE controls the trustee's right to read the property or property set.
ADS_RIGHT_DS_WRITE_PROP 0x0000020	The ObjectTypeGUID identifies a property set or property of the object. The ACE controls the trustee's right to write the property or property set.
ADS_RIGHT_DS_SELF 0x0000008	The ObjectTypeGUID identifies a validated write.

InheritedObjectType: This member exists only if the ACE_INHERITED_OBJECT_TYPE_PRESENT bit is set in the **Flags** member. If this member exists, it is a GUID structure that identifies the type of child object that can inherit the ACE. Inheritance is also controlled by the inheritance flags in the **ACE_HEADER**, as well as by any protection against inheritance placed on the child objects.

The offset of this member can vary. If the **Flags** member does not contain the ACE_OBJECT_TYPE_PRESENT flag, the **InheritedObjectType** member starts at the offset specified by the **ObjectType** member.

SidStart: The first **DWORD** of a trustee's SID. The remaining bytes of the SID are stored in contiguous memory after the **SidStart** member. This SID can be appended with application data.

2.4.4.4 ACCESS_DENIED_ACE

The **ACCESS_DENIED_ACE** structure defines an ACE for the discretionary access-control list (DACL) that controls access to an object. An access-denied ACE denies access to an object for a specific trustee identified by a security identifier (SID).

```
typedef struct _ACCESS_DENIED_ACE {
    ACE_HEADER Header;
    ACCESS_MASK Mask;
    DWORD SidStart;
} ACCESS_DENIED_ACE,
*PACCESS_DENIED_ACE;
```

Header: An [ACE HEADER](#) structure that specifies the size and type of ACE. It also contains flags that control inheritance of the ACE by child objects.

Mask: An [ACCESS MASK](#) structure that specifies the user rights denied by this ACE.

SidStart: The first [DWORD](#) of a trustee's [SID](#). The remaining bytes of the SID are stored in contiguous memory after the **SidStart** member. This SID can be appended with application data.

2.4.4.5 ACCESS_ALLOWED_CALLBACK_ACE

The **ACCESS_ALLOWED_CALLBACK_ACE** structure defines an access control entry (ACE) for the discretionary access control list (DACL) that controls access to an object. An access-allowed ACE allows access to an object for a specific trustee identified by a security identifier (SID).

```
typedef struct _ACCESS_ALLOWED_CALLBACK_ACE {
    ACE_HEADER Header;
    ACCESS_MASK Mask ;
    DWORD SidStart;
} ACCESS_ALLOWED_CALLBACK_ACE,
*PACCESS_ALLOWED_CALLBACK_ACE;
```

Header: An [ACE HEADER](#) structure that specifies the size and type of ACE. It also contains flags that control inheritance of the ACE by child objects.

Mask : An [ACCESS MASK](#) structure that specifies the user rights allowed by this ACE.

SidStart: The first [DWORD](#) of a trustee's SID. The remaining bytes of the SID are stored in contiguous memory after the **SidStart** member. This SID can be appended with application data.

2.4.4.6 ACCESS_DENIED_CALLBACK_ACE

The **ACCESS_DENIED_CALLBACK_ACE** structure defines an access control entry (ACE) for the discretionary access control list (DACL) that controls access to an object. An access-denied ACE denies access to an object for a specific trustee identified by a security identifier (SID).

```
typedef struct _ACCESS_DENIED_CALLBACK_ACE {
    ACE_HEADER Header;
    ACCESS_MASK Mask ;
    DWORD SidStart;
} ACCESS_DENIED_CALLBACK_ACE,
*PACCESS_DENIED_CALLBACK_ACE;
```

Header: An [ACE_HEADER](#) structure that specifies the size and type of ACE. It also contains flags that control inheritance of the ACE by child objects.

Mask : An [ACCESS_MASK](#) structure that specifies the user rights denied by this ACE.

SidStart: The first [DWORD](#) of a trustee's SID. The remaining bytes of the SID are stored in contiguous memory after the **SidStart** member. This SID can be appended with application data.

2.4.4.7 ACCESS_ALLOWED_CALLBACK_OBJECT_ACE

The **ACCESS_ALLOWED_CALLBACK_OBJECT_ACE** structure defines an access control entry (ACE) that controls allowed access to an object, property set, or property. The ACE contains a set of user rights, a GUID that identifies the type of object, and a security identifier (SID) that identifies the trustee to whom the system will grant access. The ACE also contains a GUID and a set of flags that control inheritance of the ACE by child objects.

```
typedef struct _ACCESS_ALLOWED_CALLBACK_OBJECT_ACE {
    ACE_HEADER Header;
    ACCESS_MASK Mask ;
    DWORD Flags ;
    GUID ObjectType;
    GUID InheritedObjectType;
    DWORD SidStart;
} ACCESS_ALLOWED_CALLBACK_OBJECT_ACE,
*PACCESS_ALLOWED_CALLBACK_OBJECT_ACE;
```

Header: An [ACE_HEADER](#) structure that specifies the size and type of ACE. It also contains flags that control inheritance of the ACE by child objects.

Mask : An [ACCESS_MASK](#) structure that specifies the user rights allowed by this ACE.

Flags : A set of bit flags that indicate whether the **ObjectType** and **InheritedObjectType** members are present. This parameter can be one or more of the following values.

Value	Meaning
0x00000000	Neither ObjectType nor InheritedObjectType are present. The SidStart member follows immediately after the Flags member.
ACE_OBJECT_TYPE_PRESENT 0x00000001	ObjectType is present and contains a GUID. If this value is not specified, the InheritedObjectType member follows immediately after the Flags member.
ACE_INHERITED_OBJECT_TYPE_PRESENT 0x00000002	InheritedObjectType is present and contains a GUID. If this value is not specified, all types of child objects can inherit the ACE.

ObjectType: This member exists only if the ACE_OBJECT_TYPE_PRESENT bit is set in the **Flags** member. Otherwise, the **InheritedObjectType** member follows immediately after the **Flags** member. If this member exists, it is a GUID structure that identifies a property set, property, extended right, or type of child object. The purpose of this GUID depends on the user rights specified in the **Mask** member.

Value	Meaning
ADS_RIGHT_DS_CONTROL_ACCESS 0X00000100	The ObjectTypeGUID identifies an extended access right.
ADS_RIGHT_DS_CREATE_CHILD 0X00000001	The ObjectTypeGUID identifies a type of child object. The ACE controls the trustee's right to create this type of child object.
ADS_RIGHT_DS_READ_PROP 0x00000010	The ObjectTypeGUID identifies a property set or property of the object. The ACE controls the trustee's right to read the property or property set.
ADS_RIGHT_DS_WRITE_PROP 0x00000020	The ObjectTypeGUID identifies a property set or property of the object. The ACE controls the trustee's right to write the property or property set.
ADS_RIGHT_DS_SELF 0x00000008	The ObjectTypeGUID identifies a validated write.

InheritedObjectType: This member exists only if the ACE_INHERITED_OBJECT_TYPE_PRESENT bit is set in the **Flags** member. If this member exists, it is a GUID structure that identifies the type of child object that can inherit the ACE. Inheritance is also controlled by the inheritance flags in the **ACE_HEADER**, as well as by any protection against inheritance placed on the child objects.

The offset of this member can vary. If the **Flags** member does not contain the ACE_OBJECT_TYPE_PRESENT flag, the **InheritedObjectType** member starts at the offset specified by the **ObjectType** member.

SidStart: The first **DWORD** of a trustee's SID. The remaining bytes of the SID are stored in contiguous memory after the **SidStart** member. This SID can be appended with application data.

2.4.4.8 ACCESS_DENIED_CALLBACK_OBJECT_ACE

The **ACCESS_DENIED_CALLBACK_OBJECT_ACE** structure defines an access control entry (ACE) that controls denied access to an object, a property set, or property. The ACE contains a set of user rights, a GUID that identifies the type of object, and a security identifier (SID) that identifies the trustee to whom the system will deny access. The ACE also contains a GUID and a set of flags that control inheritance of the ACE by child objects.

```
typedef struct _ACCESS_DENIED_CALLBACK_OBJECT_ACE {
    ACE_HEADER Header;
    ACCESS_MASK Mask ;
    DWORD Flags ;
    GUID ObjectType;
    GUID InheritedObjectType;
    DWORD SidStart;
} ACCESS_DENIED_CALLBACK_OBJECT_ACE,
*PACCESS_DENIED_CALLBACK_OBJECT_ACE;
```

Header: An **ACE_HEADER** structure that specifies the size and type of ACE. It also contains flags that control inheritance of the ACE by child objects.

Mask : An [ACCESS_MASK](#) structure that specifies the user rights denied by this ACE.

Flags : A set of bit flags that indicate whether the **ObjectType** and **InheritedObjectType** members are present. This parameter can be one or more of the following values.

Value	Meaning
0x00000000	Neither ObjectType nor InheritedObjectType are present. The SidStart member follows immediately after the Flags member.
ACE_OBJECT_TYPE_PRESENT 0x00000001	ObjectType is present and contains a GUID. If this value is not specified, the InheritedObjectType member follows immediately after the Flags member.
ACE_INHERITED_OBJECT_TYPE_PRESENT 0x00000002	InheritedObjectType is present and contains a GUID. If this value is not specified, all types of child objects can inherit the ACE.

ObjectType: This member exists only if the ACE_OBJECT_TYPE_PRESENT bit is set in the **Flags** member. Otherwise, the **InheritedObjectType** member follows immediately after the **Flags** member. If this member exists, it is a GUID structure that identifies a property set, property, extended right, or type of child object. The purpose of this GUID depends on the user rights specified in the **Mask** member.

Value	Meaning
ADS_RIGHT_DS_CONTROL_ACCESS 0x00000100	The ObjectTypeGUID identifies an extended access right.
ADS_RIGHT_DS_CREATE_CHILD 0x00000001	The ObjectTypeGUID identifies a type of child object. The ACE controls the trustee's right to create this type of child object.
ADS_RIGHT_DS_READ_PROP 0x00000010	The ObjectTypeGUID identifies a property set or property of the object. The ACE controls the trustee's right to read the property or property set.
ADS_RIGHT_DS_WRITE_PROP 0x00000020	The ObjectTypeGUID identifies a property set or property of the object. The ACE controls the trustee's right to write the property or property set.
ADS_RIGHT_DS_SELF 0x00000008	The ObjectTypeGUID identifies a validated write.

InheritedObjectType: This member exists only if the ACE_INHERITED_OBJECT_TYPE_PRESENT bit is set in the **Flags** member. If this member exists, it is a GUID structure that identifies the type of child object that can inherit the ACE. Inheritance is also controlled by the inheritance flags in the **ACE_HEADER**, as well as by any protection against inheritance placed on the child objects.

The offset of this member can vary. If the **Flags** member does not contain the ACE_OBJECT_TYPE_PRESENT flag, the **InheritedObjectType** member starts at the offset specified by the **ObjectType** member.

SidStart: The first **DWORD** of a trustee's SID. The remaining bytes of the SID are stored in contiguous memory after the **SidStart** member. This SID can be appended with application data.

2.4.4.9 SYSTEM_AUDIT_ACE

The **SYSTEM_AUDIT_ACE** structure defines an access control entry (ACE) for the system access control list (SACL) that specifies what types of access cause system-level notifications. A system-audit ACE causes an audit message to be logged when a specified trustee attempts to gain access to an object. The trustee is identified by a security identifier (SID).

```
typedef struct _SYSTEM_AUDIT_ACE {
    ACE_HEADER Header;
    ACCESS_MASK Mask;
    DWORD SidStart;
} SYSTEM_AUDIT_ACE,
*PSYSTEM_AUDIT_ACE;
```

Header: An **ACE_HEADER** structure that specifies the size and type of ACE. It also contains flags that control inheritance of the ACE by child objects.

Mask : An **ACCESS_MASK** structure that specifies the user rights that cause audit messages to be generated.

SidStart: The first **DWORD** of a trustee's SID. The remaining bytes of the SID are stored in contiguous memory after the **SidStart** member. This SID can be appended with application data. An access attempt of a kind specified by the **Mask** member by any trustee whose SID matches the **SidStart** member causes the system to generate an audit message. If an application does not specify a SID for this member, audit messages are generated for the specified access rights for all trustees.

2.4.4.10 SYSTEM_AUDIT_CALLBACK_ACE

The **SYSTEM_AUDIT_CALLBACK_ACE** structure defines an access control entry (ACE) for the system access control list (SACL) that specifies what types of access cause system-level notifications. A system-audit ACE causes an audit message to be logged when a specified trustee attempts to gain access to an object. The trustee is identified by a security identifier (SID).

```
typedef struct _SYSTEM_AUDIT_CALLBACK_ACE {
    ACE_HEADER Header;
    ACCESS_MASK Mask;
    DWORD SidStart;
} SYSTEM_AUDIT_CALLBACK_ACE,
*PSYSTEM_AUDIT_CALLBACK_ACE;
```

Header: An **ACE_HEADER** structure that specifies the size and type of ACE. It also contains flags that control inheritance of the ACE by child objects.

Mask : An **ACCESS_MASK** structure that specifies the user rights that cause audit messages to be generated.

SidStart: The first **DWORD** of a trustee's SID. The remaining bytes of the SID are stored in contiguous memory after the **SidStart** member. This SID can be appended with application data. An access attempt of a kind specified by the **Mask** member by any trustee whose SID matches the **SidStart** member causes the system to generate an audit message. If an application does not specify a SID for this member, audit messages are generated for the specified user rights for all trustees.

2.4.4.11 SYSTEM_MANDATORY_LABEL_ACE

The **SYSTEM_MANDATORY_LABEL_ACE** structure defines an access control entry (ACE) for the system access control list (SACL) that specifies the mandatory access level and policy for a securable object. [<11>](#)

```
typedef struct _SYSTEM_MANDATORY_LABEL_ACE {
    ACE_HEADER Header;
    ACCESS_MASK Mask ;
    DWORD SidStart;
} SYSTEM_MANDATORY_LABEL_ACE,
*PSYSTEM_MANDATORY_LABEL_ACE;
```

Header: An **ACE HEADER** structure that specifies the size and type of ACE. It also contains flags that control inheritance of the ACE by child objects.

Mask : An **ACCESS_MASK** structure that specifies the access policy for principals with a mandatory integrity level lower than the object associated with the SACL that contains this ACE.

Value	Meaning
SYSTEM_MANDATORY_LABEL_NO_WRITE_UP 0x00000001	A principal with a lower mandatory level than the object cannot write to the object.
SYSTEM_MANDATORY_LABEL_NO_READ_UP 0x00000002	A principal with a lower mandatory level than the object cannot read the object.
SYSTEM_MANDATORY_LABEL_NO_EXECUTE_UP 0x00000004	A principal with a lower mandatory level than the object cannot execute the object.

SidStart: Specifies the first **DWORD** of a SID. The remaining bytes of the SID are stored in contiguous memory after the **SidStart** member. The identifier authority of the SID must be SECURITY_MANDATORY_LABEL_AUTHORITY. The RID of the SID specifies the mandatory integrity level of the object associated with the SACL that contains this ACE. The RID must be one of the following values.

Value	Meaning
0x00000001	Low integrity level.
0x00000002	Medium integrity level.
0x00000003	High integrity level.

2.4.4.12 SYSTEM_AUDIT_CALLBACK_OBJECT_ACE

The **SYSTEM_AUDIT_CALLBACK_OBJECT_ACE** structure defines an access control entry (ACE) for a system access control list (SACL). The ACE can audit access to an object or subobjects, such as property sets or properties. The ACE contains a set of user rights, a GUID that identifies the type of object or subobject, and a security identifier (SID) that identifies the trustee for whom the system will audit access. The ACE also contains a GUID and a set of flags that control inheritance of the ACE by child objects.

```
typedef struct _SYSTEM_AUDIT_CALLBACK_OBJECT_ACE {
    ACE_HEADER Header;
    ACCESS_MASK Mask ;
    DWORD Flags ;
    GUID ObjectType;
    GUID InheritedObjectType;
    DWORD SidStart;
} SYSTEM_AUDIT_CALLBACK_OBJECT_ACE,
*PSYSTEM_AUDIT_CALLBACK_OBJECT_ACE;
```

Header: An [ACE HEADER](#) structure that specifies the size and type of ACE. It contains flags that control inheritance of the ACE by child objects.

Mask : An [ACCESS MASK](#) structure that specifies the user rights that cause audit messages to be generated.

Flags : A set of bit flags that indicate whether the **ObjectType** and **InheritedObjectType** members contain GUIDs. This member can be a combination of the following values.

Value	Meaning
ACE_OBJECT_TYPE_PRESENT 0x00000001	The ObjectType member contains a GUID.
ACE_INHERITED_OBJECT_TYPE_PRESENT 0x00000002	The InheritedObjectType member contains a GUID.

ObjectType: A [GUID](#) structure that identifies a property set, property, extended right, or type of child object. This member is valid only if the ACE_OBJECT_TYPE_PRESENT bit is set in the **Flags** member. Otherwise, **ObjectType** is ignored.

The purpose of this GUID depends on the user rights specified in the **Mask** member.

This member can be one of the following values.

Value	Meaning
ADS_RIGHT_DS_CONTROL_ACCESS 0X00000100	The ObjectTypeGUID identifies an extended access right.
ADS_RIGHT_DS_CREATE_CHILD 0X00000001	The ObjectTypeGUID identifies a type of child object. The ACE controls the trustee's right to create this type of child object.
ADS_RIGHT_DS_READ_PROP	The ObjectTypeGUID identifies a property set or property of the object. The ACE controls the trustee's right to read the

Value	Meaning
0x00000010	property or property set.
ADS_RIGHT_DS_WRITE_PROP 0x00000020	The ObjectTypeGUID identifies a property set or property of the object. The ACE controls the trustee's right to write the property or property set.
ADS_RIGHT_DS_SELF 0x00000008	The ObjectTypeGUID identifies a validated write.

InheritedObjectType: A **GUID** structure that identifies the type of child object that can inherit the ACE.

This member is valid only if the `ACE_INHERITED_OBJECT_TYPE_PRESENT` bit is set in the **Flags** member. If that bit is not set, **InheritedObjectType** is ignored and all types of child objects can inherit the ACE. In either case, inheritance is also controlled by the inheritance flags in the **ACE_HEADER**, and also by any protection against inheritance placed on the child objects.

SidStart: The first **DWORD** of a trustee's SID. The remaining bytes of the SID are stored in contiguous memory after the **SidStart** member. This SID can be appended with application data.

2.4.5 ACL

The access control list, or **ACL**, is used to specify a list of individual access control entries ([ACEs](#)).

The individual ACEs in an ACL are numbered from 0 to n, where n+1 is the number of ACEs in the ACL. When editing an ACL, an application refers to an ACE within the ACL by the ACE's index.

There are two types of ACL.

- A **discretionary access control list (DACL)** is controlled by the owner of an object or anyone granted `WRITE_DAC` access to the object. It specifies the access particular users and groups can have to an object. For example, the owner of a file can use a DACL to control which users and groups can and cannot have access to the file..
- A **system access control list (SACL)** is similar to the DACL, except that the SACL is used to audit rather than control access to an object. When an audited action occurs, the operating system records the event in the security log. Each ACE in a SACL has a header that indicates whether auditing is triggered by success, failure, or both; a SID that specifies a particular user or security group to monitor; and an access mask that lists the operations to audit.

The SACL also MAY contain [<12>](#) a label ACE that defines the integrity level of the object.

The only valid ACE types for a SACL are the auditing ones (`SYSTEM_AUDIT_ACE_TYPE` and `SYSTEM_AUDIT_OBJECT_ACE_TYPE`) and the label one (`SYSTEM_MANDATORY_LABEL_ACE_TYPE`), as specified in section [2.4.4.1](#).

The SACL must not contain ACEs that belong in the DACL, and the DACL must not contain ACE types that belong in the SACL. Doing so results in unspecified behavior.

The **ACL** structure is the header of an access control list (ACL). A complete ACL consists of an ACL structure followed by an ordered list of zero or more access control entries (ACEs).

```
typedef struct _ACL {
```

```

    UCHAR AclRevision;
    UCHAR Sbz1;
    USHORT AclSize;
    USHORT AceCount;
    USHORT Sbz2;
} ACL,
*PACL;

```

AclRevision: Revision of the **ACL** structure, MUST be set to one of the following values.

Value	Meaning
0x02	When set to 0x02, only ACE types 0x00, 0x01, 0x02, and 0x03 should be present in the ACL. For more information on ACE types, see section 2.4.4.1 .
0x04	When set to 0x04, ACE types 0x05, 0x06, 0x07, and 0x08 are allowed. ACLs of revision 0x04 should be applied only to Directory Service objects.

Sbz1: The **Sbz1** field is reserved and MUST be set to 0.

AclSize: The size of the complete ACL, including all ACEs, in bytes.

AceCount: The count of the number of ACE records in the **ACL**.

Sbz2: The **Sbz2** field is reserved and MUST be set to 0.

2.4.6 SECURITY_DESCRIPTOR

The **SECURITY_DESCRIPTOR** structure defines an object's security attributes. These attributes specify who owns the object, who can access the object and what they can do with it, what level of audit logging should be applied to the object, and what kind of restrictions apply to the use of the security descriptor.

Security descriptors appear in one of two forms, absolute or self-relative.

A security descriptor is said to be in absolute format if it stores all of its security information via pointer fields.

```

typedef struct _SECURITY_DESCRIPTOR {
    UCHAR Revision;
    UCHAR Sbz1;
    USHORT Control;
    PSID Owner;
    PSID Group;
    PACL Sacl;
    PACL Dacl;
} SECURITY_DESCRIPTOR,
*PSECURITY_DESCRIPTOR;

```

A security descriptor is said to be in self-relative format if it stores all of its security information in a contiguous block of memory, and expresses all of its pointer fields as offsets from its beginning.

The self relative form of the security descriptor is required if one wants to transmit the **SECURITY_DESCRIPTOR** structure as an opaque data structure for transmission in communication protocols over a wire, or for storage on secondary media; the absolute form cannot be transmitted because it contains pointers to objects that are generally not accessible to the recipient.

When a self-relative security descriptor is transmitted over a wire, it is sent in little-endian format and requires no padding.

```
typedef struct _SECURITY_DESCRIPTOR {
    UCHAR Revision;
    UCHAR Sbz1;
    USHORT Control;
    ULONG Owner;
    ULONG Group;
    ULONG Sacl;
    ULONG Dacl;
} SECURITY_DESCRIPTOR,
 *PSECURITY_DESCRIPTOR;
```

Revision: The revision of the **SECURITY_DESCRIPTOR** structure. This field **MUST** be set to 0.

Sbz1: The **Sbz1** field is reserved and **MUST** be set to 0.

Control: A 16-bit field for bit flags.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
S	R	P	P	S	D	S	D	D	S	S	S	D	D	G	O
R	M	S	D	I	I	R	R	T	S	D	P	D	P	D	D

SR (Self-Relative): Set when the security descriptor is in self-relative format. Cleared when the security descriptor is in absolute format.

RM (RM Control Valid): Set when the resource manager control bits are valid.

PS (SACL Protected): Set when the SACL should be protected from inherit operations.

PD (DACL Protected): Set when the DAACL should be protected from inherit operations.

SI (SACL Auto-Inherited): Set when the SACL was created through inheritance.

DI (DAACL Auto-Inherited): Set when the DAACL was created through inheritance.

SR (Inheritance Required): Set when the SACL should be computed through inheritance.

DR (DAACL Inheritance Required): Set when the DAACL should be computed through inheritance.

DT (DAACL Trusted): Set when ACL pointed to by the **DAACL** field was provided by a trusted source and does not require any editing of compound ACEs.

SS (Server Security): Set when the caller wants the system to create a Server ACL based on the input ACL, regardless of its source (explicit or defaulting).

SD (SACL Defaulted): Set when the SACL was established by default means.

SP (SACL Present): Set when the SACL is present on the object.

DD (DACL Defaulted): Set when the DACL was established by default means.

DP (DACL Present): Set when the DACL is present on the object.

GD (Group Defaulted): Set when the group was established by default means.

OD (Owner Defaulted): Set when the owner was established by default means.

Owner: An offset to the [SID](#) that specifies the owner of the object to which the security descriptor is associated.

Group: An offset to the **SID** that specifies the group of the object to which the security descriptor is associated.

Sacl: An offset to the [ACL](#) that contains system ACEs. Typically, the system **ACL** contains auditing ACEs (such as [SYSTEM_AUDIT_ACE](#), [SYSTEM_AUDIT_CALLBACK_ACE](#), or [SYSTEM_AUDIT_CALLBACK_OBJECT_ACE](#)), and at most one Label ACE (as specified in section [2.4.4.11](#)). This must be a valid offset if the SP flag is set; if the SP flag is not set, this field **MUST** be set to 0.

Dacl: An offset to the **ACL** that contains ACEs that control access. Typically, the discretionary ACL contains ACEs that grant or deny access to principals or groups. This must be a valid offset if the DP flag is set; if the DP flag is not set, this field **MUST** be set to 0.

2.4.7 SECURITY_INFORMATION

The **SECURITY_INFORMATION** data type identifies the object-related security information being set or queried. This security information includes:

- The owner of an object.
- The primary group of an object.
- The discretionary access control list (DAACL) of an object.
- The system access control list (SACL) of an object.

An unsigned 32-bit integer specifies portions of a [SECURITY_DESCRIPTOR](#) by means of bit flags. Individual bit values (combinable with the bitwise OR operation) are as follows:

Value	Meaning
OWNER_SECURITY_INFORMATION 0x00000001	The owner identifier of the object is being referenced.
GROUP_SECURITY_INFORMATION 0x00000002	The primary group identifier of the object is being referenced.
DAACL_SECURITY_INFORMATION 0x00000004	The DAACL of the object is being referenced.
SACL_SECURITY_INFORMATION 0x00000008	The SACL of the object is being referenced.

Value	Meaning
LABEL_SECURITY_INFORMATION 0x00000010	The mandatory integrity label is being referenced.
UNPROTECTED_SACL_SECURITY_INFORMATION 0x10000000	The SACL inherits access control entries (ACEs) from the parent object.
UNPROTECTED_DACL_SECURITY_INFORMATION 0x20000000	The DACL inherits ACEs from the parent object.
PROTECTED_SACL_SECURITY_INFORMATION 0x40000000	The SACL cannot inherit ACEs.
PROTECTED_DACL_SECURITY_INFORMATION 0x80000000	The DACL cannot inherit ACEs.

This type is declared as follows:

```
typedef DWORD SECURITY_INFORMATION, *PSECURITY_INFORMATION;
```

2.4.8 RPC_SID

The **RPC_SID** structure is a representation of a security identifier (SID), as specified by the [SID](#) structure. This structure is defined by using the interface definition language (IDL) as follows.

```
typedef struct _RPC_SID {
    unsigned char Revision;
    unsigned char SubAuthorityCount;
    SID_IDENTIFIER_AUTHORITY IdentifierAuthority;
    [size_is(SubAuthorityCount)] unsigned long SubAuthority[];
} RPC_SID,
*PRPC_SID;
```

For individual member semantics, see the **SID** structure.

2.5 Additional Information for Security Types

2.5.1 Security Descriptor Description Language

The [SECURITY_DESCRIPTOR](#) structure is a compact binary representation of the security associated with an object in a directory or on a file system, or in other stores. It is not, however, convenient for use in tools that operate primarily on text strings. Therefore, a text-based form of the security descriptor is available for situations when a security descriptor must be carried by a text method. This format is the Security Descriptor Description Language (SDDL). [<13>](#)

2.5.1.1 Syntax

An SDDL string is a single sequence of characters. The format may be ANSI or Unicode; the actual protocol MUST specify the character set used. Regardless of the character set used, the possible characters that may be used are alphanumeric and punctuation.

The format for an SDDL string is described by the following ABNF grammar, as specified in [\[RFC4234\]](#).

Where the elements are:

```
sddl = [owner-string] [group-string] [dacl-string] [sacl-string]
owner-string = "O:" sid-string
group-string = "G:" sid-string
dacl-string = "D:" [acl-flag-string] [acl-string]
sacl-string = "S:" [acl-flag-string] [acl-string]
sid-string:= sid-token / sid-value
sid-value = <SID string representation>
sid-token = <selection from table below>
acl-flag-string = acl-flag acl-flag-string / acl-flag / ""
acl-flag = "P" / "AR" / "AI"
acl-string = "" / ace acl-string / ace
ace = "(" ace-type ";" [ace-flag-string] ";" ace-rights ";"
[object-guid] ";" [inherit-object-guid] ";" sid-string ")"
ace-type = "A" / "D" / "OA" / "OD" / "AU" / "AL" / "OU" / "OL"
ace-flag-string = ace-flag ace-flag-string / ""
ace-flag = "CI" / "OI" / "NP" / "IO" / "ID" / "SA" / "FA"
ace-rights = absolute-rights / text-rights-string
absolute-rights = %x00000000 - %xFFFFFFFF ; 32 bit hexadecimal
number indicating access right bits
text-rights-string = text-right text-rights-string / ""
text-right = "RP" / "WP" / "CC" / "DC" / "LC" / "SW" / "LO" /
"DT" / "CR" / "RC" / "WD" / "WO" / "SD" / "GA" / "GW" / "GR" /
"GX" / "FA" / "FR" / "FW" / "FX" / "KA" / "KR" / "KW" / "KX"
object-guid = "" / ((%x30-39 / %x41-46) (%x30-39 / %x41-46)
(%x30-39 / %x41-46) (%x30-39 / %x41-46) (%x30-39 / %x41-46)
(%x30-39 / %x41-46) (%x30-39 / %x41-46) (%x30-39 / %x41-46) "-"
(%x30-39 / %x41-46) (%x30-39 / %x41-46) (%x30-39 / %x41-46)
```

```
(%x30-39 / %x41-46) "-" (%x30-39 / %x41-46) (%x30-39 / %x41-46)
(%x30-39 / %x41-46) (%x30-39 / %x41-46) "-" (%x30-39 / %x41-46)
(%x30-39 / %x41-46) (%x30-39 / %x41-46) (%x30-39 / %x41-46) "-"
(%x30-39 / %x41-46) (%x30-39 / %x41-46) (%x30-39 / %x41-46)
(%x30-39 / %x41-46) (%x30-39 / %x41-46) (%x30-39 / %x41-46)
(%x30-39 / %x41-46) (%x30-39 / %x41-46) (%x30-39 / %x41-46)
(%x30-39 / %x41-46) (%x30-39 / %x41-46) (%x30-39 / %x41-46) ;
The second option is the GUID of the object in the form
"XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX" Where each "X" is a Hex digit
```

```
inherit-object-guid = "" / ((%x30-39 / %x41-46) (%x30-39 / %x41-
46) (%x30-39 / %x41-46) (%x30-39 / %x41-46) (%x30-39 / %x41-46)
(%x30-39 / %x41-46) (%x30-39 / %x41-46) (%x30-39 / %x41-46) "-"
(%x30-39 / %x41-46) (%x30-39 / %x41-46) (%x30-39 / %x41-46)
(%x30-39 / %x41-46) "-" (%x30-39 / %x41-46) (%x30-39 / %x41-46)
(%x30-39 / %x41-46) (%x30-39 / %x41-46) "-" (%x30-39 / %x41-46)
(%x30-39 / %x41-46) (%x30-39 / %x41-46) (%x30-39 / %x41-46) "-"
(%x30-39 / %x41-46) (%x30-39 / %x41-46) (%x30-39 / %x41-46)
(%x30-39 / %x41-46) (%x30-39 / %x41-46) (%x30-39 / %x41-46)
(%x30-39 / %x41-46) (%x30-39 / %x41-46) (%x30-39 / %x41-46)
(%x30-39 / %x41-46) (%x30-39 / %x41-46) (%x30-39 / %x41-46)
(%x30-39 / %x41-46) (%x30-39 / %x41-46) (%x30-39 / %x41-46) ;
The second option is the GUID of the object in the form
"XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX" Where each "X" is a Hex digit
```

acl-flag: Flags for the **SECURITY_DESCRIPTOR** structure, context dependent on whether a SACL or DACL is being processed. "P" indicates Protected, the PS or PD flags above. "AR" corresponds to SR or DR. "AI" indicates SI or DI.

ace-type: String that indicates the type of ACE that is being presented.

String	ACE type	Numeric value
A	Access Allowed	0
D	Access Denied	1
AU	Audit	2
AL	Alarm	3
OA	Object Access Allowed	5
OD	Object Access Denied	6
OU	Object Audit	7
OL	Object Alarm	8

ace-flag-string: A set of ACE flags that define the behavior of the ACE. The strings correlate exactly to the flags as specified in section [2.4.4.1](#).

text-rights-string: A set of individual user rights.

String	Access right
GA	Generic All
GX	Generic Execute
GR	Generic Read
GW	Generic Write
RC	Read Control
SD	Delete
WD	Write DAC
WO	Write Owner
FA	File All Access
FR	File Read
FW	File Write
FX	File Execute
KA	Key All Access
KR	Key Read
KW	Key Write
KX	Key Execute
RP	Read Property
WP	Write Property
CC	Create Child
DC	Delete Child
LC	List Children
SW	Self Write
LO	List Object
DT	Delete Tree
CR	Control Access

sid-token: An abbreviated form of a well known SID, per the following table.

Account name	SDDL alias	Well-known SIDs
DOMAIN ADMINISTRATORS	"DA"	s-1-5-<domain>-512
DOMAIN GUESTS	"DG"	s-1-5-<domain>-501

Account name	SDDL alias	Well-known SIDs
DOMAIN_USERS	"DU"	s-1-5-<domain>-513
ENTERPRISE_DOMAIN_CONTROLLERS	"ED"	s-1-5-9
DOMAIN_DOMAIN_CONTROLLERS	"DD"	s-1-5-<domain>-516
DOMAIN_COMPUTERS	"DC"	s-1-5-<domain>-515
BUILTIN_ADMINISTRATORS	"BA"	s-1-5-32-544
BUILTIN_GUESTS	"BG"	s-1-5-32-546
BUILTIN_USERS	"BU"	s-1-5-32-545
ACCOUNT_OPERATORS	"AO"	s-1-5-32-548
BACKUP_OPERATORS	"BO"	s-1-5-32-551
PRINTER_OPERATORS	"PO"	s-1-5-32-550
SERVER_OPERATORS	"SO"	s-1-5-32-549
AUTHENTICATED_USERS	"AU"	s-1-5-11
PRINCIPAL_SELF	"PS"	s-1-5-10
CREATOR_OWNER	"CO"	s-1-3-0
CREATOR_GROUP	"CG"	s-1-3-1
LOCAL_SYSTEM	"SY"	s-1-5-18
POWER_USERS	"PU"	s-1-5-32-547
EVERYONE	"WD"	s-1-1-0
REPLICATOR	"RE"	s-1-5-32-552
INTERACTIVE	"IU"	s-1-5-4
NETWORK	"NU"	s-1-5-2
SERVICE	"SU"	s-1-5-6
RESTRICTED_CODE	"RC"	s-1-5-12
SCHEMA_ADMINISTRATORS	"SA"	s-1-<root domain>-518
RAS_SERVERS	"RS"	s-1-5-<domain>-553
ENTERPRISE_ADMINS	"EA"	s-1-5-<root domain>-519

The following string:

```
"O:BAG:BAD:P(A;CIOI;GRGX;;;BU)(A;CIOI;GA;;;BA)(A;CIOI;GA;;;SY)(A;CIOI;GA;;;CO)S:P(AU;FA;GR;;;WD)""
```

Yields the following, which is an encoded output of the security descriptor ordered as little-endian.

```

00000000 01 00 14 b0 90 00 00 00 a0 00 00 00 14 00 00 00 .....
00000010 30 00 00 00 02 00 1c 00 01 00 00 00 02 80 14 00 0.....
00000020 00 00 00 80 01 01 00 00 00 00 01 00 00 00 00 .....
00000030 02 00 60 00 04 00 00 00 00 03 18 00 00 00 00 a0 ..'.....
00000040 01 02 00 00 00 00 00 05 20 00 00 00 21 02 00 00 .....!...
00000050 00 03 18 00 00 00 10 01 02 00 00 00 00 00 05 .....
00000060 20 00 00 00 20 02 00 00 00 03 14 00 00 00 10 .....
00000070 01 01 00 00 00 00 05 12 00 00 00 00 03 14 00 .....
00000080 00 00 00 10 01 01 00 00 00 00 03 00 00 00 00 .....
00000090 01 02 00 00 00 00 05 20 00 00 00 20 02 00 00 .....
000000a0 01 02 00 00 00 00 05 20 00 00 00 20 02 00 00 .....

```

The **SECURITY_DESCRIPTOR** starts with the SD revision number (1 byte long) at address 0x00, followed by reserved bits and the SD control flags (2 bytes long). As mentioned above, this is followed by owner, group, SACL, and DACL offsets.

```
01 00 14 b0 90 00 00 00-a0 00 00 00 14 00 00 00
```

01	00	14	B0	90	00	00	00	a0	00	00	00	14	00	00	00
Revision Number	Reserved	Control flags	Owner offset				Group offset				SACL offset				

Figure 1: Security descriptor field offsets example

Control Flags

Control flags for the DACL are represented as a bitmask and the resultant set of flags is computed by a logical OR of the flags. In our case, the control flag value is set to:

```
1011000000010100
```

This value maps to:

BIT	Meaning
0	OD Owner defaulted
0	GD Group defaulted
1	DP DACL present
0	DD DACL defaulted
1	SP SACL present
0	SD SACL defaulted
0	SS Server Security
0	DT DACL Trusted

BIT	Meaning
0	DR DACL Inheritance Required
0	SR Inheritance Required
0	DI DACL auto-inherited
0	SI SACL auto-inherited
1	PD DACL-protected
1	PS SACL-protected
0	RM Control Valid
1	SR Self-Relative

SACL

This is followed by the SACL, in this example -" S:P(AU;FA;GR;;;WD)"

DACL

This is followed by the **SECURITY_DESCRIPTOR** DACL, in this example:

```
(A;C;IOI;GRGX;;;BU) (A;C;IOI;GA;;;BA) (A;C;IOI;GA;;;SY) (A;C;IOI;GA;;;CO)
```

Note The string representation for the DACL (D:), and the DACL control flags are consumed as not part of the DACL structure in the SD, but instead as the security descriptor control flags. The same applies for SACL.

02 00 60 00 04 00 00 00-00 03 18 00 00 00 00 a0 01 02 00 00 00 00 00 00 05-20 00 00 00 21 02 00 00 00 03 18 00 00 00 00 10-01 02 00 00 00 00 05 20 00 00 00 20 02 00 00-00 03 14 00 00 00 10 01 01 00 00 00 00 00 05-12 00 00 00 00 03 14 00 00 00 00 10 01 01 00 00-00 00 00 03 00 00 00 00
DACL

Figure 2: Security access control list data example

The ACL can be further dissected into the ACL header and the individual ACEs. For more information, see section [2.4.5](#).

ACL HEADER

```
02 00 60 00 04 00 00 00
AclRevision (1 byte): 0x02
Reserved           : 0x00
AclSize            : 0x0060
AceCount           : 0x0004
```

Reserved : 0x0000

ACE Structure

This is followed by the ACES in the ACL. For more information about the ACE structure, see section [2.4.4.1](#).

In this example, there are four ACEs for the DACL.

(A;CIOI;GRGX;;;BU) (A;CIOI;GA;;;BA) (A;CIOI;GA;;;SY) (A;CIOI;GA;;;CO)

First, look at the first access control entry (ACE) as an example. "(A;CIOI;GRGX;;;BU)" maps to the following in the binary structure (in little-endian order):

00 03 18 00 00 00 00 00 a0 01 02 00 00 00 00 00 00 05-20 00 00 00 21 02 00 00

01	03	18 00	00 00 00 a0	01 02 00 00 00 00 00 05 20 00 00 00 21 02 00 00
Access Allowed ACE Type	Ace Flags - CI 0I	Ace Size	Access Mask	SID S-1-5-32-545

Figure 3: ACE field offsets

Owner

The owner begins at offset 0x90. In our example, owner is set to "BA" (Built-in Admin).

01	02	00	00	00	00	00	05	20	00	00	00	20	02	00	00
Owner															

Figure 4: ACE owner field offsets example

Group

The group begins at offset 0xA0. In our example, group is set to "BA" (Built in Admin).

01	02	00	00	00	00	00	05	20	00	00	00	20	02	00	00
Group															

Figure 5: ACE group field offsets example

2.5.2 Security Descriptor Algorithms

The security descriptor is the basis for specifying the security associated with an object. The server that "owns" the object is responsible for verifying that a client has sufficient access to the object in order to open or manipulate the object. In order to create a server that maintains the same guarantees of authorization to clients, the access check algorithm should be the same.

To perform the access check, the server must have an authorization context that represents the user that requests access. For the purposes of this specification, that context is referred to as a token. The token should be considered as having a number of fields associated with it. The `Token.Sids[]` field is an array that contains the SID of the user account itself, and the SIDs of all the groups of which the user is a member.

Several SIDs in this set are considered notable during security descriptor creation.

- One SID is designated as an Owner - a SID that is allowed to own resources. The definition and control of which SIDs are allowed to own resources, such as files, is a policy decision outside the control of these algorithms. **Token.OwnerIndex** is a field indexing `Token.SIDs[]` and indicates which SID in the possible set of SIDs should be used to indicate ownership.
- One SID is designated as the primary group and is likewise controlled by policy outside of these algorithms. `Token.PrimaryGroup` is an index into the `Token.SIDs[]` array that indicates which SID should be considered the primary group of this user.

The token also contains an ACL, `Token.DefaultDACL`, that serves as the DACL assigned by default to any objects created by the user.

In addition, there are two flags associated with the token that are relevant for checking access—the `SystemACLAccess` flag and the `TakeOwnership` flag. Whether the flags are set is a policy decision outside of the access check algorithm; the results of this policy are used by the algorithm, and hence are referenced here. **Note** For more information about tokens in Windows, see [\[MSDN-ACCTOKENS\]](#).

When creating new objects, the security descriptor from the parent container of the new object is used as the template for the security descriptor of the new object.

2.5.2.1 Access Check Algorithm Pseudocode

A support function, `SidInToken`, takes the authorization context, a [SID](#) (referenced below as the `SidToTest` parameter), and an optional `PrincipalSelfSubstitute` parameter, and returns TRUE if the **SidToTest** is present in the authorization context; otherwise, it returns FALSE. The well-known SID `PRINCIPAL_SELF`, if passed as **SidToTest**, is replaced by the `PrincipalSelfSubstituteSID` prior to the examination of the authorization context.

Any plug-in replacement is required to use this exact algorithm, which is described using the pseudo-code syntax as specified in [\[DALB\]](#).

```
BOOLEAN SidInToken(  
    Token,  
    SidToTest,  
    PrincipalSelfSubstitute )  
--  
-- On entry  
--     Token is an authorization context containing all SIDs  
--     that represent the security principal  
--     SidToTest, the SID for which to search in Token
```

```

--      PrincipalSelfSubstitute, a SID with which SidToTest may be
--      replaced

IF SidToTest is the Well Known SID PRINCIPAL_SELF THEN
    set SidToTest to be PrincipalSelfSubstitute
END IF

FOR EACH SID s in Token DO
    IF s equals SidToTest THEN
        return TRUE
    END IF
END FOR

Return FALSE

END-SUBROUTINE

```

With the SidInToken support function available, the access check proceeds as follows:

On entrance:

- SecurityDescriptor: [SECURITY_DESCRIPTOR](#) structure that is assigned to the object.
- Token: Authorization context as described above.
- Access Request mask: Set of permissions requested on the object.
- Object Tree: A tree representation of the hierarchy of objects for which to check access. Each node represents an object with two values. A GUID that represents the object itself and a value called Remaining, which indicates the user rights request for that node that have not yet been satisfied. It can be NULL.
- PrincipalSelfSubst SID: A **SID** that logically replaces the SID in any ACE that contains the well-known PRINCIPAL_SELF SID. It can be NULL.

```

Set DACL to SecurityDescriptor Dacl field
Set RemainingAccess to Access Request mask

IF RemainingAccess contains ACCESS SYSTEM SECURITY access flag THEN
    IF the Token.SystemACLAccess flag is set THEN
        Remove ACCESS SYSTEM SECURITY access bit from RemainingAccess
    END IF
END IF

IF RemainingAccess contains WRITE OWNER access bit THEN
    IF the Token.TakeOwnership flag is set THEN
        Remove WRITE OWNER access bit from RemainingAccess
    END IF
END IF

CALL SidInToken( Token, SecurityDescriptor.Owner, PrincipalSelfSubst)
IF SidInToken returns True THEN
    Remove READ CONTROL and WRITE DAC from RemainingAccess
END IF

IF Object Tree is not NULL THEN
    Set LocalTree to Object Tree
    FOR each node in LocalTree DO
        Set node.Remaining to RemainingAccess
    END FOR
END IF

```

```

    END FOR
END IF

FOR each ACE in DACL DO
    IF ACE.flag does not contain INHERIT ONLY ACE THEN

        CASE ACE.Type OF

            CASE Allow Access:

                CALL SidInToken( Token, ACE.Sid, and PrincipalSelfSubst )
                IF SidInToken returns True THEN
                    Remove ACE.AccessMask from RemainingAccess
                    FOR each node in LocalTree DO
                        Remove ACE.AccessMask from node.Remaining
                    END FOR
                END IF

            CASE Deny Access:

                CALL SidInToken( Token, ACE.Sid, PrincipalSelfSubst )
                IF SidInToken returns True THEN
                    IF any bit of RemainingAccess is in ACE.AccessMask THEN
                        Return access denied
                    END IF
                END IF

            CASE Object Allow Access:

                CALL SidInToken( Token, ACE.Sid, PrincipalSelfSubst )
                IF SidInToken returns True THEN
                    IF ACE.Object is contained in LocalTree THEN
                        Locate node n in LocalTree such that
                            n.GUID is the same as ACE.Object
                        Remove ACE.AccessMask from n.Remaining
                        FOR each node ns such that ns is a descendent of n DO
                            Remove ACE.AccessMask from ns.Remaining
                        END FOR
                        FOR each node np such that np is an ancestor of n DO
                            Set np.Remaining = np.Remaining or np-1.Remaining

                                -- the 'or' above is a logical bitwise OR operator. For
                                -- Some uses (like Active Directory), a hierarchical list
                                -- of types can be passed in; if the requestor is granted
                                -- access to a specific node, this will grant access to
                                -- all its children. The preceding lines implement this by
                                -- removing, from each child, the permissions just found for
                                -- the parent. The change is propagated upwards in
                                -- the tree: once a permission request has been satisfied
                                -- we can tell the next-higher node that we do not need
                                -- to inherit it from the higher node (we already have it
                                -- in the current node). And since we must not blindly
                                -- replace the parent's RemainingAccess, we BIT OR the
                                -- parent's RemainingAccess with the current node's. This
                                -- way, if the parent needs, say, READ_CONTROL, and the
                                -- current node was just granted that, the parent's
                                -- RemainingAccess still contains this bit since satisfying
                                -- the request at a lower level does nothing to affect
                                -- the higher level node.

                            END FOR
                        END IF
                    END IF

            CASE Object Deny Access:

```

```

CALL SidInToken( Token, ACE.Sid, PrincipalSelfSubst )
IF SidInToken returns True THEN
  Locate node n in LocalTree such that
    n.GUID is the same as ACE.Object
  IF n exists THEN
    If any bit of n.Remaining is in ACE.AccessMask THEN
      Return access_denied
    END IF
  END IF
END IF

END CASE

END IF
END FOR

IF RemainingAccess = 0 THEN
  Return success
Else
  Return access denied
END IF

```

2.5.2.2 Algorithm for Creating a Security Descriptor

ACL inheritance: The algorithm for computing the system and discretionary [ACL](#) (SACL and DACL respectively) in the security descriptor for the new object is governed by the logic illustrated in the following table.

	Explicit (non default) ACL specified by creator	Explicit default ACL specified by creator	No ACL specified by the creator
Inheritable ACL from parent	Assign specified ACL (1) (2)	Assign inherited ACL	Assign inherited ACL
No inheritable ACL from parent	Assign specified ACL (1)	Assign Default ACL	Assign no ACL

1. Any ACEs with the INHERITED_ACE bit set are NOT copied to the assigned security descriptor.
2. If *AutoInheritFlags* is set to automatically inherit ACEs from the parent (SEF_DACL_AUTO_INHERIT or SEF_SACL_AUTO_INHERIT), inherited ACEs from the parent are appended after explicit ACEs from the *CreatorDescriptor*. For further details, see the parameter list for [CreateSecurityDescriptor \(section 2.5.2.3\)](#).

Note An explicitly specified ACL, whether a default ACL or not, may be empty or NULL. [<14>](#)

The rest of this section documents the details of the algorithm outlined above as a set of nested sub-procedures.

2.5.2.3 CreateSecurityDescriptor

Parameters

- *ParentDescriptor*: Security descriptor for the parent (container) object of the new object. If the object has no parent, this parameter is NULL.

- *CreatorDescriptor*: Security descriptor for the new object provided by the creator of the object. Caller can pass NULL.
- *IsContainerObject*: **BOOLEAN**: TRUE when the object is a container, FALSE otherwise.
- *ObjectTypes*: An array of pointers to GUID structures that identify the object types or classes of the object associated with *NewDescriptor* (the return value). For Active Directory objects, this array contains pointers to the class GUIDs of the object's structural class and all attached auxiliary classes. If the object for which this descriptor is being created does not have a GUID, this field **MUST** be set to null.
- *AutoInheritFlags*: A set of bit flags that control how access control entries (ACEs) are inherited from *ParentDescriptor*. This parameter can be a combination of the following values.
 - **DACL_AUTO_INHERIT**: If set, inheritable ACE's from the parent security descriptor DACL are merged with the explicit ACE's in the *CreatorDescriptor*.
 - **SACL_AUTO_INHERIT**: If set, inheritable ACE's from the parent security descriptor SACL are merged with the explicit ACE's in the *CreatorDescriptor*.
 - **DEFAULT_DESCRIPTOR_FOR_OBJECT**: Selects the *CreatorDescriptor* as the default security descriptor provided that no object type specific ACEs are inherited from the parent. If such ACEs do get inherited, *CreatorDescriptor* is ignored.
 - **DEFAULT_OWNER_FROM_PARENT**: Relevant only when the owner field is not specified in *CreatorDescriptor*. If this flag is set, the owner field in *NewDescriptor* is set to the owner of *ParentDescriptor*. If not set, the owner from the token is selected.
 - **DEFAULT_GROUP_FROM_PARENT**: Relevant only when the primary group field is not specified in *CreatorDescriptor*. If this flag is set, the primary group of *NewDescriptor* is set to the primary group of *ParentDescriptor*. If not set, the default group from the token is selected.
- *Token*: Token supplied by the caller for default security information for the new object.
- *GenericMapping*: Mapping of generic permissions to resource manager-specific permissions supplied by the caller.

Returns

- *NewDescriptor*: Output security descriptor for the object computed by the algorithm.

```
// Step 1:Compute the Owner field
IF CreatorDescriptor.Owner is NULL THEN

    IF AutoInheritFlags contains DEFAULT_OWNER_FROM_PARENT THEN
        Set NewDescriptor.Owner to ParentDescriptor.Owner
    ELSE
        Set NewDescriptor.Owner to Token.SIDs[Token.OwnerIndex]
    ENDIF

ELSE
    Set NewDescriptor.Owner to CreatorDescriptor.Owner
ENDIF

// Step 2:Compute the Group field
IF CreatorDescriptor.Group is NULL THEN
```

```

    IF AutoInheritFlags contains DEFAULT_GROUP_FROM_PARENT THEN
        Set NewDescriptor.Group to ParentDescriptor.Group
    ELSE
        Set NewDescriptor.Group to Token.PrimaryGroup
    ENDIF

ELSE
    Set NewDescriptor.Group to CreatorDescriptor.Group
ENDIF

// Step 3:Compute the DACL

CALL ComputeACL WITH
    COMPUTE_DACL, ParentDescriptor.DACL, ParentDescriptor.Control,
    CreatorDescriptor.DACL,CreatorDescriptor.Control
    IsContainerObject, ObjectTypes, GenericMapping,
    NewDescriptor.Owner, NewDescriptor.Group, Token
RETURNING NewDACL, NewControl

Set NewDescriptor.DACL to NewDACL
Set NewDescriptor.Control to NewControl

// Step 4:Compute the SACL

CALL ComputeACL WITH
    COMPUTE_SACL, ParentDescriptor.SACL, ParentDescriptor.Control,
    CreatorDescriptor.SACL,CreatorDescriptor.Control
    IsContainerObject, ObjectTypes, GenericMapping,
    NewDescriptor.Owner, NewDescriptor.Group, Token
RETURNING NewSACL, NewControl

Set NewDescriptor.SACLto NewSACL
Set NewDescriptor.Control to (NewDescriptor.Control OR NewControl)

RETURN NewDescriptor
// END CreateSecurityDescriptor

```

2.5.2.4 ComputeACL

Parameters

- *ComputeType*: Enumeration of COMPUTE_DACL and COMPUTE_SACL.
- *ParentACL*: ACL from the parent security descriptor.
- *ParentControl*: Control flags from the parent security descriptor.
- *CreatorACL*: ACL supplied in the security descriptor by the creator.
- *CreatorControl*: Control flags supplied in the security descriptor by the creator.
- *IsContainerObject*: TRUE if the object is a container, FALSE otherwise.
- *ObjectTypes*: Array of GUIDs for the object type being created.

- *GenericMapping*: Mapping of generic permissions to resource manager-specific permissions supplied by the caller.
- *Owner*: Owner to use in substituting the *CreatorOwner* SID.
- *Group*: Group to use in substituting the *CreatorGroup* SID.
- *Token*: Token for default values.

Returns

- Computed ACL
- ComputedControl

```
// The details of the algorithm to merge the parent ACL and the supplied ACL.
// The Control flags computed are slightly different based on whether it is the
// ACL in the DACL or the SACL field of the descriptor.
// The caller specifies whether it is a DACL or a SACL using the parameter,
// ComputeType.
Set ComputedACL to NULL
Set ComputedControl to NULL

CALL ContainsInheritableACEs WITH ParentACL RETURNING result

IF result = TRUE THEN
  // ParentACL contains inheritable ACEs
  IF (CreatorACL is not present) OR
    ((CreatorACL is present) AND
     (AutoInheritFlags contains DEFAULT DESCRIPTOR))
  THEN
    // Use only the inherited ACEs from the parent
    CALL ComputeInheritedACLFromParent WITH
      ParentACL, IsContainerObject, ObjectTypes
    RETURNING NextACL
    CALL PostProcessACL WITH
      NextACL, Owner, Group, GenericMapping
    RETURNING FinalACL
    Set ComputedACL to FinalACL
  ENDIF

  IF ((CreatorACL is present) AND
      (AutoInheritFlags does not contain DEFAULT DESCRIPTOR))
  THEN
    CALL PreProcessACLFromCreator WITH CreatorACL
    RETURNING PreACL

    CALL ComputeInheritedACLFromCreator WITH
      PreACL, IsContainerObject, ObjectTypes
    RETURNING TmpACL

    IF ((ComputeType = DACL COMPUTE) AND
        (CreatorControl does not contain DACL PROTECTED) AND
        (AutoInheritFlags contains DACL AUTO INHERIT flag))
    THEN
      // Compute the inherited ACEs from the parent
      CALL ComputeInheritedACLFromParent WITH
        ParentACL, IsContainerObject, ObjectTypes
      RETURNING ParentACL
      Append ParentACL.ACEs to TmpACL.ACEs
      Add DACL_AUTO_INHERITED to ComputedControl
    ELSE
      IF ((ComputeType = SACL COMPUTE) AND
```

```

        (CreatorControl does not contain SACL PROTECTED) AND
        (AutoInheritFlags contains SACL_AUTO_INHERIT flag))
    THEN
        // Compute the inherited ACEs from the parent
        CALL ComputeInheritedACLFromParent WITH
            ParentACL, IsContainerObject, ObjectTypes
        RETURNING ParentACL
        Append ParentACL.ACEs to TmpACL.ACEs
        Add SACL_AUTO_INHERITED to ComputedControl
    ENDIF
ENDIF

CALL PostProcessACL WITH
    TmpACL, Owner, Group, GenericMapping
RETURNING ProcessedACL

Set ComputedACL to ProcessedACL

ELSE // ParentACL does not contain inheritable ACEs

IF CreatorACL = NULL THEN
    // No ACL supplied for the object
    IF (ComputeType = DACL COMPUTE) THEN
        Set ComputedACL to Token.DefaultDACL
    ELSE
        // No default for SACL; left as NULL
    ENDIF
ENDIF

ELSE
    // Explicit ACL was supplied for the object - either default or not.
    // In either case, use it for the object, since there are no inherited ACEs.
    CALL PreProcessACLFromCreator WITH CreatorACL
    RETURNING TmpACL
    CALL PostProcessACL WITH
        TmpACL, Owner, Group, GenericMapping
    RETURNING ProcessedACL
    Set ComputedACL to ProcessedACL
ENDIF

ENDIF
// END ComputeACL

```

2.5.2.5 ContainsInheritableACEs

Parameters

- *ACL*

Returns

- TRUE or FALSE

```

// Computes whether the ACL parameter contains any ACEs that are inheritable
// by a child
// True: if it contains any inheritable ACEs
// False: otherwise

FOR each ACE in ACL DO
    IF(ACE.Flags contains CONTAINER_INHERIT) OR
       (ACE.Flags contains OBJECT_INHERIT)

```

```

        THEN
            RETURN TRUE
        ENDIF
    END FOR

    RETURN FALSE
    // END ContainsInheritableACEs

```

2.5.2.6 ComputeInheritedACLfromParent

Parameters

- *ACL*: [ACL](#) that contains the parent's ACEs from which to compute the inherited **ACL**.
- *IsContainerObject*: TRUE if the object is a container, FALSE otherwise.
- *ObjectTypes*: Array of GUIDs for the object type being created.

Returns

- The computed **ACL** that also includes the inherited ACEs.

```

// Computes the inheritable and inherited ACEs to propagate to the new object
// from the inheritable ACEs in the parent container object

Initialize ExplicitACL to Empty ACL

FOR each ACE in ACL DO

    IF ACE.Flags contains INHERIT ONLY
    THEN
        CONTINUE
    ENDIF

    IF(((ACE.Flags contains CONTAINER INHERIT) AND
        (IsContainerObject = TRUE))OR
        ((ACE.Flags contains OBJECT_INHERIT) AND
        (IsContainerObject = FALSE)))
    THEN

        CASE ACE.Type OF

            ALLOW:
            DENY:
                Set NewACE to ACE
                Set NewACE.Flags to INHERITED
                Append NewACE to ExplicitACL

            OBJECT_ALLOW:
            OBJECT_DENY:
                IF (ObjectTypes contains ACE.ObjectGUID) THEN
                    Set NewACE to ACE
                    Set NewACE.Flags to INHERITED
                    Append NewACE to ExplicitACL
                ENDIF
            ENDCASE
        ENDIF
    END FOR

    Initialize InheritableACL to Empty ACL

```

```

IF (IsContainerObject = TRUE) THEN
    FOR each ACE in ACL DO
        IF ACE.Flags contains NO PROPAGATE THEN
            CONTINUE
        ENDIF
        IF((ACE.Flags contains CONTAINER_INHERIT) OR
           (ACE.Flags contains OBJECT_INHERIT))
        THEN
            Set NewACE to ACE
            Add INHERITED to NewACE.Flags
            Add INHERIT_ONLY to NewACE.Flags
            Append NewACE to InheritableACL
        ENDIF
    END FOR
ENDIF

RETURN concatenation of ExplicitACL and InheritableACL
// END ComputeInheritedACLFromParent

```

2.5.2.7 ComputeInheritedACLfromCreator

Parameters

- *ACL*: [ACL](#) supplied in the security descriptor by the caller.
- *IsContainerObject*: TRUE if the object is a container, FALSE otherwise.
- *ObjectTypes*: Array of GUIDs for the object type being created.

Returns

- The computed **ACL** that also includes the inherited ACEs.

```

// Computes the inheritable and inherited ACEs to propagate to the new object
// from any inheritable ACEs in the ACL supplied by the caller

Initialize ExplicitACL to Empty ACL

FOR each ACE in ACL DO
    IF((ACE.Flags contains CONTAINER_INHERIT) AND
       (IsContainerObject = TRUE))OR
       ((ACE.Flags contains OBJECT_INHERIT) AND
        (IsContainerObject = FALSE))
    THEN
        CASE ACE.Type OF
            ALLOW:
            DENY:
                Set NewACE to ACE
                Set NewACE.Flags to NULL
                Append NewACE to ExplicitACL

            OBJECT ALLOW
            OBJECT DENY:
                IF (ObjectTypes contains ACE.ObjectGUID) THEN
                    Set NewACE to ACE
                ENDIF
        END CASE
    ENDIF
END FOR

```

```

        Set NewACE.Flags to NULL
        Append NewACE to ExplicitACL
    ENDIF

    ENDCASE
ENDIF
END FOR

Initialize InheritableACL to Empty ACL

IF (IsContainerObject = TRUE) THEN

    FOR each ACE in ACL DO
        IF((ACE.Flags contains CONTAINER_INHERIT) OR
           (ACE.Flags contains OBJECT_INHERIT))
            THEN
                Set NewACE to ACE
                Add INHERIT ONLY to NewACE.Flags
                Append NewACE to InheritableACL
            ENDIF
        END FOR
    ENDIF

RETURN concatenation of ExplicitACL and InheritableACL
// END ComputeInheritedACLFromCreator

```

2.5.2.8 PreProcessACLfromCreator

Parameters

- *ACL*: [ACL](#) to preprocess.

Returns

- Processed **ACL**.

```

Initialize NewACL to Empty ACL

FOR each ACE in ACL DO
    IF ACE.Flags does not contain INHERITED THEN
        Append ACE to NewACL
    ENDIF
END FOR

RETURN NewACL
// END PreProcessACLFromCreator

```

2.5.2.9 PostProcessACL

Parameters

- *ACL*: [ACL](#) on which to substitute SIDs.
- *Owner*: Owner to use in substituting the *CreatorOwner* SID.
- *Group*: Group to use in substituting the *CreatorGroup* SID.

- *GenericMapping*: Mapping of generic permissions to resource manager-specific permissions supplied by the caller.

Returns

- The computed **ACL** with the [SID](#) substitutions performed.

```
// Substitute CreatorOwner and CreatorGroup SIDs and do GenericMapping in ACL
Initialize NewACL to Empty ACL

FOR each ACE in ACL DO
  Set NewACE to ACE
  IF (ACE.Flags = INHERITED) OR
    (ACE.Flags = NULL)
  THEN

    CASE ACE.SID OF
      CREATOR OWNER:
        NewACE.SID = Owner
      CREATOR GROUP:
        NewACE.SID = Group
    ENDCASE

    IF (ACE.Flags contains GENERIC_READ) THEN
      Add GenericMapping.GenericRead to NewACE.Flags
    ENDIF
    IF (ACE.Flags contains GENERIC_WRITE) THEN
      Add GenericMapping.GenericWrite to NewACE.Flags
    ENDIF
    IF (ACE.Flags contains GENERIC_EXECUTE) THEN
      Add GenericMapping.GenericExecute to NewACE.Flags
    ENDIF
    Append NewACE to NewACL
  ENDIF
END FOR
RETURN NewACL
// END PostProcessACL
```

3 Structure Examples

There are no structure examples.

4 Security Considerations

There are no security considerations.

5 Appendix A: Full MS-DTYP IDL

For ease of implementation and to allow re-use of the common data types and structure in other protocols, a full IDL is provided.

```
typedef int BOOL, *PBOOL, *LPBOOL;
typedef unsigned char BYTE, *PBYTE, *LPBYTE;
typedef BYTE BOOLEAN, *PBOOLEAN;
typedef wchar_t WCHAR, *PWCHAR;
typedef WCHAR* BSTR;
typedef char CHAR, *PCHAR;
typedef double DOUBLE;
typedef unsigned long DWORD, *PDWORD, *LPDWORD;
typedef unsigned int DWORD32;
typedef __int64 DWORD64;
typedef unsigned __int64 ULONGLONG;
typedef ULONGLONG DWORDLONG, *PDWORDLONG;
typedef unsigned long error_status_t;
typedef float FLOAT;
typedef unsigned char UCHAR, *PUCHAR;
typedef short SHORT;

typedef void *HANDLE;
typedef DWORD HCALL;
typedef DWORD HRESULT;
typedef int INT, *LPINT;
typedef signed char INT8;
typedef signed short INT16;
typedef signed int INT32;
typedef __int64 INT64;
typedef const wchar_t* LMCSTR;
typedef WCHAR* LMSTR;
typedef long LONG, *PLONG, *LPLONG;
typedef INT64 LONGLONG;

#ifdef _WIN64
    typedef __int64 LONG_PTR;
#else
    typedef LONG LONG_PTR;
#endif

typedef signed int LONG32;
typedef signed __int64 LONG64;
typedef const char* LPCSTR;

#ifdef Unicode
    typedef const wchar_t* LPCTSTR;
#else
    typedef const char* LPCTSTR;
#endif

typedef const wchar_t* LPCWSTR;
typedef char* PSTR, *LPSTR;

#ifdef Unicode
    typedef WCHAR* LPTSTR;

```

```

#else
    typedef CHAR* LPTSTR;
#endif

typedef wchar_t* LPWSTR, *PWSTR;
typedef DWORD NET_API_STATUS;
typedef long NTSTATUS;
typedef [context_handle] void * PCONTEXT_HANDLE, *PPCONTEXT_HANDLE;
typedef unsigned __int64 QWORD;
typedef void* RPC_BINDING_HANDLE;
typedef UCHAR* STRING;

#ifdef Unicode
    typedef WCHAR TCHAR;
#else
    typedef CHAR TCHAR;
#endif

typedef __int64 TIME;
typedef unsigned int UINT;
typedef unsigned char UINT8;
typedef unsigned short UINT16;
typedef unsigned int UINT32;
typedef unsigned __int64 UINT64;
typedef unsigned long ULONG, *PULONG;

#ifdef _WIN64

typedef unsigned __int64 ULONG_PTR;

#else

typedef ULONG ULONG_PTR;

#endif

typedef ULONG_PTR DWORD_PTR;
typedef unsigned int ULONG32;
typedef unsigned __int64 ULONG64;
typedef wchar_t UNICODE;
typedef unsigned short USHORT;
typedef void VOID, *PVOID;
typedef unsigned short WORD, *PWORD, *LPWORD;

typedef struct _FILETIME {
    DWORD dwLowDateTime;
    DWORD dwHighDateTime;
} FILETIME,
*PFILETIME,
*LPFILETIME;

typedef struct _GUID {
    unsigned long Data1;
    unsigned short Data2;
    unsigned short Data3;
    byte Data4[8];
} GUID,

```

```

    UUID,
    *PGUID;

typedef struct _LARGE_INTEGER {
    __int64 QuadPart;
} LARGE_INTEGER, *PLARGE_INTEGER;

typedef DWORD LCID;

typedef struct _RPC_UNICODE_STRING {
    unsigned short Length;
    unsigned short MaximumLength;
    [size_is(MaximumLength/2), length_is(Length/2)]
    WCHAR* Buffer;
} RPC_UNICODE_STRING,
*PRPC_UNICODE_STRING;

typedef struct _SYSTEMTIME {
    WORD wYear;
    WORD wMonth;
    WORD wDayOfWeek;
    WORD wDay;
    WORD wHour;
    WORD wMinute;
    WORD wSecond;
    WORD wMilliseconds;
} SYSTEMTIME,
*PSYSTEMTIME;

typedef struct _UINT128 {
    UINT64 lower;
    UINT64 upper;
} UINT128,
*PUINT128;

typedef struct _ULARGE_INTEGER {
    unsigned __int64 QuadPart;
} ULARGE_INTEGER, *PULARGE_INTEGER;

typedef struct _UNICODE_STRING {
    USHORT Length;
    USHORT MaximumLength;
    [size_is(MaximumLength/2), length_is(Length/2)]
    WCHAR* Buffer;
} UNICODE_STRING,
*PUNICODE_STRING;

typedef struct _SID_IDENTIFIER_AUTHORITY {
    BYTE Value[6];
} SID_IDENTIFIER_AUTHORITY;

typedef struct _SID{
    BYTE Revision;
    BYTE SubAuthorityCount;
    SID_IDENTIFIER_AUTHORITY IdentifierAuthority;
    [size_is(SubAuthorityCount)] ULONG SubAuthority[*];
} SID,
*PSID;

```

```

typedef struct _ACCESS_MASK {
    unsigned long ACCESS_MASK;
} ACCESS_MASK,
*PACCESS_MASK;

typedef struct _ACE_HEADER {
    UCHAR AceType;
    UCHAR AceFlags;
    USHORT AceSize;
} ACE_HEADER,
*PACE_HEADER;

typedef struct _ACCESS_ALLOWED_ACE {
    ACE_HEADER Header;
    ACCESS_MASK Mask;
    DWORD SidStart;
} ACCESS_ALLOWED_ACE,
*PACCESS_ALLOWED_ACE;

typedef struct _ACCESS_ALLOWED_OBJECT_ACE {
    ACE_HEADER Header;
    ACCESS_MASK Mask ;
    DWORD Flags ;
    GUID ObjectType;
    GUID InheritedObjectType;
    DWORD SidStart;
} ACCESS_ALLOWED_OBJECT_ACE,
*PACCESS_ALLOWED_OBJECT_ACE;

typedef struct _ACCESS_DENIED_ACE {
    ACE_HEADER Header;
    ACCESS_MASK Mask;
    DWORD SidStart;
} ACCESS_DENIED_ACE,
*PACCESS_DENIED_ACE;

typedef struct _ACCESS_ALLOWED_CALLBACK_ACE {
    ACE_HEADER Header;
    ACCESS_MASK Mask ;
    DWORD SidStart;
} ACCESS_ALLOWED_CALLBACK_ACE,
*PACCESS_ALLOWED_CALLBACK_ACE;

typedef struct _ACCESS_DENIED_CALLBACK_ACE {
    ACE_HEADER Header;
    ACCESS_MASK Mask ;
    DWORD SidStart;
} ACCESS_DENIED_CALLBACK_ACE,
*PACCESS_DENIED_CALLBACK_ACE;

typedef struct _ACCESS_ALLOWED_CALLBACK_OBJECT_ACE {
    ACE_HEADER Header;
    ACCESS_MASK Mask ;
    DWORD Flags ;
    GUID ObjectType;
    GUID InheritedObjectType;
    DWORD SidStart;
}

```

```

} ACCESS_ALLOWED_CALLBACK_OBJECT_ACE,
*PACCESS_ALLOWED_CALLBACK_OBJECT_ACE;

typedef struct _ACCESS_DENIED_CALLBACK_OBJECT_ACE {
    ACE_HEADER Header;
    ACCESS_MASK Mask ;
    DWORD Flags ;
    GUID ObjectType;
    GUID InheritedObjectType;
    DWORD SidStart;
} ACCESS_DENIED_CALLBACK_OBJECT_ACE,
*PACCESS_DENIED_CALLBACK_OBJECT_ACE;

typedef struct _SYSTEM_AUDIT_ACE {
    ACE_HEADER Header;
    ACCESS_MASK Mask ;
    DWORD SidStart;
} SYSTEM_AUDIT_ACE,
*PSYSTEM_AUDIT_ACE;

typedef struct _SYSTEM_AUDIT_CALLBACK_ACE {
    ACE_HEADER Header;
    ACCESS_MASK Mask ;
    DWORD SidStart;
} SYSTEM_AUDIT_CALLBACK_ACE,
*PSYSTEM_AUDIT_CALLBACK_ACE;

typedef struct _SYSTEM_MANDATORY_LABEL_ACE {
    ACE_HEADER Header;
    ACCESS_MASK Mask ;
    DWORD SidStart;
} SYSTEM_MANDATORY_LABEL_ACE,
*PSYSTEM_MANDATORY_LABEL_ACE;

typedef struct _SYSTEM_AUDIT_CALLBACK_OBJECT_ACE {
    ACE_HEADER Header;
    ACCESS_MASK Mask ;
    DWORD Flags ;
    GUID ObjectType;
    GUID InheritedObjectType;
    DWORD SidStart;
} SYSTEM_AUDIT_CALLBACK_OBJECT_ACE,
*PSYSTEM_AUDIT_CALLBACK_OBJECT_ACE;

typedef struct _ACL {
    UCHAR AclRevision;
    UCHAR Sbz1;
    USHORT AclSize;
    USHORT AceCount;
    USHORT Sbz2;
} ACL,
*PACL;

typedef struct _SECURITY_DESCRIPTOR {
    UCHAR Revision;
    UCHAR Sbz1;
    USHORT Control;
    ULONG Owner;

```

```
    ULONG Group;
    ULONG Sacl;
    ULONG Dacl;
} SECURITY_DESCRIPTOR,
 *PSECURITY_DESCRIPTOR;

typedef DWORD SECURITY_INFORMATION, *PSECURITY_INFORMATION;

typedef struct _RPC_SID {
    unsigned char Revision;
    unsigned char SubAuthorityCount;
    SID_IDENTIFIER_AUTHORITY IdentifierAuthority;
    [size_is(SubAuthorityCount)] unsigned long SubAuthority[];
} RPC_SID,
 *PRPC_SID;
```

6 Appendix B: Windows Behavior

The information in this specification is applicable to the following versions of Windows:

- Windows Server 2003
- Windows NT
- Windows Vista
- Windows XP
- Windows 2000

Exceptions, if any, are noted below. Unless otherwise specified, any statement of optional behavior in this specification that is prescribed by using the terms SHOULD or SHOULD NOT implies Windows behavior in accordance with the SHOULD or SHOULD NOT prescription. Unless otherwise specified, the term MAY implies that Windows does not follow the prescription.

[<1> Section 2.4.2:](#) While in abstract, the **SID** is a hierarchical sequence that, in practice, is difficult to manage and deploy. When an account domain is created in Windows, three subauthorities are generated randomly to serve as the account domain portion of the SID. Windows uses the Identifier Authority value 5 to denote SIDs created by Windows. An initial **SubAuthority** value of 21 is used to denote SIDs that use this set of three random subauthorities. Thus, a Windows domain would have SIDs that appear as "S-1-5-21-x1-x2-x3-rid", where x1 through x3 are random numbers assigned during domain creation, and the RID is a domain-specific unique number assigned to the user or group during its creation.

[<2> Section 2.4.4.1:](#) Windows NT 4.0: Not supported.

[<3> Section 2.4.4.1:](#) Windows NT 4.0: Not supported.

[<4> Section 2.4.4.1:](#) Windows NT 4.0: Not supported.

[<5> Section 2.4.4.1:](#) Windows NT 4.0 and Windows 2000: Not supported.

[<6> Section 2.4.4.1:](#) Windows NT 4.0 and Windows 2000: Not supported.

[<7> Section 2.4.4.1:](#) Windows NT 4.0 and Windows 2000: Not supported.

[<8> Section 2.4.4.1:](#) Windows NT 4.0 and Windows 2000: Not supported.

[<9> Section 2.4.4.1:](#) Windows NT 4.0 and Windows 2000: Not supported.

[<10> Section 2.4.4.1:](#) Windows NT 4.0: Not supported.

[<11> Section 2.4.4.11:](#) This construct is supported only by Windows Server 2008 and Windows Vista.

[<12> Section 2.4.5:](#) This is applicable for Windows Vista or later releases of Windows.

[<13> Section 2.5.1:](#) SDDL was introduced in Windows 2000.

[<14> Section 2.5.2.2:](#) Assigning the owner and group fields in the security descriptor MUST follow the following logic.

1. If the security descriptor supplied for the object by the caller includes an owner, it is assigned as the new object's owner. Otherwise, if the DEFAULT_OWNER_FROM_PARENT flag (see section

- [2.5.2.3](#)) is set, the new object is assigned the same owner as the parent object. If this flag is not set, the default owner specified by the token (see section [2.5.2.3](#)) is assigned.
2. If the security descriptor supplied for the object by the caller includes a group, it is assigned as the new object's group. Otherwise, if the `DEFAULT_GROUP_FROM_PARENT` flag (see section [2.5.2.3](#)) is set, the new object is assigned the same primary group as the parent object. If this flag is not set, the default group specified by the token (see section [2.5.2.3](#)) is assigned.

7 Index

A

[ACCESS_ALLOWED_ACE structure](#)
[ACCESS_ALLOWED_CALLBACK_ACE structure](#)
[ACCESS_ALLOWED_CALLBACK_OBJECT_ACE structure](#)
[ACCESS_ALLOWED_OBJECT_ACE structure](#)
[ACCESS_DENIED_ACE structure](#)
[ACCESS_DENIED_CALLBACK_ACE structure](#)
[ACCESS_DENIED_CALLBACK_OBJECT_ACE structure](#)
[ACCESS_MASK structure](#)
[ACE_HEADER structure](#)
[ACL structure](#)
[Applicability](#)

C

[Capability negotiation](#)
[Common base types](#)
[Common data structures](#)
[Common data types](#)
[Constructed security types](#)

D

[Data types](#)
 [common base types](#)
 [common data structures](#)
 [common data types](#)
 [constructed security types](#)
 [security types - additional information](#)

E

[Examples - structure](#)

F

[Fields - vendor-extensible](#)
[FILETIME structure](#)

G

[Glossary](#)
[GUID structure](#)

I

[Informative references](#)
[Introduction](#)

L

[LARGE_INTEGER structure](#)
[LPFILETIME](#)

N

[Normative references](#)

P

[PACCESS_ALLOWED_ACE](#)
[PACCESS_ALLOWED_CALLBACK_ACE](#)
[PACCESS_ALLOWED_CALLBACK_OBJECT_ACE](#)
[PACCESS_ALLOWED_OBJECT_ACE](#)
[PACCESS_DENIED_ACE](#)
[PACCESS_DENIED_CALLBACK_ACE](#)
[PACCESS_DENIED_CALLBACK_OBJECT_ACE](#)
[PACCESS_MASK](#)
[PACE_HEADER](#)
[PACL](#)
[PFILETIME](#)
[PGUID](#)
[PLARGE_INTEGER](#)
[PRPC_SID](#)
[PRPC_UNICODE_STRING](#)
[PSECURITY_DESCRIPTOR](#)
[PSID](#)
[PSYSTEM_AUDIT_ACE](#)
[PSYSTEM_AUDIT_CALLBACK_ACE](#)
[PSYSTEM_AUDIT_CALLBACK_OBJECT_ACE](#)
[PSYSTEM_MANDATORY_LABEL_ACE](#)
[PSYSTEMTIME](#)
[PUINT128](#)
[PULARGE_INTEGER](#)
[PUNICODE_STRING](#)

R

References
 [informative](#)
 [normative](#)
 [overview](#)
[Relationship to other protocols](#)
[RPC_SID structure](#)
[RPC_UNICODE_STRING structure](#)

S

[Security considerations](#)
[Security types - data types](#)
[SECURITY_DESCRIPTOR structure](#)
[SID structure](#)
[SID_IDENTIFIER_AUTHORITY structure](#)
[Structure examples](#)
[SYSTEM_AUDIT_ACE structure](#)
[SYSTEM_AUDIT_CALLBACK_ACE structure](#)
[SYSTEM_AUDIT_CALLBACK_OBJECT_ACE structure](#)
[SYSTEM_MANDATORY_LABEL_ACE structure](#)
[SYSTEMTIME structure](#)

U

[UINT128 structure](#)
[ULARGE_INTEGER structure](#)
[UNICODE_STRING structure](#)
[UUID](#)

V

[Vendor-extensible fields
Versioning](#)

W

[Windows behavior](#)