IV

CHAPTER

Data Files

This chapter focuses on one of C's strongest assets: disk input and output. For many years, the fastest and leanest professional programs have been developed in C and have benefited from the language's optimized file I/O routines.

File manipulation can be a difficult task sometimes, and this chapter presents some of the most frequently asked questions regarding data files. Subjects such as streams, file modes (text and binary), file and directory manipulation, and file sharing are addressed. Most of today's professional programs are network-aware, so pay close attention to those questions at the end of the chapter that deal with file sharing and concurrency control topics. In addition, some diverse file-related topics, such as file handles in DOS and installing a hardware error handling routine, are covered in this chapter. Enjoy!

IV.1: If *errno* contains a nonzero number, is there an error?

Answer:

The global variable `errno` is used by many standard C library functions to pass back to your program an error code that denotes specifically which error occurred. However, your program *should not* check the value of `errno` to determine whether an error occurred. Usually, the standard C library function you are calling returns with a return code which denotes that an error has occurred and that the value of `errno` has been set to a specific error number. If no error has occurred or if you are using a library function that does not reference `errno`, there is a good chance that `errno` will contain an erroneous value. For performance enhancement, the `errno` variable is sometimes not cleared by the functions that use it.

You should *never* rely on the value of `errno` alone; always check the return code from the function you are calling to see whether `errno` should be referenced. Refer to your compiler's library documentation for references to functions that utilize the `errno` global variable and for a list of valid values for `errno`.

Cross Reference:

None.

IV.2: What is a stream?

Answer:

A stream is a continuous series of bytes that flow into or out of your program. Input and output from devices such as the mouse, keyboard, disk, screen, modem, and printer are all handled with streams. In C, all streams appear as files—not physical disk files necessarily, but rather *logical* files that refer to an input/output source. The C language provides five "standard" streams that are always available to your program. These streams do not have to be opened or closed. These are the five standard streams:

Name	*Description*	*Example*
`stdin`	Standard Input	Keyboard
`stdout`	Standard Output	Screen
`stderr`	Standard Error	Screen
`stdprn`	Standard Printer	LPT1: port
`stdaux`	Standard Auxiliary	COM1: port

Note that the `stdprn` and `stdaux` streams are not always defined. This is because LPT1: and COM1: have no meaning under certain operating systems. However, `stdin`, `stdout`, and `stderr` are always defined. Also, note that the `stdin` stream does not have to come from the keyboard; it can come from a disk file or some other device through what is called redirection. In the same manner, the `stdout` stream does not have to appear on-screen; it too can be redirected to a disk file or some other device. See the next FAQ for an explanation of redirection.

Cross Reference:

IV.3: How do you redirect a standard stream?

IV.4: How can you restore a redirected standard stream?

IV.5: Can `stdout` be forced to print somewhere other than the screen?

IV.3: How do you redirect a standard stream?

Answer:

Most operating systems, including DOS, provide a means to redirect program input and output to and from different devices. This means that rather than your program output (`stdout`) going to the screen, it can be redirected to a file or printer port. Similarly, your program's input (`stdin`) can come from a file rather than the keyboard. In DOS, this task is accomplished using the redirection characters, < and >. For example, if you wanted a program named PRINTIT.EXE to receive its input (`stdin`) from a file named STRINGS.TXT, you would enter the following command at the DOS prompt:

```
C:>PRINTIT < STRINGS.TXT
```

Notice that the name of the executable file always comes first. The less-than sign (<) tells DOS to take the strings contained in STRINGS.TXT and use them as input for the PRINTIT program. See FAQ IV.5 for an example of redirecting the `stdout` standard stream.

Redirection of standard streams does not always have to occur at the operating system. You can redirect a standard stream from *within your program* by using the standard C library function named `freopen()`. For example, if you wanted to redirect the `stdout` standard stream within your program to a file named OUTPUT.TXT, you would implement the `freopen()` function as shown here:

```
...
freopen(“output.txt”, “w”, stdout);
...
```

Now, every output statement (`printf()`, `puts()`, `putch()`, and so on) in your program will appear in the file OUTPUT.TXT.

Cross Reference:

IV.2: What is a stream?

IV.4: How can you restore a redirected standard stream?

IV.5: Can `stdout` be forced to print somewhere other than the screen?

IV.4: How can you restore a redirected standard stream?

Answer:

The preceding example showed how you can redirect a standard stream from within your program. But what if later in your program you wanted to restore the standard stream to its *original* state? By using the standard C library functions named `dup()` and `fdopen()`, you can restore a standard stream such as `stdout` to its original state.

The `dup()` function duplicates a file handle. You can use the `dup()` function to save the file handle corresponding to the `stdout` standard stream. The `fdopen()` function opens a stream that has been duplicated with the `dup()` function. Thus, as shown in the following example, you can redirect standard streams and restore them:

```
#include <stdio.h>
```

```
void main(void);

void main(void)
{

     int orig_stdout;

     /* Duplicate the stdout file handle and store it in orig_stdout. */

     orig_stdout = dup(fileno(stdout));

     /* This text appears on-screen. */

     printf("Writing to original stdout...\n");

     /* Reopen stdout and redirect it to the "redir.txt" file. */

     freopen("redir.txt", "w", stdout);

     /* This text appears in the "redir.txt" file. */

     printf("Writing to redirected stdout...\n");

     /* Close the redirected stdout. */

     fclose(stdout);

     /* Restore the original stdout and print to the screen again. */

     fdopen(orig_stdout, "w");

     printf("I'm back writing to the original stdout.\n");

}
```

Cross Reference:

IV.2: What is a stream?

IV.3: How do you redirect a standard stream?

IV.5: Can `stdout` be forced to print somewhere other than the screen?

IV.5: Can *stdout* be forced to print somewhere other than the screen?

Answer:

Although the `stdout` standard stream defaults to the screen, you can force it to print to another device using something called *redirection* (see FAQ IV.3 for an explanation of redirection). For instance, consider the following program:

```
/* redir.c */

#include <stdio.h>

void main(void);

void main(void)
{

     printf("Let's get redirected!\n");

}
```

At the DOS prompt, instead of entering just the executable name, follow it with the redirection character `>`, and thus redirect what normally would appear on-screen to some other device. The following example would redirect the program's output to the `prn` device, usually the printer attached on LPT1:

```
C:>REDIR > PRN
```

Alternatively, you might want to redirect the program's output to a file, as the following example shows:

```
C:>REDIR > REDIR.OUT
```

In this example, all output that would have normally appeared on-screen will be written to the file REDIR.OUT.

Refer to FAQ IV.3 for an example of how you can redirect standard streams from *within* your program.

Cross Reference:

IV.2: What is a stream?

IV.3: How do you redirect a standard stream?

IV.4: How can you restore a redirected standard stream?

IV.6: What is the difference between text and binary modes?

Answer:

Streams can be classified into two types: *text* streams and *binary* streams. Text streams are interpreted, with a maximum length of 255 characters. With text streams, carriage return/line feed combinations are translated to the newline `\n` character and vice versa. Binary streams are uninterpreted and are treated one byte at a time with no translation of characters. Typically, a text stream would be used for reading and writing standard text files, printing output to the screen or printer, or receiving input from the keyboard. A binary text stream would typically be used for reading and writing binary files such as graphics or word processing documents, reading mouse input, or reading and writing to the modem.

Cross Reference:

IV.18: How can I read and write comma-delimited text?

IV.7: How do you determine whether to use a stream function or a low-level function?

Answer:

Stream functions such as `fread()` and `fwrite()` are buffered and are more efficient when reading and writing text or binary data to files. You generally gain better performance by using stream functions rather than their unbuffered low-level counterparts such as `read()` and `write()`.

In multiuser environments, however, when files are typically shared and portions of files are continuously being locked, read from, written to, and unlocked, the stream functions do not perform as well as the low-level functions. This is because it is hard to buffer a shared file whose contents are constantly changing.

Generally, you should always use buffered stream functions when accessing nonshared files, and you should always use the low-level functions when accessing shared files.

Cross Reference:

None.

IV.8: How do you list files in a directory?

Answer:

Unfortunately, there is no built-in function provided in the C language such as `dir_list()` that would easily provide you with a list of all files in a particular directory. By utilizing some of C's built-in directory functions, however, you can write your own `dir_list()` function.

First of all, the include file dos.h defines a structure named `find_t`, which represents the structure of the DOS file entry block. This structure holds the name, time, date, size, and attributes of a file. Second, your C compiler library contains the functions `_dos_findfirst()` and `_dos_findnext()`, which can be used to find the first or next file in a directory.

The `_dos_findfirst()` function requires three arguments. The first argument is the file mask for the directory list. A mask of `*.*` would be used to list all files in the directory. The second argument is an attribute mask, defining which file attributes to search for. For instance, you might want to list only files with the Hidden or Directory attributes. See FAQ IV.11 for a more detailed explanation of file attributes. The last argument of the `_dos_findfirst()` function is a pointer to the variable that is to hold the directory information (the `find_t` structure variable).

The second function you will use is the `_dos_findnext()` function. Its only argument is a pointer to the `find_t` structure variable that you used in the `_dos_findfirst()` function. Using these two functions and the `find_t` structure, you can iterate through the directory on a disk and list each file in the directory. Here is the code to perform this task:

```
#include <stdio.h>
#include <direct.h>
```

```
#include <dos.h>
#include <malloc.h>
#include <memory.h>
#include <string.h>

typedef struct find_t FILE_BLOCK;

void main(void);

void main(void)
{

     FILE_BLOCK f_block;      /* Define the find_t structure variable */
     int ret_code;     /* Define a variable to store the return codes */

     printf("\nDirectory listing of all files in this directory:\n\n");

     /* Use the "*.*" file mask and the 0xFF attribute mask to list
        all files in the directory, including system files, hidden
        files, and subdirectory names. */

     ret_code = _dos_findfirst("*.*", 0xFF, &f_block);

     /* The _dos_findfirst() function returns a 0 when it is successful
        and has found a valid filename in the directory. */

     while (ret_code == 0)
     {

          /* Print the file's name */

          printf("%-12s\n", f_block.name);

          /* Use the _dos_findnext() function to look
             for the next file in the directory. */

          ret_code = _dos_findnext(&f_block);

     }

     printf("\nEnd of directory listing.\n");

}
```

Cross Reference:

IV.9: How do you list a file's date and time?

IV.10: How do you sort filenames in a directory?

IV.11: How do you determine a file's attributes?

IV.9: How do you list a file’s date and time?

Answer:

A file’s date and time are stored in the `find_t` structure returned from the `_dos_findfirst()` and `_dos_findnext()` functions (see FAQ IV.8). Using the example from IV.8, the source code can be modified slightly so that the date and time stamp of each file, as well as its name, is printed.

The date and time stamp of the file is stored in the `find_t.wr_date` and `find_t.wr_time` structure members. The file date is stored in a two-byte unsigned integer as shown here:

Element	*Offset*	*Range*
Seconds	5 bits	0–9 (multiply by 2 to get the seconds value)
Minutes	6 bits	0–59
Hours	5 bits	0–23

Similarly, the file time is stored in a two-byte unsigned integer, as shown here:

Element	*Offset*	*Range*
Day	5 bits	1–31
Month	4 bits	1–12
Year	7 bits	0–127 (add the value “1980” to get the year value)

Because DOS stores a file’s seconds in *two-second* intervals, only the values 0 to 29 are needed. You simply multiply the value by 2 to get the file’s true seconds value. Also, because DOS came into existence in 1980, no files can have a time stamp prior to that year. Therefore, you must add the value “1980” to get the file’s true year value.

The following example program shows how you can get a directory listing along with each file’s date and time stamp:

```
#include <stdio.h>
#include <direct.h>
#include <dos.h>
#include <malloc.h>
#include <memory.h>
#include <string.h>

typedef struct find_t FILE_BLOCK;

void main(void);

void main(void)
{

     FILE_BLOCK f_block;   /* Define the find_t structure variable */
     int ret_code;         /* Define a variable to store return codes */
     int hour;             /* We’re going to use a 12-hour clock! */
     char* am_pm;          /* Used to print “am” or “pm” */

     printf(“\nDirectory listing of all files in this directory:\n\n”);
     /* Use the "*.*" file mask and the 0xFF attribute mask to list
        all files in the directory, including system files, hidden
        files, and subdirectory names. */

     ret_code = _dos_findfirst("*.*", 0xFF, &f_block);

     /* The _dos_findfirst() function returns a 0 when it is successful
        and has found a valid filename in the directory. */

     while (ret_code == 0)
     {

          /* Convert from a 24-hour format to a 12-hour format. */

          hour = (f_block.wr_time >> 11);

          if (hour > 12)
          {
               hour  = hour - 12;
               am_pm = "pm";
          }
          else
               am_pm = "am";

          /* Print the file's name, date stamp, and time stamp. */

          printf("%-12s  %02d/%02d/%4d  %02d:%02d:%02d %s\n",
                    f_block.name,                      /* name   */
                    (f_block.wr_date >> 5) & 0x0F,     /* month */
                    (f_block.wr_date) & 0x1F,          /* day    */
                    (f_block.wr_date >> 9) + 1980,     /* year   */
                    hour,                              /* hour   */
                    (f_block.wr_time >> 5) & 0x3F,     /* minute  */
                    (f_block.wr_time & 0x1F) * 2,      /* seconds */
                    am_pm);

          /* Use the _dos_findnext() function to look
             for the next file in the directory. */

          ret_code = _dos_findnext(&f_block);

     }

     printf("\nEnd of directory listing.\n");

}
```

Notice that a lot of bit-shifting and bit-manipulating had to be done to get the elements of the time variable and the elements of the date variable. If you happen to suffer from *bitshiftophobia* (fear of shifting bits), you can optionally code the preceding example by forming a union between the `find_t` structure and your own user-defined structure, such as this:

```
/* This is the find_t structure as defined by ANSI C. */

struct find_t
{
     char reserved[21];
     char attrib;
     unsigned wr_time;
     unsigned wr_date;
     long size;
     char name[13];
}

/* This is a custom find_t structure where we
   separate out the bits used for date and time. */

struct my_find_t
{
     char reserved[21];
     char attrib;
     unsigned seconds:5;
     unsigned minutes:6;
     unsigned hours:5;
     unsigned day:5;
     unsigned month:4;
     unsigned year:7;
     long size;
     char name[13];
}

/* Now, create a union between these two structures
   so that we can more easily access the elements of
   wr_date and wr_time. */

union file_info
{
     struct find_t ft;
     struct my_find_t mft;
}
```

Using the preceding technique, instead of using bit-shifting and bit-manipulating, you can now extract date and time elements like this:

```
...
file_info my_file;
...

printf("%-12s  %02d/%02d/%4d  %02d:%02d:%02d %s\n",
          my_file.mft.name,              /* name    */
          my_file.mft.month,             /* month   */
          my_file.mft.day,               /* day     */
          (my_file.mft.year + 1980),     /* year    */
          my_file.mft.hours,             /* hour    */
          my_file.mft.minutes,           /* minute  */
          (my_file.mft.seconds * 2),     /* seconds */
          am_pm);
```

Cross Reference:

IV.8: How do you list files in a directory?

IV.10: How do you sort filenames in a directory?

IV.11: How do you determine a file's attributes?

IV.10: How do you sort filenames in a directory?

Answer:

The example in FAQ IV.8 shows how to get a list of files one at a time. The example uses the `_dos_findfirst()` and `_dos_findnext()` functions to walk through the directory structure. As each filename is found, it is printed to the screen.

When you are sorting the filenames in a directory, the one-at-a-time approach does not work. You need some way to *store* the filenames and then sort them when all filenames have been obtained. This task can be accomplished by creating an array of pointers to `find_t` structures for each filename that is found. As each filename is found in the directory, memory is allocated to hold the `find_t` entry for that file. When all filenames have been found, the `qsort()` function is used to sort the array of `find_t` structures by filename.

The `qsort()` function can be found in your compiler's library. This function takes four parameters: a pointer to the array you are sorting, the number of elements to sort, the size of each element, and a pointer to a function that compares two elements of the array you are sorting. The comparison function is a user-defined function that you supply. It returns a value less than zero if the first element is less than the second element, greater than zero if the first element is greater than the second element, or zero if the two elements are equal. Consider the following example:

```
#include <stdio.h>
#include <direct.h>
#include <dos.h>
#include <malloc.h>
#include <memory.h>
#include <string.h>

typedef struct find_t FILE_BLOCK;

int  sort_files(FILE_BLOCK**, FILE_BLOCK**);
void main(void);

void main(void)
{

     FILE_BLOCK f_block;       /* Define the find_t structure variable */
     int ret_code;             /* Define a variable to store the return
                                  codes */
     FILE_BLOCK** file_list;   /* Used to sort the files */
     int file_count;           /* Used to count the files */
     int x;                    /* Counter variable */

     file_count = -1;

     /* Allocate room to hold up to 512 directory entries. */

     file_list = (FILE_BLOCK**) malloc(sizeof(FILE_BLOCK*) * 512);

     printf("\nDirectory listing of all files in this directory:\n\n");

     /* Use the "*.*" file mask and the 0xFF attribute mask to list
        all files in the directory, including system files, hidden
        files, and subdirectory names. */
```

```
          ret_code = _dos_findfirst("*.*", 0xFF, &f_block);

          /* The _dos_findfirst() function returns a 0 when it is successful
             and has found a valid filename in the directory. */

          while (ret_code == 0 && file_count < 512)
          {

               /* Add this filename to the file list */

               file_list[++file_count] =
                   (FILE_BLOCK*) malloc(sizeof(FILE_BLOCK));

               *file_list[file_count] = f_block;

               /* Use the _dos_findnext() function to look
                  for the next file in the directory. */

               ret_code = _dos_findnext(&f_block);

          }

          /* Sort the files */

          qsort(file_list, file_count, sizeof(FILE_BLOCK*), sort_files);

          /* Now, iterate through the sorted array of filenames and
             print each entry. */

          for (x=0; x<file_count; x++)
          {

               printf("%-12s\n", file_list[x]->name);

          }

          printf("\nEnd of directory listing.\n");

     }

     int sort_files(FILE_BLOCK** a, FILE_BLOCK** b)
     {

          return (strcmp((*a)->name, (*b)->name));

     }
```

This example uses the user-defined function named `sort_files()` to compare two filenames and return the appropriate value based on the return value from the standard C library function `strcmp()`. Using this same technique, you can easily modify the program to sort by date, time, extension, or size by changing the element on which the `sort_files()` function operates.

Cross Reference:

IV.8: How do you list files in a directory?

IV.9: How do you list a file's date and time?

IV.11: How do you determine a file's attributes?

IV.11: How do you determine a file's attributes?

Answer:

The file attributes are stored in the `find_t.attrib` structure member (see FAQ IV.8). This structure member is a single character, and each file attribute is represented by a single bit. Here is a list of the valid DOS file attributes:

Value	*Description*	*Constant*
0x00	Normal	(none)
0x01	Read Only	`FA_RDONLY`
0x02	Hidden File	`FA_HIDDEN`
0x04	System File	`FA_SYSTEM`
0x08	Volume Label	`FA_LABEL`
0x10	Subdirectory	`FA_DIREC`
0x20	Archive File	`FA_ARCHIVE`

To determine the file's attributes, you check which bits are turned on and map them corresponding to the preceding table. For example, a read-only hidden system file will have the first, second, and third bits turned on. A "normal" file will have none of the bits turned on. To determine whether a particular bit is turned on, you do a bit-wise AND with the bit's constant representation.

The following program uses this technique to print a file's attributes:

```
#include <stdio.h>
#include <direct.h>
#include <dos.h>
#include <malloc.h>
#include <memory.h>
#include <string.h>

typedef struct find_t FILE_BLOCK;

void main(void);

void main(void)
{

     FILE_BLOCK f_block;  /* Define the find_t structure variable */
     int ret_code;        /* Define a variable to store the return codes */

     printf("\nDirectory listing of all files in this directory:\n\n");

     /* Use the "*.*" file mask and the 0xFF attribute mask to list
        all files in the directory, including system files, hidden
        files, and subdirectory names. */

     ret_code = _dos_findfirst("*.*", 0xFF, &f_block);

     /* The _dos_findfirst() function returns a 0 when
        it is successful and has found a valid filename
        in the directory. */
```

```
          while (ret_code == 0)
          {

               /* Print the file’s name */

               printf(“%-12s  “, f_block.name);

               /* Print the read-only attribute */

               printf(“%s “, (f_block.attrib & FA_RDONLY) ? “R” : “.”);

               /* Print the hidden attribute */

               printf(“%s “, (f_block.attrib & FA_HIDDEN) ? “H” : “.”);

               /* Print the system attribute */

               printf(“%s “, (f_block.attrib & FA_SYSTEM) ? “S” : “.”);

               /* Print the directory attribute */

               printf(“%s “, (f_block.attrib & FA_DIREC)  ? “D” : “.”);

               /* Print the archive attribute */

               printf(“%s\n”, (f_block.attrib & FA_ARCH)  ? “A” : “.”);

               /* Use the _dos_findnext() function to look
                  for the next file in the directory. */

               ret_code = _dos_findnext(&f_block);

          }

          printf(“\nEnd of directory listing.\n”);

     }
```

Cross Reference:

IV.8: How do you list files in a directory?

IV.9: How do you list a file’s date and time?

IV.10: How do you sort filenames in a directory?

IV.12: How do you view the *PATH*?

Answer:

Your C compiler library contains a function called `getenv()` that can retrieve any specified environment variable. It has one argument, which is a pointer to a string containing the environment variable you want to retrieve. It returns a pointer to the desired environment string on successful completion. If the function cannot find your environment variable, it returns `NULL`.

The following example program shows how to obtain the `PATH` environment variable and print it on-screen:

```
#include <stdio.h>
#include <stdlib.h>

void main(void);

void main(void)
{

     char* env_string;

     env_string = getenv("PATH");

     if (env_string == (char*) NULL)
          printf("\nYou have no PATH!\n");
     else
          printf("\nYour PATH is: %s\n", env_string);

}
```

Cross Reference:

None.

IV.13: How can I open a file so that other programs can update it at the same time?

Answer:

Your C compiler library contains a *low-level* file function called `sopen()` that can be used to open a file in shared mode. Beginning with DOS 3.0, files could be opened in shared mode by loading a special program named SHARE.EXE. Shared mode, as the name implies, allows a file to be shared with other programs as well as your own. Using this function, you can allow other programs that are running to update the same file you are updating.

The `sopen()` function takes four parameters: a pointer to the filename you want to open, the operational mode you want to open the file in, the file sharing mode to use, and, if you are creating a file, the mode to create the file in. The second parameter of the `sopen()` function, usually referred to as the "operation flag" parameter, can have the following values assigned to it:

Constant	*Description*
`O_APPEND`	Appends all writes to the end of the file
`O_BINARY`	Opens the file in binary (untranslated) mode
`O_CREAT`	If the file does not exist, it is created
`O_EXCL`	If the `O_CREAT` flag is used and the file exists, returns an error
`O_RDONLY`	Opens the file in read-only mode
`O_RDWR`	Opens the file for reading and writing

Constant	*Description*
`O_TEXT`	Opens the file in text (translated) mode
`O_TRUNC`	Opens an existing file and writes over its contents
`O_WRONLY`	Opens the file in write-only mode

The third parameter of the `sopen()` function, usually referred to as the "sharing flag," can have the following values assigned to it:

Constant	*Description*
`SH_COMPAT`	No other program can access the file
`SH_DENYRW`	No other program can read from or write to the file
`SH_DENYWR`	No other program can write to the file
`SH_DENYRD`	No other program can read from the file
`SH_DENYNO`	Any program can read from or write to the file

If the `sopen()` function is successful, it returns a non-negative number that is the file's handle. If an error occurs, –1 is returned, and the global variable `errno` is set to one of the following values:

Constant	*Description*
`ENOENT`	File or path not found
`EMFILE`	No more file handles are available
`EACCES`	Permission denied to access file
`EINVACC`	Invalid access code

The following example shows how to open a file in shared mode:

```
#include <stdio.h>
#include <fcntl.h>
#include <sys\stat.h>
#include <io.h>
#include <share.h>

void main(void);

void main(void)
{

     int file_handle;

     /* Note that sopen() is not ANSI compliant */

     file_handle = sopen("C:\\DATA\\TEST.DAT", O_RDWR, SH_DENYNO);

     close(file_handle);

}
```

Whenever you are sharing a file's contents with other programs, you should be sure to use the standard C library function named `locking()` to lock a portion of your file when you are updating it. See FAQ IV.15 for an explanation of the `locking()` function.

Cross Reference:

IV.14: How can I make sure that my program is the only one accessing a file?

IV.15: How can I prevent another program from modifying part of a file that I am modifying?

IV.14: How can I make sure that my program is the only one accessing a file?

Answer:

By using the `sopen()` function (see FAQ IV.13), you can open a file in shared mode and explicitly deny reading and writing permissions to any other program but yours. This task is accomplished by using the `SH_DENYWR` shared flag to denote that your program is going to deny any writing or reading attempts by other programs. For example, the following snippet of code shows a file being opened in shared mode, denying access to all other files:

```
/* Note that the sopen() function is not ANSI compliant... */

fileHandle = sopen("C:\\DATA\\SETUP.DAT", O_RDWR, SH_DENYWR);
```

By issuing this statement, all other programs are denied access to the SETUP.DAT file. If another program were to try to open SETUP.DAT for reading or writing, it would receive an EACCES error code, denoting that access is denied to the file.

Cross Reference:

IV.13: How can I open a file so that other programs can update it at the same time?

IV.15: How can I prevent another program from modifying part of a file that I am modifying?

IV.15: How can I prevent another program from modifying part of a file that I am modifying?

Answer:

Under DOS 3.0 and later, file sharing can be implemented by using the SHARE.EXE program (see FAQ IV.13). Your C compiler library comes with a function named `locking()` that can be used to lock and unlock portions of shared files.

The locking function takes three arguments: a handle to the shared file you are going to lock or unlock, the operation you want to perform on the file, and the number of bytes you want to lock. The file lock is placed relative to the current position of the file pointer, so if you are going to lock bytes located anywhere but at the beginning of the file, you need to reposition the file pointer by using the `lseek()` function.

The following example shows how a binary index file named SONGS.DAT can be locked and unlocked:

```
#include <stdio.h>
#include <io.h>
#include <fcntl.h>
#include <process.h>
#include <string.h>
#include <share.h>
#include <sys\locking.h>

void main(void);

void main(void)
{

     int      file_handle, ret_code;
     char* song_name = “Six Months In A Leaky Boat”;
     char  rec_buffer[50];

     file_handle = sopen(“C:\\DATA\\SONGS.DAT”, O_RDWR, SH_DENYNO);

     /* Assuming a record size of 50 bytes, position the file
        pointer to the 10th record. */

     lseek(file_handle, 450, SEEK_SET);

     /* Lock the 50-byte record. */

     ret_code = locking(file_handle, LK_LOCK, 50);

     /* Write the data and close the file. */

     memset(rec_buffer, ‘\0’, sizeof(rec_buffer));

     sprintf(rec_buffer, “%s”, song_name);

     write(file_handle, rec_buffer, sizeof(rec_buffer));

     lseek(file_handle, 450, SEEK_SET);

     locking(file_handle, LK_UNLCK, 50);

     close(file_handle);

}
```

Notice that before the record is locked, the record pointer is positioned to the 10th record (450th byte) by using the `lseek()` function. Also notice that to write the record to the file, the record pointer has to be repositioned to the beginning of the record before unlocking the record.

Cross Reference:

IV.13: How can I open a file so that other programs can update it at the same time?

IV.14: How can I make sure that my program is the only one accessing a file?

IV.16: How can I have more than 20 files open at once?

Answer:

The DOS configuration file, CONFIG.SYS, usually contains a FILES entry that tells DOS how many file handles to allocate for your programs to use. By default, DOS allows 20 files to be open at once. In many cases, especially if you are a user of Microsoft Windows or a database program, 20 file handles is not nearly enough. Fortunately, there is an easy way to allocate more file handles to the system. To do this, replace your `FILES=` statement in your CONFIG.SYS file with the number of file handles you want to allocate. If your CONFIG.SYS file does not contain a `FILES` entry, you can append such an entry to the end of the file. For example, the following statement in your CONFIG.SYS file will allocate 100 file handles to be available for system use:

```
FILES=100
```

On most systems, 100 file handles is sufficient. If you happen to be encountering erratic program crashes, you might have too few file handles allocated to your system, and you might want to try allocating more file handles. Note that each file handle takes up memory, so there is a cost in having a lot of file handles; the more file handles you allocate, the less memory your system will have to run programs. Also, note that file handles not only are allocated for data files, but also are applicable to binary files such as executable programs.

Cross Reference:

None.

IV.17: How can I avoid the *Abort*, *Retry*, *Fail* messages?

Answer:

When DOS encounters a critical error, it issues a call to *interrupt 24*, the critical error handler. Your C compiler library contains a function named `harderr()` that takes over the handling of calls to interrupt 24. The `harderr()` function takes one argument, a pointer to a function that is called if there is a hardware error.

Your user-defined hardware error-handling function is passed information regarding the specifics of the hardware error that occurred. In your function, you can display a user-defined message to avoid the ugly `Abort, Retry, Fail` message. This way, your program can elegantly handle such simple user errors as your not inserting the disk when prompted to do so.

When a hardware error is encountered and your function is called, you can either call the C library function `hardretn()` to return control to your application or call the C library function `hardresume()` to return control to DOS. Typically, disk errors can be trapped and your program can continue by using the `hardresume()` function. Other device errors, such as a bat FAT (file allocation table) error, are somewhat fatal, and your application should handle them by using the `hardretn()` function. Consider the following example, which uses the `harderr()` function to trap for critical errors and notifies the user when such an error occurs:

```
#include <stdio.h>
#include <dos.h>
#include <fcntl.h>
```

```
#include <ctype.h>

void main(void);
void far error_handler(unsigned, unsigned, unsigned far*);

void main(void)
{

     int file_handle, ret_code;

     /* Install the custom error-handling routine. */

     _harderr(error_handler);

     printf("\nEnsure that the A: drive is empty, \n");
     printf("then press any key.\n\n");

     getch();

     printf("Trying to write to the A: drive...\n\n");

     /* Attempt to access an empty A: drive... */

     ret_code = _dos_open("A:FILE.TMP", O_RDONLY, &file_handle);

     /* If the A: drive was empty, the error_handler() function was
        called. Notify the user of the result of that function. */

     switch (ret_code)
     {

          case 100:     printf("Unknown device error!\n");
                              break;

          case 2:       printf("FILE.TMP was not found on drive A!\n");
                              break;

          case 0:       printf("FILE.TMP was found on drive A!\n");
                              break;

          default:      printf("Unknown error occurred!\n");
                              break;

     }

}

void far error_handler(unsigned device_error, unsigned error_val,
                                     unsigned far* device_header)
{

     long x;

     /* This condition will be true only if a nondisk error occurred. */

     if (device_error & 0x8000)
          _hardretn(100);

     /* Pause one second. */
     for (x=0; x<2000000; x++);

     /* Retry to access the drive. */

     _hardresume(_HARDERR_RETRY);

}
```

In this example, a custom hardware error handler is installed named `error_handler()`. When the program attempts to access the A: drive and no disk is found there, the `error_handler()` function is called. The `error_handler()` function first checks to ensure that the problem is a disk error. If the problem is *not* a disk error, it returns 100 by using the `hardretn()` function. Next, the program pauses for one second and issues a `hardresume()` call to retry accessing the A: drive.

Cross Reference:

None.

IV.18: How can I read and write comma-delimited text?

Answer:

Many of today's popular programs use comma-delimited text as a means of transferring data from one program to another, such as the exported data from a spreadsheet program that is to be imported by a database program. Comma-delimited means that all data (with the exception of numeric data) is surrounded by double quotation marks ("") followed by a comma. Numeric data appears as-is, *with no surrounding double quotation marks*. At the end of each line of text, the comma is omitted and a newline is used.

To read and write the text to a file, you would use the `fprintf()` and `fscanf()` standard C library functions. The following example shows how a program can write out comma-delimited text and then read it back in.

```
#include <stdio.h>
#include <string.h>

typedef struct name_str
{
     char       first_name[15];
     char       nick_name[30];
     unsigned years_known;
} NICKNAME;

NICKNAME nick_names[5];

void main(void);
void set_name(unsigned, char*, char*, unsigned);

void main(void)
{

     FILE*     name_file;
     int       x;
     NICKNAME tmp_name;
```

```
printf("\nWriting data to NICKNAME.DAT, one moment please...\n");

/* Initialize the data with some values... */

set_name(0,    "Sheryl",      "Basset",      26);
set_name(1,    "Joel",        "Elkinator",    1);
set_name(2,    "Cliff",       "Shayface",    12);
set_name(3,    "Lloyd",       "Lloydage",    28);
set_name(4,    "Scott",       "Pie",          9);

/* Open the NICKNAME.DAT file for output in text mode. */

name_file = fopen("NICKNAME.DAT", "wt");

/* Iterate through all the data and use the fprintf() function
   to write the data to a file. */

for (x=0; x<5; x++)
{

     fprintf(name_file, "\"%s\", \"%s\", %u\n",
                 nick_names[x].first_name,
                 nick_names[x].nick_name,
                 nick_names[x].years_known);

}

/* Close the file and reopen it for input. */

fclose(name_file);

printf("\nClosed NICKNAME.DAT, reopening for input...\n");

name_file = fopen("NICKNAME.DAT", "rt");

printf("\nContents of the file NICKNAME.DAT:\n\n");

/* Read each line in the file using the scanf() function
   and print the file's contents. */

while (1)
{

     fscanf(name_file, "%s %s %u",
                tmp_name.first_name,
                tmp_name.nick_name,
                &tmp_name.years_known);

     if (feof(name_file))
          break;

     printf("%-15s %-30s %u\n",
                tmp_name.first_name,
                tmp_name.nick_name,
                tmp_name.years_known);

}
```

```
     fclose(name_file);

}

void set_name(unsigned name_num, char* f_name,
              char* n_name, unsigned years)
{

     strcpy(nick_names[name_num].first_name, f_name);
     strcpy(nick_names[name_num].nick_name,  n_name);

     nick_names[name_num].years_known = years;

}
```

Cross Reference:

IV.6: What is the difference between text and binary modes?

IV.7: How do you determine whether to use a stream function or a low-level function?