

# Re: newbie: I/O with nasm

---

*Source:* <http://coding.derkeiler.com/Archive/Assembler/alt.lang.asm/2006-03/msg00252.html>

---

- *From:* Herbert Kleebauer <[klee@xxxxxxxxxx](mailto:klee@xxxxxxxxxx)>
  - *Date:* Wed, 08 Mar 2006 16:08:14 +0100
- 

Frank Kotler wrote:

ArarghMail603NOSPAM@xxxxxxxxxxxxxxxxxxxxx wrote:

Windows Readfile & WriteFile (from MSDN):

```
BOOL ReadFile(  
HANDLE hFile, // handle to file  
LPVOID lpBuffer, // data buffer  
DWORD nNumberOfBytesToRead, // number of bytes to read  
LPDWORD lpNumberOfBytesRead, // number of bytes read  
LPOVERLAPPED lpOverlapped // overlapped buffer  
);
```

```
BOOL WriteFile(  
HANDLE hFile, // handle to file  
LPCVOID lpBuffer, // data buffer  
DWORD nNumberOfBytesToWrite, // number of bytes to write  
LPDWORD lpNumberOfBytesWritten, // number of bytes written  
LPOVERLAPPED lpOverlapped // overlapped buffer  
);  
<snip>
```

There we go! Always nice to hear from somebody who's not afraid to RTFM!  
Myself, between "too lazy" and "too down on MS", I haven't learned much  
Windows code.

Now, if Windows is what TK is looking for, we've got something to go on.

But for a start I think it is better to use a simple `getc` (return the  
read character (or -1 if EOF) in `eax`) and `putc` (write character in `al`)  
for a simple IO to `stdin/stdout` (based on `ReadFile` and `WriteFile`, but  
hiding the internals):

```
004010f7: 50 putc: move.l r0,-(sp)  
004010f8: a2 00401157 move.b r0,_buf
```

Re: newbie: I/O with nasm

```
004010fd: 31 c0 eor.l r0,r0
004010ff: 03 05 00401164 add.l _handle,r0
00401105: 75 0d bne.b _10
00401107: 6a f5 moveq.l #-11,-(sp)
00401109: ff 15 0040100c jsr.l (GetStdHandle)
0040110f: a3 00401164 move.l r0,_handle
00401114: 6a 00 _10: moveq.l #0,-(sp)
00401116: 68 00401168 move.l #_count,-(sp)
0040111b: 6a 01 moveq.l #1,-(sp)
0040111d: 68 00401157 move.l #_buf,-(sp)
00401122: 50 move.l r0,-(sp)
00401123: ff 15 00401014 jsr.l (WriteFile)
00401129: 09 c0 or.l r0,r0
0040112b: 75 1c bne.b _20
0040112d: 6a 00 _30: moveq.l #0,-(sp)
0040112f: 68 00401158 move.l #_text,-(sp)
00401134: 68 00401158 move.l #_text,-(sp)
00401139: 6a 00 moveq.l #0,-(sp)
0040113b: ff 15 00401000 jsr.l (MessageBoxA)
00401141: 6a 00 moveq.l #0,-(sp)
00401143: ff 15 00401008 jsr.l (ExitProcess)
00401149: 81 3d 00401168
0040114f: 00000001 _20: cmp.l #1,_count
00401153: 75 d8 bne.b _30
00401155: 58 move.l (sp)+,r0
00401156: c3 rts.l

00401157: 00 _buf: dc.b 0
00401158: 77 72 69 74 65 20
0040115e: 65 72 72 6f 72 00 _text: dc.b 'write error',0
even4
00401164: 00000000 _handle:dc.l 0
00401168: 00000000 _count: dc.l 0

0040116c: 31 c0 getc: eor.l r0,r0
0040116e: 03 05 004011e0 add.l _handle,r0
00401174: 75 0d bne.b _10
00401176: 6a f6 moveq.l #-10,-(sp)
00401178: ff 15 0040100c jsr.l (GetStdHandle)
0040117e: a3 004011e0 move.l r0,_handle
00401183: 6a 00 _10: moveq.l #0,-(sp)
00401185: 68 004011e4 move.l #_count,-(sp)
0040118a: 6a 01 moveq.l #1,-(sp)
0040118c: 68 004011d1 move.l #_buf,-(sp)
00401191: 50 move.l r0,-(sp)
00401192: ff 15 00401010 jsr.l (ReadFile)
00401198: 09 c0 or.l r0,r0
0040119a: 75 1c bne.b _20
0040119c: 6a 00 moveq.l #0,-(sp)
0040119e: 68 004011d2 move.l #_text,-(sp)
```

Re: newbie: I/O with nasm

```
004011a3: 68 004011d2 move.l #_text,-(sp)
004011a8: 6a 00 moveq.l #0,-(sp)
004011aa: ff 15 00401000 jsr.l (MessageBoxA)
004011b0: 6a 00 moveq.l #0,-(sp)
004011b2: ff 15 00401008 jsr.l (ExitProcess)
004011b8: 0f b6 05 004011d1 _20: movu.bl _buf,r0
004011bf: 81 3d 004011e4
004011c5: 00000001 cmp.l #1,_count
004011c9: 74 05 beq.b _30
004011cb: b8 ffffffff move.l #-1,r0
004011d0: c3 _30: rts.l

004011d1: 00 _buf: dc.b 0
004011d2: 72 65 61 64 20 65
004011d8: 72 72 6f 72 00 _text: dc.b 'read error',0
004011dd: 00 00 00 even4
004011e0: 00000000 _handle:dc.l 0
004011e4: 00000000 _count: dc.l 0
```

I don't know what "overlapped buffer" means.

That wouldn't be necessary to know if people would DOS to learn assembly programming!

```
=====
= ReadFile =
=====
```

The ReadFile function reads data from a file, starting at the position indicated by the file pointer. After the read operation has been completed, the file pointer is adjusted by the number of bytes actually read, unless the file handle is created with the overlapped attribute. If the file handle is created for overlapped input and output (I/O), the application must adjust the position of the file pointer after the read operation.

This function is designed for both synchronous and asynchronous operation. The ReadFileEx function is designed solely for asynchronous operation. It lets an application perform other processing during a file read operation.

```
BOOL ReadFile(
HANDLE hFile,
LPVOID lpBuffer,
DWORD nNumberOfBytesToRead,
LPDWORD lpNumberOfBytesRead,
LPOVERLAPPED lpOverlapped
);
```

## Parameters

=====

### hFile [in]

-----

Handle to the file to be read. The file handle must have been created with the `GENERIC_READ` access right. For more information, see [File Security and Access Rights](#).

For asynchronous read operations, hFile can be any handle opened with the `FILE_FLAG_OVERLAPPED` flag by the `CreateFile` function, or a socket handle returned by the `socket` or `accept` function.

Windows Me/98/95: For asynchronous read operations, hFile can be a communications resource opened with the `FILE_FLAG_OVERLAPPED` flag by `CreateFile`, or a socket handle returned by `socket` or `accept`. You cannot perform asynchronous read operations on mailslots, named pipes, or disk files.

### lpBuffer [out]

-----

Pointer to the buffer that receives the data read from the file.

### nNumberOfBytesToRead [in]

-----

Number of bytes to be read from the file.

### lpNumberOfBytesRead [out]

-----

Pointer to the variable that receives the number of bytes read. `ReadFile` sets this value to zero before doing any work or error checking. If this parameter is zero when `ReadFile` returns `TRUE` on a named pipe, the other end of the message-mode pipe called the `WriteFile` function with `nNumberOfBytesToWrite` set to zero.

If `lpOverlapped` is `NULL`, `lpNumberOfBytesRead` cannot be `NULL`. If `lpOverlapped` is not `NULL`, `lpNumberOfBytesRead` can be `NULL`. If this is an overlapped read operation, you can get the number of bytes read by calling `GetOverlappedResult`. If hFile is associated with an I/O completion port, you can get the number of bytes read by calling `GetQueuedCompletionStatus`.

If I/O completion ports are used and you are using a callback routine to free the memory allocated to the `OVERLAPPED` structure pointed to by the `lpOverlapped` parameter, specify `NULL` as the value of this parameter to avoid a memory corruption problem during the deallocation. This memory corruption problem will cause an invalid number of bytes to be returned in this parameter.

Windows Me/98/95: This parameter cannot be NULL.

#### lpOverlapped [in]

-----  
Pointer to an OVERLAPPED structure. This structure is required if hFile was created with FILE\_FLAG\_OVERLAPPED.

If hFile was opened with FILE\_FLAG\_OVERLAPPED, the lpOverlapped parameter must not be NULL. It must point to a valid OVERLAPPED structure. If hFile was created with FILE\_FLAG\_OVERLAPPED and lpOverlapped is NULL, the function can incorrectly report that the read operation is complete.

If hFile was opened with FILE\_FLAG\_OVERLAPPED and lpOverlapped is not NULL, the read operation starts at the offset specified in the OVERLAPPED structure and ReadFile may return before the read operation has been completed. In this case, ReadFile returns FALSE and the GetLastError function returns ERROR\_IO\_PENDING. This allows the calling process to continue while the read operation finishes. The event specified in the OVERLAPPED structure is set to the signaled state upon completion of the read operation.

If hFile was not opened with FILE\_FLAG\_OVERLAPPED and lpOverlapped is NULL, the read operation starts at the current file position and ReadFile does not return until the operation has been completed.

If hFile is not opened with FILE\_FLAG\_OVERLAPPED and lpOverlapped is not NULL, the read operation starts at the offset specified in the OVERLAPPED structure. ReadFile does not return until the read operation has been completed.

Windows 95/98/Me: For operations on files, disks, pipes, or mailslots, this parameter must be NULL; a pointer to an OVERLAPPED structure causes the call to fail. However, you can perform overlapped I/O on serial and parallel ports.

#### Return Values

=====

The ReadFile function returns when one of the following conditions is met: a write operation completes on the write end of the pipe, the number of bytes requested has been read, or an error occurs.

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.

If the return value is nonzero and the number of bytes read is zero, the

## Re: newbie: I/O with nasm

file pointer was beyond the current end of the file at the time of the read operation. However, if the file was opened with `FILE_FLAG_OVERLAPPED` and `lpOverlapped` is not `NULL`, the return value is zero and `GetLastError` returns `ERROR_HANDLE_EOF` when the file pointer goes beyond the current end of file.

### Remarks

=====

If part of the file is locked by another process and the read operation overlaps the locked portion, this function fails.

An application must meet certain requirements when working with files opened with `FILE_FLAG_NO_BUFFERING`:

- \* File access must begin at byte offsets within the file that are integer multiples of the volume's sector size. To determine a volume's sector size, call the `GetDiskFreeSpace` function.
- \* File access must be for numbers of bytes that are integer multiples of the volume's sector size. For example, if the sector size is 512 bytes, an application can request reads and writes of 512, 1024, or 2048 bytes, but not of 335, 981, or 7171 bytes.
- \* Buffer addresses for read and write operations must be sector aligned (aligned on addresses in memory that are integer multiples of the volume's sector size). One way to sector align buffers is to use the `VirtualAlloc` function to allocate the buffers. This function allocates memory that is aligned on addresses that are integer multiples of the system's page size. Because both page and volume sector sizes are powers of 2, memory aligned by multiples of the system's page size is also aligned by multiples of the volume's sector size.

Accessing the input buffer while a read operation is using the buffer may lead to corruption of the data read into that buffer. Applications must not read from, write to, reallocate, or free the input buffer that a read operation is using until the read operation completes.

Characters can be read from the console input buffer by using `ReadFile` with a handle to console input. The console mode determines the exact behavior of the `ReadFile` function.

If a named pipe is being read in message mode and the next message is longer than the `nNumberOfBytesToRead` parameter specifies, `ReadFile` returns `FALSE` and `GetLastError` returns `ERROR_MORE_DATA`. The remainder of the message may be read by a subsequent call to the `ReadFile` or `PeekNamedPipe` function.

When reading from a communications device, the behavior of `ReadFile` is governed by the current communication time-outs as set and retrieved using the `SetCommTimeouts` and `GetCommTimeouts` functions. Unpredictable results can occur if you fail to set the time-out values. For more

information about communication time-outs, see COMMTIMEOUTS.

If ReadFile attempts to read from a mailslot whose buffer is too small, the function returns FALSE and GetLastError returns ERROR\_INSUFFICIENT\_BUFFER.

If the anonymous write pipe handle has been closed and ReadFile attempts to read using the corresponding anonymous read pipe handle, the function returns FALSE and GetLastError returns ERROR\_BROKEN\_PIPE.

The ReadFile function may fail and return ERROR\_INVALID\_USER\_BUFFER or ERROR\_NOT\_ENOUGH\_MEMORY whenever there are too many outstanding asynchronous I/O requests.

The ReadFile code to check for the end-of-file condition (eof) differs for synchronous and asynchronous read operations.

When a synchronous read operation reaches the end of a file, ReadFile returns TRUE and sets \*lpNumberOfBytesRead to zero. The following sample code tests for end-of-file for a synchronous read operation:

```
// Attempt a synchronous read operation.
bResult = ReadFile(hFile, &inBuffer, nBytesToRead, &nBytesRead, NULL) ;
// Check for end of file.
if (bResult && nBytesRead == 0, )
{
// we're at the end of the file
}
```

An asynchronous read operation can encounter the end of a file during the initiating call to ReadFile, or during subsequent asynchronous operation.

If EOF is detected at ReadFile time for an asynchronous read operation, ReadFile returns FALSE and GetLastError returns ERROR\_HANDLE\_EOF.

If EOF is detected during subsequent asynchronous operation, the call to GetOverlappedResult to obtain the results of that operation returns FALSE and GetLastError returns ERROR\_HANDLE\_EOF.

To cancel all pending asynchronous I/O operations, use the CancelIo function. This function only cancels operations issued by the calling thread for the specified file handle. I/O operations that are canceled complete with the error ERROR\_OPERATION\_ABORTED.

If you are attempting to read from a floppy drive that does not have a floppy disk, the system displays a message box prompting the user to retry the operation. To prevent the system from displaying this message box, call the SetErrorMode function with SEM\_NOOPENFILEERRORBOX.

The following sample code illustrates testing for end-of-file for an asynchronous read operation:

```
// set up overlapped structure fields
gOverLapped.Offset = 0;
gOverLapped.OffsetHigh = 0;
gOverLapped.hEvent = hEvent;

// attempt an asynchronous read operation
bResult = ReadFile(hFile, &inBuffer, nBytesToRead, &nBytesRead,
&gOverlapped) ;

// if there was a problem, or the async. operation's still pending ...
if (!bResult)
{
// deal with the error code
switch (dwError = GetLastError())
{
case ERROR_HANDLE_EOF:
{
// we have reached the end of the file
// during the call to ReadFile

// code to handle that
}

case ERROR_IO_PENDING:
{
// asynchronous i/o is still in progress

// do something else for a while
GoDoSomethingElse() ;

// check on the results of the asynchronous read
bResult = GetOverlappedResult(hFile, &gOverlapped,
&nBytesRead, FALSE) ;

// if there was a problem ...
if (!bResult)
{
// deal with the error code
switch (dwError = GetLastError())
{
case ERROR_HANDLE_EOF:
{
// we have reached the end of
// the file during asynchronous
// operation
}

// deal with other error cases
} //end switch (dwError = GetLastError()) }
} // end case
```

```
// deal with other error cases

} // end switch (dwError = GetLastError())
} // end if
```

Example Code

=====

For an example, see Reading and Writing Asynchronously.

Requirements

=====

Client: Included in Windows XP, Windows 2000 Professional, and Windows NT Workstation, Windows Me, Windows 98, and Windows 95.  
 Server: Included in Windows Server 2003, Windows 2000 Server, and Windows NT Server.  
 Header: Declared in Winbase.h; include Windows.h.  
 Library: Use Kernel32.lib.

See Also

=====

File Management Functions, CancelIo, CreateFile, GetCommTimeouts, GetOverlappedResult, GetQueuedCompletionStatus, OVERLAPPED, PeekNamedPipe, ReadFileEx, SetCommTimeouts, SetErrorMode, WriteFile

=====

= WriteFile =

=====

The WriteFile function writes data to a file and is designed for both synchronous and asynchronous operation. The function starts writing data to the file at the position indicated by the file pointer. After the write operation has been completed, the file pointer is adjusted by the number of bytes actually written, except when the file is opened with FILE\_FLAG\_OVERLAPPED. If the file handle was created for overlapped input and output (I/O), the application must adjust the position of the file pointer after the write operation is finished.

This function is designed for both synchronous and asynchronous operation. The WriteFileEx function is designed solely for asynchronous operation. It lets an application perform other processing during a file write operation.

```
BOOL WriteFile(
HANDLE hFile,
```

```
LPCVOID lpBuffer,  
DWORD nNumberOfBytesToWrite,  
LPDWORD lpNumberOfBytesWritten,  
LPOVERLAPPED lpOverlapped  
);
```

#### Parameters

=====

#### hFile [in]

-----

Handle to the file. The file handle must have been created with the `GENERIC_WRITE` access right. For more information, see File Security and Access Rights.

For asynchronous write operations, `hFile` can be any handle opened with the `FILE_FLAG_OVERLAPPED` flag by the `CreateFile` function, or a socket handle returned by the socket or accept function.

Windows Me/98/95: For asynchronous write operations, `hFile` can be a communications resource opened with the `FILE_FLAG_OVERLAPPED` flag by `CreateFile`, or a socket handle returned by socket or accept. You cannot perform asynchronous write operations on mailslots, named pipes, or disk files.

#### lpBuffer [in]

-----

Pointer to the buffer containing the data to be written to the file.

#### nNumberOfBytesToWrite [in]

-----

Number of bytes to be written to the file.

A value of zero specifies a null write operation. The behavior of a null write operation depends on the underlying file system. To truncate or extend a file, use the `SetEndOfFile` function.

Named pipe write operations across a network are limited to 65,535 bytes.

#### lpNumberOfBytesWritten [out]

-----

Pointer to the variable that receives the number of bytes written. `WriteFile` sets this value to zero before doing any work or error checking.

If `lpOverlapped` is `NULL`, `lpNumberOfBytesWritten` cannot be `NULL`. If `lpOverlapped` is not `NULL`, `lpNumberOfBytesWritten` can be `NULL`. If this is an overlapped write operation, you can get the number of bytes written by calling `GetOverlappedResult`. If `hFile` is associated with an I/O

completion port, you can get the number of bytes written by calling `GetQueuedCompletionStatus`.

If I/O completion ports are used and you are using a callback routine to free the memory allocated to the `OVERLAPPED` structure pointed to by the `lpOverlapped` parameter, specify `NULL` as the value of this parameter to avoid a memory corruption problem during the deallocation. This memory corruption problem will cause an invalid number of bytes to be returned in this parameter.

Windows Me/98/95: This parameter cannot be `NULL`.

#### `lpOverlapped` [in]

Pointer to an `OVERLAPPED` structure. This structure is required if `hFile` was opened with `FILE_FLAG_OVERLAPPED`.

If `hFile` was opened with `FILE_FLAG_OVERLAPPED`, the `lpOverlapped` parameter must not be `NULL`. It must point to a valid `OVERLAPPED` structure. If `hFile` was opened with `FILE_FLAG_OVERLAPPED` and `lpOverlapped` is `NULL`, the function can incorrectly report that the write operation is complete.

If `hFile` was opened with `FILE_FLAG_OVERLAPPED` and `lpOverlapped` is not `NULL`, the write operation starts at the offset specified in the `OVERLAPPED` structure and `WriteFile` may return before the write operation has been completed. In this case, `WriteFile` returns `FALSE` and the `GetLastError` function returns `ERROR_IO_PENDING`. This allows the calling process to continue processing while the write operation is being completed. The event specified in the `OVERLAPPED` structure is set to the signaled state upon completion of the write operation.

If `hFile` was not opened with `FILE_FLAG_OVERLAPPED` and `lpOverlapped` is `NULL`, the write operation starts at the current file position and `WriteFile` does not return until the operation has been completed.

`WriteFile` resets the event specified by the `hEvent` member of the `OVERLAPPED` structure to a nonsignaled state when it begins the I/O operation. Therefore, there is no need for the caller to do so.

If `hFile` was not opened with `FILE_FLAG_OVERLAPPED` and `lpOverlapped` is not `NULL`, the write operation starts at the offset specified in the `OVERLAPPED` structure and `WriteFile` does not return until the write operation has been completed.

Windows Me/98/95: For operations on files, disks, pipes, or mailslots, this parameter must be `NULL`; a pointer to an `OVERLAPPED` structure causes the call to fail. However, the function supports overlapped I/O on serial and parallel ports.

## Return Values

=====

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

=====

Note that the time stamps may not be updated correctly for a remote file. To ensure consistent results, use unbuffered I/O. An application must meet certain requirements when working with files opened with FILE\_FLAG\_NO\_BUFFERING:

- \* File access must begin at byte offsets within the file that are integer multiples of the volume's sector size. To determine a volume's sector size, call the GetDiskFreeSpace function.
- \* File access must be for numbers of bytes that are integer multiples of the volume's sector size. For example, if the sector size is 512 bytes, an application can request reads and writes of 512, 1024, or 2048 bytes, but not of 335, 981, or 7171 bytes.
- \* Buffer addresses for read and write operations must be sector aligned (aligned on addresses in memory that are integer multiples of the volume's sector size). One way to sector align buffers is to use the VirtualAlloc function to allocate the buffers. This function allocates memory that is aligned on addresses that are integer multiples of the system's page size. Because both page and volume sector sizes are powers of 2, memory aligned by multiples of the system's page size is also aligned by multiples of the volume's sector size.

If part of the file is locked by another process and the write operation overlaps the locked portion, this function fails.

Accessing the output buffer while a write operation is using the buffer may lead to corruption of the data written from that buffer. Applications must not write to, reallocate, or free the output buffer that a write operation is using until the write operation completes.

Characters can be written to the screen buffer using WriteFile with a handle to console output. The exact behavior of the function is determined by the console mode. The data is written to the current cursor position. The cursor position is updated after the write operation.

The system interprets zero bytes to write as specifying a null write operation and WriteFile does not truncate or extend the file. To truncate or extend a file, use the SetEndOfFile function.

When writing to a nonblocking, byte-mode pipe handle with insufficient

buffer space, WriteFile returns TRUE with  
\*lpNumberOfBytesWritten < nNumberOfBytesToWrite.

When an application uses the WriteFile function to write to a pipe, the write operation may not finish if the pipe buffer is full. The write operation is completed when a read operation (using the ReadFile function) makes more buffer space available.

If the anonymous read pipe handle has been closed and WriteFile attempts to write using the corresponding anonymous write pipe handle, the function returns FALSE and GetLastError returns ERROR\_BROKEN\_PIPE.

The WriteFile function may fail with ERROR\_INVALID\_USER\_BUFFER or ERROR\_NOT\_ENOUGH\_MEMORY whenever there are too many outstanding asynchronous I/O requests.

To cancel all pending asynchronous I/O operations, use the CancelIo function. This function only cancels operations issued by the calling thread for the specified file handle. I/O operations that are canceled complete with the error ERROR\_OPERATION\_ABORTED.

If you are attempting to write to a floppy drive that does not have a floppy disk, the system displays a message box prompting the user to retry the operation. To prevent the system from displaying this message box, call the SetErrorMode function with SEM\_NOOPENFILEERRORBOX.

#### Example Code

=====

For an example, see Reading and Writing Asynchronously.

#### Requirements

=====

Client: Included in Windows XP, Windows 2000 Professional, and Windows NT Workstation, Windows Me, Windows 98, and Windows 95.

Server: Included in Windows Server 2003, Windows 2000 Server, and Windows NT Server.

Header: Declared in Winbase.h; include Windows.h.

Library: Use Kernel32.lib.

#### See Also

=====

File Management Functions, CancelIo, CreateFile, GetLastError, GetOverlappedResult, GetQueuedCompletionStatus, OVERLAPPED, ReadFile, SetEndOfFile, SetErrorMode, WriteFileEx

.