

14 Asynchronous Input/Output and Completion Ports

Input and output are inherently slow compared with other processing due to factors such as the following:

- Delays caused by track and sector seek time on random access devices, such as disks
- Delays caused by the relatively slow data transfer rate between a physical device and system memory
- Delays in network data transfer using file servers, storage area networks, and so on

All I/O in previous examples has been *thread-synchronous*, so that the entire thread waits until the I/O operation completes.

This chapter shows how a thread can continue without waiting for an operation to complete—that is, threads can perform *asynchronous* I/O. Examples illustrate the different techniques available in Windows.

Waitable timers, which require some of the same techniques, are also described here.

Finally, and more important, once standard asynchronous I/O is understood, we are in a position to use *I/O completion ports*, which are extremely useful when building scalable servers that must be able to support large numbers of clients without creating a thread for each client. Program 14–4 modifies an earlier server to exploit I/O completion ports.

Overview of Windows Asynchronous I/O

There are three techniques for achieving asynchronous I/O in Windows; they differ in both the methods used to start I/O operations and those used to determine when operations are complete.

- **Multithreaded I/O.** Each thread within a process or set of processes performs normal synchronous I/O, but other threads can continue execution.
- **Overlapped I/O (with waiting).** A thread continues execution after issuing a read, write, or other I/O operation. When the thread requires the I/O results before continuing, it waits on either the file handle or an event specified in the `ReadFile` or `WriteFile` overlapped structure.
- **Overlapped I/O with completion routines (or “extended I/O” or “alertable I/O”).** The system invokes a specified *completion routine* callback function within the thread when the I/O operation completes. The term “extended I/O” is easy to remember because it requires extended functions such as `WriteFileEx` and `ReadFileEx`.

The terms “overlapped I/O” and “extended I/O” are used for the last two techniques; they are, however, two forms of overlapped I/O that differ in the way Windows indicates completed operations.

The threaded server in Chapter 11 uses multithreaded I/O on named pipes. `grepMT` (Program 7–1) manages concurrent I/O to several files. Thus, we have existing programs that perform multithreaded I/O to achieve a form of asynchronous I/O.

Overlapped I/O is the subject of the next section, and the examples implement file conversion (simplified Caesar cipher, first used in Chapter 2) with this technique in order to illustrate sequential file processing. The example is a modification of Program 2–3. Following overlapped I/O, we explain extended I/O with completion routines.

Note: Overlapped and extended I/O can be complex and seldom yield large performance benefits on Windows XP. Threads frequently overcome these problems, so *some readers might wish to skip ahead to the sections on waitable timers and I/O completion ports (but see the next note)*, referring back as necessary. Before doing so, however, you will find asynchronous I/O concepts in both old and very new technology, so it can be worthwhile to learn the techniques. Also, the asynchronous procedure call (APC) operation (Chapter 10) is very similar to extended I/O. There’s a final significant advantage to the two overlapped I/O techniques: you can cancel outstanding I/O operations, allowing cleanup.

NT6 Note: NT6 (including Windows 7) provides an exception to the comment about performance. NT6 extended and overlapped I/O provide good performance compared to simple sequential I/O; we’ll show the results here.

Finally, since I/O performance and scalability are almost always the principal objectives (in addition to correctness), remember that memory-mapped I/O can be very effective when processing files (Chapter 5), although it is not trivial to recover from memory-mapped I/O errors.

Overlapped I/O

The first requirement for asynchronous I/O, whether overlapped or extended, is to set the overlapped attribute of the file or other handle. Do this by specifying the `FILE_FLAG_OVERLAPPED` flag on the `CreateFile` or other call that creates the file, named pipe, or other handle. Sockets (Chapter 13), whether created by socket or accept, have the attribute set by default. An overlapped socket can be used asynchronously in all Windows versions.

Until now, overlapped structures have only been used with `LockFileEx` and as an alternative to `SetFilePointerEx` (Chapter 3), but they are essential for overlapped I/O. These structures are optional parameters on four I/O functions that can potentially block while the operation completes:

```
ReadFile
WriteFile
TransactNamedPipe
ConnectNamedPipe
```

Recall that when you're specifying `FILE_FLAG_OVERLAPPED` as part of `dwAttrsAndFlags` (for `CreateFile`) or as part of `dwOpenMode` (for `CreateNamedPipe`), the pipe or file is to be used only in overlapped mode. Overlapped I/O does not work with anonymous pipes.

Consequences of Overlapped I/O

Overlapped I/O is asynchronous. There are several consequences when starting an overlapped I/O operation.

- I/O operations do not block. The system returns immediately from a call to `ReadFile`, `WriteFile`, `TransactNamedPipe`, or `ConnectNamedPipe`.
- A returned `FALSE` value does not necessarily indicate failure because the I/O operation is most likely not yet complete. In this normal case, `GetLastError()` will return `ERROR_IO_PENDING`, indicating no error. Windows provides a different mechanism to indicate status.

- The returned number of bytes transferred is also not useful if the transfer is not complete. Windows must provide another means of obtaining this information.
- The program may issue multiple reads or writes on a single overlapped file handle. Therefore, the handle's file pointer is meaningless. There must be another method to specify file position with each read or write. This is not a problem with named pipes, which are inherently sequential.
- The program must be able to wait (synchronize) on I/O completion. In case of multiple outstanding operations on a single handle, it must be able to determine which operation has completed. I/O operations do not necessarily complete in the same order in which they were issued.

The last two issues—file position and synchronization—are addressed by the overlapped structures.

Overlapped Structures

The `OVERLAPPED` structure (specified, for example, by the `lpOverlapped` parameter of `ReadFile`) indicates the following:

- The file position (64 bits) where the read or write is to start, as discussed in Chapter 3
- The event (manual-reset) that will be signaled when the operation completes

Here is the `OVERLAPPED` structure.

```
typedef struct _OVERLAPPED {
    DWORD Internal;
    DWORD InternalHigh;
    DWORD Offset;
    DWORD OffsetHigh;
    HANDLE hEvent;
} OVERLAPPED
```

The file position (pointer) must be set in both `Offset` and `OffsetHigh`. Do not set `Internal` and `InternalHigh`, which are reserved for the system. Currently, Windows sets `Internal` to the I/O request error code and `InternalHigh` to the number of bytes transferred. However, MSDN warns that

this behavior may change in the future, and there are other ways to get the information.

`hEvent` is an event handle (created with `CreateEvent`). The event can be named or unnamed, but it *must* be a manual-reset event (see Chapter 8) when used for overlapped I/O; the reasons are explained soon. The event is signaled when the I/O operation completes.

Alternatively, `hEvent` can be `NULL`; in this case, the program can wait on the file handle, which is also a synchronization object (see the upcoming list of cautions). *Note:* For convenience, the term “file handle” is used to describe the handle with `ReadFile`, `WriteFile`, and so on, even though this handle could refer to a pipe or device rather than to a file.

This event is immediately reset (set to the nonsignaled state) by the system when the program makes an I/O call. When the I/O operation completes, the event is signaled and remains signaled until it is used with another I/O operation. The event needs to be manual-reset because multiple threads might wait on it (although our example uses only one thread).

Even if the file handle is synchronous (it was created without `FILE_FLAG_OVERLAPPED`), the overlapped structure is an alternative to `SetFilePointer` and `SetFilePointerEx` for specifying file position. In this case, the `ReadFile` or other call does not return until the operation is complete. This feature was useful in Chapter 3.

Notice also that an outstanding I/O operation is uniquely identified by the combination of file handle and overlapped structure.

Here are a few cautions to keep in mind.

- Do not reuse an `OVERLAPPED` structure while its associated I/O operation, if any, is outstanding.
- Similarly, do not reuse an event while it is part of an `OVERLAPPED` structure.
- If there is more than one outstanding request on an overlapped handle, use events, rather than the file handle, for synchronization. We provide examples of both forms.
- As with any automatic variable, if the `OVERLAPPED` structure or event is an automatic variable in a block, be certain not to exit the block before synchronizing with the I/O operation. Also, close the event handle before leaving the block to avoid a resource leak.

Overlapped I/O States

An overlapped `ReadFile` or `WriteFile` operation—or, for that matter, one of the two named pipe operations—returns immediately. In most cases, the I/O will not

be complete, and the read or write returns `FALSE`. `GetLastError` returns `ERROR_IO_PENDING`. However, test the read or write return; if it's `TRUE`, you can get the transfer count immediately and proceed without waiting.

After waiting on a synchronization object (an event or, perhaps, the file handle) for the operation to complete, you need to determine how many bytes were transferred. This is the primary purpose of `GetOverlappedResult`.

```
BOOL GetOverlappedResult (
    HANDLE hFile,
    LPOVERLAPPED lpOverlapped,
    LPWORD lpcbTransfer,
    BOOL bWait)
```

The handle and overlapped structure combine to indicate the specific I/O operation. `bWait`, if `TRUE`, specifies that `GetOverlappedResult` will wait until the specified operation is complete; otherwise, it returns immediately. In either case, the function returns `TRUE` only if the operation has completed successfully. `GetLastError` returns `ERROR_IO_INCOMPLETE` in case of a `FALSE` return from `GetOverlappedResult`, so it is possible to poll for I/O completion with this function.

The number of bytes transferred is in `*lpcbTransfer`. Be certain that the overlapped structure is unchanged from when it was used with the overlapped I/O operation.

Canceling Overlapped I/O Operations

The Boolean NT6 function `CancelIoEx` cancels outstanding overlapped I/O operations on the specified handle in the current process. The arguments are the handle and the overlapped structure. All pending operations issued by the calling thread using the handle and overlapped structure are canceled. Use `NULL` for the overlapped structure to cancel all operations using the handle.

`CancelIoEx` cancels I/O requests in the calling thread only.

The canceled operations will usually complete with error code `ERROR_OPERATION_ABORTED` and status `STATUS_CANCELLED`, although the status would be `STATUS_SUCCESS` if the operation completed before the cancellation call.

`CancelIoEx` does not, however, wait for the cancellation to complete, so it's still essential to wait in the normal way before reusing the `OVERLAPPED` structure for another I/O operation.

Program 14-4 (serverCP) exploits `CancelIoEx`.

Example: Synchronizing on a File Handle

Overlapped I/O can be useful and relatively simple when there is only one outstanding operation. The program can synchronize on the file handle rather than on an event.

The following code fragment shows how a program can initiate a read operation to read a portion of a file, continue to perform other processing, and then wait on the handle.

```
OVERLAPPED ov = { 0, 0, 0, 0, NULL /* No event. */ };
HANDLE hF;
DWORD nRead;
BYTE buffer [BUF_SIZE];
...
hF = CreateFile ( ..., FILE_FLAG_OVERLAPPED, ... );
ReadFile (hF, buffer, sizeof (buffer), &nRead, &ov);
/* Perform other processing. nRead probably not valid. */
/* Wait for the read to complete. */
WaitForSingleObject (hF, INFINITE);
GetOverlappedResult (hF, &ov, &nRead, FALSE);
```

Example: File Conversion with Overlapped I/O and Multiple Buffers

Program 2-3 (cci) encrypted a file to illustrate sequential file conversion, and Program 5-3 (ccimm) showed how to perform the same sequential file processing with memory-mapped files. Program 14-1 (cciOV) performs the same task using overlapped I/O and multiple buffers holding fixed-size records.

Figure 14-1 shows the program organization and an operational scenario with four fixed-size buffers. The program is implemented so that the number of buffers is defined in a preprocessor variable, but the following discussion assumes four buffers.

First, the program initializes all the overlapped structures with events and file positions. There is a separate overlapped structure for each input and each output buffer. Next, an overlapped read is issued for each of the four input buffers. The program then uses `WaitForMultipleObjects` to wait for a single event, indicating either a read or a write completed. When a read completes, the buffer is copied and converted into the corresponding output buffer and the write is initiated. When a write completes, the next read is initiated. Notice that the events associated with the input and output buffers are arranged in a single array to be used as an argument to `WaitForMultipleObjects`.

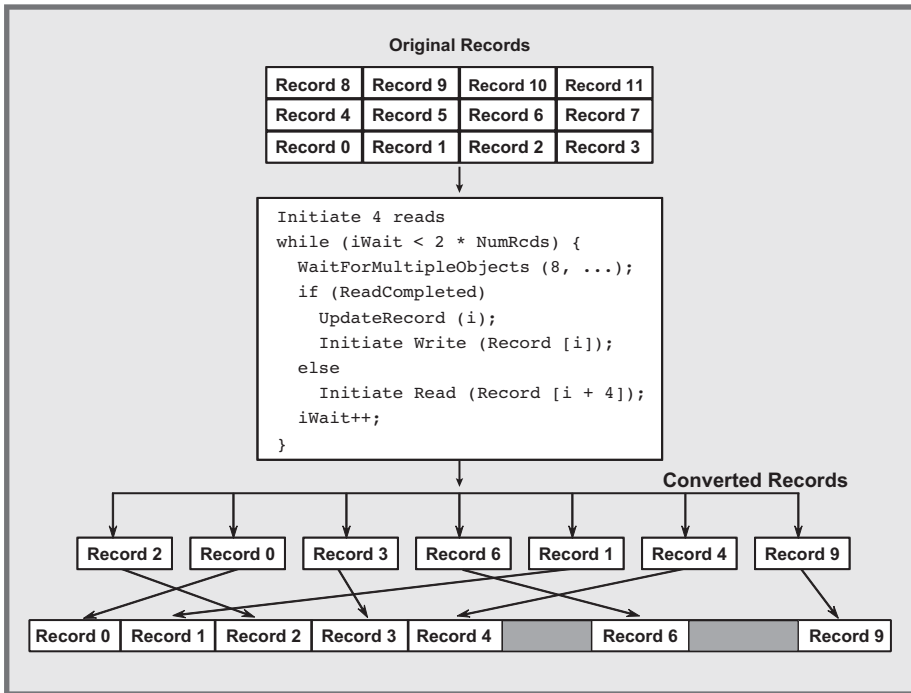


Figure 14-1 An Asynchronous File Update Model

Program 14-1 cciOV: File Conversion with Overlapped I/O

```

/* Chapter 14. cciOV.
   OVERLAPPED simplified Caesar cipher file conversion. */
/* cciOV shift file1 file2 */
/* This program illustrates overlapped asynch I/O. */

#include "Everything.h"

#define MAX_OVRLP 4/* Must be <= 32 due to WaitForMultipleObjects */
#define REC_SIZE 0x4000 /* Selected experimentally for performance */

int _tmain (int argc, LPTSTR argv[])
{
    HANDLE hInputFile, houtputFile;
    DWORD shift, nIn[MAX_OVRLP], nOut[MAX_OVRLP], ic, i;
    /* There is a copy of each of the following structures */
    /* for each outstanding overlapped I/O operation */
    OVERLAPPED overLapIn[MAX_OVRLP], overLapOut[MAX_OVRLP];
    /* The first event index is 0 for read, 1 for write */
    /* WaitForMultipleObjects requires a contiguous array */
  
```



```

HANDLE hEvents[2][MAX_OVRLP];
/* The first index on these two buffers is the I/O operation */
CHAR rawRec[MAX_OVRLP][REC_SIZE], ccRec[MAX_OVRLP][REC_SIZE];
LARGE_INTEGER curPosIn, curPosOut, fileSize;
LONGLONG nRecords, iWaits;

shift = _ttoi(argv[1]);

hInputFile = CreateFile (argv[2], GENERIC_READ,
    0, NULL, OPEN_EXISTING, FILE_FLAG_OVERLAPPED, NULL);
hOutputFile = CreateFile (argv[3], GENERIC_WRITE,
    0, NULL, CREATE_ALWAYS, FILE_FLAG_OVERLAPPED, NULL);

/* Compute total number of records - There may be a
   partial record at the end. */
GetFileSizeEx (hInputFile, &fileSize);
nRecords = (fileSize.QuadPart + REC_SIZE - 1) / REC_SIZE;

/* Create the manual-reset, unsignaled events for
   the overlapped structures. Initiate a read on
   each buffer, corresponding to the overlapped structure. */
curPosIn.QuadPart = 0;
for (ic = 0; ic < MAX_OVRLP; ic++) {
    /* Input and output complete events. */
    hEvents[0][ic] = overLapIn[ic].hEvent =
        CreateEvent (NULL, TRUE, FALSE, NULL);
    hEvents[1][ic] = overLapOut[ic].hEvent =
        CreateEvent (NULL, TRUE, FALSE, NULL);
    /* Set file position. */
    overLapIn[ic].Offset = curPosIn.LowPart;
    overLapIn[ic].OffsetHigh = curPosIn.HighPart;
    if (curPosIn.QuadPart < fileSize.QuadPart)
        ReadFile (hInputFile, rawRec[ic], REC_SIZE,
            &nIn[ic], &overLapIn[ic]);
    curPosIn.QuadPart += (LONGLONG)REC_SIZE;
}

/* All read operations running. Wait for read & write events.
   Continue until all records have been processed. */

iWaits = 0;
while (iWaits < 2 * nRecords) {
    ic = WaitForMultipleObjects (2 * MAX_OVRLP, hEvents[0],
        FALSE, INFINITE) - WAIT_OBJECT_0;
    iWaits++;

    if (ic < MAX_OVRLP) { /* A read completed. */
        GetOverlappedResult (hInputFile, &overLapIn[ic],
            &nIn[ic], FALSE);
    }
}

```

```

/* Reset event before the next WFMO call. Otherwise,
 * event won't be reset until next Read for this ic value */
ResetEvent (hEvents[0][ic]);
/* Process record and start write at same position. */
curPosIn.LowPart = overLapIn[ic].Offset;
curPosIn.HighPart = overLapIn[ic].OffsetHigh;
curPosOut.QuadPart = curPosIn.QuadPart;
overLapOut[ic].Offset = curPosOut.LowPart;
overLapOut[ic].OffsetHigh = curPosOut.HighPart;

/* Encrypt the record. */
for (i = 0; i < nIn[ic]; i++)
    ccRec[ic][i] = (rawRec[ic][i] + shift) % 256;
WriteFile (hOutputFile, ccRec[ic], nIn[ic],
    &nOut[ic], &overLapOut[ic]);

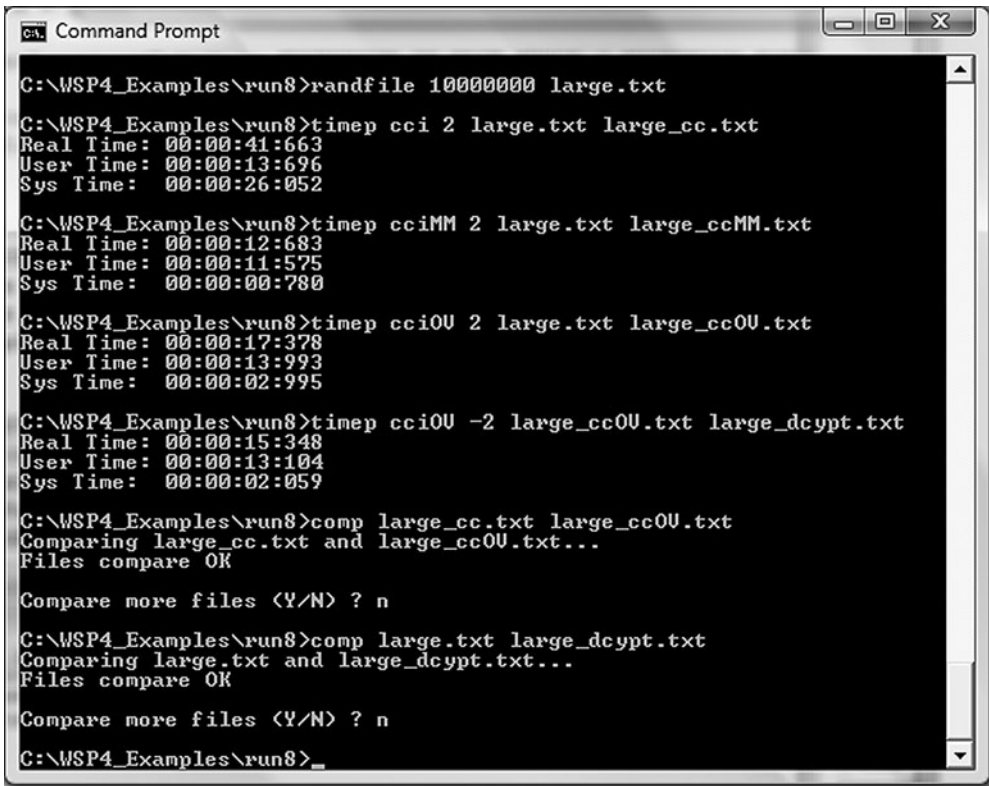
/* Prepare input overlapped structure for next read, which
 * is initiated after the write completes. */

curPosIn.QuadPart += REC_SIZE * (LONGLONG) (MAX_OVRLP);
overLapIn[ic].Offset = curPosIn.LowPart;
overLapIn[ic].OffsetHigh = curPosIn.HighPart;

} else if (ic < 2 * MAX_OVRLP) { /* A write completed. */
    /* Start the read. Check first to not read past file end */
    ic -= MAX_OVRLP; /* Set the output buffer index. */
    GetOverlappedResult (hOutputFile, &overLapOut[ic],
        &nOut[ic], FALSE);
    ResetEvent (hEvents[1][ic]);
    curPosIn.LowPart = overLapIn[ic].Offset;
    curPosIn.HighPart = overLapIn[ic].OffsetHigh;
    /* Start a new read. */
    if (curPosIn.QuadPart < fileSize.QuadPart) {
        ReadFile (hInputFile, rawRec[ic], REC_SIZE,
            &nIn[ic], &overLapIn[ic]);
    }
}
else /* Impossible unless wait failed error. */
    ReportError (_T ("Multiple wait error."), 0, TRUE);
}

/* Close all events. */
for (ic = 0; ic < MAX_OVRLP; ic++) {
    CloseHandle (hEvents[0][ic]);
    CloseHandle (hEvents[1][ic]);
}
CloseHandle (hInputFile);
CloseHandle (hOutputFile);
return 0;
}

```



```
C:\WSP4_Examples\run8>randfile 10000000 large.txt

C:\WSP4_Examples\run8>ttimep cci 2 large.txt large_cc.txt
Real Time: 00:00:41:663
User Time: 00:00:13:696
Sys Time: 00:00:26:052

C:\WSP4_Examples\run8>ttimep cciMM 2 large.txt large_ccMM.txt
Real Time: 00:00:12:683
User Time: 00:00:11:575
Sys Time: 00:00:00:780

C:\WSP4_Examples\run8>ttimep cciOV 2 large.txt large_ccOV.txt
Real Time: 00:00:17:378
User Time: 00:00:13:993
Sys Time: 00:00:02:995

C:\WSP4_Examples\run8>ttimep cciOV -2 large_ccOV.txt large_dcypt.txt
Real Time: 00:00:15:348
User Time: 00:00:13:104
Sys Time: 00:00:02:059

C:\WSP4_Examples\run8>comp large_cc.txt large_ccOV.txt
Comparing large_cc.txt and large_ccOV.txt...
Files compare OK

Compare more files (Y/N) ? n

C:\WSP4_Examples\run8>comp large.txt large_dcypt.txt
Comparing large.txt and large_dcypt.txt...
Files compare OK

Compare more files (Y/N) ? n

C:\WSP4_Examples\run8>_
```

Run 14-1 cciOV: Comparing Performance and Testing Results

Run 14-1 shows `cci` timings converting the same 640MB file with `cci`, `cciMM`, and `cciOV` on a four-processor Windows Vista machine.

Memory mapping and overlapped I/O provide the best performance, with memory-mapped I/O showing a consistent advantage (12.7 seconds compared to about 16 seconds in this test). Run 14-1 also compares the converted files and the decrypted file as an initial correctness test.

The `cciOV` timing results in Run 14-1 depend on the record size (the `REC_SIZE` macro in the listing). The `0x4000` (16K) value worked well, as did 8K. However, 32K required twice the time. An exercise suggests experimenting with the record size on different systems and file sizes. Appendix C shows additional timing results on several systems for the different `cci` implementations.

Caution: The elapsed time in these tests can occasionally increase significantly, sometimes by factors of 2 or more. However, Run 14-1 contains typical results that I've been able to reproduce consistently. Nonetheless, be aware that you might see much longer times, depending on numerous factors such as other machine activity.

Extended I/O with Completion Routines

There is an alternative to using synchronization objects. Rather than requiring a thread to wait for a completion signal on an event or handle, the system can invoke a user-specified completion, or callback, routine when an I/O operation completes. The completion routine can then start the next I/O operation and perform any other bookkeeping. The completion or callback routine is similar to Chapter 10's asynchronous procedure call and requires alertable wait states.

How can the program specify the completion routine? There are no remaining `ReadFile` or `WriteFile` parameters or data structures to hold the routine's address. There is, however, a family of extended I/O functions, identified by the `Ex` suffix and containing an extra parameter for the completion routine address. The read and write functions are `ReadFileEx` and `WriteFileEx`, respectively. It is also necessary to use one of five alertable wait functions:

- `WaitForSingleObjectEx`
- `WaitForMultipleObjectsEx`
- `SleepEx`
- `SignalObjectAndWait`
- `MsgWaitForMultipleObjectsEx`

Extended I/O is sometimes called *alertable I/O*, and Chapter 10 used alertable wait states for thread cancellation. The following sections show how to use the extended functions.

ReadFileEx, WriteFileEx, and Completion Routines

The extended read and write functions work with open file, named pipe, and mailslot handles if `FILE_FLAG_OVERLAPPED` was used at open (create) time. Notice that the flag sets a handle attribute, and while overlapped I/O and extended I/O are distinguished, a single overlapped flag enables both types of asynchronous I/O on a handle.

Overlapped sockets (Chapter 12) operate with `ReadFileEx` and `WriteFileEx`.