BDS C File I/O Tutorial

Leor Zolman
BD Software

The file I/O library functions provided with BDS C fall into two catagories: "raw" and "buffered." The raw file functions, typically coded in assembly language for best performance, are essentially a CP/M-oriented low-level interface where data transfers always occur in multiples of full CP/M logical sector (128 byte) quantities. The buffered functions (written in C) provide a byte-oriented, sequential file I/O system geared especially for "filter"-type applications; buffering allows you to read and write data in whatever sized quantities are most convenient while invisible mechanisms worry about things like sector buffering and actual disk I/O; thus the buffered I/O functions are usually more convenient to deal with than the raw functions, but they generate a lot of overhead by being slow and hogging up quite a bit of memory for code and buffer space.

Since buffered I/O is composed of raw I/O functions plus some extra code, I'll first present the raw I/O in detail, and then go onto the buffered functions.

The raw functions are characterized by their concern with "file descriptors". A file descriptor (fd) is a small integer value that becomes associated with a currently active file. This fd is always obtained by calling either the "open" or "creat" functions; their usage is:

```
        fd = open(filename,mode);    /* `filename' can be either a literal */
                                     /*  string or any expression that     */
        fd = creat(filename);        /*  evaluates to a character pointer  */
```

The former is used to open an already existing file (usually, a file that has some data in it) for reading or writing or both, and the latter is used to create a brand new file and open it for writing. In both cases, the fd is the value returned by the call. If some kind of error occurs and the specified file cannot be opened or created, a value of ERROR (-1) is returned instead. For example, if "open" cannot find the file on disk whose name is pointed to by the first argument, ERROR will be returned.

All other raw functions require an fd to specify the file to be operated on (except "unlink" and "rename", which take filename pointers). The "read" and "write" functions are used to transfer data to and from disk. Their typical usage is:

```
        i = read(fd, buffer, nsects);      /* `fd' must have been obtained by */
        j = write(fd2, buffer2, nsects2); /*  a previous call to "open"      */
```

The first call would try to read, into memory at `buffer', `nsects' sectors from the file whose `fd' is specified. The second call would try to write `nsects2' sectors from memory at `buffer2' to the disk file whose fd is `fd2'. Unless an error occurs (such as when an illegal fd is given or an attempt is made to read past the end of a file), the above functions cause an immediate disk transfer to happen. This is one of the main differences between raw and buffered I/O: raw functions always cause immediate disk activity, as long as what they are asked to do is possible, while buffered functions only go to disk when a buffer fills up (when writing) or becomes exhausted (when reading.)

For each file opened under raw I/O, there exists an invisible "r/w pointer" to keep track of the next sector to be written or read. Immediately after a file is opened, the r/w pointer always starts at sector 0 (the first sector) of the file; it is bumped after "read" and "write" calls by the number of successfully transfered sectors, so that (by default) the next transfer happens sequentially. One nice extension of the BDS C raw I/O functions over their REALLY-raw CP/M equivalents is the elimination of the concept of "extents"; Instead of "extent numbers" and "sector numbers within the current extent" to be reckoned with for every file, there is only a single 16-bit r/w pointer to be considered. The value of a file's r/w pointer may be obtained by calling the "tell" function, and modified by calling "seek".

To illustrate the use of raw I/O in a program, let's build a simple utility to make a copy of a file. The command format for this utility (which we'll call "copy") shall be:

```
        A>copy filename newname <cr>
```

This will take the file named by `filename' and create a copy of it named by `newname'. Since this is to be a classy utility, we want full error diagnostics in case something goes wrong (such as running out of disk space, not being able to find the master file, etc.) This includes checking to make sure that the correct number of arguments were typed on the command line. It is sometimes convenient to summarize a program in a half-C/half-English pseudo code form to avoid going in blind; Here is such a summary of the copy program:

```
        copy(file1,file2) {
                if (exactly 2 args weren't given) { complain and abort }
                if (can't open file1) { complain and abort }
                if (can't create file2) { complain and abort }
                while (not end of file1) {
                        Read a hunk from file1 and write it out to file2;
                        if (any error has ocurred) { complain and abort }
                }
                close all files;
        }
```

And here is the actual C program that implements the above procedure:

```
        #include "bdscio.h"       /* The standard header file               */
        #define BUFSECTS 64       /* Buffer up to 64 sectors in memory      */

        int fd1, fd2;             /* File descriptors for the two files     */
        char buffer[BUFSECTS * SECSIZ];          /* The transfer buffer     */

        main(argc,argv)
        int argc;                 /* Arg count    */
        char **argv;              /* Arg vector   */
        {
                int oksects;      /* A temporary variable */

                                  /* make sure exactly 2 args were given    */
                if (argc != 3)
                        perror("Usage: A>copy file1 file2 <cr>\n");

                                  /* try to open 1st file; abort on error */
                if ((fd1 = open(argv[1],0)) == ERROR)
                        perror("Can't open: %x\n",argv[1]);

                                  /* create 2nd file, abort on error:       */
                if ((fd2 = creat(argv[2])) == ERROR)
                        perror("Can't create: %s\n",argv[2]);

                                  /* Now we're ready to move the data:      */
                while (oksects = read(fd1, buffer, BUFSECTS)) {
                        if (oksects == ERROR)
                                perror("Error reading: %s\n",argv[1]);
                        if (write(fd2, buffer, oksects) != oksects)
                                perror("Error; probably out of disk space\n");
                }

                                  /* Copy is complete. Now close the files: */
                close(fd1);
                if (close(fd2) == ERROR)
                        perror("Error closing %s\n",argv[2]);
                printf("Copy complete\n");
        }

        perror(format,arg)        /* print error message and abort          */
        {
                printf(format, arg);      /* print message   */
                fabort(fd2);              /* abort file operations */
                exit();                   /* return to CP/M  */
        }
```

Now let's take a look at the program. First come the declarations: we need a file descriptor for each file involved in the copying process, and a large array to buffer up the data as we shuffle chunks of disk files through memory. The size of the buffer is computed as the sector size (defined in BDSCIO.H) times the number of sectors of buffering desired (defined at the top of this program as BUFSECTS).

In the "main" function, the first thing to do is make sure the correct number of arguments were given on the command line. Since the `argc' parameter is provided free by the run-time package to every main program, and is always equal to the number of arguments given PLUS ONE, we test to make sure it is equal to three (i.e, that two arguments were given). If argc is not equal to three, we call "perror" to print out a complaint and abort the program. "Perror" interprets its arguments as if they were the first two arguments to a "printf" call, performs the required "printf" call, aborts operations on the output file (this wouldn't have any effect if called before the file is opened; this would be the case if the "argc != 3" test succeeds), and exits to CP/M.

If we make it past the argc test, it is time to try opening files. The next statement opens the master file for reading, assigns the file descriptor returned by "open" to the variable `fd1', and causes the program to be aborted if "open" returned an error. This can all done at one time thanks to the power of the C expression evaluator; if you aren't used to seeing this much happen in one statement, take a moment to follow the parenthesization carefully. First the call to "open" is performed, then the assignment to `fd1' of the return value from "open", and then the test to see if that value was ERROR. If the value was NOT equal to ERROR, control will pass onto the next `if' statement; otherwise, the appropriate call to "perror" diagnoses the problem and terminates the program. Creating the output file follows exactly the same pattern.

Having made it through all the preliminaries, it is time to start copying some data (finally!). Each time through the `while' loop, we read as much as we can get (up to BUFSECTS sectors) into memory from the master file. The "read" function returns the number of sectors successfully read; this may range from 0 (indicating an end-of-file [EOF] condition) up to the number of sectors requested (in this case, BUFSECTS), with a value of ERROR being returned on disaster (when the disk drive door pops open or something). Whatever this value may be, it is assigned to `oksects' for later examination. In the special case when it is equal to zero, indicating EOF, the "while" loop will be exited. Otherwise, we enter the loop and attempt to write back out the data that we just read in. First, though, we want to make sure no gross error occurred, so a check is performed to see if ERROR was returned by the "read" call. If so, it's Abortsville. Having safely circumnavigated Abortsville, we call "write" to dump the data into the output file. If we don't succeed in writing the number of sectors we want to write, it's back to Abortsville with an appropriate error message (most write errors are caused by running out of disk space.) If the "write" succeeds, we go back to the top of the loop and try to read some more data.

The last thing to do, once the "while" loop has been left, is to mop up by closing the files; just to be complete, we check to make sure the output file has closed correctly. And that's it.

The raw file I/O functions are most useful when large amounts of data, preferably in even sector-sized chunks, need to be manipulated. The preceding file-copy program is a typical application. Raw file I/O requires you to always think in terms of "sectors"--while this poses no particular problem in, say, the file-copy example, it does add quite a bit of complexity to shuffling bits and pieces of randomly-sized data. Consider, for example, the unit known as the "text-line": A line's worth of ASCII data may vary in size anywhere from 1 byte (in the case of a null string, represented by the terminating null only) up to somewhere around 133 bytes, or maybe even more if you're dealing with some really fancy printing device. Anyway, some convenient method to read and write these text-lines to and from disk files would be a very useful thing for text processing applications. Ideally we'd like to be able to call a single function, passing to it some kind of file descriptor and a pointer to a text-line, and let the function write the text-line into the file so that it immediately follows the last line written to that file. Also, to prevent a time-consuming disk access every time a line is written, it would be nice to have our function collect up a bunch of lines and toss them all to disk at once when the "buffer" fills up. Analogously there would have to be a function to read a text-line from some disk file into a given place in memory; here, also, it would greatly improve performance if an invisible buffer was managed by the text-line-grabbing function so that disk activity is minimized. The functions described here are, in fact, "fputs" and "fgets" from the library: two of the "buffered I/O" functions.

The spotlight in the world of buffered I/O is a structure called, amazingly, an "I/O buffer". Within this structure is a large, even-sector sized character array within which the data being transferred is stored, and several assorted pointers and descriptors to keep track of "what's happening" in the data array portion of the buffer. There's a file descriptor to identify the file in raw I/O operations, there's a pointer into the data array to tell where the next byte shall be read from or written to, and there's a counter to tell how many bytes of either data or space (depending on whether you're reading or writing) are left before it becomes necessary to reload or dump the buffer. (1)

Buffered I/O functions use pointers to I/O buffers just as the raw functions use file descriptors. There are six functions that perform all actual buffered I/O for single bytes of data; the other buffered I/O functions (such as "fputs" and "fgets") do their stuff in terms of the six "backbone" functions.

For reading files we have "fopen", "getc", and "fclose". "Fopen" is called to associate an existing input file with a user-provided I/O buffer area by initializing all the variables in that buffer. "Getc" grabs a byte from the buffer, first refilling the data array from disk whenever the array is found to be empty, and returns a special value (EOF) when the end of the file is reached. "Fclose" closes the file associated with an I/O buffer.

For writing files there are "fcreat", "putc", "fflush", and "fclose" again ("fclose" leads a double existence.) "Fcreat" creates a new file and prepares an associated I/O buffer structure for recieving data. The data is written to the buffer via calls to "putc", one byte at a time. When all the data has been "putc"-ed, "fflush" is called to dump out the contents of the not-yet-full I/O buffer to the disk file. Finally, "fclose" wraps things up by closing the associated file.

The only functions that actually read and write data are "getc" and "putc"; functions such as "fgets", "fputs", "fprintf", etc. do their reading and writing in terms of "getc" and "putc".

Let's look at a simple first example. The following program prints a given text file out on the console, with line numbers generated on the left margin:

----

1. The devious user may wonder why there is space taken for a byte counter, when the data pointer could just as well be compared to the last array address to detect a full/empty buffer. Actually, it ends up being more efficient with the counter, because the code required to compare two addresses is usually bulkier than the code required to decrement a counter and test for zero.

```
/*
        PNUM.C: Program to print out a text file with
                automatic generation of line numbers.
*/

#include "bdscio.h"

main(argc,argv)
char **argv;
{
        char ibuf[BUFSIZ];      /* declare I/O buffer       */
        char linbuf[MAXLINE];   /* temporary line buffer    */
        int lineno;             /* line number variabele    */

        if (argc != 2) {        /* make sure file was given */
                printf("Usage: A>pnum filename <cr> \n");
                exit();
         }

        if (fopen(argv[1],ibuf) == ERROR) {
                printf("Can't open %s\n",argv[1]);
                exit();
         }

        lineno = 1;             /* initialize line number   */

        while (fgets(linbuf,ibuf))
                printf("%3d: %s",lineno++,linbuf);

        fclose(ibuf);
}
```

The declaration of `ibuf' provides the I/O buffer area for use with "fopen", "getc" and "fclose". The symbolic constant "BUFSIZ", defined within the BDSCIO.H header file, tells how many bytes an I/O buffer must contain; this value will vary with the number of sectors desired for data buffering. See BDSCIO.H for instructions on how to customize the buffered I/O mechanism for a different buffer size (the default is eight sectors).

After checking the argument count and opening the specified file for buffered input, all the REAL work takes place in one simple "while" statement. First the "fgets" function reads a line of text from the file and places it into the `linbuf' array. As long as the end of file isn't encountered, "fgets" will return a non-zero (true) value and the body of the "while" statement will be executed. The body consists of a single call to "printf", in which the current line number is printed out followed by a colon, space, and the current text line. After the value of `lineno' is used, it is incremented (by the ++ operator) in preperation for the next iteration. The cycle of reading and printing lines continues until "fgets" returns zero; at that point the "while" loop is abandoned and "fclose" wraps things up.

For our final example we have the kind of program known as a "filter". Generally, a filter reads an input file, performs some kind of transformation on it, and writes the result out into a new output file. The transformation might be quite complex (like a C compilation) or it might be as trivial as the conversion of an input text file to upper case. Since printing costs are pretty high these days, let's skip the C compiler for the time being and take a look at a To-Upper-Case filter program:

```
#include "bdscio.h"

main(argc,argv)
char **argv;
{
        char ibuf[BUFSIZ], obuf[BUFSIZ];
        int c;

        if (argc != 3) {
                printf("Usage: A>ucase file newfile <cr> \n");
                exit();
         }
        if (fopen(argv[1],ibuf) == ERROR) {
                printf("Can't open %s\n",argv[1]);
                exit();
         }
        if (fcreat(argv[2],obuf) == ERROR)
                printf("Can't create %s\n",argv[2]);
                exit();
         }

        while ((c = getc(ibuf)) != EOF && c != CPMEOF)
                if (putc(toupper(c),obuf) == ERROR) {
                        printf("Write error; disk probably full\n");
                        exit();
                }

        putc(CPMEOF,obuf);
        fflush(obuf);
        fclose(obuf);
        fclose(ibuf);
}
```

This time there are two buffered I/O streams to be dealt with: the input file and the output file. The first thing to do is check for the correct number of arguments (in this case, two: the name of an existing input file, and the name of the output file to be created). Then "fopen" and "fcreat" are called, to open and create the two files for buffered I/O. If that much succeeds, the main loop is entered and the fun begins. On each iteration of the loop, a single byte is grabbed from the input file and compared with the two possible end-of-text-file values: EOF and CPMEOF. Normally, the last thing in a text file SHOULD be a CPMEOF (control-Z) character. But, some text editors (none that I use) neglect to place the CPMEOF character at the end of a file if the file happens to end exactly on a sector boundary; in this case, CPMEOF will never be seen and the physical end-of-file value (EOF) must be detected. The complication this causes is rather tricky...the EOF value returned by "getc" is -1, which must be represented as a 16-bit value because "char" variables in BDS C cannot take on negative values. This is why the variable `c' is declared as an "int" instead of a "char" in the above program; if it were declared as a "char", then the sub-expression

```
        c = getc(ibuf)
```

would result in a value having the type "char" and could never possibly equal EOF as tested for in the program. Should "getc" ever return EOF in such a case, `c' would end up being equal to 255 (the "char" interpretation of the low order 8 bits of the value EOF). Thus, `c' is declared as an "int" so that the EOF comparison can make sense. This is awkward because `c' is used here for holding characters, and it would be nice to have it declared as a character variable. There's actually a way to do it, at the price of complete generality: if the EOF in the comparison were changed to 255, then `c' would have to be be declared as a "char", and the program would work...EXCEPT for when an actual hex FF (decimal 255) byte is encountered in the input file! Now, while it is a pretty safe bet to assume there aren't any hex FF bytes in your average text file, there may be exceptions. Also, there's no law that says filters can only be written for text files. Consider a program to take a binary file and "unload" it, creating an Intel-format HEX file. Would we want it to halt when the first hex FF is encountered? No, the original method is clearly the most general.

Once having determined that the end-of-file has not been encountered, the body of the "while" statement is executed. Here we use "toupper" to convert the character obtained from "getc" to upper case, and then we use "putc" to write the resulting byte out to the output file. To be neat, errors are checked for: the program terminates if "putc" returns ERROR.

As soon as an end-of-file condition is detected, we write out a final CPMEOF (control-Z) character to terminate the output file. The way this particular program is set up, the CPMEOF will be appended to the output file whether or not the input file ended with a CPMEOF. Next, "fflush" is called to flush the output buffer. This must always be done before closing a buffered output file, to make sure that all characters sent to "putc" since that last time the buffer filled up get written to disk. Finally, "fclose" is used to close the input and output files.

For more examples of the usage of buffered I/O, see CONVERT.C, CCOT.C, TABIFY.C and TELNET.C. Also, take some time to inspect the files BDSCIO.H, STDLIB1.C and STDLIB2.C, which contain the sources of all the buffered I/O functions.