Portable Systems Group

NT OS/2 Object Management Specification

Author: *Steven R. Wood*

Revision 1.6, May 24, 1991
Original Draft February 17, 1989

1. Overview

This specification describes the Object Management for the NT OS/2 system. Object Management is provided by a set of routines that are available within the NT OS/2 executive and invoked from kernel mode. This specification also describes generic object management user level NT routines and support for directories.

1.1 What is an Object?

An object is an opaque data structure that defines a protected entity that is implemented and manipulated by the operating system. A particular object type is described by the set of operations that may be performed upon it (wait, create, clear, set, cancel,...) and its relationship to other objects. All objects have the same standard set of rules for creation, deletion, protection, access, management, and naming.

1.2 Object Management Goals

- Provide an extensible, well defined mechanism for the definition and manipulation of executive data structures.

- Provide uniform rules for object retention. This is especially important in a multiprocessor system.

- Provide uniform security and protection that allows certification at C2 and beyond without modification.

- Provide a mechanism to add new object types to the system without modifying existing system code. This means that only the object type specific routines should have knowledge of the internal structure of a particular object type.

- Provide orthogonal specification of APIs which operate on objects.

- Provide attributes on objects to support POSIX compatibility.

- Provide a naming hierarchy which is integrated with the file system and mimics the OS/2 and POSIX file system directory hierarchy.

1.3 Object Data Structures

An instance of an object type is represented by a data structure which contains a standard object header and an object type specific object body. The object management routines operate on the object header, while the object type's specific routines operate on the object body.

1.4 Object Header

The object header contains information used by the object management routines to manipulate the object. The following items are maintained in the object header:

- Pointer to the name of the object, if any.
- Pointer to the directory object which contains this object's name, if any.
- Pointer to the SecurityDescriptor for the object, if any.
- AccessMode of the object, either KernelMode only or UserMode and KernelMode.
- Pointer to the Owner Process of the object for exclusive objects, if any.
- Retention counts for the object.
- Pointer to an optional handle count data base, that maintains a per process handle count for a given object.
- Pointer to the object type structure that defines the type of the object.
- Permanent / temporary attribute.
- Paged and nonpaged pool quota charges associated with the object.
- Structure control linking all objects of the same type together.

1.5 Object Types

Every object has an object type. The object type is defined by an Object Type Descriptor structure. An object type is nothing more than an object whose object body contains the following information:

- Type specific mutex.
- Structure control linking all objects of the same type together.

- Dispatcher object offset.

- Pool type to use when allocating space for objects of this type.

- Invalid object attribute bits.

- Mapping vector to map generic access bits into standard and/or specific access bits.

- Valid access bits.

- Pointer to a type specific dump procedure, if any.

- Pointer to a type specific delete procedure, if any.

- Pointer to a type specific open procedure, if any.

- Pointer to a type specific close procedure, if any.

- Pointer to a type specific parse procedure, if any.

- Pointer to a type specific security procedure, if any.

These items are used to manage type specific attributes of each object. The type specific mutex is acquired whenever an object of that type is being created, deleted, or having its security descriptor examined or modified. This prevents race conditions between object creation and deletion.

The SecurityDescriptor associated with an object type descriptor is examined for OBJECT_TYPE_CREATE access by the ObCreateObject function every time an object of the corresponding object type is created. This provides a mechanism to grant or deny the ability to create objects of a specific type on an individual identifier basis using the SecurityDescriptor associated with the object type descriptor structure.

The name is used to uniquely identify the type. All type names are stored in the \ ObjectTypes object directory.

The pool type determines whether the object header and object body are allocated from paged pool or non-paged pool.

The dispatcher offset is used to implement a generic wait function. Waiting on an object waits on the offset within the object body specified by the dispatcher offset. This allows a program to wait on multiple objects of different types or a single object of unknown type, without having to know the object type.

The six type specific procedures are called whenever a type specific action must be performed from within the context of the object manager.

1.6 Object Handles

An object handle is a 32-bit opaque pointer to an object. There may be more than one handle for a given object, as a result of sharing via inheritance or naming. Associated with each handle is a pointer to the object, a granted access mask that was computed at the time the handle was created and handle attributes such as where the handle should be inherited on child process creation.

Object handles are created by inserting an object into an object table. An object table consists of an array of object table entries. An object handle is an index into an object table to the object table entry for that handle. The object table entry contains the information associated with the handle (i.e. the pointer to the object, the granted access mask and the handle attributes). There is an object table associated with each process. Thus handles are process specific, and meaningless outside of the context in which the handle was created. All object handles associated with a process are automatically "closed" upon that process terminating.

Each object table has a mutex associated with it. This mutex is acquired any time the object table is examined or modified.

The low order 2 bits of a 32-bit object handle are set to zero by the system when a handle is created and are ignored by all system services that accept a handle. This allows applications to encode application specific type information in the low order two bits.

In the debugging version of the system, part of each 32-bit object handle is reserved for a serial number that is also stored in the associated object table entry. When an object handle is used to reference an object, the serial number in the 32-bit handle is compared with that in the object table entry and an error is returned if they don't match. This will catch cases when an old handle is reused inadvertantly.

When creating a handle to an object, the caller may specify a *DesiredAccess* parameter. The Object Manager probes the security descriptor associated with an object with the *DesiredAccess* parameter. If all requested access bits are allowed by the security descriptor then the access is granted, and the *DesiredAccess* parameter is stored in the object table entry as the granted access mask.

Some objects may require a more sophisticated access control scheme than simply checking the bits in the security descriptor. For example, a particular kind of access to an object may be granted by being given explicit permission via the security descriptor, or by having a privilege, or by having a particular kind of access to the object's container. In order to accomodate access schemes such as these, the caller may create an AccessState structure (via SeCreateAccessState). An AccessState structure contains the desired access mask, a record of the currently granted access mask, and room for a set of privileges. The caller performs whatever kind of access checking is necessary to suit it's needs, clearing bits in the imbedded *DesiredAccess* mask as appropriate. When all of the object specific logic is complete, the structure is then passed to the object manager for whatever security processing remains.

When referencing an object via an object handle, the caller also specifies a *DesiredAccess* parameter. However, in this case, the test for access is nothing more than a bit test against the granted access mask stored in the associated object table entry. Thus object handle creation encapsulates the security check for NT OS/2. Please refer to the Local Security chapter for a description of the bits defined for *DesiredAccess*, and for a description of the AccessState structure.

1.7 Object Attributes Structure

When a handle to an object is created, the object is specified with an Object Attributes structure. The structure identifies the object by name, specifies attributes about the object and/or handle being created and specifies an optional security descriptor to associate with the created object.

```
typedef struct _OBJECT_ATTRIBUTES {
        ULONG Length;
        HANDLE RootDirectory;
        PSTRING ObjectName;
        ULONG Attributes;
        PVOID SecurityDescriptor;
        PVOID SecurityQualityOfService;   \par} OBJECT_ATTRIBUTES,
*typedefOBJECT_ATTRIBUTES
```

OBJECT_ATTRIBUTES Structure:

Length —Specifies the length of this structure. Must be set to sizeof(OBJECT_ATTRIBUTES).

RootDirectory —An optional handle to a directory object that specifies where to start the name lookup. If this field is specified, then the *ObjectName* field must also be specified.

If this field is not specified and the *ObjectName* field is specified, then the name lookup begins in the root directory of the object name space.

ObjectName —A pointer to an object name string. The form of the name is:

[\name...\name]\object_name

The name must begin with a leading path separator character (\) if the *RootDirectory* field is NOT specified. If the *RootDirectory* field is specified, then it must NOT begin with a leading path separator as the name is relative to that directory.

Attributes —A set of flags that control attributes about the object and the handle.

Attributes Flags:

OBJ_INHERIT —The open handle is to be inherited by child process's whenever the calling process creates a new process.

OBJ_EXCLUSIVE —The object is to be accessed exclusively by the current process. Invalid if *OBJ_INHERIT* also specified.

OBJ_PERMANENT —The object is to be created as a permanent object.

OBJ_CASE_INSENSITIVE —Indicates that the name lookup should be performed in a manner which ignores the case of *ObjectName* rather than performing an exact match search.

OBJ_OPENIF —Return a handle to an already existing object if an object by the same name already exists. If the name does not exist, and the call is a create, then create the name.

SecurityDescriptor —An optional pointer to a security descriptor to associate with this object. See the Local Security Specification for a description of a Security Descriptor. If an object is created without a security descriptor, then access to the object will be uncontrolled.

SecurityQualityOfService —An optional pointer to the security quality of service parameters specified by the client for this communication session.

1.8 Resource Quotas and Objects

Objects are allocated from system memory, either paged or nonpaged pool. When an object is created the resource charges are specified and stored in the object's header. When a process creates a handle for an object, the resource charges stored in the object's header are levied against the process. This occurs whenever any handle is created to an object. So if process A creates an object and a handle to go with it, it gets charged quota for that object. If process A then creates process B, such that process B inherits a handle to the object, then process B is also charged quota for the same object. The same is true if process A creates a second handle to the same object.

The resource charge is removed whenever a handle is closed. The resource charge includes the space for the object header, the object body, the handle table entry, the object name, if any and the security descriptor, if any. If there is no security descriptor, then a fixed amount is charged (256 bytes) in case the process later attaches a security descriptor to the object with the **NtSetSecurityObject** system service.

1.9 Object Retention

Once an instance of an object has been created, two fields and the permanent flag contained within the object's header, control retention. The fields are named *HandleCount* and *PointerCount*.

The *HandleCount* represents the number of references to this object from various object tables. This count is incremented each time an object is inserted into an object table. It is decremented each time a handle is closed, either with **NtClose** or as a result of process termination. If this count becomes zero, a check is made to determine if an attempt should be made to delete the object's name. If the permanent flag in the object's header is false and the object has a name, then an attempt is made to delete the object's name by conditionally removing its directory entry. Conditional deletion means that the necessary mutexes are released, the directory mutex is acquired, the directory entry is located and the *HandleCount* is checked again. If the count is still zero, the object's name is deleted. This is done because the object was declared as temporary and the last handle to the object has been closed.

Once the conditional deletion of the object's name has occurred, the *PointerCount* for the object is decremented.

The *PointerCount* represents the number of pointers in existence which refer to the object. When an object is first created with the ObCreateObject function, this count is set to one to account for the reference returned to the caller. In addition, if the object has a name, the count is set to two to account for the pointer from the directory object which contains the name. This count is incremented for each object table that refers to the object.

The *PointerCount* is also updated as the object is referenced and dereferenced. When the *PointerCount* is decremented to zero, the object is deleted as there are no pointers outstanding. The *PointerCount* is never allowed to be decremented below the value of the *HandleCount*.

1.10 Exclusive Object Handles

Exclusive object handles provide a method of obtaining exclusive access to a system wide resource such as a tape drive. The semantics provided by exclusive handles cannot be provided by access protection because access protection determines who can access an object, while an exclusive handle essentially "reserves" an object.

Exclusive object handles are provided by specifying *OBJ_EXCLUSIVE* in the object attributes structure.

Exclusive object creation has the following rules:

- Any instance of an object whose type allows exclusion, may be opened or created for exclusion provided the *HandleCount* is zero.

- Any instance of an object which has a non-zero *HandleCount* and is not marked as exclusive cannot be opened for exclusion.

- Any instance of an object which has a non-zero *HandleCount* and is marked as exclusive can only be opened for exclusion from the owning process. This allows the owning process to open an exclusive object multiple times.

Finally, exclusive object handles may not be inherited by other processes. This means that an error will be returned if both *OBJ_EXCLUSIVE* and *OBJ_INHERIT* are specified in the object attributes structure.

1.11 Object Name Space

The Object Manager manages the global name space for NT OS/2. This name space is used to access all named objects that are contained in the local machine environment. Some of the objects that can have names are:

directory objects

object type objects

symbolic link objects

semaphore and event objects

process and thread objects

section and segment objects

port objects

device objects

file system objects

file objects

The object name space is modelled after OS/2 file naming convention, where directory names in a path are separated by a backslash (\). Case insensitivity is optional whenever a name lookup is performed. Case is always preserved when a name is inserted into a directory.

During system initialization, the Object Manager creates the root directory of the object name space. The **NtCreateDirectoryObject** system service can be used to create other directories within the object name space. The **ObInsertObject** function can be used to create object names within a directory object.

The entire object name space is guarded by a single mutex. This mutex is acquired whenever an portion of the directory structure is examined or modified.

A name lookup occurs whenever a new object is being inserted or an existing object is being opened by name. The name lookup is accomplished by searching in the root directory for the first name in the path. If no matching name is found, an error is returned.

The root directory defaults to the actual root directory of the global name space. However, then specifying an object name, a root directory handle may also be specified. This is the only form of relative name lookup supported by the Object Manager.

If a matching name is found and there are more tokens left in the name string, the corresponding object header is examined. If the object is not a directory object, its corresponding object type structure is examined for a parse routine. If no parse routine exists, an error status code is returned. Otherwise, the directory mutex is released, and the parse routine is called.

The parse routine can return one of three values: STATUS_SUCCESS to indicate that the object was found, STATUS_REPARSE to indicate that a reparse should occur or an error status code to indicate that the name was not found or invalid.

The parse procedure is passed pointers to both the complete name string and the remaining portion of the name string. If the parse routine returns reparse it should deallocate the original string and allocate the new string to parse, or modify the original string.

After the Object Manager's system initialization, the object name space looks like:

```
\                   - Root Directory
\ObjectTypes           - Object Type Name Directory
\ObjectTypes\Type       - Type Object Type
\ObjectTypes\Directory    - Directory Object Type
\ObjectTypes\SymbolicLink - Symbolic Link Object Type
```

Other components of system initialization will create additional type, directory and object names within the object name space.

1.12 Preventing Deadlock

To detect deadlock, the kernel associates a level number with each mutex. If an attempt is made to acquire a mutex with a level number less than a currently owned mutex a system bugcheck occurs. Associated with the Object Management routines are three levels of mutex.

o The lowest level is the object table mutex.

o The next higher level is the directory mutex.

o The highest level is the type specific mutex.

2. Object Executive APIs

2.1 Creating Object Types

New object types can be added to the system with the **ObCreateObjectType** function:

NTSTATUS
ObCreateObjectType(
    IN PSTRING *TypeName*,
    IN POBJECT_TYPE_INITIALIZER *ObjectTypeInitializer*,
    IN PULONG *DispatcherObjectOffset* **OPTIONAL**,
    IN PSECURITY_DESCRIPTOR *SecurityDescriptor* **OPTIONAL**,
    OUT POBJECT_TYPE ***ObjectType*
)

Parameters:

TypeName —A required pointer to a name string. This name must not contain the path separator character (OBJ_NAME_PATH_SEPARATOR), otherwise the STATUS_INVALID_OBJECT_NAME error status code is returned.

ObjectTypeInitializer —A required pointer to a structure that specifies type specific information about the new object type being created.

OBJECT_TYPE_INITIALIZER Structure:

ULONG *Length* —Specifies the size of this data structure in bytes.

ULONG *InvalidAttributes* —Specifies object attributes that are invalid for objects of this type. An attempt to specify any these attributes when creating an object of this type will result in the STATUS_INVALID_PARAMETER error status code being returned. This field may not specify any bits that are not contained in OBJ_VALID_ATTRIBUTES.

GENERIC_MAPPING *GenericMapping* —Specifies the mapping of the GENERIC_READ, GENERIC_WRITE and GENERIC_EXECUTE access rights for this object type.

ULONG *ValidAccessMask* **—**Specifies the valid access bits that may be specified with the *DesiredAccess* parameter when creating a handle to an object of this type. The mask is only used to remove unsupported access bits and does not cause an error if an unsupported access bit is specified. Thus specifying a *DesiredAccess* of -1 (all ones) will result in requesting a *DesiredAccess* equal to the *ValidAccessMask* for the type of object being created.

POOL_TYPE *PoolType* **—**Specifies the type of pool, one of NonPagedPool or PagedPool. This parameter must specify NonPagedPool if the *DispatcherObjectOffset* parameter is specified. The STATUS_INVALID_PARAMETER error status code is returned if the later condition is not met.

BOOLEAN *MaintainHandleCount* **—**Specifies whether a handle count data base should be maintained. If TRUE, then for each object of this type, a data base is kept that keeps track of how many handles to that object each process currently has. This allows the Open/Close object type procedures to implement special logic when the first handle to an object is created and when the last handle to an object within a process is closed. If this field is TRUE then at least one of the OpenProcedure or CloseProcedure fields must be non-NULL, otherwise the STATUS_INVALID_PARAMETER error status code is returned.

OB_DUMP_METHOD *DumpProcedure* **—**An optional pointer to the procedure to invoke on object dumping. This procedure is useful for the debugging version of NT OS/2 to allow a uniform way to dump the contents of an object in human readable form.

If this field is NULL, no routine is called when an object is dumped.

OB_OPEN_METHOD *OpenProcedure* **—**An optional pointer to the procedure to invoke whenever a handle to an object of this type is created.

If this field is NULL, no routine is called when a handle to an object of this type is created.

OB_CLOSE_METHOD *CloseProcedure* —An optional pointer to the procedure to invoke whenever a handle to an object of this type is destroyed.

If this field is NULL, no routine is called when a handle to an object of this type is destroyed.

OB_DELETE_METHOD *DeleteProcedure* —An optional pointer to the procedure to invoke on object deletion. This procedure is responsible for deallocating any pool which was allocated by object type specific routines and performing any "cleanup" operations. When the *DeleteProcedure* returns, the object management routines deallocate the object structure, unlinks the object from its object type structure, etc.

If this field is NULL, no routine is called before deallocating the object structure.

OB_PARSE_METHOD *ParseProcedure* —An optional pointer to the parse routine for this object type. If, during name parsing, an object of this type is encountered and additional parse tokens exist, this routine is invoked.

OB_SECURITY_METHOD *SecurityProcedure* —An optional pointer to the procedure to invoke whenever the *SecurityDescriptor* associated with an object is set or queried via the **NtSetSecurityObject** and **NtQuerySecurityObject** system services. Note that another procedure (SeAssignSecurity) and not this procedure is used to insert an original security descriptor on an object.

If this field is NULL, then the SeDefaultObjectMethod will be called instead.

SecurityDescriptor —An optional pointer to a Security Descriptor. This descriptor will be attached to the type object. Any attempt to create an object of this type will require the OBJECT_TYPE_CREATE access right.

DispatcherObjectOffset —An optional pointer to the offset into the object body of a kernel dispatcher object for wait operations. If this value is not specified then an object of this type cannot be used as an argument to the **NtWaitForSingleObject** and **NtWaitForMultipleObjects** system services.

ObjectType —A pointer to a variable which receives the location of the object type structure created.

Return Value:

Status code that indicates whether or not the operation was successful.

The create object type function creates an object type structure. This function returns a pointer to the object type structure via the *ObjectType* parameter.

The *TypeName* is inserted into the \ObjectTypes object directory. If the name already exists, then this function will return an error.

This function returns one of the following status codes:

- STATUS_SUCCESS —normal, successful completion.
- STATUS_INVALID_PARAMETER —one of the parameters was invalid.
- STATUS_OBJECT_NAME_INVALID —the type name string contained a path separator character (OBJ_NAME_PATH_SEPARATOR).
- STATUS_NO_MEMORY —unable to allocate NonPagedPool for the object type structure.

2.2 Object Type Procedure Templates

This section describes the six different procedure types that can be associated with an object type. These procedures are called whenever certain actions are performed upon an object whose object type structure contains the addresses of these procedures.

2.2.1 Object Dump Procedure

VOID
typedef
(***OB_DUMP_METHOD**)(
    IN PVOID *Object*,
    IN POB_DUMP_CONTROL *DumpControl* **OPTIONAL**
)

Parameters:

Object —A pointer to the object's body.

DumpControl —An optional pointer to a dump control structure. This structure specifies the output stream and the detail level. If not specified then output should be sent to the standard output stream. Default detail level is 1.

OB_DUMP_CONTROL Structure:

PVOID *Stream* —an opaque pointer to an output stream.

ULONG *DetailLevel* —level of detail to show, along with some modifiers. See ObDumpObject description for values.

This function is called whenever one of the ObDumpObject functions is called for an object of this type. This procedure is free to write to the output stream an ASCII representation of its contents. The content is governed by the *DetailLevel* parameter.

2.2.2 Object Open Procedure

VOID
typedef
(***OB_OPEN_METHOD**)(
    IN OB_OPEN_REASON *OpenReason*,
    IN PEPROCESS *Process*,
    IN PVOID *Object*,
    IN ACCESS_MASK *GrantedAccess*,
    IN ULONG *HandleCount* **OPTIONAL**
)

Parameters:

OpenReason —Indicates one of four specific reasons for the handle being created. These are:

OpenReason Values:

ObCreateHandle —a handle to a new object is being created via the **ObInsertObject** interface.

ObOpenHandle —a handle to an existing object is being created via the **ObInsertObject**, **ObOpenObjectByName** or the **ObOpenObjectByPointer** interface.

ObDuplicateHandle —a handle to an existing object is being created via the **NtDuplicateObject** system service.

ObInheritHandle —a handle to an existing object is being created a a result of object inheritence during process creation.

Process —Specifies a pointer to the process for which the handle has been created.

Object —Specifies a pointer to the object for which the handle has been created.

GrantedAccess —Specifies the granted access mask associated with the newly created handle.

HandleCount —Optional parameter, that is non-zero if the *MaintainHandleCount* in the associated object type structure is TRUE. If non-zero then represents the number of handles to the specified *Object* that have been created in the object table associated with the specified *Process*. Interesting value is 1, which means this is the first handle to the specified *Object* for the specified *Process*.

This function is called whenever a handle to an object is created. The *OpenReason* parameter specifies the reason the handle is being created.

This function is called after the handle has actually been inserted in the object table for the specified process, but before the object type mutex has been released. This means that the function must not attempt to manipulate any object handles itself, as it my result in an attempt to recusively acquire the object type mutex.

2.2.3 Object Close Procedure

VOID
typedef
(***OB_CLOSE_METHOD**)(
    IN PEPROCESS *Process* **OPTIONAL**,
    IN PVOID *Object*,
    IN ACCESS_MASK *GrantedAccess*,
    IN ULONG *HandleCount*
)

Parameters:

Process —Specifies a pointer to the process for which the handle has been destroyed.

Object —Specifies a pointer to the object for which the handle is been destroyed.

GrantedAccess —Specifies the granted access mask that was associated with the destroyed handle.

HandleCount —Optional parameter, that is non-zero if the *MaintainHandleCount* in the associated object type structure is TRUE. If non-zero then represents the number of handles to the specified *Object* that have been created in the object table associated with the specified *Process*, including the handle that has just been destroyed. Interesting value is 1, which means this is the last handle to the specified *Object* for the specified *Process*.

This function is called whenever a handle to an object is destroyed.

This function is called after the handle has actually been deleted from the object table for the specified process, but before the object type mutex has been released. This means that the function must not attempt to manipulate any object handles itself, as it my result in an attempt to recusively acquire the object type mutex. Also, the object name, if any, is still valid when this function is called.

2.2.4 Object Delete Procedure

VOID
typedef
(***OB_DELETE_METHOD**)(
        IN PVOID *Object*
)

Parameters:

Object —A pointer to the object's body.

This function is called whenever the *PointerCount* associated with the object is decremented to zero, and the object is a temporary object. See the section on *Object* Retention for a description of how the *PointerCount* can become zero.

2.2.5 Object Parse Procedure

NTSTATUS
typedef
(*OB_PARSE_METHOD)(
    IN PVOID *ParseObject*,
    IN POBJECT_TYPE *ObjectType,*
    IN OUT PACCESS_STATE *AccessState*,
    IN KPROCESSOR_MODE *AccessMode*,
    IN ULONG *Attributes*,
    IN OUT PSTRING *CompleteName*,
    IN OUT PSTRING *RemainingName*,
    IN OUT PVOID *Context* **OPTIONAL**,
    IN PSECURITY_QUALITY_OF_SERVICE *SecurityQos* **OPTIONAL**,
    OUT PVOID ***Object**
)

Parameters:

ParseObject —a pointer to the object, whose type contains this procedure as its *ParseProcedure*.

ObjectType —A pointer that supplies the type of object being referenced.

AccessState —A pointer to a structure that contains a record of desired types of access, already granted access types, and a list of privileges that may have been used to obtain some of the granted access types. If privileges are passed, a control flag in the argument indicates whether any of the privileges or all of the privileges are needed to open the object.

AccessMode —Indicates the access mode to use for the access check. One of UserMode or KernelMode.

Attributes —A set of flags that control the object attributes.

OBJ_CASE_INSENSITIVE —Indicates that the name lookup should be performed in a manner which ignores the case of the *ObjectName* rather than performing an exact match search.

CompleteName —A pointer to the complete path name being parsed.

RemainingName —A pointer to the portion of the complete path name that remains to be parsed.

Context —An optional pointer that is passed uninterpreted to the *ParseProcedure*. It is the same *Context* parameter that was passed to the routine that triggered the name lookup.

SecurityQos —An optional pointer to the security quality of service parameters specified by the client for this communication session.

Object —A pointer to a variable which receives the address of the object that the remaining name parsed to.

Return Value:

Status code that indicates whether or not the operation was successful.

CompleteName and *RemainingName* both point to the same string, with *RemainingName* describing a suffix of the *CompleteName*. Storage for the name string is from paged or nonpaged pool. This allows parse routines to allocate storage for a new name, copy any information necessary into the newly allocated storage, and deallocate the storage containing the previous name string. The Buffer fields in the *CompleteName* and *RemainingName* structures would then be updated to point to the newly allocated string and the *Length* fields would be updated as appropriate.

This function is called whenever an object is looked up by name. See the *Object* Name Space section for a description about how name lookup is performed.

This function may return one of the following status codes:

- o STATUS_SUCCESS —normal, successful completion.

- o STATUS_REPARSE —a success status code that tells the object manager to start the parse over at the beginning of the *CompleteName* string. The assumption being that the function modified the *CompleteName* string to point to a new name, such as the target of a symbolic link.

- o STATUS_OBJECT_PATH_SYNTAX_BAD —if the parse failed because of an ill formed path name.
- o STATUS_OBJECT_PATH_NOT_FOUND —if the parse was terminated because a path component was not found and there were characters remaining to parse.
- o STATUS_OBJECT_NAME_NOT_FOUND —if the parse was terminated because a path component was not found and there were no more characters remaining to parse.
- o STATUS_OBJECT_PATH_INVALID —if the parse succeeded and matched an object, but there were more characters remaining to be parsed.
- o STATUS_ACCESS_DENIED —if any of the access tests involved in creating the object failed.

2.2.6 Object Security Procedure

```
NTSTATUS
typedef
(*OB_SECURITY_METHOD)(
    IN PVOID Object,
    IN SECURITY_OPERATION_CODE OperationCode,
    IN PSECURITY_INFORMATION SecurityInformation,
    IN OUT PSECURITY_DESCRIPTOR SecurityDescriptor,
    IN OUT PULONG CapturedLength,
    IN OUT PSECURITY_DESCRIPTOR *ObjectsSecurityDescriptor,
    IN POOL_TYPE PoolType,
    IN PGENERIC_MAPPING GenericMapping
    )
```

Parameters:

Object —A pointer to an object

OperationCode —Indicates one of three specific operations that the method can perform.

OperationCode Values:

SetSecurityDescriptor —used to alter the security descriptor protecting an operation. The security method will take the input security descriptor and apply the portions of it specified by the *SecurityInformation* argument to the object.

QuerySecurityDescriptor —used to return to the caller a copy of the portions of object's security descriptor requested by the *SecurityInformation* argument. The information will be returned in the form of a security descriptor in the *SecurityDescriptor* buffer.

DeleteSecurityDescriptor —used when an instance of an object is being deleted. The method will cleanup (and delete as necessary) any storage associated with the object's security descriptor.

AssignSecurityDescriptor —used when an instance of an object is being created and security is being assigned to the object for the first time. The method will take the contents of the *SecurityDescriptor* field and assign it to the object.

SecurityInformation —Specifies which security information is being set or queried.

SecurityDescriptor —Points to buffer to either set or read the security descriptor from. This buffer will be probed and captured as necessary by this procedure. This parameter is ignored for the delete operation.

This parameter is ignored for the delete operation.

CapturedLength —For a query operation this specifies the size, in bytes, of the output security descriptor buffer and on return contains the number of bytes needed to store the complete security descriptor. If the length needed is greater than the length supplied the operation will fail. This parameter is ignored for the set and delete operations. It is expected to be point into kernel space, ie, it need not be probed and it will not change.

ObjectsSecurityDescriptor —This supplies the address of a variable pointing to the current object's security descriptor. This parameter will be used if the object's security descriptor is stored as part of the object header (this occurs as the default method). If this parameter is used then the procedure will deallocate and reallocate pool as necessary to hold the object's security descriptor. Alternate methods (e.g., the file system) will not use this parameter and instead will have the underlying file system store the descriptor (this means that system wide file object handles are not allowed).

This parameter is ignored for the assign operation.

PoolType —Specifies the type of pool to allocate for the object's security descriptor if needed. This parameter is ignored for the query and delete operations.

Return Value:

Status code that indicates whether or not the operation was successful.

Before calling this procedure the object manager will have determined that the requested action is allowed according to the granted access rights and privileges of the caller.

2.3 Creating An Object

The data structures for an object are created with the ObCreateObject function:

```
NTSTATUS
ObCreateObject(
    IN KPROCESSOR_MODE ProbeMode,
    IN POBJECT_TYPE ObjectType,
    IN POBJECT_ATTRIBUTES ObjectAttributes OPTIONAL,
    IN KPROCESSOR_MODE OwnershipMode,
    IN OUT PVOID ParseContext OPTIONAL,
    IN ULONG ObjectBodySize,
    IN ULONG PagedPoolCharge,
    IN ULONG NonPagedPoolCharge,
    OUT PVOID *Object
    )
```

Parameters:

ProbeMode —Specifies one of UserMode or KernelMode. This is the mode used when probing the *ObjectAttributes* structure.

ObjectType —An pointer to the object type structure describing the type of object to create.

ObjectAttributes —An optional pointer to an *Object Attributes* structure. Refer to the *Object Attributes* discussion for details.

OwnershipMode —Specifies one of UserMode or KernelMode. For existing objects, this parameter is ignored.

The *OwnershipMode* controls the interpretation of the *SecurityDescriptor*. If the *OwnershipMode* is KernelMode and the object does not have a *SecurityDescriptor* then no access to the object with an *AccessMode* of UserMode is allowed. If the *OwnershipMode* is KernelMode and the *AccessMode* is KernelMode then the *SecurityDescriptor* is examined to determine access.

If the *OwnershipMode* is UserMode and the *AccessMode* is KernelMode then the access is always allowed. If the *OwnershipMode* is UserMode and the *AccessMode* is UserMode then the *SecurityDescriptor* is examined to determine access.

ParseContext —An optional pointer that is passed uninterpreted to any *ParseProcedure* that is called during the course of performing the name lookup.

ObjectBodySize —Size of the object body in bytes.

PagedPoolCharge —The number of bytes of paged pool to charge to the current process.

NonPagedPoolCharge —The number of bytes of nonpaged pool to charge to the current process.

Object —A pointer to a variable which receives the address of the newly created object.

Return Value:

Status code that indicates whether or not the operation was successful.

Creating an object causes a block of storage from pool to be allocated. The size of the block is the sum of the object header size and the object body size. The object header is initialized and the *PointerCount* is set to 1 and the *HandleCount* is set to zero.

The address of the uninitialized object body is returned via the *Object* parameter. It is the responsibility of the object type specific creation routine to initialize the object body.

The *ObjectAttributes* parameter is considered unprobed and thus is probed by this function, using the mode specified in the *ProbeMode* parameter.

The *Attributes* field of the *ObjectAttributes* parameter is validated and stored in the object header.

The *RootDirectory* field of the *ObjectAttributes* parameter is captured into the object header at this time. The handle is not referenced at this time. It will be referenced when **ObInsertObject** is called to insert the object into an object table.

If specified, any string structure specified by the *ObjectName* field of the *ObjectAttributes* parameter is captured into the object header at this time. The actual buffer pointer to by the string structure is not probed at this time. Instead it is probed when **ObInsertObject** is called to insert the object into an object table.

The *SecurityDescriptor* field of the *ObjectAttributes* parameter is captured into the object header at this time. The pointer is not probed until **ObInsertObject** is called to insert the object into an object table. If for some reason the attempt to insert the object fails, **ObInsertObject** will clear the field in the object header before attempting to dereference the object.

The *SecurityQualityOfService* field of the *ObjectAttributes* parameter is captured into the object header at this time. The purpose of capturing it into the object header is to facilitate passing the QOS information to **ObInsertObject**. Rather than put a pointer to the QOS information into the *Object* header, the *SecurityQos* field is temporarily used to hold the pointer to the QOS structure. Note that in the case of an error, this field must be zero'd out before the object is freed, to prevent the pointer from being interpreted as a quantity of pool memory to be freed.

The *ParseContext* parameter is also captured into the object header for later use when **ObInsertObject** is called.

Memory for the object header and object body is allocated from the pool type specified in the object type descriptor. The amount of quota to charge is calculated. Quota includes the memory for the object header and body, plus any additional quota specified by the *PagedPoolCharge* and *NonPagedPoolCharge* parameters. The total quota to charge is remembered in the object header. This will allow the quota to be charged each time a handle is created for this object, using the either **ObOpenObjectByName** function or the **ObInsertObject** function.

This function may return one of the following status codes:

- STATUS_SUCCESS —normal, successful completion.

- STATUS_INVALID_PARAMETER —one of the parameters was invalid.

- STATUS_OBJECT_NAME_INVALID —an object name was specified in the *ObjectAttributes* structure, but it has a zero length.

- STATUS_NO_MEMORY —no memory to allocate the object.

- STATUS_ACCESS_VIOLATION —Either the *ObjectAttributes* pointer or the *ObjectAttributes*->ObjectName pointer or the *ObjectAttributes*->ObjectName->Buffer pointer were invalid.

- STATUS_DATATYPE_MISALIGNMENT —Either the *ObjectAttributes* pointer or the *ObjectAttributes*->ObjectName pointer were not aligned on a 4 byte boundary.

2.4 Creating an Instance of an Object

An instance of an object is created by inserting the new created object into the calling process's object table and obtaining an object handle. This is accomplished with the **ObInsertObject** function:

NTSTATUS
ObInsertObject(
    IN PVOID *Object*,
    IN PACCESS_STATE *PassedAccessState* **OPTIONAL**,
    IN ACCESS_MASK *DesiredAccess* **OPTIONAL**,
    IN ULONG *ObjectPointerBias*,
    OUT PVOID **NewObject* **OPTIONAL**,
    OUT PHANDLE *Handle*
)

Parameters:

Object —A pointer to the object's body. The object must be one that was returned by **ObCreateObject**.

PassedAccessState —An optional pointer to a structure that contains a record of desired types of access, already granted access types, and a list of privileges that may have been used to obtain some of the granted access types. If privileges are passed, a control flag in the argument indicates whether any of the privileges or all of the privileges are needed to open the object.

DesiredAccess —An optional parameter describing the desired types of access to the object. The interpretation of this field is object type dependent. Simple access requests (ie, those that intend to compare the desired access to the Dacl on the object) need only pass a *DesiredAccess* mask, rather than constructing an *AccessState* structure.

ObjectPointerBias —Value to increment the *PointerCount* by. This occurs whether or not the object is successfully inserted into the object table.

NewObject —An optional pointer to a variable that will will receive the pointer to the referenced object's body. A pointer to the referenced object's body is returned only if the *ObjectPointerBias* field is not zero and the argument is present. If the argument is supplied and the *ObjectPointerBias* is zero, then NULL is returned in the pointer.

Handle —A pointer to a variable that will receive the object handle value.

Return Value:

Status code that indicates whether or not the operation was successful.

Inserting the object into a table causes an object handle to be allocated from the appropriate table thereby making the object visible. If the object was given a name, the name is visible to all threads that have "read" or "execute" access to the directory path that contains the name.

The *ObjectName* field of the *ObjectAttributes* parameter to **ObCreateObject** is extracted from the object header and probed for accessiblility. Storage is then allocated for a copy of the string, so that any parse procedures called can reallocate the string for reparse operations. The *Attributes* and *ParseContext* fields that were captured into the object header are used along with the captured *ObjectName* as additional parameters to the name lookup procedure.

During the creation of a new object's instance, checks are performed to ensure that the name of the object, if any, is unique within the specified directory. If the name is not unique, the newly created object is deleted and the *OBJ_OPENIF* option is used to determine the appropriate action.

If *OBJ_OPENIF* was specified, the object instance with the collided name is examined to see if the desired access can be granted. If so, a handle is created to the collided object. If *OBJ_OPENIF* was not specified, an error status is returned to the caller.

In the process of creating or opening a named object, several different security operations may be performed. For each subdirectory in the object's path, the current subject must have TRAVERSE access to that subdirectory in order for the name search to continue. The interface to perform this test is **ObCheckTraverseAccess**. **ObCheckTraverseAccess** will be called by the object manager as appropriate if the object does not have an object-specific parse routine. For those objects that do specify parse routines, it is the responsiblity of the parse routine to check traverse access to each subdirectory. **ObCheckTraverseAccess** may generate audit messages.

If the object is being created, it is necessary to check to make sure that the subject has the ability to create an object in the specified directory. Note that this is a different access type than the ability to traverse the parent directory. The interface that performs this test is **ObCheckCreateObjectAccess**. Like **ObCheckTraverseAccess**, this routine will be called by the object manager unless there exists an object-specific parse routine, in which case it is the responsibility of the parse routine to make the call.

Finally, a new handle to the object is created, and the count of outstanding handles to the object is incremented in the object header. Depending on whether the object is being created or simply opened, the parse routine must call either **ObCheckCreateInstanceAccess** or ObpCheckObjectAccess respectively.

The **ObInsertObject** function automatically dereferences the specified object, even if the operation fails for any reason. This means that the *Object* value is no longer usable when this function returns. This is due to the fact that at the completion of the **ObInsertObject** function, the object handle could now be deleted by another thread of execution causing the storage for the object to be deallocated or the name could have collided, causing the original object to be deleted.

The *ObjectPointerBias* parameter provides a mechanism for ensuring a pointer to the object can be utilized. When the *ObjectPointerBias* is not zero, the value is added to the *PointerCount* in the object header referenced by the handle. This prevents the object from being deleted. The *NewObject* parameter receives the pointer to the object body referred to by the object. This may be a different object then the one which was inserted due to name collisions.

This is typically the last operation that is performed when an instance of an object is created, and the handle and status value are returned to the caller.

This function may return one of the following status codes:

- o STATUS_SUCCESS —normal, successful completion.
- o STATUS_OBJECT_NAME_EXISTS —the object name already existed and *OBJ_OPENIF* was specified. This is a warning status code.
- o STATUS_OBJECT_TYPE_MISMATCH —the object name already existed, but was a different type than specified by the *ObjectType* parameter.
- o STATUS_OBJECT_NAME_COLLISION —the object name already existed and *OBJ_OPENIF* was not specified.
- o STATUS_OBJECT_PATH_SYNTAX_BAD —if the parse failed because of an ill formed path name.

- o STATUS_OBJECT_PATH_NOT_FOUND —if the parse was terminated because a path component was not found and there were characters remaining to parse.
- o STATUS_OBJECT_NAME_NOT_FOUND —if the parse was terminated because a path component was not found and there were no more characters remaining to parse.
- o STATUS_OBJECT_PATH_INVALID —if the parse succeeded and matched an object, but there were more characters remaining to be parsed.
- o STATUS_ACCESS_DENIED
- o STATUS_QUOTA_EXCEEDED
- o STATUS_NO_MEMORY

2.5 Open Object by Name

An object can be opened by name with the **ObOpenObjectByName** function:

NTSTATUS
ObOpenObjectByName(
  IN POBJECT_ATTRIBUTES *ObjectAttributes*,
  IN POBJECT_TYPE *ObjectType* **OPTIONAL**,
  IN KPROCESSOR_MODE *AccessMode*,
  IN OUT PACCESS_STATE *PassedAccessState* **OPTIONAL**,
  IN ACCESS_MASK *DesiredAccess* **OPTIONAL**,
  IN OUT PVOID *ParseContext* **OPTIONAL**,
  OUT PHANDLE *Handle*
)

Parameters:

ObjectAttributes —A pointer to a structure that specifies the object's attributes. Refer to the *Object Attributes* discussion for details.

ObjectType —A optional pointer to the object type structure for the object's type.

AccessMode —Indicates the access mode to use for the access check. One of UserMode or KernelMode.

ParseContext —An optional pointer that is passed uninterpreted to any *ParseProcedure* that is called during the course of performing the name lookup.

PassedAccessState —An optional pointer to a structure that contains a record of desired types of access, already granted access types, and a list of privileges that may have been used to obtain some of the granted access types. If privileges are passed, a control flag in the argument indicates whether any of the privileges or all of the privileges are needed to open the object.

DesiredAccess —The desired types of access to the object. The interpretation of this field is object type dependent. Simple access requests (ie, those that intend to compare the desired access to the Dacl on the object) need only pass a *DesiredAccess* mask, rather than constructing an *AccessState* structure.

Handle —A pointer to a variable that will receive the object handle.

Return Value:

Status code that indicates whether or not the operation was successful.

Opening an object by name causes a name search to be performed. If this function completes successfully, a pointer to the named object's body is inserted into the specified object table.

Successful opening of an object by name causes the *HandleCount* and *PointerCount* for the specified object to be incremented.

This function may return one of the following status codes:

- STATUS_SUCCESS —normal, successful completion.
- STATUS_OBJECT_TYPE_MISMATCH —the object name was found, but was a different type than specified by the *ObjectType* parameter.
- STATUS_OBJECT_PATH_SYNTAX_BAD —if the parse failed because of an ill formed path name.

- STATUS_OBJECT_PATH_NOT_FOUND —if the parse was terminated because a path component was not found and there were characters remaining to parse.
- STATUS_OBJECT_NAME_NOT_FOUND —if the parse was terminated because a path component was not found and there were no more characters remaining to parse.
- STATUS_OBJECT_PATH_INVALID —if the parse succeeded and matched an object, but there were more characters remaining to be parsed.
- STATUS_ACCESS_DENIED
- STATUS_QUOTA_EXCEEDED
- STATUS_NO_MEMORY

2.6 Open Object by Pointer

A handle to an object can be opened by pointer with the **ObOpenObjectByPointer** function:

NTSTATUS
ObOpenObjectByPointer(
    IN PVOID *Object*,
    IN ULONG *HandleAttributes*,
    IN PACCESS_STATE *PassedAccessState* **OPTIONAL**,
    IN ACCESS_MASK *DesiredAccess* **OPTIONAL**,
    IN POBJECT_TYPE *ObjectType* **OPTIONAL**,
    IN KPROCESSOR_MODE *AccessMode*,
    OUT PHANDLE *Handle*
)

Parameters:

Object —A pointer to the object that is being opened.

HandleAttributes —The attributes to associated with the handle. Same as the *Attributes* field in the *ObjectAttributes* structure. Refer to the *Object Attributes* discussion for details.

PassedAccessState —An optional pointer to a structure that contains a record of desired types of access, already granted access types, and a list of privileges that may have been used to obtain some of the granted access types. If privileges are passed, a control flag in the argument indicates whether any of the privileges or all of the privileges are needed to open the object.

DesiredAccess —The desired types of access to the object. The interpretation of this field is object type dependent. Simple access requests (ie, those that intend to compare the desired access to the Dacl on the object) need only pass a *DesiredAccess* mask, rather than constructing an *AccessState* structure.

ObjectType —A optional pointer to the object type structure for the object's type.

AccessMode —Indicates the access mode to use for the access check. One of UserMode or KernelMode.

Handle —A pointer to a variable that will receive the object handle.

Return Value:

Status code that indicates whether or not the operation was successful.

Opening an object by pointer the *HandleCount* and *PointerCount* for the specified object to be incremented and a handle to the object created.

This function may return one of the following status codes:

- STATUS_SUCCESS —normal, successful completion.
- STATUS_OBJECT_TYPE_MISMATCH
- STATUS_ACCESS_DENIED
- STATUS_QUOTA_EXCEEDED
- STATUS_NO_MEMORY

2.7 Referencing An Object

A user mode routine refers to an instance of an object through an object handle. In order for the executive to operate upon the object, access validation must be performed on the object handle, and the object handle must be converted to a pointer to the desired object's body. This is accomplished with the **ObReferenceObjectByHandle** function:

NTSTATUS
ObReferenceObjectByHandle(
  IN HANDLE *Handle*,
  IN ACCESS_MASK *DesiredAccess*,
  IN POBJECT_TYPE *ObjectType* **OPTIONAL**,
  IN KPROCESSOR_MODE *AccessMode*,
  OUT PVOID *****Object*,
  OUT POBJECT_HANDLE_INFORMATION *HandleInformation* **OPTIONAL**
)

Parameters:

Handle —An open handle to an object.

DesiredAccess —The desired types of access to the object. The interpretation of this field is object type dependent.

ObjectType —An optional pointer to the object type structure for the object's type. If this value is omitted, no type check is performed.

AccessMode —Indicates the access mode to use for the access check. One of UserMode or KernelMode.

Object —A pointer to a variable that will receive a pointer to the object's body.

HandleInformation —An optional pointer to XXXXXXXXXX

Return Value:

Status code that indicates whether or not the operation was successful.

This function uses the specified object handle as an index into the process object table. The index is validated against the object table bounds and converted into a pointer to a specific entry in the object table.

If the *AccessMode* is KernelMode, the desired access is always allowed.

If the *AccessMode* is UserMode, the desired access is compared to the granted access field stored within the table. If all of the bits in the *DesiredAccess* mask are set in the granted access mask, then access is granted. Otherwise the STATUS_ACCESS_DENIED error status code is returned.

If the desired access is allowed, a pointer to the object header is obtained from the table. If the specified *ObjectType* is supplied, it is compared to the object type field within the object header, and if they are equal a pointer to the object body is returned to the caller as the function value, and the *PointerCount* field in the object header is incremented.

Incrementing the *PointerCount* field prevents the object from being deleted while it is being operated upon.

A pointer to the object body is retreived from the object table entry and returned to the caller via the *Object* parameter.

This function may return one of the following status codes:

- o STATUS_SUCCESS —normal, successful completion.
- o STATUS_OBJECT_TYPE_MISMATCH
- o STATUS_ACCESS_DENIED
- o STATUS_INVALID_HANDLE

2.8 Reference Object by Name

An object can be referenced by name with the **ObReferenceObjectByName** function:

NTSTATUS
ObReferenceObjectByName(
    IN PSTRING *ObjectName,*
    IN ULONG *Attributes,*
    IN PACCESS_STATE *PassedAccessState* **OPTIONAL**,
    IN ACCESS_MASK *DesiredAccess* **OPTIONAL**,
    IN POBJECT_TYPE *ObjectType,*
    IN KPROCESSOR_MODE *AccessMode,*
    IN OUT PVOID *ParseContext* **OPTIONAL**,
    OUT PVOID **Object*
)

Parameters:

ObjectName —A pointer to a string which specifies the name of the object to open.

Attributes —A set of flags that control the object attributes.

> *OBJ_CASE_INSENSITIVE* —Indicates that the name lookup should be performed in a manner which ignores the case of the *ObjectName* rather than performing an exact match search.

PassedAccessState —An optional pointer to a structure that contains a record of desired types of access, already granted access types, and a list of privileges that may have been used to obtain some of the granted access types. If privileges are passed, a control flag in the argument indicates whether any of the privileges or all of the privileges are needed to open the object.

DesiredAccess —The desired types of access to the object. The interpretation of this field is object type dependent. Simple access requests (ie, those that intend to compare the desired access to the Dacl on the object) need only pass a *DesiredAccess* mask, rather than constructing an *AccessState* structure.

ObjectType —A pointer to the object type structure for the object's type.

AccessMode —Indicates the access mode to use for the access check. One of UserMode or KernelMode.

ParseContext —An optional pointer that is passed uninterpreted to any *ParseProcedure* that is called during the course of performing the name lookup.

Object —A pointer to a variable that will receive a pointer to the object's body.

Return Value:

Status code that indicates whether or not the operation was successful.

Referencing an object by name causes a name search to be performed. If this function completes successfully, a pointer to the named object's body is returned as the function value. The name search is accomplished by acquiring the directory mutex, and searching in the root directory for the first name in the path. If no matching name is found, an error status code is returned.

If a matching name is found and there are more tokens left in the name string, the corresponding object header is examined. If the object is not a directory object, its corresponding object type structure is examined for a parse routine. If no parse routine exists, an error status code is returned. Otherwise, the directory mutex is released, and the parse routine is called.

The parse routine is responsible for either returning a pointer to an object, which can be referenced as a result of the parse, or returning a unique value, OBJ_REPARSE to indicate that the name lookup should start over from the beginning of the string.

If the value returned is OBJ_REPARSE, the directory mutex is acquired and name parsing beings using the complete string as the name. This requires the parse routine to deallocate the previous string and allocate the new string to parse, or modify the original string.

Successful referencing of an object by name causes the *PointerCount* for the specified object to be incremented.

This function may return one of the following status codes:

- STATUS_SUCCESS —normal, successful completion.
- STATUS_OBJECT_TYPE_MISMATCH —the object name was found, but was a different type than specified by the *ObjectType* parameter.

- o STATUS_OBJECT_PATH_SYNTAX_BAD —if the parse failed because of an ill formed path name.
- o STATUS_OBJECT_PATH_NOT_FOUND —if the parse was terminated because a path component was not found and there were characters remaining to parse.
- o STATUS_OBJECT_NAME_NOT_FOUND —if the parse was terminated because a path component was not found and there were no more characters remaining to parse.
- o STATUS_OBJECT_PATH_INVALID —if the parse succeeded and matched an object, but there were more characters remaining to be parsed.
- o STATUS_ACCESS_DENIED
- o STATUS_NO_MEMORY

2.9 Reference Object by Pointer

NTSTATUS
ObReferenceObjectByPointer(
    IN PVOID *Object*,
    IN ACCESS_MASK *DesiredAccess*,
    IN POBJECT_TYPE *ObjectType*,
    IN KPROCESSOR_MODE *AccessMode*
)

Parameters:

Object —A pointer to the object's body.

DesiredAccess —A mask representing the desired access to the object.

ObjectType —A pointer to the object type structure for the object.

AccessMode —Indicates the access mode to use for the access check. One of UserMode or KernelMode.

<u>Return Value:</u>

Status code that indicates whether or not the operation was successful.

This function may return one of the following status codes:

- o STATUS_SUCCESS —normal, successful completion.
- o STATUS_OBJECT_TYPE_MISMATCH

2.10 Making an Object Temporary

An object can be made temporary with the ObMakeTemporaryObject function:

VOID
ObMakeTemporaryObject(
    IN PVOID *Object*
)

<u>Parameters:</u>

Object —A pointer to an object.

This is a generic function and operates on any type of object.

Making an object temporary causes the permanent flag of the associated object to be cleared. A temporary object has a name as long as its *HandleCount* is greater than zero. When the *HandleCount* becomes zero, the name is deleted and the *PointerCount* adjusted appropriately.

2.11 Dereferencing an Object

A referenced object is dereferenced with the ObDereferenceObject function:

VOID
ObDereferenceObject(
    IN PVOID *Object*
)

Parameters:

Object **—**A pointer to the object's body.

When an object is dereferenced, its *PointerCount* is decremented and retention checks are performed.

2.12 Object Management during Process Creation and Deletion

The *Process* Structure component uses these function during process creation and deletion to initialize and cleanup the object table associated with a process.

2.12.1 Process Creation Hook

The *Process* Structure component calls the *Object* Management component at process creation time via the **ObInitProcess** function.

NTSTATUS
ObInitProcess(
    PEPROCESS *ParentProcess* **OPTIONAL**,
    PEPROCESS *NewProcess*
)

Parameters:

ParentProcess **—**An optional pointer to the process to inherit any handles from.

NewProcess **—**A pointer to the process that is being created.

Return Value:

Status code that indicates whether or not the operation was successful.

This functions creates an object table for the *NewProcess*. It then scans the object table associated with the *ParentProcess*, if any, and creates copies of all handles that were created with the *OBJ_INHERIT* attribute.

This function may return one of the following status codes:

- STATUS_SUCCESS **—**normal, successful completion.
- STATUS_QUOTA_EXCEEDED
- STATUS_NO_MEMORY

2.12.2 Process Deletion Hook

The *Process* Structure component calls the *Object* Management component at process deletion time via the **ObKillProcess** function.

VOID
ObKillProcess(
    PEPROCESS *Process*
)

Parameters:

Process **—**A pointer to the process that is being destroyed.

This function scans the object table associated with the process being destroyed and calls **NtClose** for each valid handle.

2.13 Dump Object Support

Objects are displayed using the **ObDumpObjectByHandle**, **ObDumpObjectByName** and **ObDumpObjectByPointer** functions. These functions display the contents of an object or objects to a specified output stream with a specified level of information. The default output stream is standard output.

NTSTATUS
ObDumpObjectByHandle(
    IN HANDLE *Handle*,
    IN POB_DUMP_CONTROL *DumpControl* **OPTIONAL**
)

Parameters:

Handle —An open handle to an object.

DumpControl —An optional pointer to a dump control structure. This structure specifies the output stream and the detail level. If not specified then output should be sent to the standard output stream. Default detail level is 1.

OB_DUMP_CONTROL Structure:

PVOID *Stream* —an opaque pointer to an output stream.

ULONG *DetailLevel* —level of detail to show, along with some modifiers.

Return Value:

Status code that indicates whether or not the operation was successful.

This function may return one of the following status codes:

- STATUS_SUCCESS —normal, successful completion.
- STATUS_ACCESS_DENIED
- STATUS_INVALID_HANDLE

NTSTATUS
ObDumpObjectByName(
    IN PSTRING *ObjectName*,
    IN ULONG *Attributes*,
    IN POB_DUMP_CONTROL *DumpControl* **OPTIONAL**
)

Parameters:

ObjectName —A pointer to a string which specifies the name of the object to open.

Attributes —A set of flags that control the object attributes.

OBJ_CASE_INSENSITIVE —Indicates that the name lookup should be performed in a manner which ignores the case of the *ObjectName* rather than performing an exact match search.

DumpControl —See **ObDumpObjectByHandle** description for meaning of this parameter.

Return Value:

Status code that indicates whether or not the operation was successful.

This function may return one of the following status codes:

- STATUS_SUCCESS —normal, successful completion.
- STATUS_ACCESS_DENIED

NTSTATUS
ObDumpObjectByPointer(
    IN PVOID *Object*,
    IN POB_DUMP_CONTROL *DumpControl* **OPTIONAL**
)

Parameters:

Object —A pointer to the object's body.

DumpControl —See **ObDumpObjectByHandle** description for meaning of this parameter.

Return Value:

Status code that indicates whether or not the operation was successful.

This function may return one of the following status codes:

- STATUS_SUCCESS —normal, successful completion.

- STATUS_ACCESS_DENIED

2.14 Check Traverse Access

A parse routine calls **ObCheckTraverseAccess** for each section of a pathname to see if the caller has Traverse access to that directory.

BOOLEAN
ObCheckTraverseAccess(
    IN PVOID *DirectoryObject*,
    IN ACCESS_MASK *TraverseAccess*,
    IN PACCESS_STATE *AccessState*,
    IN BOOLEAN *TypeMutexLocked*,
    IN KPROCESSOR_MODE *PreviousMode*,
    OUT PNTSTATUS *AccessStatus*
)

Parameters:

DirectoryObject —The object header of the object being examined.

TraverseAccess —The access mask corresponding to traverse access for this directory type.

AccessState —Checks for traverse access will typically be incidental to some other access attempt. Information on the current state of that access attempt is required so that the constituent access attempts may be associated with each other in the audit log.

TypeMutexLocked —Supplies a boolean indicating whether or not the object's type mutext is locked.

AccessMode —The previous processor mode.

AccessStatus —Pointer to a variable to return the status code of the access attempt. In the case of failure this status code must be propagated back to the user.

Return Value:

BOOLEAN —TRUE if access is allowed and FALSE otherwise. *AccessStatus* contains the status code to be passed back to the caller. It is not correct to simply pass back STATUS_ACCESS_DENIED, since this will have to change with the advent of mandatory access control.

This routine is to be called by *Object* parse methods as they parse the component subdirectories of a path. On each subdirectory, they must call **ObCheckTraverseAccess**, which will examine the security descriptors on the object to determine if it is legal to traverse that directory. If it returns failure, the value returned in *AccessStatus* must be propogated back to the user.

This routine will generate audit records as appropriate.

2.15 Check Create Instance access

A parse routine calls ObCheckCreateInstance to determine if the caller is allowed to create an instance of an object.

BOOLEAN
ObCheckCreateInstanceAccess(
    IN PVOID *Object*,
    IN ACCESS_MASK *CreateInstanceAccess*,
    IN PACCESS_STATE *AccessState* **OPTIONAL**,
    IN BOOLEAN *TypeMutexLocked*,
    IN KPROCESSOR_MODE *PreviousMode*,
    OUT PNTSTATUS *AccessStatus*
)

Parameters:

Object —The object header of the object being examined.

CreateInstanceAccess —The access mask corresponding to create access for this object type.

AccessState —Checks for create access will typically be incidental to some other access attempt. Information on the current state of that access attempt is required so that the constituent access attempts may be associated with each other in the audit log.

TypeMutexLocked —Indicates whether the type mutex for this object's type is locked. The type mutex is used to protect the object's security descriptor from being modified while it is being accessed.

AccessMode —The previous processor mode.

AccessStatus —Pointer to a variable to return the status code of the access attempt. In the case of failure this status code must be propagated back to the user.

Return Value:

BOOLEAN —TRUE if access is allowed and FALSE otherwise. *AccessStatus* contains the status code to be passed back to the caller.

Routine Description:

Parse routines must call this routine to check for Create Instance access to the object. If the attempt fails, the caller must propagate the result returned in *AccessStatus* back to the user, rather than simply returning STATUS_ACCESS_DENIED.

Note that checking for the ability to create an object of a given type is different from creating the object itself. This attempt may be audited, even if the attempt to create the object ultimately fails.

2.16 Check Create Object Access

A parse routine calls **ObCheckCreateObjectAccess** to see if it may create an object in the passed directory.

BOOLEAN
ObCheckCreateObjectAccess(
    IN PVOID *DirectoryObject*,
    IN ACCESS_MASK *CreateAccess*,
    IN PACCESS_STATE *AccessState* **OPTIONAL**,
    IN BOOLEAN *TypeMutexLocked*,
    IN KPROCESSOR_MODE *PreviousMode*,
    OUT PNTSTATUS *AccessStatus*
)

Parameters:

DirectoryObject —The object header of the object being examined.

CreateAccess —The access mask corresponding to create access for this directory type.

AccessState —Checks for traverse access will typically be incidental to some other access attempt. Information on the current state of that access attempt is required so that the constituent access attempts may be associated with each other in the audit log.

TypeMutexLocked —Indicates whether the type mutex for this object's type is locked. The type mutex is used to protect the object's security descriptor from being modified while it is being accessed.

AccessMode —The previous processor mode.

AccessStatus —Pointer to a variable to return the status code of the access attempt. In the case of failure this status code must be propagated back to the user.

Return Value:

BOOLEAN —TRUE if access is allowed and FALSE otherwise. *AccessStatus* contains the status code to be passed back to the caller.

Routine Description:

This routine checks to see if we are allowed to create an object in the given directory. If the attempt fails, the caller must propagate the result returned in *AccessStatus* back to the user, rather than simply returning STATUS_ACCESS_DENIED.

This routine may generate audit messages as appropriate.

2.17 Check Implicit Object Access

Check object access when there will be no handle allocated.

BOOLEAN
ObCheckImplicitObjectAccess(
    IN PVOID *Object*,
    IN OUT PACCESS_STATE *AccessState*,
    IN BOOLEAN *TypeMutexLocked*,
    IN KPROCESSOR_MODE *AccessMode*,
    OUT PNTSTATUS *AccessStatus*
)

Parameters:

ObjectHeader —The object header of the object being examined.

AccessState —The ACCESS_STATE structure containing accumulated information about the current access attempt.

TypeMutexLocked —Indicates whether the type mutex for this object's type is locked. The type mutex is used to protect the object's security descriptor from being modified while it is being accessed.

AccessMode —The previous processor mode.

AccessStatus —Pointer to a variable to return the status code of the access attempt. In the case of failure this status code must be propagated back to the user.

Return Value:

BOOLEAN —TRUE if access is allowed and FALSE otherwise

Routine Description:

This routine is used to perform access validation for reasons other than opening or creating an object. For example, a file system may want to determine of a subject has FILE_LIST_DIRECTORY access to a directory as part of some other access validation.

For access operations on objects that are being opened or created, use ObpCheckObjectAccess.

The routine performs access validation on the passed object. The remaining desired access mask is extracted from the *AccessState* parameter and passed to the appropriate security routine to perform the access check.

Note that the RemainingDesiredAccess field in the *AccessState* parameter is not modified.

2.18 Checking Access for Object Reference

This routine is to be used to determine if a reference by name should be permitted.

BOOLEAN
ObCheckObjectReference(
    IN PVOID *Object*,
    IN OUT PACCESS_STATE *AccessState*,
    IN BOOLEAN *TypeMutexLocked*,
    IN KPROCESSOR_MODE *AccessMode*,
    OUT PNTSTATUS *AccessStatus*
)

Parameters:

ObjectHeader —The object header of the object being examined.

AccessState —The ACCESS_STATE structure containing accumulated information about the current attempt to gain access to the object.

TypeMutexLocked —Indicates whether the type mutex for this object's type is locked. The type mutex is used to protect the object's security descriptor from being modified while it is being accessed.

AccessMode —The previous processor mode.

AccessStatus —Pointer to a variable to return the status code of the access attempt. In the case of failure this status code must be propagated back to the user.

Return Value:

BOOLEAN —TRUE if access is allowed and FALSE otherwise

Routine Description:

The routine performs access validation on the passed object. The remaining desired access mask is extracted from the *AccessState* parameter and passes to the appropriate security routine to perform the access check.

If the access attempt is successful, SeAccessCheck returns a mask containing the granted accesses. The bits in this mask are turned on in the PreviouslyGrantedAccess field of the *AccessState*, and are turned off in the RemainingDesiredAccess field.

This routine differs from ObpCheckObjectAccess in that it calls a different audit routine.

2.19 Locking a security descriptor

Call **ObLockSecurityDescriptor** before reading or writing an object's security descriptor.

VOID
ObLockSecurityDescriptor(
    IN PVOID *Object*
)

Parameters:

Object —supplies a pointer to the object whose security descriptor is being examined.

Return Value: None.

Routine Description:

This function acquires the object type mutex for the passed object, which will protect the object's security descriptor from modification by another thread.

2.20 Unlocking a security descriptor

Call **ObLockSecurityDescriptor** before reading or writing an object's security descriptor.

VOID
ObUnlockSecurityDescriptor(
    IN PVOID *Object*
)

Parameters:

Object **—**supplies a pointer to the object whose security descriptor is being examined.

Return Value: None.

Routine Description:

This function releases the object type mutex for the passed object, which has been protecting the object's security descriptor from modification by another thread.

2.21 Query an object's Security Descriptor field

This routine allows components outside of OB to retrieve the Security Descriptor pointer in an object's header. The contents of this pointer does not necessarily reflect the actual security descriptor attached to an object.

VOID
ObQueryObjectSecurityDescriptor(
    IN PVOID *Object*,
    OUT PSECURITY_DESCRIPTOR ***SecurityDescriptor*
)

Parameters:

Object **—**Supplies a pointer to the object

SecurityDescriptor **—**Returns the contents of the object header's *SecurityDescriptor* field, which may be NULL.

Routine Description:

Takes a pointer to an object and returns a pointer to the security descriptor contained in the header.

2.22 Set an object's Security Descriptor field

This routine permits components outside of OB to set the security descriptor field in an object's header.

VOID
ObAssignObjectSecurityDescriptor(
    IN PVOID *Object*,
    IN PSECURITY_DESCRIPTOR *SecurityDescriptor*,
    IN POOL_TYPE *PoolType*
)

Parameters:

Object —Supplies a pointer to the object

SecurityDescriptor —Supplies a pointer to the security descriptor to be assigned to the object.

PoolType —Supplies the type of pool memory used to allocate the security descriptor.

Routine Description:

Takes a pointer to an object and sets the *SecurityDescriptor* field in the object's header. Performs security quota calculations and places the security quota for this object into the object's header.

2.23 Query an object's Security information

This routine will return a copy of the passed object's security descriptor, regardless of where the security descriptor is stored.

NTSTATUS
ObGetObjectSecurity(
    IN PVOID *Object*,
    OUT PSECURITY_DESCRIPTOR **SecurityDescriptor*,
    OUT PBOOLEAN *MemoryAllocated*
)

Parameters:

Object —Supplies the object being queried.

SecurityDescriptor —Returns a pointer to the object's security descriptor.

MemoryAllocated —indicates whether we had to allocate pool memory to hold the security descriptor or not. This should be passed back into **ObReleaseObjectSecurity**.

Return Value:

STATUS_SUCCESS —The operation was successful. Note that the operation may be successful and still return a NULL security descriptor.

STATUS_INSUFFICIENT_RESOURCES —Insufficient memory was available to satisfy the request.

Routine Description:

Given an object, this routine will find its security descriptor. It will do this by calling the object's security method.

It is possible for an object not to have a security descriptor at all. Unnamed objects such as events that can only be referenced by a handle are an example of an object that does not have a security descriptor.

2.24 Release an object's Security information

This routine frees the memory allocated by a previous call to **ObGetObjectSecurity**.

VOID
ObReleaseObjectSecurity(
    IN PSECURITY_DESCRIPTOR *SecurityDescriptor*,
    IN BOOLEAN *MemoryAllocated*
)

Parameters:

SecurityDescriptor **—**Supplies a pointer to the security descriptor to be freed.

MemoryAllocated **—**Supplies whether or not we should free the memory pointed to by *SecurityDescriptor*.

Routine Description:

This function will free up any memory associated with a queried security descriptor.

2.25 Set Security Quota Charged for object

Each object, when it is created, is alloted a certain amount of pool memory for security information. The amount is a function of the size of the Group and Dacl information in the object's security descriptor. The sum of the sizes of these items is passed to this routine, which will calculate the amount of pool memory to charge based on that sum, and place the resultant quantity into the object's header.

VOID
ObSetSecurityQuotaCharged(
    IN PVOID *Object*,
    IN OUT PULONG *SecurityQuotaCharged*,
    IN POOL_TYPE *PoolType*
)

Parameters:

Object **—**Supplies the object to be updated.

SecurityQuotaCharged —Supplies the proposed amount of quota to be charged for security information for each handle to this object. Will return the actual amount charged.

PoolType —The type of pool memory that will be allocated to hold the security information for this object.

Routine Description:

Sets the *SecurityQuotaCharged* field for the passed object. Updates the *PagedPoolCharge* or *NonPagedPoolCharge* with the new amount, depending on the value of *PoolType*.

2.26 Validate security information against quota

Any attempt to grow the security information on an object must have the resulting size checked against the maximum amount of pool memory that may be used for the object's security information.

NTSTATUS
ObValidateSecurityQuota(
    IN PVOID *Object*,
    IN ULONG *NewSize*
)

Parameters:

Object —Supplies a pointer to the object whose information is to be modified.

NewSize —Supplies the size of the proposed new security information.

Return Value:

STATUS_SUCCESS —New size is within alloted quota.

STATUS_QUOTA_EXCEEDED —The desired adjustment would have exceeded the permitted security quota for this object.

Routine Description:

This routine will check to see if the new security information is larger than is allowed by the object's pre-allocated quota.

3. Object System Services

The following routines provide an interface for user mode applications to manipulate and query objects.

3.1 Create Directory Object

Directory objects are created with the **NtCreateDirectoryObject** function:

NTSTATUS
NtCreateDirectoryObject(
    OUT PHANDLE *DirectoryHandle*,
    IN ULONG *DesiredAccess*,
    IN POBJECT_ATTRIBUTES *ObjectAttributes*
)

Parameters:

DirectoryHandle **—**A pointer to a variable that will receive the directory object handle.

DesiredAccess **—**The desired types of access to the directory. The following object type specific access flags can be specified in addition to the STANDARD_RIGHTS_REQUIRED flags described in the *Object Attributes* section.

DesiredAccess Flags:

DIRECTORY_QUERY **—**Query access to the directory is desired.

DIRECTORY_TRAVERSE **—**Name lookup access to the directory is desired.

DIRECTORY_CREATE_OBJECT **—**Name creation access to the directory is desired.

DIRECTORY_CREATE_SUBDIRECTORY **—**Subdirectory creation access to the directory is desired.

ObjectAttributes —A pointer to a structure that specifies the object's attributes. Refer to the *Object Attributes* discussion for details.

Return Value:

Status code that indicates whether or not the operation was successful.

Directory objects are an integral part of the object management functions and as such are manipulated indirectly as a result of other operations. For example, when an object is created, its name, if any, is "inserted" in a directory object and the *PointerCount* fields of both the directory object and the named object are incremented. The named object's header contains a pointer to the directory object which contains the name.

A single mutex is utilized to guard the directory structure. It must be acquired any time a directory is accessed for examination or manipulation.

The directory object's body contains the information necessary to translate an object name to a pointer to the object. Incrementing the *PointerCount* field in the directory object's header for each name in the directory prevents the directory object from being "deallocated" with outstanding names.

If a directory object is temporary and the *HandleCount* becomes zero, then an attempt is made to delete the directory object's name by conditionally removing its directory entry. Conditional deletion means that the necessary mutexes are released, the directory mutex is acquired, the directory entry which contains the directory object is located and the *HandleCount* is checked again. If the count is still zero, the directory object's name is deleted. This is done because the directory object was declared as temporary and the last handle to the object has been closed.

If the directory's name is deleted, the *PointerCount* has not yet been decremented to account for the lack of a name. Any names which still reside within the directory object are deleted. This is accomplished by acquiring the directory mutex and finding a valid name within the directory. From the valid name, the corresponding object is located and its name field and backpointer are removed, its *PointerCount* is decremented, and the permanent flag is set false. If the resulting *PointerCount* of the named object is now zero, the directory mutex is released and the object type specific delete routine is invoked.

This procedure is repeated until all valid names within the directory have been deleted, at which time the directory mutex is released, and the *PointerCount* for the directory is decremented.

Even though a directory object's name has been removed, the directory object remains until all names contained within it have been removed. This means that certain objects which had names will no longer have names once the directory object's name has been removed. This condition is detected by a NULL backpointer in the path of directory objects.

This function may return one of the following status codes:

- o STATUS_SUCCESS —normal, successful completion.
- o STATUS_ACCESS_DENIED
- o STATUS_ACCESS_VIOLATION —Either the *ObjectAttributes* pointer or the *ObjectAttributes*->ObjectName pointer or the *ObjectAttributes*->ObjectName->Buffer pointer were invalid. Or the *DirectoryHandle* pointer was invalid.
- o STATUS_DATATYPE_MISALIGNMENT —Either the *ObjectAttributes* pointer or the *ObjectAttributes*->ObjectName pointer were not aligned on a 4 byte boundary. Or the *DirectoryHandle* pointer was not aligned on a 4 byte boundary.

3.2 Open Object Directory

NTSTATUS
NtOpenDirectoryObject(
    OUT PHANDLE *DirectoryHandle*,
    IN ACCESS_MASK *DesiredAccess*,
    IN POBJECT_ATTRIBUTES *ObjectAttributes*
)

<u>Parameters:</u>

DirectoryHandle —A pointer to a variable that will receive the directory object handle.

DesiredAccess —The desired types of access to the directory. The following object type specific access flags can be specified in addition to the STANDARD_RIGHTS_REQUIRED flags described in the *Object Attributes* section.

DesiredAccess Flags:

DIRECTORY_QUERY —Query access to the directory is desired.

DIRECTORY_TRAVERSE —Name lookup access to the directory is desired.

DIRECTORY_CREATE_OBJECT —Name creation access to the directory is desired.

DIRECTORY_CREATE_SUBDIRECTORY —Subdirectory creation access to the directory is desired.

ObjectAttributes —A pointer to a structure that specifies the object's attributes. Refer to the *Object Attributes* discussion for details.

Return Value:

Status code that indicates whether or not the operation was successful.

This function may return one of the following status codes:

- STATUS_SUCCESS —normal, successful completion.
- STATUS_ACCESS_DENIED
- STATUS_ACCESS_VIOLATION —Either the *ObjectAttributes* pointer or the *ObjectAttributes*->ObjectName pointer or the *ObjectAttributes*->ObjectName->Buffer pointer were invalid. Or the *DirectoryHandle* pointer was invalid.
- STATUS_DATATYPE_MISALIGNMENT —Either the *ObjectAttributes* pointer or the *ObjectAttributes*->ObjectName pointer were not aligned on a 4 byte boundary. Or the *DirectoryHandle* pointer was not aligned on a 4 byte boundary.

3.3 Query Object Directory

The names in a directory object can be queried using the **NtQueryDirectoryObject** function:

NTSTATUS
NtQueryDirectoryObject(
  IN HANDLE *DirectoryHandle*,
  OUT PVOID *Buffer*,
  IN ULONG *Length*,
  IN BOOLEAN *ReturnSingleEntry*,
  IN BOOLEAN *RestartScan*,
  IN OUT PULONG *Context*,
  OUT PULONG *ReturnLength* **OPTIONAL**
)

Parameters:

DirectoryHandle —handle of directory object being queried.

Buffer —pointer to where directory entries are to be returned. The format is array of structures containing the following fields:

OBJECT_DIRECTORY_INFORMATION Structure:

STRING *Name* —*Name* of an object in the directory

STRING *TypeName* —Type name of the object

The *Buffer* fields of each name string point to memory allocated at the end of the storage pointed to by the *Buffer* parameter. This the array of Directory Entries grows down and the actual characters for each string grow up and if they meet in the middle, then the operation stops and this function returns to the caller.

Length —maximum number of bytes that can be stored in the location pointed to by the *Buffer* parameter.

ReturnSingleEntry —TRUE forces the query to stop after a single entry has been returned. Otherwise the query will return as many entries as there is room for in the output buffer.

RestartScan —TRUE forces the query to start with the first name in the directory. Otherwise the query picks up with the next name after the last name returned by the previous call to **NtQueryDirectoryObject** for this directory object.

Context —A pointer to a context value. This value is used by this system service to remember its position within a directory object. The input value is ignored if the *RestartScan* parameter is TRUE.

ReturnLength —optional pointer to a variable that will receive the actual number of bytes stored in the location pointed to by the *Buffer* parameter.

Return Value:

Status code that indicates whether or not the operation was successful.

This function returns one or more entries from the directory object specified by the *DirectoryHandle* parameter.

This function remembers its current position across calls by storing a 32-bit number into the location pointed to by the *Context* parameter. This number is a logical index into the directory. It is not a pointer. This will prevent deletions that happen between calls from turning a *Context* value into a garbage quantity. It may become inaccurate due to insertions and deletions, but it will not bug check the system.

This function may return one of the following status codes:

o STATUS_SUCCESS —normal, successful completion.

o STATUS_ACCESS_DENIED

o STATUS_INVALID_HANDLE

3.4 Create Symbolic Link

NTSTATUS
NtCreateSymbolicLinkObject(
OUT PHANDLE *LinkHandle*,
IN ULONG *DesiredAccess*,
IN POBJECT_ATTRIBUTES *ObjectAttributes*,
IN PSTRING *LinkTarget*
)

Parameters:

LinkHandle —Supplies a pointer to a variable that will receive the symbolic link object handle.

DesiredAccess —The desired types of access to the symbolic link object. The following object type specific access flags can be specified in addition to the STANDARD_RIGHTS_REQUIRED flags described in the *Object Attributes* section.

DesiredAccess Flags:

SYMBOLIC_LINK_QUERY —Query access to the symbolic link is desired.

ObjectAttributes —A pointer to a structure that specifies the object's attributes. Refer to the *Object Attributes* discussion for details.

LinkTarget —Supplies the target name for the symbolic link object.

Return Value:

Status code that indicates whether or not the operation was successful.

This function creates a symbolic link object, sets its initial value to value specified in the *LinkTarget* parameter, and opens a handle to the object with the specified desired access.

The symbolic link object type has a parse procedure that implements the symbolic link semantics. Basically if the parse procedure is called and if the remaining string is not null, then the remaining string value is concatenated with the target name string stored in the symbolic link object, separated by a path separator character. The result replaces the complete string and the OBJ_REPARSE is returned to trigger the reparse.

If the remaining string is null, then it assumes the caller is trying to open the symbolic link and returns a pointer to the symbolic link object body. This will fail with STATUS_OBJECT_TYPE_MISMATCH if the caller did not specify the symbolic link object type.

Otherwise the symbolic link parse procedure returns NULL to indicate an error.

This function may return one of the following status codes:

- o STATUS_SUCCESS —normal, successful completion.
- o STATUS_ACCESS_DENIED
- o STATUS_ACCESS_VIOLATION —Either the *ObjectAttributes* pointer or the *ObjectAttributes*->ObjectName pointer or the *ObjectAttributes*->ObjectName->Buffer pointer were invalid. Or the *LinkTarget*, *LinkTarget*->Buffer or the *LinkHandle* pointer were invalid.
- o STATUS_DATATYPE_MISALIGNMENT —Either the *ObjectAttributes* pointer or the *ObjectAttributes*->ObjectName pointer were not aligned on a 4 byte boundary. Or the *LinkTarget* or *LinkHandle* pointer were not aligned on a 4 byte boundary.

3.5 Open Symbolic Link

NTSTATUS
NtOpenSymbolicLinkObject(
    OUT PHANDLE *LinkHandle*,
    IN ULONG *DesiredAccess*,
    IN POBJECT_ATTRIBUTES *ObjectAttributes*
)

<u>Parameters:</u>

LinkHandle —Supplies a pointer to a variable that will receive the symbolic link object handle.

DesiredAccess —The desired types of access to the symbolic link object. The following object type specific access flags can be specified in addition to the STANDARD_RIGHTS_REQUIRED flags described in the *Object Attributes* section.

DesiredAccess Flags:

SYMBOLIC_LINK_QUERY —Query access to the symbolic link is desired.

ObjectAttributes —A pointer to a structure that specifies the object's attributes. Refer to the *Object Attributes* discussion for details.

Return Value:

Status code that indicates whether or not the operation was successful.

This function opens a handle to a symbolic link object with the specified desired access.

This function may return one of the following status codes:

- STATUS_SUCCESS —normal, successful completion.
- STATUS_ACCESS_DENIED
- STATUS_ACCESS_VIOLATION —Either the *ObjectAttributes* pointer or the *ObjectAttributes*->ObjectName pointer or the *ObjectAttributes*->ObjectName->Buffer pointer were invalid. Or the *LinkHandle* pointer was invalid.
- STATUS_DATATYPE_MISALIGNMENT —Either the *ObjectAttributes* pointer or the *ObjectAttributes*->ObjectName pointer were not aligned on a 4 byte boundary. Or the *LinkHandle* pointer was not aligned on a 4 byte boundary.

3.6 Query Symbolic Link

NTSTATUS
NtQuerySymbolicLinkObject(
  IN HANDLE *LinkHandle*,
  OUT PSTRING *LinkTarget*
)

Parameters:

LinkHandle **—**Supplies a handle to a symbolic link object.

LinkTarget **—**Supplies a pointer to a record that is to receive the target name of the symbolic link object.

Return Value:

Status code that indicates whether or not the operation was successful.

This function queries the state of an symbolic link object and returns the requested information in the string pointed to by the *LinkTarget* parameter.

3.7 Wait For Single Object

A wait operation on a waitable object is accomplished with the **NtWaitForSingleObject** function:

NTSTATUS
NtWaitForSingleObject(
  IN HANDLE *Handle*,
  IN BOOLEAN *Alertable*,
  IN PTIME *TimeOut* **OPTIONAL**
)

Parameters:

Handle **—**An open handle to a waitable object.

Alertable **—**A boolean value that specifies whether the wait is alertable.

TimeOut —An optional pointer to a time-out value that specifies the absolute or relative time over which the wait is to be completed.

Return Value:

Status code that indicates whether or not the operation was successful.

Waiting on an object checks the current state of the object. If the current state of the object allows continued execution, any adjustments to the object state are made (for example, decrementing the semaphore count for a semaphore object) and the thread continues execution. If the current state of the object does not allow continued execution, the thread is placed into the wait state pending the change of the object's state or time-out.

This function requires SYNCHRONIZE access to the passed handle.

This function may return one of the following success status codes that indicates how the wait was satisfied:

- A value of STATUS_TIME_OUT indicates that the wait was terminated due to the *TimeOut* conditions.
- A value of STATUS_SUCCESS indicates the specified object attained a Signaled state thus completing the wait.
- A value of STATUS_ABANDONED indicates the specified object attained a Signaled state but was abandoned.

This function may return one of the following error status codes if the wait was not satisfied:

- STATUS_ALERTED
- STATUS_USER_APC
- STATUS_HANDLE_NOT_WAITABLE
- STATUS_ACCESS_DENIED
- STATUS_INVALID_HANDLE

- o STATUS_ACCESS_VIOLATION —The Timeout pointer was invalid.
- o STATUS_DATATYPE_MISALIGNMENT —The Timeout pointer was not aligned on a 4 byte boundary.

3.8 Wait for Multiple Objects

A wait operation on multiple waitable objects (up to MAXIMUM_WAIT_OBJECTS) is accomplished with the **NtWaitForMultipleObjects** function:

NTSTATUS
NtWaitForMultipleObjects(
    IN ULONG *Count*,
    IN HANDLE *Handles*[],
    IN WAIT_TYPE *WaitType*,
    IN BOOLEAN *Alertable*,
    IN PTIME *TimeOut* **OPTIONAL**
)

Parameters:

Count —A count of the number of objects that are to be waited on.

Handles —An array of object handles. An error status is returned if more than one of the handles refers to the same object. This can occur even if two handle values are different but both refer to the same object.

WaitType —The type of operation that is to be performed (WaitAny or WaitAll).

Alertable —A boolean value that specifies whether the wait is alertable.

TimeOut —An optional pointer to a time-out value that specifies the absolute or relative time over which the wait is to be completed.

Return Value:

Status code that indicates whether or not the operation was successful.

This function requires SYNCHRONIZE access to the passed handle.

This function may return one of the following success status codes that indicates how the wait was satisfied:

- A value of STATUS_TIME_OUT indicates that the wait was terminated due to the *TimeOut* conditions.

- A value from 0 to MAXIMUM_WAIT_OBJECTS - 1, indicates, in the case of wait for any object, the object number which satisfied the wait. In the case of wait for all objects, the value only indicates that the wait was completed successfully.

- A value from STATUS_ABANDONED to STATUS_ABANDONED + (MAXIMUM_WAIT_OBJECTS - 1), indicates, in the case of wait for any object, the object number which satisfied the event, and that the object which satisfied the event was abandoned. In the case of wait for all objects, the value indicates that the wait was completed successfully and at least one of the objects was abandoned.

This function may return one of the following error status codes if the wait was not satisfied:

- STATUS_ALERTED

- STATUS_USER_APC

- STATUS_INVALID_PARAMETER

- STATUS_HANDLE_NOT_WAITABLE

- STATUS_ACCESS_DENIED

- STATUS_INVALID_HANDLE

- STATUS_QUOTA_EXCEEDED

- STATUS_NO_MEMORY

- STATUS_INVALID_PARAMETER_MIX —One or more of the handle values in the *Handles* array referenced the same object.

- STATUS_ACCESS_VIOLATION —The *Handles* or Timeout pointer was invalid.

- o STATUS_DATATYPE_MISALIGNMENT —The *Handles* or Timeout pointer was not aligned on a 4 byte boundary.

3.9 Duplicate Handle

A duplicate handle can be created with the **NtDuplicateObject** function:

NTSTATUS
NtDuplicateObject(
    IN HANDLE *SourceProcessHandle,*
    IN HANDLE *SourceHandle,*
    IN HANDLE *TargetProcessHandle,*
    OUT PHANDLE *TargetHandle,*
    IN ACCESS_MASK *DesiredAccess,*
    IN ULONG *HandleAttributes,*
    IN ULONG *Options*
)

Parameters:

SourceProcessHandle —An open handle to a process object or NtCurrentProcess().

SourceHandle —An open handle valid in the context of the source process.

TargetProcessHandle —An open handle to a process object or NtCurrentProcess().

TargetHandle —A pointer to a variable which receives the new handle that points to the same object as *SourceHandle* does.

DesiredAccess —The access requested to for the new handle. This access must be equal to or a proper subset of the granted access associated with the *SourceHandle*. This parameter is ignored if the DUPLICATE_SAME_ACCESS option is specified.

HandleAttributes —The attributes to associated with the new handles. Only *OBJ_INHERIT* is relevant.

Options —Specifies optional behaviors for the caller.

Options Flags:

DUPLICATE_CLOSE_SOURCE —The *SourceHandle* will be closed by this server prior to returning to the caller. This occurs regardless of any error status returned.

DUPLICATE_SAME_ACCESS —The *DesiredAccess* parameter is ignored and instead the *GrantedAccess* associated with *SourceHandle* is used as the *DesiredAccess* when creating the *TargetHandle*.

Return Value:

Status code that indicates whether or not the operation was successful.

This is a generic function and operates on any type of object.

This function requires PROCESS_DUP_ACCESS to both the *SourceProcessHandle* and the *TargetProcessHandle*.

This function may return one of the following status codes:

- STATUS_SUCCESS —normal, successful completion.
- STATUS_ACCESS_DENIED
- STATUS_INVALID_HANDLE
- STATUS_QUOTA_EXCEEDED
- STATUS_NO_MEMORY
- STATUS_ACCESS_VIOLATION —The *TargetHandle* pointer was invalid.
- STATUS_DATATYPE_MISALIGNMENT —The *TargetHandle* pointer was not aligned on a 4 byte boundary.

3.10 Close Handle

An open handle to any object can be closed with the **NtClose** function:

NTSTATUS
NtClose(
    IN HANDLE *Handle*
)

Parameters:

Handle —An open handle to an object.

Return Value:

Status code that indicates whether or not the operation was successful.

This is a generic function and operates on any type of object.

Closing an open handle to an object causes the handle to become invalid and the *HandleCount* of the associated object to be decremented and object retention checks to be performed.

This function may return one of the following status codes:

- STATUS_SUCCESS —normal, successful completion.
- STATUS_INVALID_HANDLE

3.11 Making an Object Temporary

An object can be made temporary with the **NtMakeTemporaryObject** function:

NTSTATUS
NtMakeTemporaryObject(
    IN HANDLE *Handle*
)

Parameters:

Handle —An open handle to an object.

Return Value:

Status code that indicates whether or not the operation was successful.

This is a generic function and operates on any type of object.

Making an object temporary causes the permanent flag of the associated object to be cleared. A temporary object has a name as long as its *HandleCount* is greater than zero. When the *HandleCount* becomes zero, the name is deleted and the *PointerCount* adjusted appropriately.

This function may return one of the following status codes:

- STATUS_SUCCESS **—**normal, successful completion.
- STATUS_ACCESS_DENIED
- STATUS_INVALID_HANDLE

3.12 Query Object

Information about an opened object can be obtained with the **NtQueryObject** function:

NTSTATUS
NtQueryObject(
    IN HANDLE *Handle*,
    IN OBJECT_INFORMATION_CLASS *ObjectInformationClass*,
    OUT PVOID *ObjectInformation*,
    IN ULONG *Length*,
    OUT ULONG ****ReturnLength* **OPTIONAL
)

Parameters:

Handle **—**Specifies the object that information is being requested from.

ObjectInformationClass **—**Specifies the type of information to retrieve from the specified object.

ObjectInformationClass Values:

ObjectBasicInformation —Returns the basic information about the specified object.

ObjectNameInformation —Returns the complete path name of the object referred to by the *Object*.

ObjectTypeInformation —Returns the name of the object type associated with the object.

ObjectInformation —A pointer to a buffer which receives the specified information. The format and content of the buffer depend on the specified object information class.

ObjectInformation Format by Information Class:

ObjectBasicInformation —Data type is POBJECT_BASIC_INFORMATION

OBJECT_BASIC_INFORMATION Structure:

ULONG *Attributes* —The attributes associated with this object. Only *OBJ_INHERIT*, *OBJ_PERMANENT* and *OBJ_EXCLUSIVE* are relevant after an object handle has been created.

ACCESS_MASK *GrantedAccess* —The access mask bits that were granted to the current process with the passed handle.

ULONG *PagedPoolCharge* —How much PagedPool is charged against a process when it creates a handle to this object.

ULONG *NonPagedPoolCharge* —How much NonPagedPool is charged against a process when it creates a handle to this object.

ULONG *NameInfoSize* —The size needed to store a copy of the name associated with this object. Zero if no name.

ULONG *TypeInfoSize* —The size needed to store a copy of the type name associated with this object.

ULONG *SecurityDescriptorSize* —The size needed to store a copy of the *SecurityDescriptor* associated with this object. See the **NtQuerySecurityObject** for a description of how to get the actual copy of the security descriptor.

ObjectNameInformation —Data type is POBJECT_NAME_INFORMATION

OBJECT_NAME_INFORMATION Structure:

STRING *Name* —The name associated with this object, if any.

ObjectTypeInformation —Data type is POBJECT_TYPE_INFORMATION

OBJECT_TYPE_INFORMATION Structure:

STRING *TypeName* —The name of the object type associated with this object.

Length —Specifies the length in bytes of the *ObjectInformation* buffer.

ReturnLength —An optional parameter that receives the number of bytes placed in the *ObjectInformation* buffer.

Return Value:

Status code that indicates whether or not the operation was successful.

This function requires READ_CONTROL access to the passed handle.

This function may return one of the following status codes:

- STATUS_SUCCESS —normal, successful completion.
- STATUS_INVALID_INFO_CLASS —The *ObjectInformationClass* parameter did not specify a valid value.
- STATUS_INFO_LENGTH_MISMATCH —The value of the ObjectInformationLength parameter did not match the length required for the information class requested by the *ObjectInformationClass* parameter.
- STATUS_ACCESS_DENIED

- o STATUS_INVALID_HANDLE

3.13 Set Security Descriptor for an Object

The function **NtSetSecurityObject** takes a well formed Security Descriptor provided by the caller and assigns specified portions of it to an object. Based on the flags set in the Security Information parameter and the caller's access rights, this procedure will replace any or all of the security information associated with an object.

This is the only function available to users and applications for changing security information, including the owner ID, group ID, and the discretionary and system ACLs of an object. The caller must have WRITE_OWNER access to the object to change the owner or primary group of the object. The caller must have WRITE_DAC access to the object to change the discretionary ACL. The caller must have the "SeSecurityPrivilege" privilege to assign a system ACL to an object.

NTSTATUS
NtSetSecurityObject(
    IN HANDLE *Handle*,
    IN SECURITY_INFORMATION *SecurityInformation*,
    IN PSECURITY_DESCRIPTOR *SecurityDescriptor*
)

<u>Parameters:</u>

Handle —A handle to an existing object.

SecurityInformation —Indicates which security information is to be applied to the object. The value(s) to be assigned are passed in the *SecurityDescriptor* parameter.

The security information is specified using the following boolean flag fields:

SecurityInformation.Owner (Object's Owner SID)
SecurityInformation.Group (Object's Group SID)
SecurityInformation.Dacl (Object's Discretionary ACL)
SecurityInformation.Sacl (Object's System ACL)

SecurityDescriptor —A pointer to a well formed Security Descriptor.

Return Value:

Status code that indicates whether or not the operation was successful.

This function may return one of the following status codes:

- o STATUS_SUCCESS —normal, successful completion.
- o STATUS_ACCESS_DENIED
- o STATUS_PRIVILEGE_NOT_HELD
- o STATUS_INVALID_HANDLE

3.14 Query Security Descriptor for an Object

The function **NtQuerySecurityObject** returns to the caller requested security information currently assigned to an object.

Based on the caller's access rights and privileges this procedure will return a security descriptor containing any or all of the object's owner ID, group ID, discretionary ACL or system ACL. To read the owner ID, group ID, or the discretionary ACL the caller must be granted READ_CONTROL access to the object. To read the system ACL the caller must have "SeSecurityPrivilege" privilege.

```
NTSTATUS
NtQuerySecurityObject(
    IN HANDLE Handle,
    IN SECURITY_INFORMATION SecurityInformation,
    OUT PSECURITY_DESCRIPTOR SecurityDescriptor,
    IN ULONG Length,
    OUT PULONG LengthNeeded
    )
```

Parameters:

Handle —A handle to an existing object.

SecurityInformation —Supplies a value describing which pieces of security information are being queried. The values that may be specified are the same as those defined in the **NtSetSecurityObject** API section.

SecurityDescriptor —A pointer to the buffer to receive a copy of the requested security information. This information is returned in the form of a security descriptor.

Length —The size, in bytes, of the Security Descriptor buffer.

LengthNeeded —A pointer to the variable to receive the number of bytes needed to store the complete security descriptor. If *LengthNeeded* is less than or equal to *Length* then the entire security descriptor is returned in the output buffer, otherwise none of the descriptor is returned.

Return Value:

Status code that indicates whether or not the operation was successful.

This function may return one of the following status codes:

o STATUS_SUCCESS —normal, successful completion.

o STATUS_BUFFER_TOO_SMALL —The value of the *Length* parameter did not specify enough memory for the requested information. The *LengthNeeded* variable will be filled in with the amount of memory needed.

o STATUS_ACCESS_DENIED

o STATUS_PRIVILEGE_NOT_HELD

o STATUS_INVALID_HANDLE

x