**I/O Completion Ports**

I/O completion ports provide an efficient threading model for processing multiple asynchronous I/O requests on a multiprocessor system. When a process creates an I/O completion port, the system creates an associated queue object for requests whose sole purpose is to service these requests. Processes that handle many concurrent asynchronous I/O requests can do so more quickly and efficiently by using I/O completion ports in conjunction with a pre-allocated thread pool than by creating threads at the time they receive an I/O request.

**How I/O Completion Ports Work**

The **CreateIoCompletionPort** function creates an I/O completion port and associates one or more file handles with that port. When an asynchronous I/O operation on one of these file handles completes, an I/O completion packet is queued in first-in-first-out (FIFO) order to the associated I/O completion port. One powerful use for this mechanism is to combine the synchronization point for multiple file handles into a single object, although there are also other useful applications.

I/O completion packet（完成消息包）在I/O completion port（完成端口）中使用FIFO方式队列。

**Note**

The term *file handle* as used here refers to a system abstraction representing an overlapped I/O endpoint, not only a file on disk. For example, it can be a network endpoint, TCP socket, named pipe, or mail slot. Any system object that supports overlapped I/O can be used. For a list of related I/O functions, see the end of this topic.

A thread (either one created by the main thread or the main thread itself) uses the **GetQueuedCompletionStatus** function to wait for a completion packet to be queued to the I/O completion port, rather than waiting directly for the asynchronous I/O to complete. Threads that block their execution on an I/O completion port are released in last-in-first-out (LIFO) order, and the next completion packet is pulled from the I/O completion port's FIFO queue for that thread. This means that, when a completion packet is released to a thread, the system releases the last (most recent) thread associated with that port, passing it the completion information for the oldest I/O completion.

Although any number of threads can call **GetQueuedCompletionStatus** for a specified I/O completion port, when a specified thread calls **GetQueuedCompletionStatus** the first time, it becomes associated with the specified I/O completion port until one of three things occurs: The thread exits, specifies a different I/O completion port, or closes the I/O completion port. In other words, a single thread can be associated with, at most, one I/O completion port.

When a completion packet is queued to an I/O completion port, the system first checks how many threads associated with that port are running. If the number of threads running is less than the concurrency value (discussed in the next section), one of the waiting threads (the most recent one) is allowed to process the completion packet. When a running thread completes its processing, it typically calls **GetQueuedCompletionStatus** again, at which point it either returns with the next completion packet or waits if the queue is empty.

Threads can use the **PostQueuedCompletionStatus** function to place completion packets in an I/O completion port's queue. By doing so, the completion port can be used to receive communications from other threads of the process, in addition to receiving I/O completion packets from the I/O system. The **PostQueuedCompletionStatus** function allows an application to queue its own special-purpose completion packets to the I/O completion port without starting an asynchronous I/O operation. This is useful for notifying worker threads of external events, for example.

The I/O completion port handle and every file handle associated with that particular I/O completion port are known as *references to the I/O completion port*. The I/O completion port is released when there are no more references to it. Therefore, all of these handles must be properly closed to release the I/O completion port and its associated system resources. After these conditions are satisfied, an application should close the I/O completion port handle by calling the **CloseHandle** function.

**Note**

An I/O completion port is associated with the process that created it and is not shareable between processes. However, a single handle is shareable between threads in the same process.

For another article about I/O completion ports, see "Inside I/O Completion Ports" in the Microsoft TechNet Library at <http://technet.microsoft.com/en-us/sysinternals/bb963891.aspx>

**Threads and Concurrency**

The most important property of an I/O completion port to consider carefully is the concurrency value. The concurrency value of a completion port is specified when it is created with **CreateIoCompletionPort** via the *NumberOfConcurrentThreads* parameter. This value limits the number of runnable threads associated with the completion port. When the total number of runnable threads associated with the completion port reaches the concurrency value, the system blocks the execution of any subsequent threads associated with that completion port until the number of runnable threads drops below the concurrency value.

The most efficient scenario occurs when there are completion packets waiting in the queue, but no waits can be satisfied because the port has reached its concurrency limit. Consider what happens with a concurrency value of one and multiple threads waiting in the **GetQueuedCompletionStatus** function call. In this case, if the queue always has completion packets waiting, when the running thread calls **GetQueuedCompletionStatus**, it will not block execution because, as mentioned earlier, the thread queue is LIFO. Instead, this thread will immediately pick up the next queued completion packet. No thread context switches will occur, because the running thread is continually picking up completion packets and the other threads are unable to run.

**Note**

In the previous example, the extra threads appear to be useless and never run, but that assumes that the running thread never gets put in a wait state by some other mechanism, terminates, or otherwise closes its associated I/O completion port. Consider all such thread execution ramifications when designing the application.

The best overall maximum value to pick for the concurrency value is the number of CPUs on the computer. If your transaction required a lengthy computation, a larger concurrency value will allow more threads to run. Each completion packet may take longer to finish, but more completion packets will be processed at the same time. You can experiment with the concurrency value in conjunction with profiling tools to achieve the best effect for your application.

The system also allows a thread waiting in **GetQueuedCompletionStatus** to process a completion packet if another running thread associated with the same I/O completion port enters a wait state for other reasons, for example the **SuspendThread** function. When the thread in the wait state begins running again, there may be a brief period when the number of active threads exceeds the concurrency value. However, the system quickly reduces this number by not allowing any new active threads until the number of active threads falls below the concurrency value. This is one reason to have your application create more threads in its thread pool than the concurrency value. Thread pool management is beyond the scope of this topic, but a good rule of thumb is to have a minimum of twice as many threads in the thread pool as there are processors on the system. For additional information about thread pooling, see Thread Pools.

**Supported I/O Functions**

The following functions can be used to start I/O operations that complete by using I/O completion ports. You must pass the function an instance of the **OVERLAPPED** structure and a file handle previously associated with an I/O completion port (by a call to **CreateIoCompletionPort**) to enable the I/O completion port mechanism:

* **ConnectNamedPipe**
* **DeviceIoControl**
* **LockFileEx**
* **ReadDirectoryChangesW**
* **ReadFile**
* **TransactNamedPipe**
* **WaitCommEvent**
* **WriteFile**

**CreateIoCompletionPort Function**

Creates an input/output (I/O) completion port and associates it with a specified file handle, or creates an I/O completion port that is not yet associated with a file handle, allowing association at a later time.

Associating an instance of an opened file handle with an I/O completion port allows a process to receive notification of the completion of asynchronous I/O operations involving that file handle.

**Note**

The term *file handle* as used here refers to a system abstraction that represents an overlapped I/O endpoint, not only a file on disk. Any system objects that support overlapped I/O—such as network endpoints, TCP sockets, named pipes, and mail slots—can be used as file handles. For additional information, see the Remarks section.

**Syntax**

HANDLE WINAPI CreateIoCompletionPort(

\_\_in HANDLE *FileHandle*,

\_\_in\_opt HANDLE *ExistingCompletionPort*,

\_\_in ULONG\_PTR *CompletionKey*,

\_\_in DWORD *NumberOfConcurrentThreads*

);

**Parameters**

*FileHandle*

An open file handle or INVALID\_HANDLE\_VALUE.

文件句柄或INVALID\_HANDLE\_VALUE

The handle must be to an object that supports overlapped I/O.

句柄必须支持overlapped I/O

If a handle is provided, it has to have been opened for overlapped I/O completion. For example, you must specify the FILE\_FLAG\_OVERLAPPED flag when using the **CreateFile** function to obtain the handle.

If INVALID\_HANDLE\_VALUE is specified, the function creates an I/O completion port without associating it with a file handle. In this case, the *ExistingCompletionPort* parameter must be NULL and the *CompletionKey* parameter is ignored.

如果是INVALID\_HANDLE\_VALUE，函数创建I/O completion port但不关联文件句柄。此时，ExistingCompletionPort参数必须为NULL，并且CompletionKey参数将被忽略。

*ExistingCompletionPort*

A handle to an existing I/O completion port or NULL.

If this parameter specifies an existing I/O completion port, the function associates it with the handle specified by the *FileHandle* parameter. The function returns the handle of the existing I/O completion port if successful; it does not create a new I/O completion port.

If this parameter is NULL, the function creates a new I/O completion port and, if the *FileHandle* parameter is valid, associates it with the new I/O completion port. Otherwise no file handle association occurs. The function returns the handle to the new I/O completion port if successful.

*CompletionKey*

The per-handle user-defined completion key that is included in every I/O completion packet for the specified file handle. For more information, see the Remarks section.

*NumberOfConcurrentThreads*

The maximum number of threads that the operating system can allow to concurrently process I/O completion packets for the I/O completion port. This parameter is ignored if the *ExistingCompletionPort* parameter is not NULL.

OS可以允许在I/O completion port（IO完成端口）同时处理I/O completion packets（IO完成信息包）的最大线程数。如果ExistingCompletionPort不是NULL，则该参数被忽略。

If this parameter is zero, the system allows as many concurrently running threads as there are processors in the system.

如果该参数为0，系统将允许的并发运行线程数等于系统处理器数目。

**Return Value**

If the function succeeds, the return value is the handle to an I/O completion port:

如果成功返回I/O completion port的句柄：

* If the *ExistingCompletionPort* parameter was NULL, the return value is a new handle.  
  *ExistingCompletionPort* 为NULL时，返回新的句柄。
* If the *ExistingCompletionPort* parameter was a valid I/O completion port handle, the return value is that same handle.  
  *ExistingCompletionPort*是有效I/O completion port句柄时，返回该句柄。
* If the *FileHandle* parameter was a valid handle, that file handle is now associated with the returned I/O completion port.  
  *FileHandle*有效时，该关联文件句柄和I/O completion port。

If the function fails, the return value is NULL. To get extended error information, call the **GetLastError** function.

如果函数失败，返回NULL。如果需要获取附加错误信息，调用GetLastError。

**Remarks**

The I/O system can be instructed to send I/O completion notification packets to I/O completion ports, where they are queued. The **CreateIoCompletionPort** function provides this functionality.

IO系统可以向I/O completion ports发送通知信息包并进入队列。

An I/O completion port and its handle are associated with the process that created it and is not shareable between processes. However, a single handle is shareable between threads in the same process.

I/O completion port及其句柄和创建它的进程相关联并且不能在进程间共享。但是一个句柄可以在同一进程中的不同线程中共享。

**CreateIoCompletionPort** can be used in three distinct modes:

* Create only an I/O completion port without associating it with a file handle.
* Associate an existing I/O completion port with a file handle.
* Perform both creation and association in a single call.

To create an I/O completion port without associating it, set the *FileHandle* parameter to INVALID\_HANDLE\_VALUE, the *ExistingCompletionPort* parameter to NULL, and the *CompletionKey* parameter to zero (which is ignored in this case). Set the *NumberOfConcurrentThreads* parameter to the desired concurrency value for the new I/O completion port, or zero for the default (the number of processors in the system).

The handle passed in the *FileHandle* parameter can be any handle that supports overlapped I/O. Most commonly, this is a handle opened by the **CreateFile** function using the FILE\_FLAG\_OVERLAPPED flag (for example, files, mail slots, and pipes). Objects created by other functions such as **socket** can also be associated with an I/O completion port. For an example using sockets, see **AcceptEx**.

For more information on I/O completion port theory, usage, and associated functions, see I/O Completion Ports.

Multiple file handles can be associated with a single I/O completion port by calling **CreateIoCompletionPort** multiple times with the same I/O completion port handle in the *ExistingCompletionPort* parameter and a different file handle in the *FileHandle* parameter each time.

Use the *CompletionKey* parameter to help your application track which I/O operations have completed. This value is not used by **CreateIoCompletionPort** for functional control; rather, it is attached to the file handle specified in the *FileHandle* parameter at the time of association with an I/O completion port. This completion key should be unique for each file handle, and it accompanies the file handle throughout the internal completion queueing process. It is returned in the **GetQueuedCompletionStatus** function call when a completion packet arrives. The *CompletionKey* parameter is also used by the **PostQueuedCompletionStatus** function to queue your own special-purpose completion packets.

After an instance of an open handle is associated with an I/O completion port, it cannot be used in the **ReadFileEx** or **WriteFileEx** function because these functions have their own asynchronous I/O mechanisms.

It is best not to share a file handle associated with an I/O completion port by using either handle inheritance or a call to the **DuplicateHandle** function. Operations performed with such duplicate handles generate completion notifications. Careful consideration is advised.

The I/O completion port handle and every file handle associated with that particular I/O completion port are known as *references to the I/O completion port*. The I/O completion port is released when there are no more references to it. Therefore, all of these handles must be properly closed to release the I/O completion port and its associated system resources. After these conditions are satisfied, close the I/O completion port handle by calling the **CloseHandle** function.

**Requirements**

|  |  |
| --- | --- |
| **Client** | Requires Windows Vista, Windows XP, or Windows 2000 Professional. |
| **Server** | Requires Windows Server 2008, Windows Server 2003, or Windows 2000 Server. |
| **Header** | Declared in WinBase.h; include Windows.h. |
| **Library** | Use Kernel32.lib. |
| **DLL** | Requires Kernel32.dll. |

**See Also**

**Overview Topics**

File Management Functions  
I/O Completion Ports  
Using the Windows Headers  
Windows Sockets 2

**Functions**

**AcceptEx**  
**CreateFile**  
**DuplicateHandle**  
**GetQueuedCompletionStatus**  
[**GetQueuedCompletionStatusEx**](ms-help://MS.VSCC.v90/MS.MSDNQTR.v90.en/fileio/fs/getqueuedcompletionstatusex_func.htm)  
**PostQueuedCompletionStatus**  
**ReadFileEx**  
**WriteFileEx**

[Send comments about this topic to Microsoft](mailto:wsddocfb@microsoft.com?subject=Documentation%20feedback%20[fileio\fs%5d:%20CreateIoCompletionPort%20%20RELEASE:%20(4/22/2008)&body=%0A%0APRIVACY%20STATEMENT%0A%0AThe%20SDK%20team%20uses%20the%20feedback%20submitted%20to%20improve%20the%20SDK%20documentation.%20We%20do%20not%20use%20your%20e-mail%20address%20for%20any%20other%20purpose.%20We%20will%20remove%20your%20e-mail%20address%20from%20our%20system%20after%20the%20issue%20you%20are%20reporting%20has%20been%20resolved.%20While%20we%20are%20working%20to%20resolve%20this%20issue,%20we%20may%20send%20you%20an%20e-mail%20message%20to%20request%20more%20information%20about%20your%20feedback.%20After%20the%20issues%20have%20been%20addressed,%20we%20may%20send%20you%20an%20e-mail%20message%20to%20let%20you%20know%20that%20your%20feedback%20has%20been%20addressed.%0A%0AFor%20more%20information%20about%20Microsoft's%20privacy%20policy,%20see%20http://privacy.microsoft.com/en-us/default.aspx.)

Build date: 4/22/2008

**PostQueuedCompletionStatus Function**

Posts an I/O completion packet to an I/O completion port.

**Syntax**

BOOL WINAPI PostQueuedCompletionStatus(

\_\_in HANDLE *CompletionPort*,

\_\_in DWORD *dwNumberOfBytesTransferred*,

\_\_in ULONG\_PTR *dwCompletionKey*,

\_\_in\_opt LPOVERLAPPED *lpOverlapped*

);

**Parameters**

*CompletionPort*

A handle to an I/O completion port to which the I/O completion packet is to be posted.

*dwNumberOfBytesTransferred*

The value to be returned through the *lpNumberOfBytesTransferred* parameter of the **GetQueuedCompletionStatus** function.

*dwCompletionKey*

The value to be returned through the *lpCompletionKey* parameter of the **GetQueuedCompletionStatus** function.

*lpOverlapped*

The value to be returned through the *lpOverlapped* parameter of the **GetQueuedCompletionStatus** function.

**Return Value**

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call **GetLastError** .

**Remarks**

The I/O completion packet will satisfy an outstanding call to the **GetQueuedCompletionStatus** function. This function returns with the three values passed as the second, third, and fourth parameters of the call to **PostQueuedCompletionStatus**. The system does not use or validate these values. In particular, the *lpOverlapped* parameter need not point to an **OVERLAPPED** structure.

**Requirements**

|  |  |
| --- | --- |
| **Client** | Requires Windows Vista, Windows XP, or Windows 2000 Professional. |
| **Server** | Requires Windows Server 2008, Windows Server 2003, or Windows 2000 Server. |
| **Header** | Declared in WinBase.h; include Windows.h. |
| **Library** | Use Kernel32.lib. |
| **DLL** | Requires Kernel32.dll. |

**See Also**

**CreateIoCompletionPort**  
File Management Functions  
**GetQueuedCompletionStatus**  
**OVERLAPPED**

**GetQueuedCompletionStatus Function**

Attempts to dequeue an I/O completion packet from the specified I/O completion port. If there is no completion packet queued, the function waits for a pending I/O operation associated with the completion port to complete.

To dequeue multiple I/O completion packets at once, use the [**GetQueuedCompletionStatusEx**](ms-help://MS.VSCC.v90/MS.MSDNQTR.v90.en/fileio/fs/getqueuedcompletionstatusex_func.htm) function.

**Syntax**

BOOL WINAPI GetQueuedCompletionStatus(

\_\_in HANDLE *CompletionPort*,

\_\_out LPDWORD *lpNumberOfBytes*,

\_\_out PULONG\_PTR *lpCompletionKey*,

\_\_out LPOVERLAPPED\* *lpOverlapped*,

\_\_in DWORD *dwMilliseconds*

);

**Parameters**

*CompletionPort*

A handle to the completion port. To create a completion port, use the **CreateIoCompletionPort** function.

*lpNumberOfBytes*

A pointer to a variable that receives the number of bytes transferred during an I/O operation that has completed.

*lpCompletionKey*

A pointer to a variable that receives the completion key value associated with the file handle whose I/O operation has completed. A completion key is a per-file key that is specified in a call to **CreateIoCompletionPort**.

*lpOverlapped*

A pointer to a variable that receives the address of the **OVERLAPPED** structure that was specified when the completed I/O operation was started.

Even if you have passed the function a file handle associated with a completion port and a valid **OVERLAPPED** structure, an application can prevent completion port notification. This is done by specifying a valid event handle for the **hEvent** member of the **OVERLAPPED** structure, and setting its low-order bit. A valid event handle whose low-order bit is set keeps I/O completion from being queued to the completion port.

*dwMilliseconds*

The number of milliseconds that the caller is willing to wait for a completion packet to appear at the completion port. If a completion packet does not appear within the specified time, the function times out, returns FALSE, and sets \**lpOverlapped* to NULL.

If *dwMilliseconds* is INFINITE, the function will never time out. If *dwMilliseconds* is zero and there is no I/O operation to dequeue, the function will time out immediately.

**Return Value**

If the function dequeues a completion packet for a successful I/O operation from the completion port, the return value is nonzero. The function stores information in the variables pointed to by the *lpNumberOfBytes*, *lpCompletionKey*, and *lpOverlapped* parameters.

If \**lpOverlapped* is NULL and the function does not dequeue a completion packet from the completion port, the return value is zero. The function does not store information in the variables pointed to by the *lpNumberOfBytes* and *lpCompletionKey* parameters. To get extended error information, call **GetLastError**. If the function did not dequeue a completion packet because the wait timed out, **GetLastError** returns WAIT\_TIMEOUT.

If \**lpOverlapped* is not NULL and the function dequeues a completion packet for a failed I/O operation from the completion port, the return value is zero. The function stores information in the variables pointed to by *lpNumberOfBytes*, *lpCompletionKey*, and *lpOverlapped*. To get extended error information, call **GetLastError**.

**Remarks**

This function associates a thread with the specified completion port. A thread can be associated with at most one completion port.

This function returns TRUE when at least one pending I/O is completed.

This function returns FALSE when no I/O operation was dequeued. This typically means that an error occurred while processing the parameters to this call, or that the *CompletionPort* handle was closed or is otherwise invalid. The **GetLastError** function provides extended error information.

Starting with Windows Vista, if a call to **GetQueuedCompletionStatus** fails because the handle associated with it is closed, the function returns FALSE and **GetLastError** will return ERROR\_ABANDONED\_WAIT\_0.

For more information on I/O completion port theory, usage, and associated functions, see I/O Completion Ports.

**Requirements**

|  |  |
| --- | --- |
| **Client** | Requires Windows Vista, Windows XP, or Windows 2000 Professional. |
| **Server** | Requires Windows Server 2008, Windows Server 2003, or Windows 2000 Server. |
| **Header** | Declared in WinBase.h; include Windows.h. |
| **Library** | Use Kernel32.lib. |
| **DLL** | Requires Kernel32.dll. |

**See Also**

**Overview Topics**

File Management Functions  
I/O Completion Ports  
Using the Windows Headers

**Functions**

**ConnectNamedPipe**  
**CreateIoCompletionPort**  
**DeviceIoControl**  
[**GetQueuedCompletionStatusEx**](ms-help://MS.VSCC.v90/MS.MSDNQTR.v90.en/fileio/fs/getqueuedcompletionstatusex_func.htm)  
**LockFileEx**  
**ReadFile**  
**PostQueuedCompletionStatus**  
**TransactNamedPipe**  
**WaitCommEvent**  
**WriteFile**