Inside NT Object Manager

Mark Russinovich

(Reprinted from WindowsItPro Magazine)

The Object Manager is the Windows NT Executive subsystem that receives the least amount of attention or recognition. Ironically, the Object Manager provides a resource management support infrastructure that all other NT Executive subsystems (including the Memory Manager, I/O Manager, and Process Manager) rely on. The Object Manager is a support subsystem that performs its work behind the scenes. As an NT systems administrator or a Win32 programmer, you probably will never interact directly with the Object Manager; however, almost everything you do, from opening files to starting a program to viewing the Registry, requires its assistance.

In this tour of the Object Manager, I'll first describe where the Object Manager fits into NT's architecture, the role it plays in common operations, and the services it provides. Next, I'll explain how NT's subsystems define object types and what kind of information the Object Manager tracks. Finally, I'll look at the Object Manager namespace, which is the doorway to file system namespaces and the Registry namespace.

Resource Management

A major part of NT's role is managing a computer's physical and logical resources, such as physical and virtual memory, disks, files, processes, threads, synchronization primitives (semaphores, events, etc.), printers, and video displays. NT must provide a mechanism whereby programs can look up resources, share them, protect them, read and modify their attributes, and interact with them. Thus, resource management encompasses tracking the state of resources, allowing only actions consistent with their state, and an API so that programs can manipulate them.

Files are a visible example of a common resource. NT provides an API for creating and opening files; modifying their attributes (hidden, read-only, etc.); and on NTFS, ensuring that programs honor file security settings. Programs expect these capabilities, and NT implements its resource management (i.e., the way NT efficiently keeps track of a file's state) hidden from applications. The operating system does the work of tracking the state and protecting resources.

In NT, the typical way a program accesses a resource such as a file is to open or create the resource and then manipulate it. Usually, a resource is assigned a name when it's created so that programs can share it. To look up or open an existing shared resource or a global resource (e.g., a file system, disk, or other physical device attached to the system), a program specifies the resource's name. However, programs often create unnamed resources, which typically are logical resources (e.g., synchronization primitives), that an application will privately use.

Regardless of whether resources are physical resources (such as disk drives and keyboards) or logical resources (such as files and shared virtual memory), NT represents them as object data structures, which the Object Manager defines. Objects are shells that other NT Executive subsystems can fill in so that they can build custom object types to represent the resources they manage. The Object Manager tracks information that is independent of the type of resource an object represents; the subsystem-specific core of an object contains data relevant to a particular resource.

The reason NT builds objects using the Object Manager's infrastructure is simple: Each subsystem does not need to reinvent the wheel to track the system-related state of an object, look up its resources, charge applications for the memory required to allocate an object, and protect resources with security. By concentrating these functions in the Object Manager, Microsoft can share code across subsystems, write and validate NT's security code once, and apply the same naming conventions to all resources.

Object Types

The Object Manager's duties involve creating object types, creating and deleting objects, querying and setting an object's attributes, and locating objects. As I mentioned previously, Executive subsystems (including the Object Manager) create objects that represent the resources they manage. Before a subsystem (e.g., the I/O Manager) can tell the Object Manager to make an object (e.g., a file), the Object Manager must define the underlying object type that the subsystem will instantiate the object from. Object types, like classes in C++ parlance, store information common to all objects representing the same type of resource. This information includes statistical information and pointers to the method procedures that the subsystems will invoke when they perform actions on an object of the corresponding type. Thus, when the Executive subsystems initialize, they call a function in the Object Manager to define their object types.

For example, open files require resource management, and the I/O Manager subsystem builds file objects (with help from the Object Manager) to track open files. Inside the file object's body, the I/O Manager keeps track of the file's name, its logical volume, and whether the file is marked for deletion; the file object also provides storage for private data associated with the file system driver that the file belongs to.

Other examples of subsystem object types include process objects, which represent active processes; section objects, which represent memory-mapped files or shared memory; and device objects, which represent physical or logical devices. Table 1 summarizes the 23 object types NT 4.0 defines.

**TABLE 1:
Object Types and Defining Subsystems**

Object Type	Represents	Defining Subsystem
Object type	Object type object	Object Manager
Directory	Object namespace	Object Manager
SymbolicLink	Object namespace	Object Manager
Event	Synchronization primitive	Executive
EventPair	Synchronization primitive	Executive
Mutant	Synchronization primitive	Executive
Semaphore	Synchronization primitive	Executive
Windows Station	Login session	Win32
Desktop	Windows desktop	Win32
Timer	Timer notifications	Executive
File	Tracks open files	I/O Manager
IoCompletion	Tracks I/O completion notifications	I/O Manager
Adapter	DMA resource	I/O Manager
Controller	DMA controller	I/O Manager
Device	Logical or physical device	I/O Manager
Driver	Device driver	I/O Manager
Key	Doorway to the Registry	Configuration Manager
Port	Communications channel	LPC Facility

Section	Memory mapping	Memory Manager
Process	Active process	Process Manager
Thread	Active thread	Process Manager
Token	Process security profile	Process Manager
Profile	Performance monitoring	Kernel

As with objects, NT implements object types as data structures. Figure 1 shows typical information an object type data structure stores. The object type's statistical data is useful for system monitoring. This data includes information such as the name of the object type, how many objects of that type currently exist, the maximum number of objects that have existed at any time, and the default amount of memory that NT charges a process each time the subsystem creates an object of that type.

DATA

Type Name
Total Objects
Total Handles
Max Objects
Max Handles
Paged Pool Charge
NonPaged Pool Charge

METHODS

Dump Procedure
Open Procedure
Close Procedure
Delete Procedure
Parse Procedure
Security Procedure
Query Name Procedure
Okay-To-Close Procedure

Object type procedures or methods are what really differentiate object types. When a subsystem creates an object type, the subsystem passes to the Object Manager a data structure that contains pointers to all the object type procedures. The Object Manager calls these procedures when the subsystem requires actions performed on an object. For example, if a subsystem wants to close an object, the Object Manager first calls the Okay-To-Close Procedure (if the subsystem has specified one). If that procedure returns a FALSE value, an error returns to the closer to signal that the object

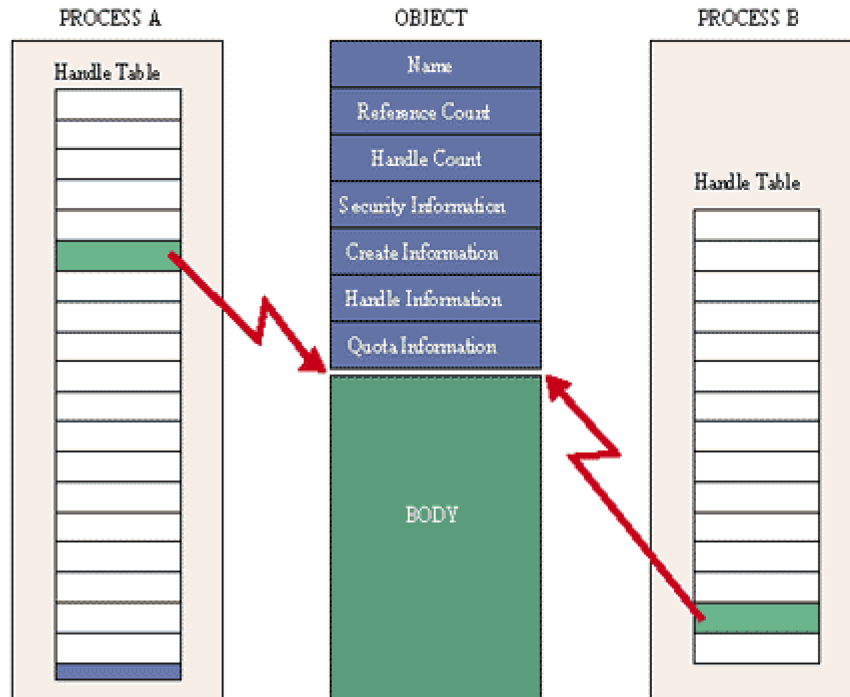
cannot be closed. If the procedure returns a TRUE value, the Object Manager then calls the Close Procedure so that the subsystem responsible for the object can clean up the object's state. Each object type procedure receives a predefined list of parameters that includes information pertinent to the requested operation. The call-out mechanism the Object Manager uses for object types lets subsystems see and control the actions taken on the objects (and therefore, the resources) that they manage.

I'll describe the purpose of most of the object type procedures throughout the rest of the article, but I'll comment on two unrelated procedures that appear in Figure 1 now. The Dump Procedure is not defined for any object type, nor does NT ever reference it. Microsoft developers probably relied on dump procedures in the early days of NT, but they removed the code after they had the basic object building blocks in place and working. The Security Procedure lets subsystems implement object security schemes that differ from NT's default security policies. NT falls back on its default security policies if a subsystem does not define a Security Procedure.

Objects

When a subsystem directs the Object Manager to create an object, the subsystem passes the Object Manager a pointer to an object type, which serves as the connection to the object type's global data and procedures. Other parameters include an optional name and security information to be applied to the new object, the size of the subsystem-specific body of the object, and pool charges that can override the default charges in the object type.

To the subsystem creating the new object, the Object Manager returns a pointer to the body of the new object. In this body area, subsystems can store data that tracks the state of the resource the object represents. Preceding this body (effectively private to the Object Manager) is an object header, which Figure 2 illustrates. The object header stores the name of the object, the parameters passed to the creation function, the object's security attributes, and a pointer to the object type.



Two fields in the object header, Handle Count and Reference Count, track different kinds of references to the object. Handle Count refers to the number of times applications have opened the object. When a program opens or creates a resource, NT's APIs return a special value, or handle, that the program can use to refer to the open resource. APIs that manipulate a resource's state use the resource's handle in lieu of its name; thus, handles provide a convenient way to refer to open resources.

Although handles are opaque values to applications, handles reference entries in a process' handle table (also shown in Figure 2). A handle table is a dynamically managed array (i.e., no hard upper ceiling exists for how large the array can become) that the Object Manager indexes via handles to locate the objects the handles refer to. Handles are process specific, so two processes can have different handle values for the same open resource.

When a program closes an object, the Object Manager calls the object type's Close Procedure and passes it the object's handle count. One example of a subsystem that monitors object handle counts is the I/O Manager, which notifies file systems when it closes all handles for a file object so that the file system can perform necessary cleanup operations. At that time, a file system will delete a file marked for deletion, because no application is using it.

The second field tracked in object headers, Reference Count, is the total number of references to an object. Operating system components can reference or create objects without going through the NT API, and consequently, do not require handles. Reference Count records the number of handles for an object plus the number of active references that operating system components make to the object. The Object Manager uses this count to determine when the system no longer needs an object. When Reference Count drops to zero, nothing in the system is using the object, so the system can remove the object's state and storage. The Object Manager will call an object type's Delete Procedure (which eliminates the object, not the resource the object represents) with the object as a parameter.

Locating Objects

Up to this point, I've avoided the details about how applications open an object by specifying the object's name. Every NT object that has a name lives within the Object Manager namespace. This namespace, which is very much like the familiar file system namespace, consists of directories that contain subdirectories or objects. In fact, you enter the file system namespace (with names like C:\temp\file.txt) and the Registry namespace (with names like Registry\Machine) via the Object Manager namespace. First, let's look at the Object Manager namespace, and then we'll look at how NT embeds alternative namespaces within it.

If you study Table 1, you'll see the Object Manager's Directory and SymbolicLink object types. NT uses these object types to define the Object Manager namespace. The optional name given to an object when it's created locates the object within the namespace. When NT is initializing, various subsystems create directories in the Object Manager namespace. The I/O Manager creates a \device subdirectory that it will use to store named device objects, and the Object Manager creates a subdirectory called \??. The \?? subdirectory holds objects accessible via the Win32 API. Thus, any Object Manager resource referenced from Win32 must have a corresponding named object in this subdirectory.

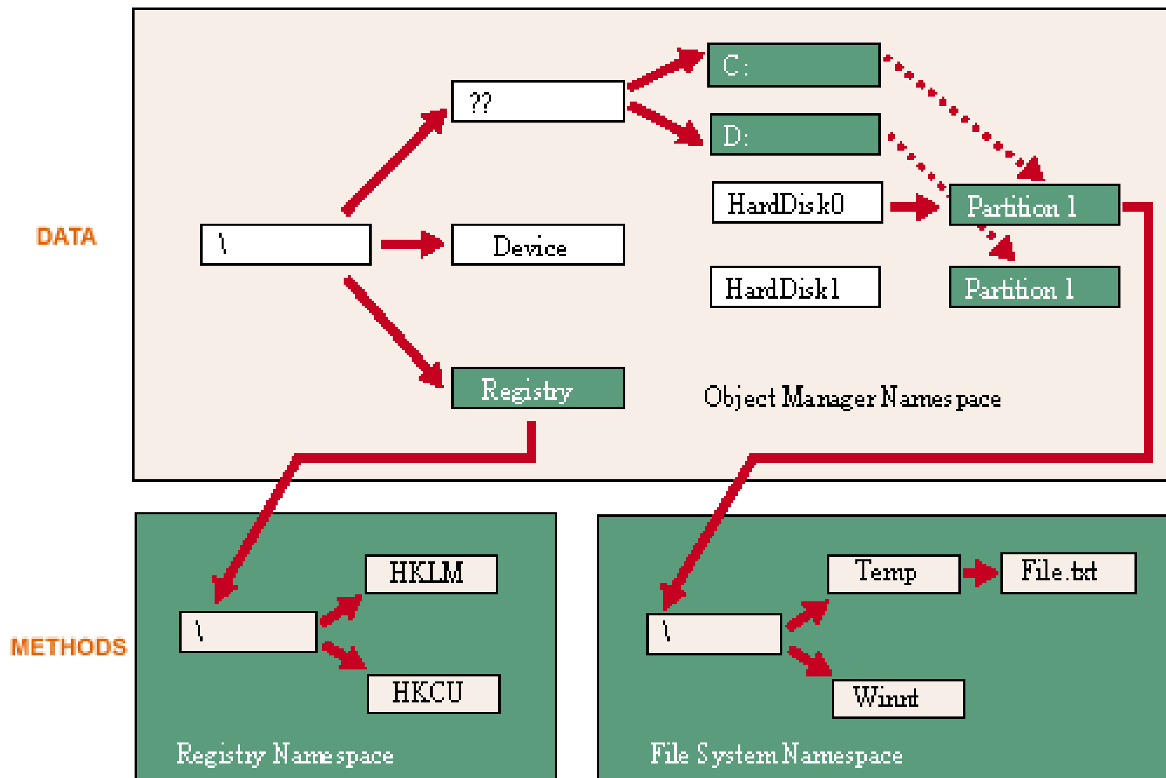
For example, serial ports named COM1, COM2, and so forth in Win32 have objects with those names in the \?? subdirectory. You'll also find C: and other drive letters in this directory. The objects with those names are symbolic link objects, which point (with alternative names) at objects elsewhere in the namespace. Drive letters point to the \device subdirectory at device objects that have names

associated with the hard disk partitions they reside on. For example, C: might point at \device\harddisk0\partition1.

An object type's Parse Procedure is what lets NT connect alternative namespaces to the Object Manager namespace. When the Object Manager performs a name lookup and encounters an object, the Object Manager checks to see whether the object's type has a Parse Procedure, and calls it. The subsystem managing the object type can then take the remaining portion of the name and perform a lookup within the subsystem's namespace.

The same sequence of events happens when you open C:\temp\file.txt from a Win32 program. First, Win32 translates the name to \??\C:\temp\file.txt. Next, NT calls the Object Manager's name-parsing routine, which locates the C: symbolic link object in the \?? directory. The Object Manager then looks up \device\harddisk0\partition1, which the symbolic link points to, and finds a device object. The Object Manager passes the rest of the name, \temp\file.txt, to the I/O Manager's device object type Parse Procedure, which locates the file system responsible for C: and hands it the name.

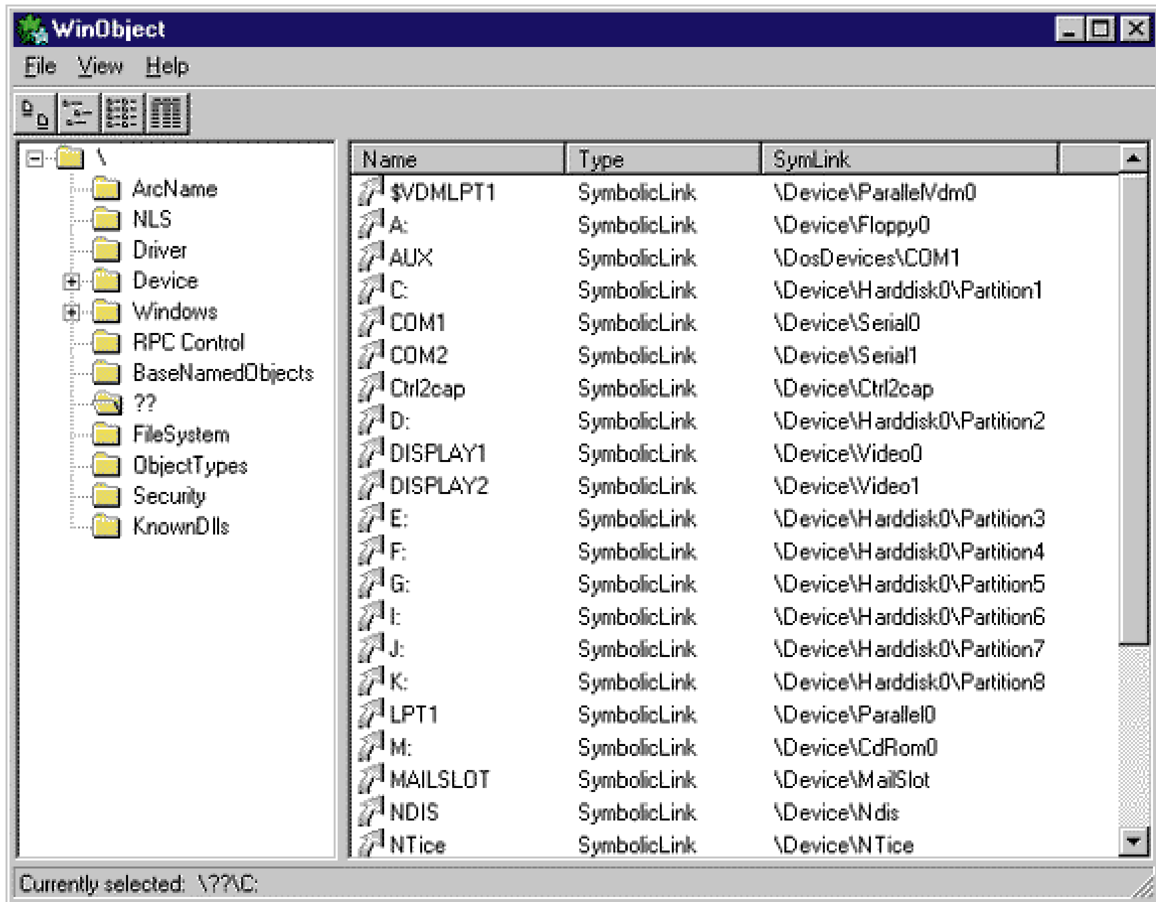
You enter the Registry namespace similarly via the \Registry key object type Parse Procedure. Figure 3 presents a simplified depiction of these three namespaces and how they are connected. (In Figure 3, HKLM stands for HKEY_CU stands for HKEY_CURRENT_USER.)



Most Win32 programmers and systems administrators don't know about the Object Manager namespace because they don't need to know about it to open files and Registry keys. However, you can use native NT system services to obtain information about what's in the namespace. With the Win32 software development kit (SDK), Microsoft provides the WinObj tool, which will display the namespace as if you were browsing with Explorer. Unfortunately, the WinObj tool has several

significant bugs that cause it to display incorrect information (e.g., inaccurate handle and reference counts).

Another version of the WinObj tool, which you can get at http://www.ntinternals.com/winobj.htm, doesn't suffer from the same problems as Microsoft's WinObj tool, and it displays additional information about certain object types. Screen 1 shows the view of the Object Manager's \?? directory that this alternative WinObj tool displays. One subdirectory worth noting is the ObjectTypes subdirectory, which contains all the defined object types.



A Little Knowledge Goes a Long Way

Although you can get along just fine managing or programming NT without knowing about the Object Manager, some familiarity with it is useful. For example, using the Control Panel's Ports applet, you unfortunately can direct NT to create invalid serial ports. The WinObj tool lets you look in the \?? subdirectory of the Object Manager namespace for COM objects and determine which serial ports really exist. Even if you don't run into such problems, knowledge of NT object management can give you a better understanding of NT's architecture and the Win32 API.