```
BOOL ReadFileEx (
   HANDLE hFile,
   LPVOID lpBuffer,
   DWORD nNumberOfBytesToRead,
   LPOVERLAPPED lpOverlapped,
   LPOVERLAPPED_COMPLETION_ROUTINE lpcr)
BOOL WriteFileEx (
   HANDLE hFile,
   LPVOID lpBuffer,
   DWORD nNumberOfBytesToWrite,
   LPOVERLAPPED lpOverlapped,
   LPOVERLAPPED_COMPLETION_ROUTINE lpcr)
```

The two functions are familiar but have an extra parameter to specify the completion routine. The completion routine could be `NULL`; there's no easy way to get the results.

The overlapped structures must be supplied, but there is no need to specify the `hEvent` member; the system ignores it. It turns out, however, that this member is useful for carrying information, such as a sequence number, to identify the I/O operation, as shown in Program 14–2.

In comparison to `ReadFile` and `WriteFile`, notice that the extended functions do not require the parameters for the number of bytes transferred. That information is conveyed as an argument to the completion routine.

The completion routine has parameters for the byte count, an error code, and the overlapped structure. The last parameter is necessary so that the completion routine can determine which of several outstanding operations has completed. Notice that the same cautions regarding reuse or destruction of overlapped structures apply here as they did for overlapped I/O.

```
VOID WINAPI FileIOCompletionRoutine (
   DWORD dwError,
   DWORD cbTransferred,
   LPOVERLAPPED lpo)
```

As was the case with `CreateThread`, which also specified a function name, *`FileIOCompletionRoutine`* is a placeholder and not an actual function name.

Common `dwError` values are `0` (success) and `ERROR_HANDLE_EOF` (when a read tries to go past the end of the file). The overlapped structure is the one used by the completed `ReadFileEx` or `WriteFileEx` call.

Two things must happen before the completion routine is invoked by the system.

1. The I/O operation must complete.
2. The calling thread must be in an alertable wait state, notifying the system that it should execute any queued completion routines.

How does a thread get into an alertable wait state? It must make an explicit call to one of the alertable wait functions described in the next section. In this way, the thread can ensure that the completion routine does not execute prematurely. A thread can be in an alertable wait state only while it is calling an alertable wait function; on return, the thread is no longer in this state.

Once these two conditions have been met, completion routines that have been queued as the result of I/O completion are executed. *Completion routines are executed in the same thread that made the original I/O call and is in the alertable wait state.* Therefore, the thread should enter an alertable wait state only when it is safe for completion routines to execute.

Alertable Wait Functions

There are five alertable wait functions, and the three that relate directly to our current needs are described here.

```
DWORD WaitForSingleObjectEx (
   HANDLE hObject,
   DWORD dwMilliseconds,
   BOOL bAlertable)

DWORD WaitForMultipleObjectsEx (
   DWORD cObjects,
   LPHANDLE lphObjects,
   BOOL fWaitAll,
   DWORD dwMilliseconds,
   BOOL bAlertable)

DWORD SleepEx (
   DWORD dwMilliseconds,
   BOOL bAlertable)
```

Each alertable wait function has a `bAlertable` flag that must be set to `TRUE` when used for asynchronous I/O. The functions are extensions of the familiar `Wait` and `Sleep` functions.

Time-outs, as always, are in milliseconds. These three functions will return as soon as *any one* of the following situations occurs.

- Handle(s) are signaled so as to satisfy one of the two wait functions in the normal way.
- The time-out period expires.
- At least one completion routine or user APC (see Chapter 10) is queued to the thread and `bAlertable` is set. Completion routines are queued when their associated I/O operation is complete (see Figure 14–2) or `QueueUserAPC` queues a user APC. Windows executes all queued user APCs and completion routines before returning from an alertable wait function. Note how this allows I/O operation cancellation with a user APC, as was done in Chapter 10.

Also notice that no events are associated with the `ReadFileEx` and `WriteFileEx` overlapped structures, so any handles in the wait call will have no direct relation to the I/O operations. `SleepEx`, on the other hand, is not associated with a synchronization object and is the easiest of the three functions to use. `SleepEx` is usually used with an `INFINITE` time-out so that the function will return only after one or more of the currently queued completion routines have finished.

Execution of Completion Routines and the Alertable Wait Return

As soon as an extended I/O operation is complete, its associated completion routine, with the overlapped structure, byte count, and error status arguments, is queued for execution.

All of a thread's queued completion routines are executed when the thread enters an alertable wait state. They are executed sequentially but not necessarily in the same order as I/O completion. The alertable wait function returns only after the completion routines return. This property is essential to the proper operation of most programs because it assumes that the completion routines can prepare for the next use of the overlapped structure and perform related operations to get the program to a known state before the alertable wait returns.

`SleepEx` and the wait functions will return `WAIT_IO_COMPLETION` if one or more queued completion routines were executed.

Here are two final points.

1. Use an `INFINITE` time-out value with any alertable wait function. Without the possibility of a time-out, the wait function will return only after all queued completion routines have been executed or the handles have been signaled.

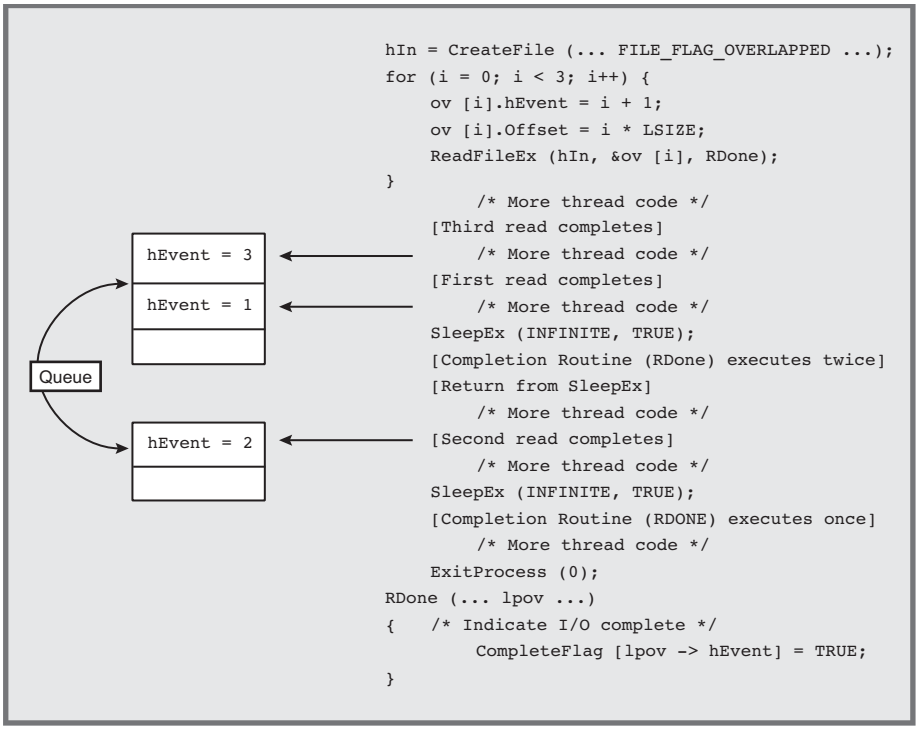


Figure 14–2 Asynchronous I/O with Completion Routines

2. It is common practice to use the `hEvent` data member of the overlapped structure to convey information to the completion routine because Windows ignores this field.

Figure 14–2 illustrates the interaction among the main thread, the completion routines, and the alertable waits. In this example, three concurrent read operations are started, and two are completed by the time the alertable wait is performed.

Example: File Conversion with Extended I/O

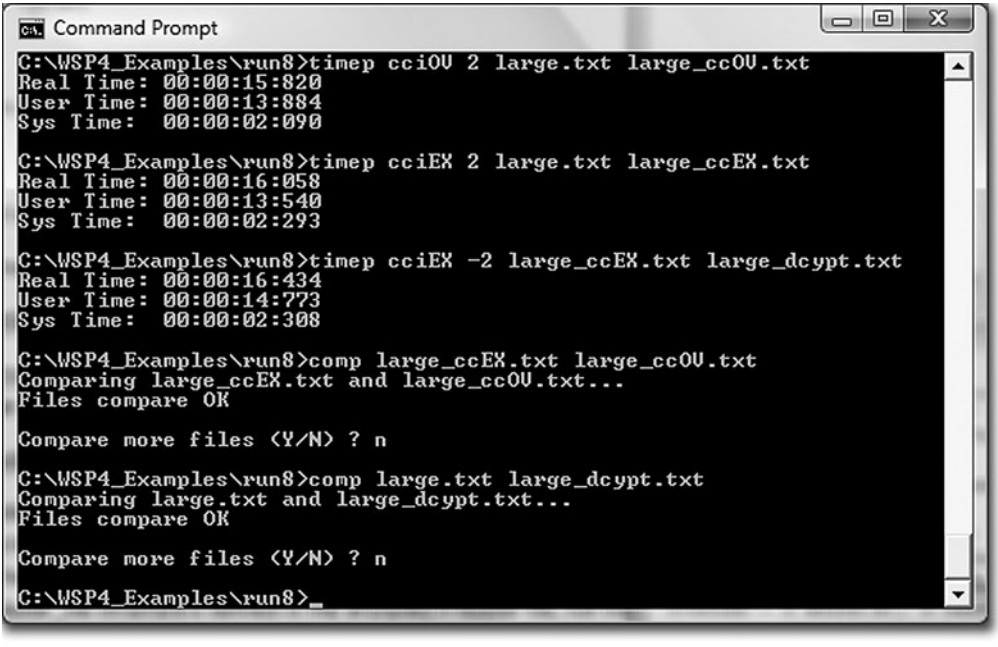
Program 14–2, `cciEX`, reimplements Program 14–1, `cciOV`. These programs show the programming differences between the two asynchronous I/O techniques. `cciEX` is similar to Program 14–1 but moves most of the bookkeeping code to the completion routines, and many variables are global so as to be accessible from the completion routines.

Program 14–2 `cciEX`: File Conversion with Extended I/O

```
/* Chapter 14. cciEX.
      EXTENDED simplified Caesar cipher conversion. */
/*  cciEX file1 file2  */
/* This program illustrates Extended (a.k.a.
   alterable or completion routine asynch) I/O.
   It was developed by restructuring cciOV. */

#include "Everything.h"

#define MAX_OVRLP 4
#define REC_SIZE 0x8000 /* Size is not as important as with cciOV */

static VOID WINAPI ReadDone (DWORD, DWORD, LPOVERLAPPED);
static VOID WINAPI WriteDone (DWORD, DWORD, LPOVERLAPPED);

/* The first overlapped structure is for reading,
   and the second is for writing. Structures and buffers are
   provided for each outstanding operation */
OVERLAPPED overLapIn[MAX_OVRLP], overLapOut[MAX_OVRLP];
CHAR rawRec[MAX_OVRLP][REC_SIZE], cciRec[MAX_OVRLP][REC_SIZE];
HANDLE hInputFile, hOutputFile;
LONGLONG nRecords, nDone;
LARGE_INTEGER fileSize;
DWORD shift;

int _tmain (int argc, LPTSTR argv[])
{
   DWORD ic;
   LARGE_INTEGER curPosIn;

   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 to process.
      There may not be a partial record at the end. */
   GetFileSizeEx (hInputFile, &fileSize);
   nRecords = (fileSize.QuadPart + REC_SIZE - 1) / REC_SIZE;

   /* Initiate read on buffer paired with an overlapped structure*/
   curPosIn.QuadPart = 0;
   for (ic = 0; ic < MAX_OVRLP; ic++) {
      overLapIn[ic].hEvent = (HANDLE)ic;/* Overload event field to */
      overLapOut[ic].hEvent = (HANDLE)ic;/* hold the event number. */
      /* Set file position. */
```

```
        overLapIn[ic].Offset = curPosIn.LowPart;
        overLapIn[ic].OffsetHigh = curPosIn.HighPart;
        if (curPosIn.QuadPart < fileSize.QuadPart)
           ReadFileEx (hInputFile, rawRec[ic], REC_SIZE,
                 &overLapIn[ic], ReadDone);
        curPosIn.QuadPart += (LONGLONG) REC_SIZE;
     }

     /*  All read operations are running. Enter an alterable
        wait state; continue until all records have been processed. */
     nDone = 0;
     while (nDone < 2* nRecords) {
        SleepEx (0, TRUE);
     }

     CloseHandle (hInputFile); CloseHandle (hOutputFile);
     return 0;
}

static VOID WINAPI ReadDone (DWORD Code, DWORD nBytes,
                             LPOVERLAPPED pOv)
{
     LARGE_INTEGER curPosIn, curPosOut;
     DWORD i, ic; /* Represents value from event field */

     nDone++;
     /* Process the record and initiate the write. */
     /* Get the overlapped structure ID from the event field. */
     ic = PtrToInt(pOv->hEvent);
     curPosIn.LowPart = overLapIn[ic].Offset;
     curPosIn.HighPart = overLapIn[ic].OffsetHigh;
     curPosOut.QuadPart = (curPosIn.QuadPart / REC_SIZE) * REC_SIZE;
     overLapOut[ic].Offset = curPosOut.LowPart;
     overLapOut[ic].OffsetHigh = curPosOut.HighPart;

     /* Encrypt the characters */
     for (i = 0; i < nBytes; i++)
        cciRec[ic][i] = (rawRec[ic][i] + shift) % 256;
     WriteFileEx (hOutputFile, cciRec[ic], nBytes,
        &overLapOut[ic], WriteDone);

     /* Prepare the input overlapped structure
        for the next read, which will be initiated
        after the write, issued above, completes. */
     curPosIn.QuadPart += REC_SIZE * (LONGLONG) (MAX_OVRLP);
     overLapIn[ic].Offset = curPosIn.LowPart;
     overLapIn[ic].OffsetHigh = curPosIn.HighPart;

     return;
}
```

```
static VOID WINAPI WriteDone (DWORD Code, DWORD nBytes,
                              LPOVERLAPPED pOv)
{
   LARGE_INTEGER curPosIn;
   DWORD ic;
   nDone++;
   /* Get the overlapped structure ID from the event field. */
   ic = PtrToInt(pOv->hEvent);

   /* Start the read. The file position was already set in the input
      overlapped structure. Check first, however, to assure not to
      read past the end of file. */
   curPosIn.LowPart = overLapIn[ic].Offset;
   curPosIn.HighPart = overLapIn[ic].OffsetHigh;
   if (curPosIn.QuadPart < fileSize.QuadPart) {
      /* Start a new read. */
      ReadFileEx (hInputFile, rawRec[ic], REC_SIZE,
            &overLapIn[ic], ReadDone);
   }
   return;
}
```



Run 14–2 `cciEX`**:** Overlapped I/O with Completion Routines

Run 14–2 and Appendix C show that `cciEX` performs competitively with `cciOV`, which was slower than the memory-mapped version on the tested four-processor Windows Vista computer. Based on these results, asynchronous overlapped I/O is a good choice for sequential and possibly other file I/O but does not compete with memory-mapped I/O. The choice between the overlapped and extended overlapped I/O is somewhat a matter of taste (I found `cciEX` slightly easier to write and debug).

Asynchronous I/O with Threads

Overlapped and extended I/O achieve asynchronous I/O within a single user thread. These techniques are common, in one form or another, in many older OSs for supporting limited forms of asynchronous operation in single-threaded systems.

Windows, however, supports threads, so the same functional effect is possible by performing synchronous I/O operations in multiple, separate threads, at the possible performance cost due to thread management overhead. Threads also provide a uniform and, arguably, much simpler way to perform asynchronous I/O. An alternative to Programs 14–1 and 14–2 is to give each thread its own handle to the file and each thread could synchronously process every fourth record.

The `cciMT` program, not listed here but included in the *Examples* file, illustrates how to use threads in this way. `cciMT` is simpler than the two asynchronous I/O programs because the bookkeeping is less complex. Each thread simply maintains its own buffers on its own stack and performs the read, convert, and write sequence synchronously in a loop. The performance is superior to the results in Run 14–2, so the possible performance impact is not realized. In particular, the 640MB file conversions, which require about 16 seconds for `cciOV` and `cciEX`, require about 10 seconds for `cciMT`, running on the same machine. This is better than the memory-mapped performance (about 12 seconds).

What would happen if we combined memory mapping and multiple threads? `cciMTMM`, also in the *Examples* file, shows even better results, namely about 4 seconds for this case. Appendix C has results for several different machines.

My personal preference is to use threads rather than asynchronous I/O for file processing, and they provide the best performance in most cases. Memory mapping can improve things even more, although it's difficult to recover from I/O errors, as noted in Chapter 5. The programming logic is simpler, but there are the usual thread risks.

There are some important exceptions to this generalization.

- The situation shown earlier in the chapter in which there is only a single outstanding operation and the file handle can be used for synchronization.
- Asynchronous I/O can be canceled. However, with small modifications, `cciMT` can be modified to use overlapped I/O, with the reading or writing thread

waiting on the event immediately after the I/O operation is started. Exercise 14–10 suggests this modification.

- Multithreaded programs have many risks and can be a challenge to get right, as Chapters 7 through 10 describe. A source file, `cciMT_dh.c`, that's in the unzipped *Examples,* documents several pitfalls I encountered while developing `cciMT`; don't try to use this program because it is not correct!
- Asynchronous I/O completion ports, described at the end of this chapter, are useful with servers.
- NT6 executes asynchronous I/O programs very efficiently compared to normal file I/O (see Run 14–1 and Appendix C).

Waitable Timers

Windows supports waitable timers, a type of waitable kernel object.

You can always create your own timing signal using a timing thread that sets an event after waking from a `Sleep` call. `serverNP` (Program 11–3) also uses a timing thread to broadcast its pipe name periodically. Therefore, waitable timers are a redundant but useful way to perform tasks periodically or at specified absolute or relative times.

As the name suggests, you can wait for a waitable timer to be signaled, but you can also use a callback routine similar to the extended I/O completion routines. A waitable timer can be either a *synchronization timer* or a *manual-reset (or notification) timer.* Synchronization and manual-reset timers are comparable to auto-reset and manual-reset events; a synchronization timer becomes unsignaled after a wait completes on it, and a manual-reset timer must be reset explicitly. In summary:

- There are two ways to be notified that a timer is signaled: either wait on the timer or have a callback routine.
- There are two waitable timer types, which differ in whether or not the timer is reset automatically after a wait.

The first step is to create a timer handle with `CreateWaitableTimer`.

```
HANDLE CreateWaitableTimer (
   LPSECURITY_ATTRIBUTES lpTimerAttributes,
   BOOL bManualReset,
   LPCTSTR lpTimerName);
```

The second parameter, bManualReset, determines whether the timer is a synchronization timer or a manual-reset notification timer. Program 14–3 uses a synchronization timer, but you can change the comment and the parameter setting to obtain a notification timer. Notice that there is also an OpenWaitableTimer function that can use the optional name supplied in the third argument.

The timer is initially inactive, but SetWaitableTimer activates it, sets the timer to unsignaled, and specifies the initial signal time and the time between periodic signals.

```
BOOL SetWaitableTimer (
   HANDLE hTimer,
   const LARGE_INTEGER *pDueTime,
   LONG lPeriod,
   PTIMERAPCROUTINE pfnCompletionRoutine,
   LPVOID lpArgToCompletionRoutine,
   BOOL fResume);
```

hTimer is a valid timer handle created using CreateWaitableTimer.

The second parameter, pointed to by pDueTime, is either a positive absolute time or a negative relative time and is actually expressed as a FILETIME with a resolution of 100 nanoseconds. FILETIME variables were introduced in Chapter 3 and were used in Chapter 6's timep (Program 6–2).

The third parameter specifies the interval between signals, using millisecond units. If this value is 0, the timer is signaled only once. A positive value indicates that the timer is a periodic timer and continues signaling periodically until you call CancelWaitableTimer. Negative interval values are not allowed.

pfnCompletionRoutine, the fourth parameter, specifies the time-out callback function (completion routine) to be called when the timer is signaled *and* the thread enters an alertable wait state. The routine is called with the pointer specified in the fifth parameter, lpArgToCompletionRoutine, as an argument.

Having set a synchronization timer, you can now call SleepEx or other alertable wait function to enter an alertable wait state, allowing the completion routine to be called. Alternatively, wait on the timer handle. As mentioned previously, a manual-reset waitable timer handle will remain signaled until the next call to SetWaitableTimer, whereas Windows resets a synchronization timer immediately after the first wait after the set.

The complete version of Program 14–3 in the *Examples* file allows you to experiment with using the four combinations of the two timer types and with choosing between using a completion routine or waiting on the timer handle.

The final parameter, `fResume`, is concerned with power conservation. See the MSDN documentation for more information.

Use `CancelWaitableTimer` to cancel the last effect of a previous `SetWaitableTimer`, although it will not change the timer's signaled state; use another `SetWaitableTimer` call to do that.

Example: Using a Waitable Timer

Program 14–3 shows how to use a waitable timer to signal the user periodically.

Program 14–3 `TimeBeep`: A Periodic Signal

```
/* Chapter 14 -  TimeBeep. Periodic alarm. */
/* Usage: TimeBeep period (in ms). */
/* This implementation uses kernel object "waitable timers".
   This program uses a console control handler to catch
   control C signals; see the end of Chapter 7.  */

#include "Everything.h"

static BOOL WINAPI Handler (DWORD CntrlEvent);
static VOID APIENTRY Beeper (LPVOID, DWORD, DWORD);
volatile static LONG exitFlag = 0;

HANDLE hTimer;

int _tmain (int argc, LPTSTR argv [])
{
   DWORD count = 0, period;
   LARGE_INTEGER dueTime;

   if (argc >= 2) period = _ttoi (argv [1]) * 1000;

   SetConsoleCtrlHandler (Handler, TRUE);

   dueTime.QuadPart = -(LONGLONG)period * 10000;
          /*  Due time is negative for first time-out relative to
              current time. Period is in ms (10**-3 sec) whereas
              the due time is in 100 ns (10**-7 sec) units to be
              consistent with a FILETIME. */

   hTimer = CreateWaitableTimer (NULL,
      FALSE /*TRUE*/,/* Not manual reset ("notification") timer, but
                        a "synchronization timer." */
      NULL);
```

```
    SetWaitableTimer (hTimer,
           &dueTime /* relative time of first signal. Positive value
                       would indicate an absolute time. */,
           period  /* Time period in ms */,
           Beeper  /* Timer function */,
           &count  /* Parameter passed to timer function */,
           FALSE);

    /* Enter the main loop */
    while (!exitFlag) {
        _tprintf (_T("count = %d\n"), count);
        /* count is increased in the timer routine */
        /*  Enter an alertable wait state, enabling the timer routine.
            The timer handle is a synchronization object, so you can
            also wait on it. */
        SleepEx (INFINITE, TRUE);
        /* or WaitForSingleObjectEx (hTimer, INFINITE); Beeper(...); */
    }

    _tprintf (_T("Shut down. count = %d"), count);
    CancelWaitableTimer (hTimer);
    CloseHandle (hTimer);
    return 0;
}

/* Waitable timer callback function */
static VOID APIENTRY Beeper (LPVOID lpCount,
   DWORD dwTimerLowValue, DWORD dwTimerHighValue)
{

    *(LPDWORD)lpCount = *(LPDWORD)lpCount + 1;

    _tprintf (_T("About to perform beep number: %d\n"),
           *(LPDWORD)lpCount);
    Beep (1000 /* Frequency */, 250 /* Duration (ms) */);
    return;
}

BOOL WINAPI Handler (DWORD CntrlEvent)
{
    InterlockedIncrement(&exitFlag);
    _tprintf (_T("Shutting Down\n"));

    return TRUE;
}
```