Coffee Into Bugs: libc From Scratch

Part 2: Standard Input and Output

Dave Allison

*“A software engineer is a device for converting coffee into bugs”*

I’m writing my own version of *libc* (the standard C library) from scratch for my 6502 project, a small 6502 based computer and with a C compiler. This is the second part of a series of articles describing how to write a simple version of *libc*.

All this code will be in C (the C99 standard) and the hope is that it is useful as a learning tool for those wanting to learn how to program in C, or those writing their own libc. This is all my own work and I have not copied anything from anywhere else. It’s also not related to my employer.

This article will present the C library’s standard input and output, or ***stdio***, facilities. I’ll show how the FILE struct is defined and how the various input and output functions operate on it and with the facilities provided by the operating system. Omitted from this article are the formatted I/O functions (*printf*, *scanf* and family) as these are sufficiently complex to warrant their own article. Stay tuned.

# Operating System I/O Facilities

All operating systems provide a way to get bytes into and out of the computer. How this is done depends on the operating system design, but most provide a way to read an array of bytes from a device into memory and to write an array of bytes to a device from memory. All POSIX based operating systems will provide two functions in a library that are implemented a system calls into the OS, usually defined something akin to this:

ssize\_t write(int fd, const void\* buf, size\_t buflen);

ssize\_t read(int fd, void\* buf, size\_t buflen);

The ***write*** function takes a buffer full of bytes of length *buflen* and writes it to the file descriptor *fd*. If returns the number of bytes written to the device or 0 if the device has been closed or -1 if an error occurs. It is possible that the function doesn’t write all the bytes as requested and happens frequently when writing to network connections.[[1]](#footnote-1)

Likewise, the ***read*** function reads from a file descriptor into the address passed in *buf*, which has a size of *buflen*. It will not read more than *buflen* bytes into the buffer and will return the number of bytes read. If if returns 0 it means that the device has been closed, or -1 means an error.

In both cases, if an error occurs, -1 is returned and the ***errno*** thread-local variable is set to the error number.

A file descriptor is an integer allocated by the operating system and refers to an open device or other operating system facility. POSIX-style operating systems present a unified device system where a terminal, or file on disk, or a network connection are all referred to by a file descriptor. A file descriptor is allocated when a file is opened, or a network connection is created, etc.

Each type of device or facility has its own function to open it and allocate a file descriptor. For example, to open a file on disk, or a device attached to a serial line, the ***open*** function is used, usually defined as:

int open(const char\* path, int oflag, ...);

This takes the name of a device, which could be the name of a file on a disk, or a character device, like /***dev/ttyS0***, or a pseudo-file like ***/proc/self/fd***, opens it and allocates an unused file descriptor that is returned from the function. An error will result in -1 being returned and *errno* being set. The oflag parameter, and any additional parameters are used to tell the operating system how to open the device, whether it’s being opened for reading or writing, etc.

Once a program is done with a file or device it can close it using the ***close*** function:

int close(int fd);

This makes the file descriptor no longer available for performing I/O.

At program startup, the C library will allocate three file descriptors that will be available for use by the program when main is called. These are always given the numbers 0, 1 and 2. The first one is connected to the *standard input* of the program – usually the terminal. The second and third are *standard output* and *standard error* respectively, which are also usually connected to the terminal in which the program is being run. The operating system will usually provide ways to redirect these to other places.

Here’s a small example function to copy the contents of a file.

#include <unistd.h>

#include <fcntl.h>

#include <errno.h>

int CopyFile(const char\* from, const char\* to) {

int infd = open(from, O\_RDONLY);

if (infd == -1) {

return errno;

}

int outfd = open(to, O\_WRONLY|O\_CREAT|O\_TRUNC, 0444);

if (outfd == -1) {

close(infd);

return errno;

}

char buf[256];

for (;;) {

ssize\_t n = read(infd, buf, sizeof(buf));

if (n < 0) {

return errno;

}

if (n == 0) {

break;

}

ssize\_t bytes\_written = write(outfd, buf, n);

if (bytes\_written < 0) {

return errno;

}

if (bytes\_written != n) {

return EIO;

}

}

close(infd);

close(outfd);

return 0;

}

The function takes two arguments:

1. The name of a file to copy from
2. The name of a file to copy to.

It opens both files and reads from the source file into a buffer, which is then written to the destination file. The loop terminates when we reach end of file (EOF), signaled by read returning 0. If an error occurs, the *CopyFile* function returns the value of *errno*, otherwise it returns 0.

The calls to open pass different values for the *oflag* argument. In the case of opening the file for read, the value O\_RDONLY (open read-only) is passed. For the destination file, we pass three flags: O\_WRONLY, O\_CREAT and O\_TRUNC. Together these mean that the file is opened write-only and if it doesn’t exist it is created. If it does exist, it is truncated to zero length.

Anyway, this article isn’t about POSIX I/O, so let’s leave it there and discuss the C library’s facilities.

# C Library Input and Output

From reading the previous section you could be forgiven for thinking that the operating system provides all the input and output facilities you could want in a program. You can open and close files, read and write data. Why, then, does the C library provide a whole parallel set of functions that basically do the same thing?

I’ve been wondering what the original reasoning was behind the abstraction that the C library designers chose back in the 1960’s. A couple of thoughts spring to mind:

1. Buffering. Reading and writing using the operating system directly can result in heavy use of system calls, especially if the amount of data being read or written isn’t known. Adding a layer of buffers between the C program and the OS system calls can improve performance.
2. Portability. POSIX operating systems have well defined functions and a unified device infrastructure using the file system and file descriptors. However, not all operating systems have the equivalent. On some, perhaps older systems, a descriptor for a file might be completely different from a descriptor for a network socket or a terminal. Also, an operating system might not provide read and write functions like POSIX does, leading to extensive changes to port code among operating systems.
3. Ease of use. Opening files on POSIX is a bit of a pain. You have to remember what flags to use and then check for errors and EOF.
4. Type safety. Being a statically typed language, using an int as a file descriptor isn’t a great choice. It would be better to have a specific *FileDescriptor* type that can only be used when talking to I/O services. This would help prevent programming errors.

The C library’s standard input and output facilities are defined in the <stdio.h> header file and comprise of a structure called ***FILE*** and a set of functions that use it.

# The FILE Structure

The ***FILE*** structure is the interface between the C program and the underlying operating system system calls. We are using a POSIX-style operating system for the implementation of this standard I/O library so the FILE struct is defined as:

typedef struct {

int fd; // OS file descriptor.

char\* buf; // Buffer (or NULL).

int bufsize; // Buffer size (or 0).

int rindex; // Read index into buf.

int rlimit; // Limit of chars to read.

int windex; // Write index.

mode\_t buffering\_mode; // \_IOFBF, \_IOLBF or \_IONBF

char buffer\_owned; // The buffer is owned by this FILE.

char unget\_index; // Index into unget\_buf for ungotten bytes.

char eof\_flag; // 1 if EOF reached.

char error\_flag; // 1 for error condition.

char unget\_buf[10]; // Buffer for ungetc.

struct FILE\* prev;

struct FILE\* next;

} FILE;

All open files contain the file descriptor allocated by the operating system and this is held in the **fd** member of *FILE*. The *buf* and *bufsize* members together describe a buffer that holds data to be written to the next *write* system call, or data read from the most recent call to *read*. The integer members *rindex* and *windex* are the current read and write indices into the buffer (the next index to read a byte from or write into). The *rlimit* member is the index of the last character present in the buffer when reading. The FILE may be in one of 3 buffering modes, held in the *buffering\_mode* member and defined as:

#define \_IOFBF 1 // Full buffering.

#define \_IOLBF 2 // Line buffering.

#define \_IONBF 3 // No buffering.

*Full buffering* means that the buffer is only written to the operating system when it is full, or when explicitly flushed by the program. *Line buffering* means that the buffer is written to the operating system when a newline character is written, or when reading from the OS, the read terminates when a newline is read. *No buffering* means that there is no buffer allocated and all calls to the stdio functions will result in a system call.

The *buffer\_owned* flag says whether this FILE owns the buffer (allocated from the heap), or if the buffer is statically allocated.

The C library contains a function to undo reads, called *ungetc*. The FILE contains a small buffer to hold the bytes pushed back by calls to *ungetc*, with *unget\_index* containing the index of the next available space for a new character.

Two flags, *eof\_flag* and *error\_flag* say whether we are at a current end-of-file or error condition.

All *FILE* structs are kept in a double linked list so that the *fflush* function can flush all open files if it’s passed a NULL pointer. Also, C runtime systems need to flush all open files before the program exits. The *prev* and *next* members point to the previous and next *FILE* in the linked list respectively.

## Opening and closing a FILE

There are 3 FILE structs statically allocated by the C library, corresponding to the file descriptors 0, 1 and 2. These are initialized before the program’s main function is called. Here’s the definition of the standard files:

#if defined(\_\_6502\_\_)

#define BUFSIZE 64

#else

#define BUFSIZE 4096

#endif

#define BUFSIZ BUFSIZE

static char s\_stdin\_buf[BUFSIZE];

static char s\_stdout\_buf[BUFSIZE];

static FILE s\_stdin, s\_stdout, s\_stderr;

static FILE s\_stdin = {.fd = 0,

.buf = s\_stdin\_buf,

.bufsize = BUFSIZE,

.buffering\_mode = \_IOLBF,

.prev = NULL,

.next = &s\_stdout};

static FILE s\_stdout = {.fd = 1,

.buf = s\_stdout\_buf,

.bufsize = BUFSIZE,

.buffering\_mode = \_IOLBF,

.prev = &s\_stdin,

.next = &s\_stderr};

static FILE s\_stderr = {.fd = 2,

.buffering\_mode = \_IONBF,

.prev = &s\_stdout};

FILE\* stdin = &s\_stdin;

FILE\* stdout = &s\_stdout;

FILE\* stderr = &s\_stderr;

FILE\* \_\_all\_files = &s\_stdin;

FILE\* \_\_last\_file = &s\_stderr;

The *BUFSIZE* macro is the default size for the *stdin* and *stdout* buffers. On a small computer, like a 6502, we can only afford 64 bytes, but normally this is something reasonable, like 4K.

The *s\_stdin\_buf* and *s\_stdout\_buf* arrays are the statically allocated buffers for *stdin* and *stdout* respectively. There is no *stderr* buffer as it unbuffered.

The three static *FILE* variables *s\_stdin*, *s\_stdout* and *s\_stderr* are initialized statically, with the three external variables *stdin*, *stdout* and *stderr* pointing to them. Both *stdin’s* and *stdout’s* buffering mode are set to \_*IOLBF* (line buffered), while *stderr* is \_*IONBF* (no buffering).

The *prev* and *next* pointers of the three *FILE* structures are arranged so that the ***\_\_all\_files*** points to a list containing *s\_stdin*, *s\_stdout* and *s\_stderr* in that order. The ***\_\_last\_file*** variable points to the element at the end of the list, *s\_stderr*. This linked list is maintained by *fopen* and *fclose*.

Aside from *stdin*, *stdout* and *stderr*, new *FILE* struct is created by calling the ***fopen*** function. This maps onto the POSIX *open* function. Here’s the *fopen* function:

FILE\* fopen(const char\* filename, const char\* mode) {

int open\_mode = 0;

const char\* m = mode;

while (\*m != '\0') {

if (\*m == 'r') {

open\_mode |= 1;

} else if (\*m == 'w') {

open\_mode |= 2;

} else if (\*m == 'a') {

open\_mode |= 4;

} else {

return NULL;

}

m++;

}

if (open\_mode == 3) {

open\_mode = O\_RDWR;

} else if (open\_mode == 1) {

open\_mode = O\_RDONLY;

} else if (open\_mode == 2) {

open\_mode = O\_WRONLY | O\_TRUNC | O\_CREAT;

} else if (open\_mode == 4) {

open\_mode = O\_WRONLY | O\_APPEND | O\_CREAT;

}

int fd = open(filename, open\_mode, 0777);

if (fd == -1) {

return NULL;

}

// Allocate the FILE and buffer in one block.

FILE\* fp = malloc(sizeof(FILE) + BUFSIZE);

if (fp == NULL) {

return NULL;

}

fp->buf = (char\*)fp + sizeof(FILE);

fp->fd = fd;

fp->bufsize = BUFSIZE;

fp->windex = 0;

fp->rindex = fp->bufsize;

fp->rlimit = 0;

fp->buffer\_owned = 0; // Buffer doesn't need to be freed.

fp->buffering\_mode = \_IOFBF; // Fully buffered.

fp->unget\_index = 0;

fp->eof\_flag = 0;

fp->error\_flag = 0;

fp->next = NULL;

// Link into global list of all files.

\_\_last\_file->next = fp;

fp->prev = \_\_last\_file;

\_\_last\_file = fp;

return fp;

}

The function takes two arguments:

1. The path of the file to open
2. A string containing the open mode. This contains a combination of the characters ‘r’, ‘w’ and ‘a’.

Since the *open* system call takes a set of flags to specify how the file should be opened, we need to translate the open mode into the flags. The character ‘r’ means open for reading, and this maps to the flag O\_RDONLY. The ‘w’ character maps to O\_WRONLY[[2]](#footnote-2) and ‘a’ is O\_APPEND. We also add O\_TRUNC for ‘w’ and O\_CREAT for both ‘w’ and ‘a’ to allow us to truncate existing files or create non-existing ones.[[3]](#footnote-3)

The file is opened by calling *open*. Assuming there is no error, we then allocate memory. There are two chunks of memory needed for an open FILE:

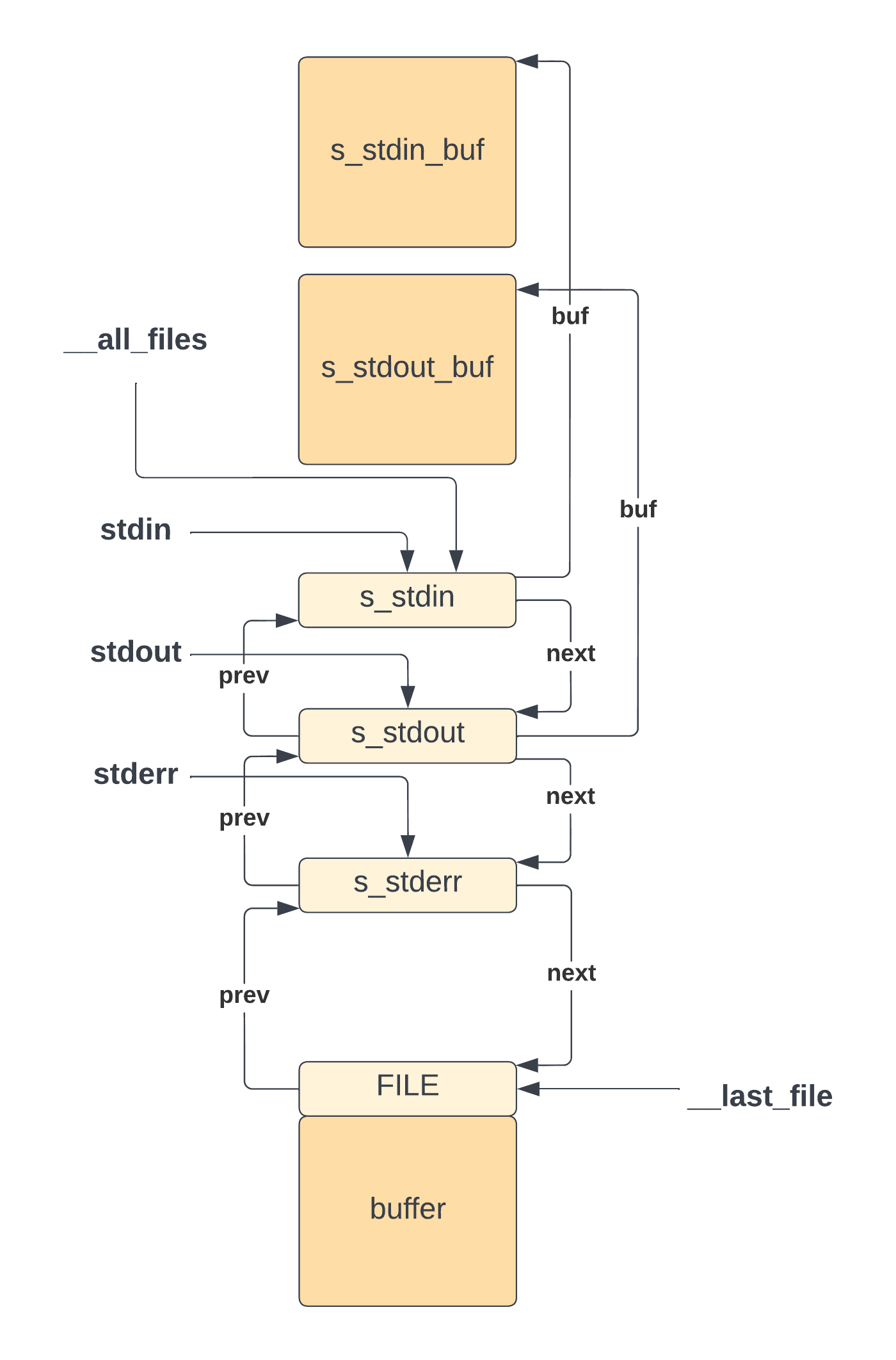
1. The FILE struct itself
2. The buffer.

Rather than allocate these using 2 calls to *malloc*, it is more efficient to allocate one piece of memory sufficiently large to hold both the FILE struct and the buffer. The *FILE* is located at the lowest address with the buffer following immediately. The buffer is always *BUFSIZE* bytes long.

Diagram

Description automatically generated

We then initialize the *FILE* struct and return its address. The buffering mode is set to \_*IOFBF* (fully buffered) but can be changed later if needed. The newly allocated FILE structure (and buffer) is inserted at the end of the *\_\_all\_files* double linked list. Here’s what the list would look like after opening a single file.



To close a file, we need to call the system call close and free up any memory allocated for the *FILE* and buffer. This is done by the function *fclose*:

int fclose(FILE\* fp) {

if (fp == NULL) {

return EOF;

}

int e = fflush(fp);

if (e == EOF) {

return EOF;

}

e = close(fp->fd);

if (e == -1) {

return EOF;

}

// Unlink from previous.

if (fp->prev == NULL) {

\_\_all\_files = fp->next;

} else {

fp->prev->next = fp->next;

}

// Unlink from next.

if (fp->next == NULL) {

\_\_last\_file = fp->prev;

} else {

fp->next->prev = fp->prev;

}

// If the buffer is owned (set by setvbuf), free it.

if (fp->buffer\_owned) {

free(fp->buf);

}

// For all except standard streams the FILE is allocated on the

// heap. Free it.

if (fp->fd > 2) {

free(fp);

}

return 0;

}

The *fclose* function first flushes the buffer by calling *fflush* (see later). It then calls the operating system function *close* to close the file descriptor. If this fails, we return EOF. We then unlink the FILE from the \_*all\_files* list by removing it from its previous and next links. Then we need to free the memory. If the file’s buffer has been changed by a call to *setvbuf*, we free up the buffer it allocated. Then, if we are not *stdin*, *stdout* or *stderr*, we free up the FILE struct (and buffer space).

## Changing the buffering

If the default buffering scheme isn’t suitable for the program, you can change it for a FILE after it is opened. The function *setvbuf* (set virtual buffer, I think) allows a *FILE’s* buffer to be changed to memory allocated by the caller, or allocated from the heap. It is passed 4 arguments:

1. The FILE to be modified
2. A pointer to the buffer to use, or NULL for a heap allocated buffer.
3. The new buffering mode to use.
4. The size of the buffer.

Here’s the definition of *setvbuf*:

int setvbuf(FILE\* restrict stream, char\* restrict buf, mode\_t mode,

size\_t size) {

if (mode != \_IONBF && mode != \_IOLBF && mode != \_IOFBF) {

return -1;

}

if (stream->buf != NULL && stream->buffer\_owned) {

free(stream->buf);

}

if (buf == NULL && mode != \_IONBF) {

stream->buf = malloc(size);

stream->buffer\_owned = 1;

} else {

stream->buf = buf;

stream->buffer\_owned = 0;

}

stream->bufsize = size;

stream->buffering\_mode = mode;

return 0;

}

After validating the buffering mode, the function frees up any old buffer if it is owned by the FILE (from a previous call to *setvbuf*, presumably). It then either uses the buffer passed or, if it is NULL, allocates a new buffer from the heap. The *buffer\_owned* member of *FILE* is set to 1 if the buffer was allocated from the heap.

There is also a shortcut function provided for the common use cases, called *setbuf*, which calls *setvbuf* with either full buffering or no buffering depending on the value passed as the buffer. If NULL is passed, there will be no buffer allocated.

void setbuf(FILE\* restrict stream, char\* restrict buf) {

setvbuf(stream, buf, buf ? \_IOFBF : \_IONBF, BUFSIZ);

}

## Flushing the buffer

The specification for the *fflush* function is as follows:

1. If the stream is NULL, all open files are flushed.
2. If the stream is open for write, the contents of the buffer are written to the operating system
3. If the stream is open for read, all unread characters in the buffer are discarded.

The first requirement means that we need to keep track of all open FILE structs, so they are linked into a double linked list anchored by *\_\_all\_files*. We traverse this list if *stream* is NULL, recursively calling *fflush* for all FILEs in the list. We ignore any errors during this traversal since stopping at an error would prevent any further file flushes.

int fflush(FILE \*stream) {

if (stream == NULL) {

// Flush all files.

FILE\* fp = \_\_all\_files;

while (fp != NULL) {

fflush(fp); // Ignore error if any.

fp = fp->next;

}

return 0;

}

if (stream->buf != NULL) {

// Flush any input chars not read.

stream->rlimit = stream->rindex = 0;

// Flush output to OS.

ssize\_t remaining = stream->windex; // Bytes remaining to write.

stream->windex = 0;

const char\* buf = stream->buf;

// The call to write may not write all bytes in one call.

while (remaining > 0) {

ssize\_t nbytes = remaining;

ssize\_t n = write(stream->fd, buf, remaining);

if (n < 0) {

stream->error\_flag = 1;

return EOF;

}

// Update for next iteration.

remaining -= n;

buf += n;

}

}

return 0;

}

The *windex* member of FILE is the index of the next available position into which a write may be done (the first free byte in the buffer) so this holds the number of buffered bytes. You may recall from the section discussing the operating system’s *write* function that it may not write all the requested bytes in one invocation. This can happen when the file descriptor is attached to a device or network socket, but will rarely, if ever, happen when the file is on a disk. When using *write*, it is necessary to look at its return value and use that to ensure that all the bytes are written. The *remaining* local variable says how many bytes remain to be written and is updated by the result of *write*. An error from *write* will result in EOF being returned.

# Writing to a FILE

The standard I/O functions generally don’t write directly to the operating system unless the *FILE* is unbuffered. Instead, they write to the internal *FILE* buffer which is only written to the OS when the buffer is full or, if line buffered, when a newline character is written.

The function to write a character to a *FILE* is ***fputc***, defined as:

int fputc(char\_t c, FILE\* stream) {

if (stream->buf == NULL) {

char ch = c;

ssize\_t remaining = 1;

while (remaining > 0) {

ssize\_t n = write(stream->fd, &ch, 1);

if (n < 0) {

return EOF;

}

remaining -= n;

}

return c;

}

// Buffer full?

if (stream->windex == stream->bufsize) {

int e = fflush(stream);

if (e != 0) {

return e;

}

}

// Add to next position in buffer.

stream->buf[stream->windex++] = c;

if (c == '\n' && stream->buffering\_mode == \_IOLBF) {

// Flush on newline.

return fflush(stream);

}

return 0;

}

It first checks if the *FILE* is unbuffered and if so, calls the operating system’s *write* function directly to write a single character. As usual, *write* may not work first time, so we loop until it returns an error or successfully writes the character.

If the *FILE* is buffered, we are going to add another byte to it. If there is no room to add the byte, we flush the buffer using *fflush* and return EOF if we get an error. The *fflush* function will reset the *windex* member of *FILE* to zero thus guaranteeing there will be space for the new byte. We add the byte to the buffer at index *windex*, and increment *windex* by 1.

Finally, if we wrote a newline character and the *FILE* is line buffered, we call *fflush* to write the current buffer to the OS. The *stdout* *FILE* is line buffered by default.

There is a shortcut for writing to *stdout*:

int putchar(char\_t c) {

return fputc(c, stdout);

}

## Writing an array

Writing one character at a time to a *FILE* is the basic operation, but it is a little inconvenient when a program wants to write more than one byte at a time. The ***fwrite*** function is provided to write a contiguous sequence of records to a *FILE* where a record is defined as a sequence of contiguous bytes[[4]](#footnote-4).

The *fwrite* function is passed 4 arguments:

1. The address of the buffer containing the records to write
2. The size of each record
3. The number of records to write
4. The FILE pointer to write to.

Here’s the *fwrite* function:

size\_t fwrite(const void\* ptr, size\_t size, size\_t n, FILE\* stream) {

char\* p = ptr;

size\_t len = size \* n;

size\_t numchars = 0;

while (len > 0) {

int v = fputc(\*p++, stream);

if (v == EOF) {

return n;

}

numchars++;

len--;

}

return numchars / size;

}

It is basically a simple loop calling *fputc*. The number of bytes to write is calculated as the product of *size* and *n* (the record size multiplied by the number of records). It returns the number of records written successfully.

## Writing a string

It is a common operation in C to write a *string*, where a string is a sequence of bytes terminated by a zero byte. If we were to use *fwrite* for this, the program would have to determine the length of the string before writing. This is a tad inconvenient, so there’s a function to make it easier: ***fputs***:

int fputs(const char\* str, FILE\* stream) {

const char\* s = str;

while (\*s != '\0') {

if (fputc(\*s++, stream) == EOF) {

return EOF;

}

}

return 0;

}

There’s also a shortcut analogous to *putchar* for writing to *stdout*. This one also writes a newline character after the string.

int puts(void) {

int e = fputs(stream, stdout);

if (e != 0) {

return e;

}

return fputc('\n', stdout);

}

# Reading from a FILE

The ***getc*** and ***fgetc*** functions read a character from an open FILE. According to the standard, they both do the same thing, but *getc* may be implemented by a macro which may be faster than *fgetc*. I’m not going to do that though, so in this case they are both the same thing.

As you are probably aware, *FILE* usually contains a buffer and the *getc* function reads from that buffer. If there are no buffered characters available, it will ask the operating system for some from the file descriptor using its *read* function. When calling *read*, we attempt to fill the *FILE’s* buffer with as many characters as are available. However, if the *FILE* is line buffered, we have to read until we get a newline character, and if the *FILE* is unbuffered we call the OS directly to read a single character.

Here’s the implementation of *getc*:

int getc(FILE\* stream) {

if (stream->buf == NULL) {

// NON-buffered, call read directly.

char rbuf[1];

ssize\_t n = read(stream->fd, rbuf, 1);

if (n <= 0) {

SetErrorOrEof(stream, n);

return EOF;

}

return rbuf[0];

}

// Any chars pushed with ungetc?

if (stream->unget\_index > 0) {

return stream->unget\_buf[--stream->unget\_index];

}

if (stream->rindex < stream->rlimit) {

// Char available in buffer.

return stream->buf[stream->rindex++];

}

// Nothing available in the buffer.

if (stream->error\_flag != 0 || stream->eof\_flag != 0) {

return EOF;

}

stream->rindex = 0;

stream->rlimit = 0;

if (stream->buffering\_mode == \_IOFBF) {

// Fully buffered, use read for full buffer.

ReadFullBuffer(stream);

} else {

char\* p = stream->buf + stream->rlimit;

for (;;) {

ssize\_t n = read(stream->fd, p, 1);

if (n <= 0) {

SetErrorOrEof(stream, n);

break;

}

char ch = \*p++;

stream->rlimit++;

if (stream->buffering\_mode == \_IOLBF && ch == '\n') {

break;

}

if (stream->rlimit == stream->bufsize) {

break;

}

}

}

if (stream->error\_flag != 0 || stream->eof\_flag != 0) {

return EOF;

}

return stream->buf[stream->rindex++];

}

The first check is for an unbuffered FILE, in which case we read a single byte from the OS. If the *read* call returns 0 or less, we call the helper function ***SetErrorOrEof*** to set the *FILE’s* flags as appropriate. Here’s that function:

static void SetErrorOrEof(FILE\* stream, ssize\_t n) {

if (n < 0) {

stream->error\_flag = 1;

} else {

stream->eof\_flag = 1;

}

}

Next, we check if that are any characters in the unget buffer, placed there by the ***ungetc*** function and if so, take the last one out of that buffer and decrement the unget counter.

If there are no ungotten characters, we look for the next unread character in the *FILE’s* buffer. If there is one, signified by *rindex* being less than *rlimit*, we take that character and increment *rindex* for the next time. If we reach the end of the available characters, *rindex* will equal *rlimit*.

If we reach the point where there are no buffered characters available, we need to get some more from the OS. How this is done depends on the buffering mode. If we are fully buffered, we try to read as many characters as possible and fill the buffer. This is done by a call to the helper function ***ReadFullBuffer***, defined as follows:

static void ReadFullBuffer(FILE\* stream) {

char\* p = stream->buf;

int remaining = stream->bufsize;

while (remaining > 0) {

ssize\_t n = read(stream->fd, p, remaining);

if (n <= 0) {

SetErrorOrEof(stream, n);

break;

}

stream->rlimit += n;

p += n;

remaining -= n;

}

}

This keeps calling *read* until we fill up the buffer or we run out of characters to read. It sets *rlimit* to the index just beyond the last character read.

The last alternative is that the *FILE* is line buffered, in which case we need to read a single character at a time, adding it to the buffer until we fill the buffer or read a newline character.

After reading into the buffer from the OS, we return the first available character, unless we are at EOF or in error.

That was the *getc* function, but we also need to define *fgetc*:

int fgetc(FILE\* stream) {

return getc(stream);

}

There’s also a shortcut for reading from *stdin*:

int getchar(void) {

return getc(stdin);

}

## Ungetting a character

Say we are writing code for a compiler’s Lexical Analyzer that is reading from a *FILE*. We’ve come across a digit and want to read the number, converting it to decimal as we read. Something like this:

int ReadNumber(FILE\* fp, int ch) {

int n = ch - '0';

while (!feof(fp)) {

ch = getc(fp);

if (ch == EOF) {

break;

}

if (!isdigit(ch)) {

ungetc(ch, fp);

break;

}

n = n \* 10 + ch - '0';

}

return n;

}

We’ve already read the first character and passed it in *ch*. We then read characters until we reach EOF or a non-digit. If we read a digit, we accumulate it into the result in decimal, which is returned. Notice, however, if we read a non-digit we need to unget the character we’ve just read. If we don’t do that, the character will be skipped because the caller has no way to know what it was.

Let’s look at how this function would be used in a hypothetical Lexical Analyzer:

void Lex(FILE\* fp) {

while (!feof(fp)) {

int ch = getc(fp);

if (isdigit(ch)) {

number = ReadNumber(fp, ch);

token = NUMBER;

break;

} else if (isalpha(ch)) {

// Handle token starting with letter...

}

}

}

So, we are processing a FILE and have read a character into *ch*. If this is a digit, we call *ReadNumber* to read it in decimal and set the variable *number* to the value and *token* to NUMBER (presumably something will look at token and know we’ve got a number). If the character wasn’t a digit, we check for other token-starting characters (letters, quote marks, etc.) and handle those too (not shown).

If the *ReadNumber* function doesn’t unget the character after the digit, the next iteration of this loop will just read the next character, ignoring the one read by *ReadNumber*, which means we’ve ignored a character in the input – potential disaster for a compiler.

To handle this in our FILE implementation we have a 10-byte buffer that is used to hold a stack of characters we’ve called *ungetc* for. We also keep track of the index into this buffer. Here’s the implementation of *ungetc*:

int ungetc(int ch, FILE\* stream) {

if (ch == EOF) {

return EOF;

}

if (stream->unget\_index == sizeof(stream->unget\_buf)) {

return EOF;

}

stream->eof\_flag = 0;

stream->unget\_buf[stream->unget\_index++] = ch;

return ch;

}

Pretty simple.

## Reading an array

Akin to writing an array, we also have the ability to read an array of bytes from a *FILE*. The function is called, no surprises here, ***fread*** and follows the same signature as its sister function *fwrite*. Here’s the definition of *fread*:

size\_t fread(void \* restrict ptr, size\_t size, size\_t n, FILE\* stream) {

size\_t nbytes = size \* n;

int ch;

size\_t nread = 0;

char\* p = ptr;

while (nbytes > 0 && (ch = fgetc(stream)) != EOF) {

\*p++ = ch;

nread++;

nbytes--;

}

return nread / size;

}

It also has the design feature of reading a sequence of contiguous records with a given size[[5]](#footnote-5). It reads characters from the stream using *fgetc* (*getc* would also work) until it has read the required number of bytes or we get an EOF or error.

## Reading a line

There’s also a function that allows you to read a line, where a line is a sequence of bytes terminated by a newline byte. There are actually two of these available: one that keeps the newline intact, and one that reads only from stdin and discards the terminating newline. The functions are called, respectively, ***fgets*** and ***gets***. It turns out that the *gets* function is really rather dangerous to use since it does take a length argument and thus you can overwrite all kinds of memory in your program if you use it. In fact, some compilers refuse to let you call it because it’s so dangerous.

Here’s the definition of *fgets*, the safe function:

char \*fgets(char \* restrict s, int n,

FILE \* restrict stream) {

n--; // Space for newline.

char\* p = s;

while (n > 0) {

char ch = fgetc(stream);

if (ch == EOF) {

stream->eof\_flag = 1;

if (p == s) {

return NULL;

}

break;

}

\*p++ = ch;

if (ch == '\n') {

\*p = '\0';

break;

}

n--;

}

return s;

}

It takes the address of where you want the data stored, the max length of the data to read, and the *FILE* from which the reading is done. It will write the bytes read sequentially into the buffer until it reaches the limit, we get an EOF or error, or we get a newline character. When a newline is read, it will append a NUL byte (0) to the end. The newline is kept intact.

For completeness, here’s the definition of *gets*:

char\* gets(char \*str) {

char\* p = str;

for (;;) {

char ch = fgetc(stdin);

if (ch == EOF) {

stdin->eof\_flag = 1;

if (p == str) {

return NULL;

}

break;

}

\*p++ = ch;

if (ch == '\n') {

p[-1] = '\0';

break;

}

}

return str;

}

The intention is that it can be used to read a string from the input stream, but the lack of a length argument makes it completely useless for that. There’s not much more to say about it, other than you would be asking for trouble if you use it in a program meant to work.

# Moving around

If your file resides on a seekable medium, like a disk, it’s possible to move the current position in the file and read or write to different locations within the file. The final stop in our exploration of C’s standard I/O facilities is a set of functions that you can use to change the current position and find out what the current position is set to.

The functions are:

1. ***fseek*** – seek to a new position
2. ***ftell*** – what is the current position
3. ***fsetpos*** – another way to seek
4. ***fgetpos*** – another way to find the current position
5. ***rewind*** – seek to the start

The reason for the existence of fseek and fsetpos (and ftell and fgetpos) is to provide a way to use an opaque position indicator that is not necessarily a long integer. This is to provide portability among operating systems and for larger files. In this implementation, we will be ignoring that and just making them equivalent.

To move to a new location in a seekable file, the operating system provides a function called ***lseek***. This is almost identical to fseek except it takes a file descriptor rather than a FILE pointer. This is almost analogous to the relationship between *fwrite* and *write*. The *fseek* function is a direct mapping onto *lseek* as follows:

int fseek(FILE \*stream, long int offset, int whence) {

fflush(stream);

int e = lseek(stream->fd, offset, whence);

if (e == -1) {

return e;

}

stream->eof\_flag = 0;

stream->unget\_index = 0;

return 0;

}

The specification for fseek says that it flushes the file and, if successful, clears the EOF and unget buffer. The *whence* argument is one of:

1. SEEK\_SET – move to the given offset
2. SEEK\_CUR – move relative to the current offset
3. SEEK\_END – more relative to the end of file

To find the current position we make use of the fact that SEEK\_CUR is the current position and the *lseek* function returns the new position. We can call *lseek* with a 0 offset from SEEK\_CUR and lseek will tell us the current position.

long int ftell(FILE \*stream) {

fflush(stream);

return lseek(stream->fd, 0, SEEK\_CUR);

}

The *rewind* function is simply an *fseek* to offset 0 in the file, but it also clears the error state for some reason.

void rewind(FILE \*stream) {

int e = fseek(stream, 0, SEEK\_SET);

if (e == 0) {

stream->error\_flag = 0;

}

}

The *fsetpos* and *fgetpos* functions provide a slightly different interface but are essentially the same as *fseek* and *ftell* respectively.

int fgetpos(FILE \* restrict stream, fpos\_t \* restrict pos) {

long p = ftell(stream);

if (p == -1) {

return -1;

}

\*pos = p;

return 0;

}

int fsetpos(FILE \*stream, const fpos\_t \*pos) {

fpos\_t p = fseek(stream, \*pos, SEEK\_SET);

if (p == -1) {

return -1;

}

return 0;

}

# Detecting EOF etc.

The final three functions allow you to detect if the FILE is in an EOF or error condition, and clear the error.

int feof(FILE\* stream) { return stream->eof\_flag; }

int ferror(FILE\* stream) { return stream->error\_flag; }

void clearerr(FILE\* stream) { stream->error\_flag = 0; }

1. It’s also possible that -1 is returned but errno is set to EAGAIN (or EWOULDBLOCK) which means that a non-blocking read or write cannot perform the action and the program should try again. [↑](#footnote-ref-1)
2. If both O\_RDONLY and O\_WRONLY are set, this corresponds to O\_RDWR (read-write) [↑](#footnote-ref-2)
3. This function doesn’t handle inconsistent open modes correctly. More error checking is needed. [↑](#footnote-ref-3)
4. I’ve always wondered by fwrite is defined this way when most C functions deal with bytes directly and don’t group them into records. I guess it was to allow an array of structs to be handled directly, but it does seem incongruous to the C style. [↑](#footnote-ref-4)
5. It occurs to me that if the designers hadn’t chosen to do this record-based approach, the functions could have avoided the integer multiplication and divisions. Those are onerous for a processor like a 6502. [↑](#footnote-ref-5)