Microsoft Windows™ 32-Bit API

Rex Jaeschke

Windows 32-Bit API

Edition: 3.0

The training materials associated with this book are available for license. Interested parties should contact the author at the address below.

Please address comments, corrections, and questions to the author:

Rex Jaeschke
2051 Swans Neck Way
Reston, VA 20191-4023
+1 (703) 860-0091
www.RexJaeschke.com
rex@RexJaeschke.com

Preface

This book presents information about various components of Microsoft's 32-bit API. All of these components are at work “behind the scenes.'' None of them involves the GUI interface; there are already plenty of sources of information for Windows programming in general. What this book is concerned with is what to do in between the time you get input into your program and when you output the results. This book covers the bread and butter of getting the real work done.

The initial research and testing for this book was done using Windows NT (hereafter referred to as WinNT). However, much of it also applies directly to Windows 95 (hereafter referred to as Win95) and all examples have been tested under that environment as well. Except where noted, the behavior described is identical for both systems. (For the purposes of this book, Windows XP and Vista are modern versions of WinNT.)

While a number of vendors sell C/C++ compilers, linkers, and other related development tools for the Windows 32-bit API environment, Microsoft's Visual C++ is perceived, at least by this author, to be the implementation of choice. Therefore, examples and narrative have been written assuming use of that compiler and its associated linker. The reader will need to extrapolate for other development systems.

Reader Assumptions

Most examples are written in C, so a good working knowledge of that language is assumed. Except where stated otherwise, references to Standard C also imply Standard C++.

Because it is not at all difficult to do so, most examples have been internationalized. Specifically, they can be compiled to support the following character modes: single-byte, double-byte, and wide character. However, all text inside string literals and character constants is written in English. Making the code completely independent of cultural environment would require placing such strings in string resources. Since the primary audience for this book is currently English-speaking I decided to not add this extra level of abstraction. If you are not familiar with Microsoft's approach to producing internationalized code and you want to know more about it, refer to §1. If you don't care to know and just want to be able to follow the examples, follow these simple guidelines: First, ignore each occurrence of the function-like macro `_T`. For example, read `_T("Hello")` as if it said `"Hello"`, and read `_T('Y')` as if it said `'Y'`. Second, names such as `_stprintf` and `_tcscpy` are simply masking macros that are replaced by the underlying library function names `sprintf` and `strcpy`, respectively, in single-byte mode. Annex A shows how to translate these names.

The reader is assumed to be comfortable with the type qualifiers `const` and `volatile`, function prototypes, and type synonyms.

Presentation Style

The approach used in this book is different from that used in many other books and training courses. Having developed and delivered programming-language training courses for more than 15 years, I have found that the best approach for my students is an incremental one. I avoid introducing things that are unnecessary at any given

time, thus making examples small, simple, and well focused. Specifically, I introduce the basic elements and constructs of the language using procedural programming examples.

I prefer the student to have an excellent chance of understanding and absorbing small amounts of new material, and reinforcing it with lab sessions, as they progress. The intent here is to eliminate any chance of their being overwhelmed, provided, of course, they meet the prerequisites.

Different styles work for different teachers and different students. I do not suggest that my approach is better than is any other; I simply know that my approach works well, and has formed the basis of my successful seminar business for more than a decade.

Exercises and Solutions

The programs shown in the text are available electronically in a directory tree named Source, where each chapter has its own subdirectory.

Some chapters contains exercises, some of which have the character ***** following their number. For each exercise so marked, a solution is provided in a directory tree named Labs, in which each chapter has its own subdirectory.[1] Exercises that are not so marked have no general solution and require experimentation or research in an implementation's documentation. Exercises having solutions contain a statement of the form "(See lab directory *xx*.)", which indicates the corresponding solution or test file in the Labs subdirectory.

You are strongly encouraged to solve all exercises in one section before continuing to the next. Also, invent your own exercises as you go and be inquisitive; don't be afraid to experiment. Try to understand why the compiler gives you each error or why a program fails at run time.

Rex Jaeschke, November 2009

[1] The solutions are only available to licensees of these materials when they are used in formal training scenarios.

1. Memory Layout

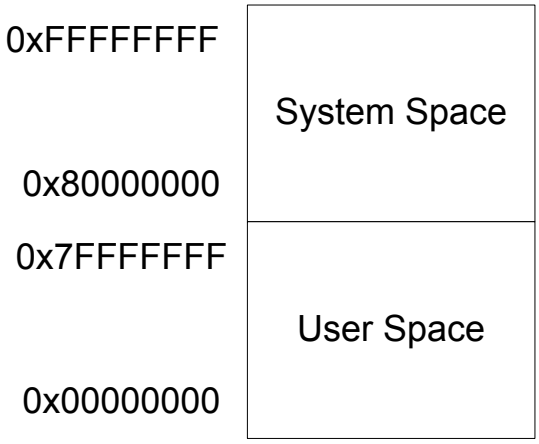
In this chapter, we'll learn how memory is laid out and how a compiler and linker organize things in memory, just what locations we can access, which are off limits, and why.

1.1 Address Space

Win95 and WinNT use 32-bit addressing,[1] resulting in a linear address space range of 0x0–0xFFFFFFFF; that's 4 gigabytes! Virtual addresses are mapped to physical addresses by the operating system allowing a process to be relocated in memory at runtime.[2]

Memory is organized in units called *pages*, the size of which can vary from one platform to another. For example, on Intel machines each page is 4KB while on Alpha-based systems each page is 8KB. Therefore, we should not hard-code the page size in our programs; instead, we should obtain it by calling the API function `GetSystemInfo`.

In WinNT, the address space is broken up into two equal-size parts: *system space* and *user space*, as follows:



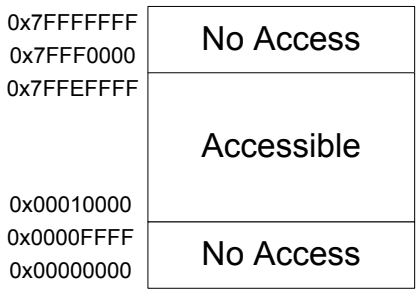
User-mode code cannot read from or write to memory located in system space. Therefore, any attempt to access memory addresses in the range 0x80000000–0xFFFFFFFF results in an *access violation*; that is, an attempt to access memory we do not own, or for which we have no access permission. (§1.2 covers access violations in detail.) As a result, if we dereference an uninitialized automatic pointer, we have a 50 percent chance of getting an access violation. And that's good, since most access violations show up very quickly.

As it happens, user-mode code cannot actually access all 2GB of the user space. Specifically, a block of low and a block of high addresses in that space are inaccessible, as shown below.

---

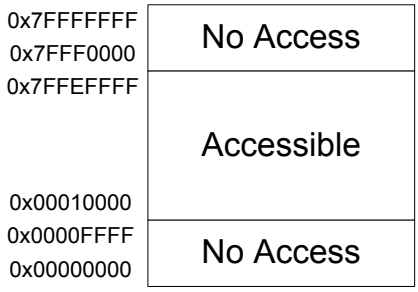
[1] This is true even when WinNT is running on 64-bit machines.

[2] For detailed information on how this mapping works, see Helen Custer's book *Inside Windows NT* from Microsoft Press, 1993 (and its subsequent revisions).



User-mode code cannot read or write to memory located in these two 64KB areas. Therefore, any attempt to access memory addresses in the ranges 0x0–0x0000FFFF and 0x7FFF0000–0x7FFFFFFF results in an access violation. Furthermore, user-mode code can only access user space allocated to its parent program. And since most programs are considerably smaller than 2GB - (2 * 64KB), most user space is inaccessible to user-mode code.

In Win95, the address space is broken up into five parts, as follows:



The parts marked “Don't Touch”' are accessible, but we should not access addresses located there. The part marked “Accessible'' is private to each process while that marked “Accessible (shared)” is shared by all 32-bit processes.

The API function `GetSystemInfo` provides useful information about an address space. The following information was produced from program sysinfo.c (which calls this function):

Property	Win95/Intel	WinNT/Intel	WinNT/Alpha
Address Space	00000000–FFFFFFFF	00000000–FFFFFFFF	00000000–FFFFFFFF
Page Size	4096 bytes	4096 bytes	8192 bytes
Number of Pages	1048576	1048576	524288
Allocation Granularity	65536 bytes	65536 bytes	65536 bytes
Min Application Address	00400000	00010000	00010000

Property	Win95/Intel	WinNT/Intel	WinNT/Alpha
Max Application Address	7FFFFFFF	7FFEFFFF	7FFEFFFF
OEM ID	0	0	2
Active Processor Mask	00000001	00000001	00000001
Number of Processors	1	1	1
Processor Type	Intel 486	Intel Pentium	Alpha 21064

While most of this information comes directly from the SYSTEM_INFO structure, some of it is computed. The Minimum and Maximum Application Addresses correspond to those mentioned earlier.[1]

Each virtual page of memory has a set of attributes assigned to it, depending on how that page is to be used. The attributes are, as follows: Read-only, Read/write, Execute only, Guard page, No-access, and Copy-on-write. Note that while we can jump to an address in an execute-only page and execute instructions contained therein, we cannot actually read that page.[2]

Program testrd.c probes all 2GB of its user space. It reports on contiguous blocks of memory that are readable and non-readable. Its output looks like the following:

```
  Address Range      Length     Attribute
=========================================
00000000-0000FFFF  (00010000)  Not Readable
00010000-00010FFF  (00001000)      Readable
...
7FFDE000-7FFE0FFF  (00003000)      Readable
7FFE1000-7FFFFFFF  (0001F000)  Not Readable
=========================================
Total memory readable:     00176000
Total memory not readable: 7FE8A000
Total memory:              80000000
```

[1] Some of the same information can be obtained from the WinNT Diagnostics program, option Hardware, in the Administrative Tools group.

[2] On systems that don't support execute-only pages (such as those based on processors from MIPS and Intel), these pages are treated as read-only.

7. Synchronization

In this chapter, we will see an overview of the various ways in which we can synchronize the execution of threads. We will also look at the API functions used to achieve synchronization.

7.1 Introduction

Historically, almost all applications ran in serial mode, one part after the other. As a result, all operations were inherently synchronized and the interaction of the parts was simple and well defined. Because only one operation was in progress at any time, there was no contention for shared resources; indeed, there were no shared resources.

Newer operating systems give applications programmers the ability to share resources—such as memory and devices—between separately executing programs. In recent years, such parallelism has been extended to a single program by the addition of a multithreading capability. As we learned in §1, a thread is a separate stream of execution. In our serial program model, the program has one thread; we say it is single-threaded.

There are a number of ways to make a program multithreaded. The obvious approach is to use some language or library facility to start up threads explicitly. The less obvious approach is via signal handlers. When a signal (caused by a divide-by-zero, for example) occurs, the current thread of execution is blocked and control transfers to a new thread, which handles the signal. Upon completion, the signal handler thread terminates and control returns to the interrupted thread.

Whatever the method used to produce concurrently executing threads, if multiple threads need access to the same resource, some form of synchronization is needed.

Clearly, it is important not to keep a resource locked any longer than necessary. Consider the case in which a related set of measurements is taken and the results are used in a lengthy series of calculations. Rather than locking up the table of values during these calculations, the locking thread can copy the table to a private array, release the lock, and then process the private copy. Now, if a new set of measurements arrives, it can be stored immediately the lock is released. That is, we take a synchronized snapshot and work on the copy. However, depending on our application, this approach may be good or bad. Consider the case in which new and better data arrives yet we are off processing the previous set. In this case it might be better to abandon that processing and start on the new set. However, if new measurements arrive at a rapid pace, we might spend considerable time starting and aborting processing without getting any computations completed.

Synchronization between threads is needed when either of the following situations occur: a resource can have only one user at a time (for example, a communications port), or where multiple users have access to a shared resource and one or more of them wants write access to non-atomic shared objects (for example, one file writer with multiple readers of that file).

7.2 Synchronization Methods

The 32-bit API supports a number of different ways of synchronizing multiple threads within a process and/or between processes running on the same machine. These methods involve the use of the following kinds of objects:

- Critical Section —This object is used to guard a piece of code that can be executed only by one thread at a time. Consider the case of multiple threads each wanting to update a non-atomic data structure in shared memory; only one thread at a time should have write access. (Note, however, a critical section does not prevent some other piece of code from accessing that shared structure for read, and that could result in a problem.) A critical section is owned by one thread at a time. Threads synchronizing via a critical section must be in the same process. For more information on critical sections, see §1.
- Event —Any thread with access to an event object can set that object so that all threads waiting on that object to be set, can resume. One use for an event is in the case in which a processing thread only processes the data in some buffer when new data has been stored. At all other times, it is waiting for that event to complete. An event is not owned by any thread, per se. Threads synchronizing via an event can be in the same process or in different processes on the same system. For more information on events, see §1.
- Mutex —This kind of object can be owned by only one thread at a time. It provides a way for threads to have mutually exclusive access to some shared resource. One use for a mutex is to control access to memory shared between multiple processes via a file map. Threads synchronizing via a mutex can be in the same process or in different processes on the same system. For more information on mutexes, see §1.
- Semaphore —A semaphore object is a more general form of a mutex in that it allows no more than a given number of threads to be accessing some shared resource simultaneously. We use a semaphore to control a shared resource that can only (or efficiently) support a limited number of users. A semaphore is not owned by any thread per se; a given thread either is or is not currently one of the set of threads having access to the resource being controlled by the semaphore. Threads synchronizing via a semaphore can be in the same process or in different processes on the same system. For more information on semaphores, see §1.

7.3 The Wait Functions

Except for critical sections, each synchronization object has a handle. A thread that wishes to make use of a resource protected by some synchronization object specifies that handle when it calls the appropriate wait function. If a thread needs access to any one or to all of a set of resources each protected by different synchronization objects, it must specify the handle of each object.

Once a synchronization object has been created, it has one of two possible states: *signaled* or *non-signaled*. When a synchronization object becomes signaled, any thread waiting on it can continue execution. The non-signaled state indicates that the synchronization object is not currently available, causing the requesting thread to be blocked—that is, put into an efficient wait state.

When a wait function is called, the thread does not continue execution until that function returns. And it does not return unless the conditions specified in its call have been met. Typically, the caller asks to continue when the objects designated by one or more handles become signaled. However, the caller can also ask for the wait to

"time out" after a given time period has elapsed. An INFINITE time period results in the thread waiting forever, if necessary.

The simplest and most commonly used wait function is WaitForSingleObject, which requires that only one handle be passed:

```
DWORD WaitForSingleObject(HANDLE hHandle, DWORD dwMilliseconds);
```

dwMilliseconds represents the time-out interval. If the wait request succeeds, a number of possible values can be returned. They are:

- WAIT_OBJECT_0 – The object designated by the handle is signaled.
- WAIT_TIMEOUT – Indicates that the time-out interval elapsed before the object designated by the handle became signaled.
- WAIT_ABANDONED – Indicates that the object designated by the handle was abandoned. A mutex is owned by one thread at a time. If the thread owning a mutex terminates before releasing that thread, the mutex is marked as being *abandoned*. Then, the next thread waiting to own that mutex, gets ownership and this macro value is returned.

If the wait request fails, WAIT_FAILED is returned. Here's an example of using WaitForSingleObject:

```
HANDLE handle;
DWORD value;

if ((value = WaitForSingleObject(handle, INFINITE)) == WAIT_FAILED)
{
    ProcError(_T("WaitForSingleObject failed"), API_Error);
}
```

Although we are concerned primarily here about waiting for events, mutexes, and semapahores, we can also wait for various other kinds of objects to become signaled. These include detection of the following: certain changes to a directory, arrival of console input, and termination of a thread or process.

To wait for any one or all of a number of objects to become signaled, we use WaitForMultipleObjects:

```
DWORD WaitForMultipleObjects(DWORD nCount, CONST HANDLE *lpHandles,
    BOOL bWaitAll, DWORD dwMilliseconds);
```

The first argument, nCount, specifies the number of handles in the array specified by the second argument, lpHandles. The array can contain no more than MAXIMUM_WAIT_OBJECTS handles. It is most important to note that the handles in the array need not all refer to objects of the same type. For example, we could wait on all (or any one) of a set containing handles to an event, a mutex, and a semaphore. If we wish to wait for all the specified objects, we make bWaitAll be TRUE. If we wish to wait for any one the specified objects, we make it be FALSE.

13. File Mapping

A file map can be used for two distinct purposes: to provide an inter-process communication facility backed up by the system-paging file and, to map a named file into the address space of a process, allowing very efficient I/O. (Of course, an application can use a combination of these approaches.)

13.1 Communicating Via Shared Memory

In this section, we will see how an almost arbitrarily large block of memory can be shared between separate programs, allowing them to have access to the same data. And with the proper synchronization, these programs can read from and/or write to that shared data, simultaneously.

13.1.1 Introduction

We can share memory between threads in different processes running on the same system by using what is called a *file map*. This technique is very efficient when a large amount of data is to be communicated from one program to another because the data is instantly available to all programs that can map to this memory; no data need actually be sent between the programs.

The memory being shared is always backed up by a disk file. As the shared memory is modified, so too is this file, because of the operating system's memory paging operations. This is most useful in certain applications. Consider the case where one program acquires data and stores it in shared memory where it can be accessed by other programs in that application. At any time, the disk version reflects the state of the shared memory. Therefore, if the application or system crashes for any reason, the state of the in-memory database can automatically be restored when the application is restarted.

In virtual operating systems such as Win95 and WinNT, each program is allocated a series of virtual address by the linker. When the program is run, pages of memory are allocated to it. Ordinarily, the pages allocated to one program are not accessible to another. However, for shared memory, the same pages are allocated to multiple programs. Therefore, a given physical page of memory might start at one virtual address in one program but have an entirely different starting address in another.

The disk file used to backup the shared memory can either be named or unnamed. If a named file is desired, the user must create it in the desired directory with the appropriate access attributes. The file might be initialized before any application maps memory to it (as in program fmapfcr.c) or it might be initialized by the application, as the programmer desires. Being a named file, it lives at the pleasure of its owner. A named file allows information to be saved and restored across different executions of the mapping application. If we do not need a permanent record of the mapped memory, we can request that the operating system swapping file be used instead. (If we do this, the swap file must be large enough to contain all unnamed file-mapped memory for all applications running at the same time.)

When a named file is used, its contents are loaded into shared memory when its parent application begins execution. From that point on, the application's programs see the file's contents as an array of bytes in memory, making that file much simpler to deal with.

A program can use a *file view* to map into as many as 2^{32} bytes of a file. However, when using 32-bit addressing, that takes up the whole of the program's address space so we really can't view that much. If we actually have files larger than that, we can use multiple file views, each mapping up to 2^{32} bytes. In this manner, we can map into a file containing 2^{64} bytes, using 2^{32} file views each mapping 2^{32} bytes.[1]

For files of any size, it is possible to specify a file's starting offset and view size, allowing views to be smaller than 2^{32} bytes. This is useful when files are very large and, either little memory is available to map into that file or, the programmer does not wish to make much memory available, thereby keeping the size of the program down. Unless we specify otherwise, the whole file will be mapped provided it will fit. Note, however, that we cannot simply view a file of any size starting at any offset; the starting offset must be a multiple of the system's memory allocation granularity.[2]

When a named file is used, it should be opened for exclusive use. (If it is already open for use by another program, the message ERROR_SHARING_VIOLATION results.) For example, program fmap2a.c contains the following:

```
hFile = CreateFile(_T("filemap.dat"), GENERIC_READ | GENERIC_WRITE,
    0, NULL, OPEN_EXISTING, 0, NULL);
```

The file filemap.dat is an existing file (created by fmapfcr.c) that should be opened for both read and write operations. The third argument is a 0, indicating that the file cannot be shared while it is open. Because this file will be updated by the operating system's paging machinery, we can never synchronize read or write access with other applications wishing to access this file at the same time. If we were opening this file for read-only mode, then it may make sense to open it for shared read. However, it never makes sense to open it for shared write. In this example, if the file does not exist, the call returns ERROR_FILE_NOT_FOUND.

13.1.2 File Maps

Before memory can be shared, we must create a map object to define and control the sharing. This is done by calling CreateFileMapping. For example:

```
hMap = CreateFileMapping(hFile, NULL, PAGE_READWRITE, 0, 0, _T("FileMap"));
```

For named map files, the first argument specifies the file handle returned from CreateFile. For unnamed map files, the first argument must be (HANDLE) 0xFFFFFFFF. Both read and write access have been requested and the map object has been called FileMap. Arguments 4 and 5 are given the default value of zero, indicating that the whole file should be mapped. These arguments must specify a size if the system-paging file is being used otherwise ERROR_INVALID_PARAMETER results.

When a file map is created using CreateFileMapping, it is given a name,[3] access permission, and a maximum size. Other programs that wish to access this map should do so by using OpenFileMapping. (It is possible to have them use CreateFileMapping instead, however.) OpenFileMapping cannot override the map object attributes established by CreateFileMapping when the map was created.

---

[1] Note, however, that in §1, we learned that a non-privileged program can really only access slightly less than 2^{32} bytes of address space.
[2] On Intel- and Alpha-based systems, this value is 64KB. Run program sysinfo.c to find out this value for your system.
[3] Refer to §4.4 for more information on object names.

If OpenFileMapping fails to find a file map with the name specified, the non-obvious message ERROR_FILE_NOT_FOUND results. In reality, this operation has nothing to do with a file; the message really means there is no such file map object.

Once a file map object is created, it exists until all programs that are using it, terminate. Consider the scenario in which program P1 opens the file and creates the file map object. Then programs p2 through P*n* access that same file map object. If Programs P2 through P*n* terminate before P1, exclusive access to the disk file is relinquished only when all programs are done. However, what happens if P1 terminates while one or more of P2 through P*n* are still executing? Yes, the file system sees the exclusive access terminate so it allows the file to be read by other programs. However, it will not allow the file to be deleted or written to; attempts to do so result in ERROR_USER_MAPPED_FILE.

There is no explicit action needed to destroy a file map. Ordinarily, a file map goes away when the file to which it is mapped, is closed. However, if a file close is requested by the program that created the file map, and other programs are still using that map, the file isn't closed, and so the map isn't destroyed until the last application program using it, terminates.

13.1.3 File Views

The creation and subsequent opening of a file map only declares our intent to access the shared memory. To actually access the memory, we must create a file view. For example:

```
int *p;

p = MapViewOfFile(hMap, FILE_MAP_READ, 0, 0, 0);
```

The first argument specifies the file map handle returned by CreateFileMapping or OpenFileMapping. The second argument indicates the access required; FILE_MAP_READ indicates read-only access while FILE_MAP_WRITE provides both read and write capability. The third and fourth arguments specify the file's base offset at which to begin, while the last argument specifies the view size. A view then is a window into the mapped file. And we can have multiple views of the same file, all using the same map object. However, if we are writing to the shared memory, we must be sure not to have views that overlap.

MapViewOfFile returns a void pointer. In C, this pointer can safely be assigned directly into a pointer of any type. However, such assignments are prohibited in C++, in which case, an explicit cast is needed just as it would be in a call to malloc. In fact, we can think of MapViewOfFile as being just like malloc in C/C++ or new in C++—both return the address of a newly allocated block of memory. Therefore, the way in which we access this memory is determined by the type of the pointer that leads to it. In the example above, p is a pointer to int so, by subscripting p, we can treat the shared memory as an array of ints, as in

```
p[elem_num] = value;
```

A view is destroyed by calling UnmapViewOfFile. For example:

```
UnmapViewOfFile(p);
```

If p does not current point to an existing view, ERROR_INVALID_ADDRESS results.

