Chapter 12: Input/Output

Programs in previous chapters have used the *input* macro to input data from the PC console keyboard and the *output* macro to output data to the console display. Input and output from an assembly language program have been limited to the keyboard and the monitor. This chapter examines the underlying operating system calls that are used by the *input* and *output* macros. It then examines similar operating system calls that make it possible to read and write sequential files to secondary storage. Next it looks at the 80x86 instructions that actually do input and output and discusses alternative I/O schemes, including memory-mapped and interrupt-driven I/O.

Chapter 12: Input/Output
12.1 Console I/O Using the Kernel32 Library

Figure 12.1 shows a simple example illustrating how kernel32 functions can write a simple message.

This example is similar to many of those seen previously in the book in its overall structure.

However, it is missing the "standard" directive INCLUDE io.h. In addition to the familiar prototype for the *ExitProcess* function, it contains two new function prototypes. These functions are needed to write to the console.

Chapter 12: Input/Output
12.1 Console I/O Using the Kernel32 Library

The Windows 95/98/NT operating systems are similar to many others in that they treat input/output devices and disk files in a uniform manner. Note that in Fig. 12.1, a *WriteFile* call is used to display a message on the console. This same function can be used to write to a disk file. The device or file used for I/O is identified by its *handle*, a doubleword value in an assembly language program. The handle value must be obtained before the *WriteFile* call is made. There is more than one way to do this for a console file; *GetStandardHandle* provides an easy method.

Chapter 12: Input/Output
12.1 Console I/O Using the Kernel32 Library

The Windows 95/98/NT operating systems are similar to many others in that they treat input/output devices and disk files in a uniform manner.

Note that in Fig. 12.1, a *WriteFile* call is used to display a message on the console. This same function can be used to write to a disk file.

The device or file used for I/O is identified by its *handle*, a doubleword value in an assembly language program.

The handle value must be obtained before the *WriteFile* call is made. There is more than one way to do this for a console file; *GetStandardHandle* provides an easy method.

Chapter 12: Input/Output
12.1 Console I/O Using the Kernel32 Library

Any *GetStdHandle* call has a single parameter; a numeric value, distinct from the handle, indicates the particular device. There are three standard devices:

one for input,

one for output, and

one to report errors (normally the same as the standard output device).

Chapter 12: Input/Output
12.1 Console I/O Using the Kernel32 Library

Each device number is usually equated to a symbol, and these symbols are used in code. We will only use the input and output devices; their numbers and names appear in Fig. 12.2. *GetStdHandle* is a function, returning in EAX a handle for the standard I/O device.

The handle value is usually stored in memory to be available later. In the sample program, the returned value is immediately copied to the doubleword referenced by *hStdOut*.

12.1 Console I/O Using the Kernel32 Library

Mnemonic	Equated Value
STD_INPUT	-10
STD_OUTPUT	-11

Figure 12.2: Standard device numbers

12.1 Console I/O Using the Kernel32 Library

With five parameters, a *WriteFile* call is more complicated:

✓ **The first Parameter is The handle that identifies the file.** This handle is returned by GetStdHandle, not the device number.

✓ **The second parameter is the address of the string.** Note the use of the NEAR32 PTR operator in the example to tell the assembler to use the address of *OldProg* rather than the value stored there.

✓**The third parameter is a doubleword containing the number of bytes to be displayed.**

✓**The forth parameter is used to return a value to the calling program.** This value indicates how many bytes were actually written. In the case of output to the console, this will be the length of the message unless an error occurs.

✓**The fifth parameter will always be 0 in this book's examples.** It can be used to indicate non-sequential access to some files, but we are going to deal only with sequential access.

Chapter 12: Input/Output
12.1 Console I/O Using the Kernel32 Library

Console input is almost as easy as output. Figure 12.3 shows a program that inputs a string of characters, converts each uppercase letter to lowercase, and displays the resulting string.

Chapter 12: Input/Output
12.1 Console I/O Using the Kernel32 Library

The new function in this example is *Readfile*. It is very similar to *WriteFile* except that the second parameter has the address of an input buffer, the third parameter gives the maximum number of characters to read, and the fourth parameter returns the number of characters actually read.

The number of characters read will normally be smaller than the size of the buffer to receive the characters. If it is larger, values in memory following the input buffer may be destroyed. An additional consideration with console input is that carriage return and linefeed characters are added to the characters that you key in. That is, if you type six characters and then press Enter, eight characters will actually be stored in the input buffer—the six characters plus the carriage return and linefeed.

12.1 Console I/O Using the Kernel32 Library

In the program from Fig. 12.3, there is a blank line of output before the line of lowercase characters, which is because of the CR/LF that is in memory before the input buffer. The starting address for output includes these two additional characters and the character count has been increased by two to include these characters. Because the original character count includes the CR/LF at the end of the characters read in, there will also be a skip to a new line after the characters are displayed.

The *input* and *output* macros that you have used in most of this book expand into procedure calls that use the kernel32 console input/output functions. The relevant portion of the file IO.ASM is shown in Fig. 12.4.

Chapter 12: Input/Output
12.1 Console I/O Using the Kernel32 Library

At this point there is nothing surprising in the input/output code in IO.ASM. It starts with the same directives that appeared in the previous two examples. The data area does not include an input buffer since this will be in the user's calling program. It does have the variable s*trAddr* to locally store the input or output buffer address that is passed as a parameter. The output procedure *outproc* expects this to be the address of a null-terminated string. After standard procedure entry code, it computes the length of that string. It then gets the handle for the console and writes to the console, exactly as in the earlier example in Fig. 12.1.

12.1 Console I/O Using the Kernel32 Library

The input procedure *inproc* is also simple. After standard procedure entry code, it gets the handle for the console and copies the two parameters (length and string address) to local variables. A *ReadFile* call does the actual input. The only complication is that the *inproc* procedure promises a null-terminated string, and the string read by *ReadFile* is terminated by the CR/LF. The code

```
mov     ecx,read
mov     BYTE PTR [esi+ecx-2],0
```

places a null byte at the end of the string, actually replacing the carriage return character by a null. It works because the starting address of the string is in ESI, so that when the character count is put in ECX, ESI+ECX–2 points to the address of the next-to-last character in the input buffer.

Chapter 12: Input/Output
12.1 Console I/O Using the Kernel32 Library

This is an appropriate time to repeat the warning from Section 6.1: some Microsoft operating system functions may require that the stack be doubleword-aligned. When these functions are used in procedures, you must only push doubleword values onto the stack. This is, for instance, why the code in Fig. 12.4 contains a pushfd instruction even though a pushf would save all the flag values that are meaningful to most programs.

Chapter 12: Input/Output
12.2 Sequential File I/O Using the Kernel32 Library

File processing applications generally involve opening the file, reading from or writing to the file, and finally closing the file. At the level of the kernel32 library, opening the file means to obtain a handle for it. Closing the file that has been read may be important to free it up for access by another user. Closing a file that has been written may be necessary to force the operating systems to save the final characters. In this section we investigate how to do some of these operations for sequential disk files. File operations like these are usually more appropriately done using a high-level language, so the primary purpose of this section is to give you a sense of what is "under the hood" of a high-level language.