CHAPTER

2 Using the Windows File System and Character I/O

The file system and simple terminal I/O are often the first OS features that the developer encounters. Early PC OSs such as MS-DOS did little more than manage files and terminal (or *console*) I/O, and these resources are also central features of nearly every OS.

Files are essential for the long-term storage of data and programs. Files are also the simplest form of program-to-program communication. Furthermore, many aspects of the file system model apply to interprocess and network communication.

The file copy programs in Chapter 1 introduced the four essential file processing functions:

`CreateFile`	`WriteFile`
`ReadFile`	`CloseHandle`

This chapter explains these and related functions and also describes character processing and console I/O functions in detail. First, we say a few words about the various file systems available and their principal characteristics. In the process, we'll see how to use Unicode wide characters for internationalization. The chapter includes an introduction to Windows file and directory management.

The Windows File Systems

Windows natively supports four file systems on directly attached devices, but only the first is important throughout the book, as it is Microsoft's primary, full-functionality file system. In addition, file systems are supported on devices such as USB drives. The file system choice on a disk volume or partition is specified when the volume is formatted.

1. The *NT* file system (NTFS) is Microsoft's modern file system that supports long file names, security, fault tolerance, encryption, compression, extended attributes, and very large files[1] and volumes. Note that diskettes, which are now rare, do not support NTFS.
2. The *File Allocation Table* (FAT and FAT32) file systems are rare on current systems and descend from the original MS-DOS and Windows 3.1 FAT (or FAT16) file systems. FAT32 supported larger disk drives and other enhancements, and the term FAT will refer to both versions. FAT does not support Windows security, among other limitations. FAT is the only supported file system for floppy disks and is often the file system on memory cards.
3. The *CD-ROM* file system (CDFS), as the name implies, is for accessing information provided on CD-ROMs. CDFS is compliant with the ISO 9660 standard.
4. The *Universal Disk Format* (UDF), an industry standard, supports DVD drives and will ultimately supplant CDFS. Windows Vista uses the term *Live File System* (LFS) as an enhancement that allows you to add new files and hide, but not actually delete, files.

Windows provides both client and server support for distributed file systems, such as the Networked File System (NFS) and Common Internet File System (CIFS). Windows Server 2003 and 2008 provide extensive support for storage area networks (SANs) and emerging storage technologies. Windows also allows custom file system development.

The file system API accesses all the file systems in the same way, sometimes with limitations. For example, only NTFS supports security. This chapter and the next point out features unique to NTFS as appropriate, but, in general, assume NTFS.

[1] "Very large" and "huge" are relative terms that we'll use to describe a file longer than 4GB, which means that you need to use 64-bit integers to specify the file length and positions in the file.

File Naming

Windows supports hierarchical file naming, but there are a few subtle distinctions for the UNIX user and basic rules for everyone.

- The full pathname of a disk file starts with a drive name, such as `A:` or `C:`. The `A:` and `B:` drives are normally diskette drives, and `C:`, `D:`, and so on are hard disks, DVDs, and other directly attached devices. Network drives are usually designated by letters that fall later in the alphabet, such as `H:` and `K:`.
- Alternatively, a full pathname, or Universal Naming Convention (UNC), can start with a double backslash (\\), indicating the global root, followed by a server name and a *share name* to indicate a path on a network file server. The first part of the pathname, then, is `\\servername\sharename`.
- The pathname *separator* is the backslash (\), although the forward slash (/) works in `CreateFile` and other low-level API pathname parameters. This may be more convenient for C/C++ programmers, although it's best simply to use backslashes to avoid possible incompatibility.
- Directory and file names cannot contain any ASCII characters with a value in the range 1–31 or any of these characters:

  `<   >  :  "   | ? * \ /`

  These characters have meaning on command lines, and their occurrences in file names would complicate command line parsing. Names can contain blanks. However, when using file names with blanks on a command line, put each file name in quotes so that the name is not interpreted as naming two distinct files.
- Directory and file names are *case-insensitive*, but they are also *case-retaining*, so that if the creation name is `MyFile`, the file name will show up as it was created, but the file can also be accessed with the name `myFILE`.
- Normally, file and directory names used as API function arguments can be as many as 255 characters long, and pathnames are limited to `MAX_PATH` characters (currently 260). You can also specify very long names with an escape sequence, which we'll describe later.
- A period (`.`) separates a file's name from its extension, and extensions (usually two to four characters after the rightmost period in the file name) conventionally indicate the file's type. Thus, `cci.EXE` would be an executable file, and `cci.C` would be a C language source file. File names can contain multiple periods.

A single period (`.`) and two periods (`..`), as directory names, indicate the current directory and its parent, respectively.

With this introduction, it is now time to learn more about the Windows functions introduced in Chapter 1.

Opening, Reading, Writing, and Closing Files

The first Windows function described in detail is `CreateFile`, which opens existing files and creates new ones. This and other functions are described first by showing the function prototype and then by describing the parameters and function operation.

Creating and Opening Files

This is the first Windows function, so we'll describe it in detail; later descriptions will frequently be much more streamlined as the Windows conventions become more familiar. This approach will help users understand the basic concepts and use the functions without getting bogged down in details that are available on MSDN.

Furthermore, `CreateFile` is complex with numerous advanced options not described here; we'll generally mention the more important options and sometimes give very brief descriptions of other options that are used in later chapters and examples.

Chapter 1's introductory Windows `cpW` program (Program 1–2) shows a simple use of `CreateFile` in which there are two calls that rely on default values for most of the parameters shown here.

```
HANDLE CreateFile (
   LPCTSTR lpName,
   DWORD dwAccess,
   DWORD dwShareMode,
   LPSECURITY_ATTRIBUTES lpSecurityAttributes,
   DWORD dwCreate,
   DWORD dwAttrsAndFlags,
   HANDLE hTemplateFile)
```

Return: A `HANDLE` to an open file object, or
`INVALID_HANDLE_VALUE` in case of failure.

Parameters

The parameter names illustrate some Windows conventions that were introduced in Chapter 1. The prefix dw describes DWORD (32 bits, unsigned) options containing flags or numerical values. lpsz (long pointer to a zero-terminated string), or, more simply, lp, is for pathnames and other strings, although the Microsoft documentation is not entirely consistent. At times, you need to use common sense or read the documentation carefully to determine the correct data types.

lpName is a pointer to the null-terminated string that names the file, pipe, or other named object to open or create. The pathname is normally limited to MAX_PATH (260) characters, but you can circumvent this restriction by prefixing the pathname with \\?\ and using Unicode characters and strings.[2] This technique allows functions requiring pathname arguments to use names as long as 32K characters. The prefix is not part of the name. Finally, the LPCTSTR data type is explained in an upcoming section that also describes generic characters and strings; just regard it as a string data type for now.

dwAccess specifies the read and write access, using GENERIC_READ and GENERIC_WRITE. Flag values such as READ and WRITE do not exist. The GENERIC_ prefix may seem redundant, but it is necessary to conform with the macro names in the Windows header file, winnt.h. Numerous other constant names may seem longer than necessary, but the long names are easily readable and avoid name collisions with other macros.

These values can be combined with a bit-wise "or" operator (|), so to open a file for read and write access:

```
GENERIC_READ | GENERIC_WRITE
```

dwShareMode is a bit-wise "or" combination of:

- 0—The file cannot be shared. Furthermore, not even this process can open a second HANDLE on this file.
- FILE_SHARE_READ—Other processes, including the one making this call, can open this file for concurrent read access.
- FILE_SHARE_WRITE—This allows concurrent writing to the file.

When relevant to proper program operation, the programmer must take care to prevent concurrent updates to the same file location by using locks or other mechanisms. Chapter 3 covers this in more detail.

---

[2] Please see the "Interlude: Unicode and Generic Characters" section later in this chapter for more information.

lpSecurityAttributes points to a SECURITY_ATTRIBUTES structure. Use NULL values with CreateFile and all other functions for now; security is treated in Chapter 15.

dwCreate specifies whether to create a new file, overwrite an existing file, and so on.

- CREATE_NEW—Create a new file. Fail if the specified file already exists.
- CREATE_ALWAYS—Create a new file, or overwrite the file if it already exists.
- OPEN_EXISTING—Open an existing file or fail if the file does not exist.
- OPEN_ALWAYS—Open the file, creating it if it does not exist.
- TRUNCATE_EXISTING—Set the file length to zero. dwCreate must specify at least GENERIC_WRITE access. Destroy all contents if the specified file exists. Fail if the file does not exist.

dwAttrsAndFlags specifies file attributes and flags. There are 32 flags and attributes. Attributes are characteristics of the file, as opposed to the open HANDLE, and these flags are ignored when an existing file is opened. Here are some of the more important attribute and flag values.

- FILE_ATTRIBUTE_NORMAL—This attribute can be used only when no other attributes are set (flags can be set, however).
- FILE_ATTRIBUTE_READONLY—Applications can neither write to nor delete the file.
- FILE_FLAG_DELETE_ON_CLOSE—This is useful for temporary files. Windows deletes the file when the last open HANDLE is closed.
- FILE_FLAG_OVERLAPPED—This attribute flag is important for asynchronous I/O (see Chapter 14).

Several additional flags also specify how a file is processed and help the Windows implementation optimize performance and file integrity.

- FILE_FLAG_RANDOM_ACCESS—The file is intended for random access, and Windows will attempt to optimize file caching.
- FILE_FLAG_SEQUENTIAL_SCAN—The file is for sequential access, and Windows will optimize caching accordingly. These last two access modes are not enforced and are hints to the Windows cache manager. Accessing a file in a manner inconsistent with these access modes may degrade performance.

- FILE_FLAG_WRITE_THROUGH and FILE_FLAG_NO_BUFFERING are two examples of advanced flags that are useful in some advanced applications.

hTemplateFile is the HANDLE of an open GENERIC_READ file that specifies extended attributes to apply to a newly created file, ignoring dwAttrsAndFlags. Normally, this parameter is NULL. Windows ignores hTemplateFile when an existing file is opened. This parameter can be used to set the attributes of a new file to be the same as those of an existing file.

The two CreateFile instances in cpW (Program 1–2) use default values extensively and are as simple as possible but still appropriate for the task. It could be beneficial to use FILE_FLAG_SEQUENTIAL_SCAN in both cases. (Exercise 2–3 explores this option, and Appendix C shows the performance results.)

Notice that if the file share attributes and security permit it, there can be numerous open handles on a given file. The open handles can be owned by the same process or by different processes. (Chapter 6 describes process management.)

Windows Vista and later versions provide the ReOpenFile function, which returns a new handle with different flags, access rights, and so on, assuming there are no conflicts with existing handles to the same file. ReOpenFile allows you to have different handles for different situations and protect against accidental misuse. For example, a function that updates a shared file could use a handle with read-write access, whereas other functions would use a read-only handle.

Closing Files

Windows has a single all-purpose CloseHandle function to close and invalidate kernel handles[3] and to release system resources. Use this function to close nearly all HANDLE objects; exceptions are noted. Closing a handle also decrements the object's handle reference count so that nonpersistent objects such as temporary files and events can be deleted. Windows will close all open handles on exit, but it is still good practice for programs to close their handles before terminating.

Closing an invalid handle or closing the same handle twice will cause an exception when running under a debugger (Chapter 4 discusses exceptions and exception handling). It is not necessary or appropriate to close the standard device handles, which are discussed in the "Standard Devices and Console I/O" section.

```
BOOL CloseHandle (HANDLE hObject)
```

Return: TRUE if the function succeeds; FALSE otherwise.

---

[3] It is convenient to use the term "handle," and the context should make it clear that we mean a Windows HANDLE.

The comparable UNIX functions are different in a number of ways. The UNIX open function returns an integer file descriptor rather than a handle, and it specifies access, sharing, create options, attributes, and flags in the single integer `oflag` parameter. The options overlap, with Windows providing a richer set.

There is no UNIX equivalent to `dwShareMode`. UNIX files are always shareable.

Both systems use security information when creating a new file. In UNIX, the `mode` argument specifies the familiar user, group, and other file permissions.

`close` is comparable to `CloseHandle`, but it is not general purpose.

The C library `stdio.h` functions use `FILE` objects, which are comparable to handles (for disk files, terminals, tapes, and other devices) connected to streams. The `fopen` mode parameter specifies whether the file data is to be treated as binary or text. There is a set of options for read-only, update, append at the end, and so on. `freopen` allows `FILE` reuse without closing it first. The Standard C library cannot set security permissions.

`fclose` closes a `FILE`. Most `stdio` `FILE`-related functions have the `f` prefix.

Reading Files

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

Return: `TRUE` if the read succeeds (even if no bytes were read due to an attempt to read past the end of file).

Assume, until Chapter 14, that the file handle does *not* have the `FILE_FLAG_OVERLAPPED` option set in `dwAttrsAndFlags`. `ReadFile`, then, starts at the current file position (for the handle) and advances the position by the number of bytes transferred.

The function fails, returning `FALSE`, if the handle or any other parameters are invalid or if the read operation fails for any reason. The function does not fail if the file handle is positioned at the end of file; instead, the number of bytes read (`*lpNumberOfBytesRead`) is set to `0`.

Parameters

Because of the long variable names and the natural arrangement of the parameters, they are largely self-explanatory. Nonetheless, here are some brief explanations.

`hFile` is a file handle with `FILE_READ_DATA` access, a subset of `GENERIC_READ` access. `lpBuffer` points to the memory buffer to receive the input data. `nNumberOfBytesToRead` is the number of bytes to read from the file.

`lpNumberOfBytesRead` points to the actual number of bytes read by the `ReadFile` call. This value can be zero if the handle is positioned at the end of file or there is an error, and message-mode named pipes (Chapter 11) allow a zero-length message.

`lpOverlapped` points to an `OVERLAPPED` structure (Chapters 3 and 14). Use `NULL` for the time being.

Writing Files

```
BOOL WriteFile (
   HANDLE hFile,
   LPCVOID lpBuffer,
   DWORD nNumberOfBytesToWrite,
   LPDWORD lpNumberOfBytesWritten,
   LPOVERLAPPED lpOverlapped)
```

Return: `TRUE` if the function succeeds; `FALSE` otherwise.

The parameters are familiar by now. Notice that a successful write does not ensure that the data actually is written through to the disk unless `FILE_FLAG_WRITE_THROUGH` is specified with `CreateFile`. If the `HANDLE` position plus the write byte count exceed the current file length, Windows will extend the file length.

UNIX `read` and `write` are the comparable functions, and the programmer supplies a file descriptor, buffer, and byte count. The functions return the number of bytes actually transferred. A value of `0` on `read` indicates the end of file; –1 indicates an error. Windows, by contrast, requires a separate transfer count and returns Boolean values to indicate success or failure.

The functions in both systems are general purpose and can read from files, terminals, tapes, pipes, and so on.

The Standard C library `fread` and `fwrite` binary I/O functions use object size and object count rather than a single byte count as in UNIX and Windows. A short transfer could be caused by either an end of file or an error; test explicitly with `ferror` or `feof`. The library provides a full set of text-oriented functions, such as `fgetc` and `fputc`, that do not exist outside the C library in either OS.

Interlude: Unicode and Generic Characters

Before proceeding, we explain briefly how Windows processes characters and differentiates between 8- and 16-bit characters and generic characters. The topic is a large one and beyond the book's scope, so we only provide the minimum detail required.

Windows supports standard 8-bit characters (type `char` or `CHAR`) and wide 16-bit characters (`WCHAR`, which is defined to be the C `wchar_t` type). The Microsoft documentation refers to the 8-bit character set as ANSI, but it is actually a misnomer. For convenience, we use the term "ASCII," which also is not totally accurate.[4]

The wide character support that Windows provides using the Unicode UTF-16 encoding is capable of representing symbols and letters in all major languages, including English, French, Spanish, German, Japanese, and Chinese.

Here are the normal steps for writing a generic Windows application that can be built to use either Unicode or 8-bit ASCII characters.

1. Define all characters and strings using the generic types `TCHAR`, `LPTSTR`, and `LPCTSTR`.
2. Include the definitions `#define UNICODE` and `#define _UNICODE` in all source modules to get Unicode wide characters (ANSI C `wchar_t`); otherwise, with `UNICODE` and `_UNICODE` undefined, `TCHAR` will be equivalent to `CHAR` (ANSI C `char`). The definition must precede the `#include <windows.h>` statement and is frequently defined on the compiler command line, the Visual Studio project properties, or the project's `stdafx.h` file. The first preprocessor variable controls the Windows function definitions, and the second variable controls the C library.
3. Byte buffer lengths—as used, for example, in `ReadFile`—can be calculated using `sizeof (TCHAR)`.

[4] The distinctions and details are technical but can be critical in some situations. ASCII codes only go to 127. There are different ASNI code pages, which are configurable from the Control Panel. Use your favorite search engine or search MSDN with a phrase such as "Windows code page 1252" to obtain more information.

4. Use the collection of generic C library string and character I/O functions in `tchar.h`. Representative functions are `_fgettc`, `_itot` (for `itoa`), `_stprintf` (for `sprintf`), `_tcscpy` (for `strcpy`), `_ttoi`, `_totupper`, `_totlower`, and `_ftprintf`.[5] See MSDN for a complete and extensive list. All these definitions depend on `_UNICODE`. This collection is not complete. `memchr` is an example of a function without a wide character implementation. New versions are provided in the *Examples* file as required.
5. Constant strings should be in one of three forms. Use these conventions for single characters as well. The first two forms are ANSI C; the third—the `_T` macro (equivalently, `TEXT` and `_TEXT`)—is supplied with the Microsoft C compiler.

```
"This string uses 8-bit characters"

L"This string uses 16-bit characters"

_T("This string uses generic text characters")
```

6. Include `tchar.h` after `windows.h` to get required definitions for text macros and generic C library functions.

Windows uses Unicode 16-bit characters throughout, and NTFS file names and pathnames are represented internally in Unicode. If the `UNICODE` macro is defined, wide character strings are required by Windows calls; otherwise, 8-bit character strings are converted to wide characters. Some Windows API functions only support Unicode, and this policy is expected to continue with new functions.

All future program examples will use `TCHAR` instead of the normal `char` for characters and character strings unless there is a clear reason to deal with individual 8-bit characters. Similarly, the type `LPTSTR` indicates a pointer to a generic string, and `LPCTSTR` indicates, in addition, a constant string. At times, this choice will add some clutter to the programs, but it is the only choice that allows the flexibility necessary to develop and test applications in either Unicode or 8-bit character form so that the program can be easily converted to Unicode at a later date. Furthermore, this choice is consistent with common, if not universal, industry practice.

It is worthwhile to examine the system include files to see how `TCHAR` and the system function interfaces are defined and how they depend on whether or not `UNICODE` and `_UNICODE` are defined. A typical entry is of the following form:

---

[5] The underscore character (_) indicates that a function or keyword is provided by Microsoft C, and the letters `t` and `T` denote a generic text character. Other development systems provide similar capability but may use different names or keywords.

```
#ifdef UNICODE
#define TCHAR WCHAR
#else
#define TCHAR CHAR
#endif
```

Alternative Generic String Processing Functions

String comparisons can use `lstrcmp` and `lstrcmpi` rather than the generic `_tcscmp` and `_tcscmpi` to account for the specific language and region, or *locale*, at run time and also to perform *word* rather than *string* comparisons. String comparisons simply compare the numerical values of the characters, whereas word comparisons consider locale-specific word order. The two methods can give opposite results for string pairs such as *coop/co-op* and *were/we're*.

There is also a group of Windows functions for dealing with Unicode characters and strings. These functions handle locale characteristics transparently. Typical functions are `CharUpper`, which can operate on strings as well as individual characters, and `IsCharAlphaNumeric`. Other string functions include `CompareString` (which is locale-specific). The generic C library functions (e.g., `_tprintf`) and the Windows functions will both appear in upcoming examples to demonstrate their use. Examples in later chapters will rely mostly on the generic C library for character and string manipulation, as the C Library has the required functionality, the Windows functions do not add value, and readers will be familiar with the C Library.

The Generic Main Function

Replace the C `main` function, with its argument list (`argv[]`), with the macro `_tmain`. The macro expands to either `main` or `wmain` depending on the `_UNICODE` definition. The `_tmain` definition is in `tchar.h`, which must be included after `windows.h`. A typical main program heading, then, would look like this:

```
#include <windows.h>
#include <tchar.h>
int _tmain(int argc, LPTSTR argv[])
{
    ...
}
```

The Microsoft C `_tmain` function also supports a third parameter for environment strings. This nonstandard extension is also common in UNIX.