Give Me a Handle, and I'll Show You an Object

Ruediger Asche
Microsoft Developer Network Technology Group

Created: July 15, 1993

Abstract
This article discusses objects and handles under Microsoft® Windows™ version 3.1 and Windows NT™. The terms objects and handles have different meanings in each operating system and imply a variety of relationships between objects and their corresponding handles. This article elaborates on the differences between the way components of Windows NT treat objects as opposed to the way Windows 3.1 treats them, with particular emphasis on shareability issues.

Introduction
You got a smile so bright, 
You know you could have been a candle. 
I'm holding you so tight, 
You know you could have been a handle.

Smokey Robinson, "The Way You Do the Things You Do"

One of the masterpieces of contemporary American literature, Webster's New World Dictionary (Paperback Edition), defines a handle as "that part of a tool, etc., by which it is held or lifted." Now, is this a precise definition or what? What on earth is etc.? By intuitive knowledge, we know that etc. can cover anything from toothbrushes to tennis rackets to cookware. While I leave it up to the imagination of the reader to figure out what Smokey Robinson wanted to hold or lift, I will tell you right away what we will be handling in this article: objects.

Object is one of the major buzzwords of today's computer reality. You find objects everywhere, from object-oriented languages to object linking and embedding. Microsoft® Windows™ has used the term object for a long time to describe several very different things. To further confuse the issue, things that are not explicitly labeled as objects behave the same way as some objects do.

Windows NT™ is essentially an object-centered operating system and has a very definite notion of what an object is. Unfortunately, Windows NT's idea of an object is not related at all to the concept that 16-bit Windows has of an object; as a result, when writing a Win32®-based application for Windows NT, you will need to deal with even more "object types," which will make the confusion about objects potentially even greater. This article untangles the different meanings of the term object and the related term handle, and makes you aware of potential incompatibilities between applications written for Windows version 3.1 and for Win32.

We display the entire object and handle layout of Windows NT in the Appendix at the end of this article. Please refer to it as necessary as you read this article.

One more note: For the sake of this discussion, we will entirely leave out C++ and Microsoft Foundation Class Library objects. Although some Foundation class objects are directly mapped to objects provided by the operating system, the way Foundation class objects are referenced, maintained, and protected is exclusively a compiler issue. In this article we will focus on objects that are provided and maintained on the operating system level.

This article is not meant to be a supplement to the documentation of Windows NT. All that is described herein applies to version 3.1 of Windows NT and may change in future versions of the operating system. Please be aware that applications written for the Win32 API are meant to be portable between all platforms that support this API, and any assumptions about a particular implementation (such as the relationships between handles and objects as described in this article) will break applications that operate under such assumptions.

Objects in 16-Bit Windows
The idea of an object that Windows 3.1 introduced was roughly "something that is accessed through a handle." In Windows 3.1, you create the object—a process that returns a magic cookie called a handle—and subsequently use the handle to access the object. This ensures that the actual implementation of the object can be changed without affecting the application.
If you have used objects and handles exactly like this, you are well off, and you will find that you will need to change almost nothing when porting your application from Windows 3.1 to 32-bit Windows NT. However, many developers for Windows 3.1 made assumptions about the relationship between handles and their objects—sometimes they had to because application programming interfaces (APIs) were missing for certain operations. Such assumptions will break applications under Windows NT.

Maybe you have used **GlobalHandle** to retrieve the handle of any given address and subsequently use the handle to reallocate the object? Works like a charm under 16-bit Windows. Maybe you have extracted the HIWORD of a given pointer and converted it into a handle, or peeked into memory without doing a **GlobalLock** using that handle? Maybe you were bold, retrieving the base of the local descriptor table and looking at descriptors by using the handle as an offset into the table? Beware, it’s these kinds of things that will leave you with major redesign work when you try to port your application.

Such things used to work under 16-bit Windows, mainly due to two widely known facts:

- First, objects under 16-bit Windows generally fall into two categories: objects that are stored in GDI’s and USER’s local heaps, and objects that are allocated from the global heap (that is, Windows internally calls **GlobalAlloc** to allocate memory for that object).
- Second, the handles that belong to global heap objects are, in fact, selectors to those global memory segments, which implies an immediate relationship between the handle and its memory location.

For example, there is no API under Windows 3.1 that returns the size of a resource, given its handle. No problem; because resources are allocated from the global heap, you can eventually use all of the global heap functions to party on the resources. You can call **GlobalPageLock** if, for whatever reason, you wish to permanently lock the resource; you can call **GlobalSize** to figure out its current size; you can call **GlobalRealloc** if you wish to dynamically change its size; and you can even **GlobalLock** it to write to or read from the resource data directly.

The entities that are explicitly labeled as objects under Windows 3.1 are GDI and USER objects. GDI objects are brushes, pens, bitmaps, fonts, palettes, and regions; USER objects are window classes, atoms, menus, and windows. Handles for those objects are, in fact, pointers into the default data segments of the GDI and USER modules, respectively. A relationship between the handles and the objects is implied in that the handles, when converted to pointers into USER’s and GDI’s default data segment, point to data structures that describe the objects internally.

Unfortunately, the API sets that treat those objects are fairly inconsistent. Although the **Createxxx** and corresponding **DeleteObject** APIs were originally designed to provide a uniform **Create/Use/Delete** sandwich for all GDI object types, not even palettes could be incorporated into the paradigm when they were introduced with Windows 3.0; **SelectObject** will not work on them. On the other hand, walking GDI’s local heap will reveal that palettes are stored in there the same way as brushes, pens, bitmaps, fonts, device contexts, and regions are; thus, although they are stored internally the same way, they have different operations that work on them.

One of the corollaries of the design of Windows 3.1 is that all objects are globally accessible. Because only one local descriptor table (LDT) is being used for all Windows-based applications, all global handles (that is, selectors) can be used by all applications and DLLs; and because all GDI and USER objects are references into the globally accessible GDI and USER default data segments, they can be seen and used by all applications as well.

This implementation detail is widely used by many applications, be it explicitly (for example, in order to share memory) or implicitly (for example, in order to enumerate windows or fonts). One of the unfortunate consequences of this design is that any application can (deliberately or inadvertently) modify or possibly destroy objects that belong to other applications. There is no protection mechanism against these kinds of practices. (“The Way We Used to Do the Things We Did.”)

**Object Management Under Windows NT**

Windows NT has a rather precise and orthogonal idea of what an object is. Windows NT has an API set that works on all kinds of objects; handles are maintained by the executive via a consistent mechanism that allows for secure access to objects, supports synchronization, and rejects attempts to delete objects that are still in use.

The objects that Windows NT knows about are processes, threads, files (including devices that look like files, such as communication devices, mail slots, or pipes), file mappings (or sections, as they are labeled on the executive level), events, semaphores, and mutexes. All of those object types have certain properties in common that we will elaborate on a bit later.

You can see that none of the things that Windows 3.1 calls objects (hereafter called "old objects") are covered by the term objects in Windows NT. The equivalents of Windows 3.1 "old objects" are not implemented on the executive level under Windows NT, so next we'll look a little bit more closely at how the Win32 subsystem implements "old objects."
other words, a window or a brush or a pen does not have anything to do with a Windows NT object, and the Windows NT executive would be totally clueless if it were passed a handle to one of those objects.

**How the Subsystem Realizes Windows GDI Objects**

Let us begin with GDI objects. Under Windows 3.1, GDI objects are stored in GDI's default data segment. They are designed to be global; that is, one application can create an object and pass it on to another application. There is no ownership of GDI objects in 16-bit Windows; thus, one application can destroy a GDI object that you created in another application.

Under Windows NT, GDI objects are stored in the client part of the Win32 subsystem's GDI module. That means that any GDI object is valid only in the context of the application that created it. However, consider this: Create an object and pass its handle to another application. Let this other application call `SelectObject` on that newly created object, and the call may succeed! However, nothing might happen; or instead of a brush, a pen might be selected into the device context! What is happening?

Well, what happens here is that handles to GDI objects are internally implemented as offsets into a handle table that resides on the client side of the Win32 subsystem. (Remember that the Win32 client is a DLL that resides in a Win32-based application's address space and is called by the application.) In other words, handle tables are kept on a per-process basis, but they are not process-tagged. That means that a handle to an object that belongs to process A might coincidentally look like a valid handle in process B's context. Thus, a call to `SelectObject` from B might succeed, but B will actually have selected a totally different object into its device context—or worse, the right one. Selecting the right object may be worse because the objects might coincidentally be the same, so you think it works, but the application will behave weirdly later on. So, do not pass handles to GDI objects between applications; they have totally different meanings in different processes.

Actually, the picture is even more complex than that. Because all applications share the same physical output device, maintaining a device context cannot be left up to the client alone. Any output call will eventually end up in the server, and thus, the server has to know about its clients' device contexts as well as their GDI objects. The subsystem divides the representation of the GDI object (including the device context) between the client and the server. For efficiency's sake, as much work as possible is done by the client—in most cases, for example, a call to `SelectObject` will not make it to the server; the client will simply update an internal data structure. As soon as the server does the drawing, it knows how to retrieve and realize the objects on the client's side.

The GDI part of the Win32 subsystem has a handle manager of its own through which all accesses to GDI objects go. This is also true for USER, as we will see later. A handle to a GDI object is, in fact, made up of two components: One part encodes the offset into an object table that resides in the client part of the subsystem, and one part is a uniqueness identifier. This identifier changes—as the name implies—whenever a handle is recycled and is stored both as part of the object and as part of the handle.

Validating an object involves, among other things, matching the unique part of the handle against the value in the object. If a handle is recycled, the new handle returned is different from the one that belonged to the destroyed object with the same table index. This mechanism is used to make sure that stray pointers or previously deleted object handles that are passed into a function that works on an object do not reference wrong objects. If the validation fails, an error 6, INVALID_HANDLE, is generated.

In future versions of Windows NT, it may well be that this implementation detail changes. Although a handle will most likely still be composed of two 16-bit components for compatibility reasons, the interpretation of the "uniqueness" part may change due to performance considerations.

You might wonder now why you don't see the INVALID_HANDLE error when you pass a GDI object handle between applications in error. The reason is that initially the "uniqueness" value is the same for all handles in all address spaces; objects become unique only after the handle is recycled. Thus, two valid objects in different processes may very well have the same "uniqueness" value. "The Way We Do the Things We Do?" You betcha.

The downside of this implementation is that it cuts down on the number of available handles. A handle has the size of a DWORD—that is, it's 32 bits. Thus, a maximum of 232 or 4 GB of objects can be theoretically addressed by a handle. Currently, the two components of the handle—the table index and the uniqueness identifier—take up 16 bits of the handle each, such that theoretically a maximum of 64K (216) of GDI objects could be accessed per process, but the maximum size of the client's handle table is currently hardcoded to 16K of entries.

**And Now for Something Completely Different: USER Objects**

The story of USER objects is different from GDI's story. Under Windows 3.1, the definition of *object* covers both objects internal to USER's default data segment (windows, window classes, menus, and so on) and resources that are actually...
allocated from the global heap. Windows NT still maintains both object types in the USER module, but because there is
no global heap anymore as there was in Windows 3.1, resources behave a little bit differently.

One of the main goals in the implementation of the USER part of Windows NT's Win32 subsystem was compatibility.
Under Windows 3.1, an application has a high degree of control over USER-maintained objects. For example, through the
EnumModules function, an application for Windows 3.1 can get access to all top-level windows in the system; with the
window handle, the menu of an application can be modified, messages can be monitored, device contexts obtained, and
so on. In order for Windows NT to maintain this degree of control, USER objects are maintained on the server side of the
Win32 subsystem, and the functions that access the objects (such as LoadIcon) need to do more work. For example, if
the object were located on the client side, accessing it would mainly be a task of translating the handle into a pointer,
but in the USER implementation, memory in the server process must be referenced.

Just like its GDI counterpart, the Windows NT USER component keeps a handle manager that works pretty much like the
one described in the GDI discussion, but the object tables are maintained on the server side of the subsystem, and
therefore the object limit is global. Currently, it is 64K entries.

Resources
The good news is that under Windows NT, resources are not handled differently from the other USER objects at all.
Under Windows 3.x, the local heaps of USER and GDI turned out to be severe system bottlenecks for object-intensive
applications. If you have worked with 16-bit Windows intensively, chances are that at more than one point you have
seen the free system resources decreasing severely when you worked with large menus or many windows. Windows 3.1
somewhat widened that bottleneck by making menus resources instead of objects within USER's default data segment.
Windows NT implements all USER objects orthogonally.

From the user's point of view, the major differences between resources and other objects is that at times resources need
to be accessed directly, without using the API functions that operate on them. (This holds especially true for user-defined
"raw" resources.)

There are actually two different ways to access resources under Windows NT—one in which the resource resides in the
server process and one in which it resides in the client. To exemplify the first case, let's assume your application calls
LoadIcon and passes the returned handle on to application B, which then calls DrawIcon on that handle. Guess what?
Works fine. Just like under Windows 3.1. This seems to be strange at first glance because the two applications have
disjoint address spaces, so how can the target application see something that the source application loaded?

Note that you cannot do much more with that handle. In particular, you cannot get your hands on the memory that
describes the icon because the handle has no meaning whatsoever to either your application or application B. It is a
handle local to the server part of the Win32 subsystem. DrawIcon eventually executes in the server part as well, and
thus it knows how to interpret the handle. In this case, the icon is a real object in the strict sense of the word: The
handle is a true magic cookie without any meaning to anyone except for the operations that are allowed on it (such as
DrawIcon).

As a side effect of this modular approach, using this technique to access resources is fairly limited. You cannot modify
the resource or look at its memory. (We will look at ways to work around this in a second.) Also, there is no way for the
subsystem to determine if the object is in use by anybody but the process that owns it. In our example, the subsystem
cannot know if somebody is using the icon, so as soon as your application terminates, the icon is gone from the server
subsystem address space, and a subsequent attempt to call DrawIcon on it will return with an error code of
ERROR_INVALID_CURSOR_HANDLE. Do not be confused about "CURSOR" in that error message—as far as USER32 is
concerned, icons and cursors are basically the same; internally, they are represented as the same data structures.

So how does one share resources among processes? Well, the bad news is that there is no easy way for applications to
share resources dynamically at run time so that a change in the resource that one application makes appears in the
other application. Aside from using the CopyIcon, CopyAcceleratorTable, and CopyCursor functions, which are
especially designed to create local copies of resources, the best one can do about this is to have one application map an
image of the other application's executable into its own memory space, locate the resource in the image, and party on it.
To aid this process, the LoadResource API family has been modified for Windows NT.

If application HAND wants to access a resource in application FOOT, it needs to use the LoadLibrary, FindResource,
and LoadResource APIs as described in the Win32 API Help file under the documentation for FindResource. If you
want to permanently change resources in the target executable file, you should use the UpdateResource,
BeginUpdateResource, and EndUpdateResource APIs.

The trick here is that LoadLibrary will map the executable file into the address space of the process that calls it and
return a pointer to that image. All that FindResource and LoadResource do is examine the executable header in that
image, figure out where within the image the resource is located, and return a pointer to the resource image.

The part that will probably confuse you here is that **LoadLibrary** also works on .EXE files. Under Windows 3.1, if you load an executable file, it will execute right away, whereas a DLL will linger in memory until it is called. **LoadLibrary**, **LoadModule**, and **WinExec** are very closely interrelated under Windows 3.1, but they are totally different under Windows NT, where **LoadLibrary** basically maps the executable file into the process's address space. **LoadLibrary** also executes the DLL entry point routine when a library is loaded, but it does not execute anything when an application is loaded. The return value from **WinExec**, which is a stripped-down version of **CreateProcess**, is a handle to a Windows NT executable object, whereas an instance handle, the value returned from **LoadLibrary**, is a virtual pointer.

**Win32 Kernel Objects**

The preceding discussion on virtual pointers leads us right into the remaining category of objects in the Win32 subsystem—kernel objects. Now, here we run into a terminology problem: Under Windows 3.1, the component that is responsible for memory management and system services is called the "kernel." Unfortunately, the same term is used for the component in Windows NT that executes in privileged mode at the lowest level. Windows NT renames the part of the Win32 subsystem that corresponds to the "base". Thus, this section does not really deal with kernel objects, but with base objects. Most of the objects that you access through the base are, in fact, native Windows NT objects, which we will discuss in the following section.

The only entity that the kernel component of Windows 3.1 dubs an object is a memory object from which a number of other object types derive, such as modules or instances; in the rest of this section, we will look at these in more detail and see how those are implemented in Windows NT.

Let us elaborate on the last statement of the preceding section a little bit more. Windows 3.1 knows instance handles, task handles, and module handles. Instance handles are, in fact, selectors of the default data segments of the application instance. (This fact is sometimes used to change the data segment so that an operation can work on the default data segment of a different application—an ugly practice, but it works.) The task handle is the selector of a global memory block that contains task-specific parameters (such as the input queue that is associated with the task). The module handle is the selector of a global memory block that contains a modified version of the executable header associated with the module. (This memory block is also known as the module database.)

For applications in Windows 3.1, there is only one module database in the system for all instances of the application, whereas both the task database and the default data segment exist separately for each instance of the application. The **GetModuleHandle** API returns the handle (selector) of the module database, whereas a call to **GetWindowWord** with **GW_HINSTANCE** passed as the second parameter returns the instance handle, and the **GetCurrentTask** API returns the task database handle. For other applications, the **TaskFirst/TaskNext** Toolhelp API functions can be employed to retrieve the task database handles.

For more information on modules, tasks, and instances under Windows 3.1, please consult the Knowledge Base articles Q76676, "Differences Between Task Handles and Instance Handles," and Q78327, "HANDLes Returned by GetModuleHandle and LoadLibrary," as well as Bob Gunderson's technical article "Modules, Instances, and Tasks." (Knowledge Base, MSDN™ Library)

Windows NT does not use task databases or module databases anymore, and the default data segment is built on top of an application's virtual address space. The **hInstance** value that is being passed to an application upon startup is basically the same as the value returned from **LoadLibrary**: a virtual address in the application's address space that contains the memory-mapped image of the executable file. Incidentally, **GetModuleHandle** (null) returns the same value. It is fairly easy to confuse this value with a process handle, which is something totally different (namely, a native Windows NT object) and cannot be used interchangeably with those virtual pointers.

The remaining kernel object type is the **heap object type**, which we'll look at now. As Randy Kath's technical article "Managing Heap Memory in Win32" outlines, the Windows NT heap memory manager resides on top of the virtual memory manager; that is, a heap is essentially a chunk of virtual memory that consists of the heap itself and some administrative information that is used to maintain it. The handle returned from the **HeapCreate** API is essentially the virtual address of this memory chunk, and a call to **HeapAlloc** will look into the administration header and return a pointer to the memory chunk. You should never attempt to write to a memory location that belongs to the heap header because it would seriously confuse the heap manager. Also, due to the very nature of disjoint address spaces, one application's local heap is of no relevance to other applications.

Because heap handles are virtual pointers, and no record is kept of the local heaps allocated by the base part of the Win32 subsystem, there is really no limit on the number of heaps you can theoretically allocate, except for the limits that the executive imposes on the number of virtual memory chunks that an application can use. For details on virtual memory management, please refer to Randy Kath's technical articles "The Virtual-Memory Manager in Windows NT" and
"Managing Virtual Memory in Win32." It is worthwhile mentioning here, however, that you cannot allocate more than 32K of virtual memory blocks in any process because the virtual address space of a process is limited to 2 GB, and the minimum size for each allocation is 64K; thus, the maximum number of allocations that can be made is 2G/64K, or 32K of virtual memory blocks.

Note that this is different from the number of allocations that can be drawn from heaps. Because heaps themselves are virtual memory blocks, you cannot create more than 32K-worth of heaps per application, but from a given heap, you can allocate as many memory blocks as you want to.

Native Windows NT Objects

We will now look at native objects in Windows NT. As Helen Custer describes in depth in Inside Windows NT (Microsoft Press, 1993), the object manager is an integral and central part of Windows NT. It is responsible for accepting and processing requests to create, access, and destroy objects—both from a process and from the executive. Some of the objects that Windows NT knows about are not directly visible to applications, such as driver objects or symbolic link objects, but internally they work in exactly the same way as the aforementioned objects. In Inside Windows NT, these invisible objects are called kernel objects, as opposed to executive objects.

We have already enumerated the native Windows NT objects that are visible to Win32 applications: processes, threads, files, file mappings, events, semaphores, and mutexes. We will not elaborate too much on these here because Chapter Three of Inside Windows NT already gives a comprehensive overview of these objects and how to use them. The worthwhile aspects to mention about them are the following:

- They can attain a signaled state so that you can call WaitForSingleObject or WaitForMultipleObjects on them.
- You can use DuplicateHandle to access the object from inside other processes. (Some restrictions apply! Refer to the Win32 API Help file for details.)
- The Windows NT executive keeps a reference count for each object and removes the object only after the last handle is closed.
- Access to the objects is monitored by the security subsystem. (I will cover "security and how you can make use of it" in a future article.)
- Most objects can be optionally named so that other processes can access them by name if security allows.

Handles to executive objects are always associated with a particular process—this statement does not always hold true for kernel objects; for example, objects that describe the drivers for bootable hard drives must exist before the system has started up. The association of handles to processes is the reason you cannot use an existing handle to access an object from another process; you need to either obtain another handle using DuplicateHandle to create a handle table entry for the same object in another process or open an object by name in the other process.

Unlike the handles that are maintained by the Win32 USER and GDI subsystem components, handles to native objects under Windows NT are not unique; that is, upon destruction of an object, the corresponding handle may be recycled and will look exactly like the handle to the destroyed object. There is one unintuitive consequence of this implementation: Process IDs under Windows NT are actually handles into a dummy system table—that is, a table that does not associate its handles with any objects. If you create a process and store its ID away, then after termination of the process, the ID may be recycled for other types of global objects. You therefore cannot use process IDs to uniquely identify processes.

Finally, there is no hardcoded limit on the number of entries a handle table can have. A handle table maintained by the executive may grow dynamically if necessary; thus, the number of handles that can be allocated per process depends only on the memory available on your machine. (Note that Windows NT may further limit this number on a per-user basis by reducing the user's quota on non pageable memory.)

Conclusion

We have discussed several types of objects and their handles under Windows NT, and how they relate to corresponding object types under Windows 3.1. The most important result from this discussion is that any assumption about the implementation of objects and the meaning of handles is very likely to break code compatibility between platforms and should, therefore, be avoided. Another important point to mention is that the shareability of an object between processes is a matter of the implementation of the operating system and varies from object type to object type and from operating system to operating system. The table below lists the object types discussed in this article, as well as their locations and their shareability.

From the point of view of implementation, a handle under Windows NT is always realized in one of two categories:

- An offset into a table that is being maintained by either the executive or a subsystem component.
- A virtual address in the address space of a process.

The concept that makes this implementation possible is the memory-mapped file (or section object). Whereas
Windows 3.1 needs to copy data from an executable file into main memory to build code segments or load resources, Windows NT only needs to create a memory-mapped file object that is backed by the executable file instead of the system pagefile. Thus, a resource can be retrieved by calculating a pointer into the memory-mapped file object that represents the executable. The USER part of the Win32 subsystem uses this technique to implement both methods of resource access: A call to LoadIcon, for example, will map the executable into the server's address space so that a subsequent call to DrawIcon will retrieve the icon data from the memory-mapped image, whereas LoadLibrary maps the executable into the client's address space so that FindResource and LoadResource will retrieve the data from there (which is why resources retrieved this way will not have any meaning to other processes).

Also, it is important to emphasize the disjoint API sets that are available for the different types of objects. You cannot call DeleteObject on a resource or an executive object; conversely, CloseHandle will fail on GDI objects and resources.

A good way to look at it is to classify each type of object as a member of something one could call an "object type class," a category of objects that are roughly characterized by being maintained by the same component of Windows NT.

Following this approach, there are the following object type classes in Windows NT:
- Executive objects (processes, threads, sections, file objects, events, semaphores, and mutexes)
- Win32 GDI objects (pens, brushes, fonts, palettes, regions, device contexts, bitmap headers)
- Win32 USER objects
  - WIN32 resources (accelerator tables, bitmap resources, dialog box templates, font resources, menu resources, raw data resources, string table entries, message table entries, cursors/icons)
  - Other USER objects (windows, menus)
- Win32 base objects (heaps)

**Summary of Object Properties**

<table>
<thead>
<tr>
<th>Shareable</th>
<th>Executive objects</th>
<th>Win32 base objects</th>
<th>Win32 GDI objects</th>
<th>Win32 USER objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shareable</td>
<td>Via APIs (DuplicateHandle or Openxxx)</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Validated</td>
<td>Yes</td>
<td>No</td>
<td>Yes, by uniqueness</td>
<td>Yes, by uniqueness</td>
</tr>
<tr>
<td>Limit</td>
<td>Only by physical memory</td>
<td>32K of heaps</td>
<td>16K per process</td>
<td>64K systemwide (including resources and internal types)</td>
</tr>
</tbody>
</table>

**Note** Shareable indicates whether an object of that type can be accessed by several processes. Validated indicates whether the handle protects the object against invalid access to it, possibly due to an invalid handle.

**Appendix**

In this diagram, three processes are running in a particular Windows NT session—the Win32 subsystem server process and two clients (that is, Win32-based applications). Note that by design of Windows NT, all processes' virtual addresses over 2 GB are mapped globally (that is, are mapped to the same physical addresses in all processes).
An instance handle is a pointer to the image of an executable file in a client process.

A resource handle as obtained by `FindResource` and `LoadResource` is a pointer to that process within the image.

A handle returned from `VirtualAlloc` or `HeapCreate` is a pointer to the beginning of the memory block in the client's address space.

A handle returned from `HeapAlloc` is a pointer into the chunk of memory allocated by 3.

A GDI handle is a relative offset into a table located in the client's address space.

A USER handle is a relative offset into a table located in the server's address space.

A handle to a native Windows NT object is a relative offset into a table located in system space. There are several of those tables—one per process and a few tables maintained by the system.

A USER object itself is located in the server's address space.

In the case of resources, 8 still holds true, but just as in 2, the resource is referenced through a memory-mapped image of the file that holds the executable, only the image resides in the server's address space this time. This is the scenario you encounter, for example, when calling `LoadIcon`.

The data structures that describe native Windows NT objects reside in system address space. Depending on the object type, part of the object may also be located in a process's address space. (This holds true, for example, for section objects.)

The data that describes a GDI object resides in the client's address space. Please also observe the restriction mentioned earlier in this article under "How the Subsystem Realizes Window GDI Objects."