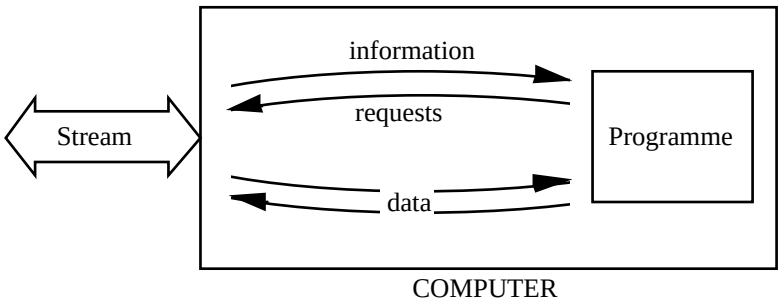
STREAMS IN PROGRAMMES

Streams ? Why not files ? – because, according to our argument in the *STREAMS* chapter, programmes are stream operators. A programme might have to refer to a file to identify it, but once the file is open and in use the programme deals with the stream of data produced from the file (or to be directed into the file) by the operating system, and this stream is at the lowest level quite indistinguishable from a stream coming from or going to a device like a keyboard or screen. In fact, we shall find it necessary to discuss both files and streams in this chapter, but within the programme streams are the important data structure – though we usually call them files anyway.

This is still true even if the stream is associated with a random access file. Within the programme, we are concerned only with the stream of records which come from or go to the file; it is up to the operating system to provide software to manage the translation between static file and stream. Of course, the programme will have to provide the operating system with the information it uses to set up the required sort of translation, and it might be that the programme will have to do other things to the translation procedures to make the stream work properly (such as provide record numbers) – but, whatever that is, it's another sequence of actions, and can always be regarded as a control stream operating together with the more obvious data stream.

And that's usually true : in any transaction between programme and external world, we commonly require both data transfer and some exchange of instructions to the device and information about the device. We can illustrate the system like this :



If we are to use streams in our programmes, the first essential is to provide means to talk about them in the programmes. A major obstacle to this requirement is that, in most cases, we *don't* talk about streams in our programmes; we saw earlier (*STREAMS*) that it was usually appropriate for programmes to be expressed in terms of files rather than streams. More precisely, then, the first essential is to make the compilers (or other translating software) convert instructions expressed in terms of file operations into actual stream operations. (Notice that this fits in very well with the analogy between files and memory made in the *STREAMS* chapter : in both cases, the compiler must convert instructions phrased in terms of the static entities into operations which move data about.) We'll assume in our discussion that the programme is written in terms of files, which is certainly the common case at present – even when dealing with terminals, which really are streams.

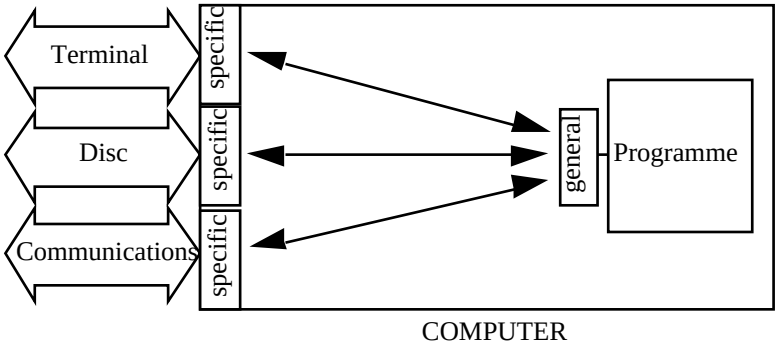
The programme will contain some instructions which refer to the files which it will use. Just how this is done is a matter for programming language design, not operating systems – some languages use names, some numbers, some assumptions – but whatever the mechanism the instructions for each file must in some way be linked to a representation of the corresponding stream maintained in memory when the programme runs.

Now, that might seem obvious, but in practice it's something of an embarrassment, because we can't use the straightforward method, which is to declare a named variable of the appropriate structure in the programme and use it as we would any other structure. We can't do it, because it's unsafe; we've already commented on that in *ONWARDS AND UPWARDS – OPERATING SYSTEMS*. The stream representation contains sensitive information which could lead to unfortunate consequences, accidental or deliberate, if it were accessible in a programme's ordinary address space. The usual solution is to store

the real representations in a table accessible only to the system, with items of the table identified in the programme by their indices alone, and communication through supervisor calls. As the indices are simple integers, they can safely be passed back to the programme, which must provide them as stream identifiers whenever a stream operation supervisor call is executed. In a well designed language such as Cobol or Pascal (not *always* well designed), the integers will be disguised as file identifiers, so that you can't do arithmetic on them; in other cases (C, Fortran) they are left as integers.

Not the least inconvenient consequence of this requirement is that we have to find another name for the index, which is difficult because it has no need to exist which is obviously connected with its function. In Unix, it's called a "file descriptor", but that name is commonly associated with something else, which we shall discuss shortly. In MS-DOS the variable is called a "file handle"; we shall adopt this name henceforth for the sole reason that it isn't used for any other purpose so far as we know. (We would rather call it a "stream handle", but we stick with the existing name to avoid yet further proliferation of terminology.)

The supervisor call gets us out of the programme, but only as far as the local table of stream information. Less visibly, we must make sure that we can link the file mentioned in instructions in the programme with an actual stream or file known to the operating system. We must elaborate the previous picture a little to include a hint of the software required to implement these links :



Notice that, for each sort of channel, there is device-specific software to convert the specific device signals into a stream, and there is general software to interface the stream with the programme. The framework for this structure must be provided by the operating system, but the details can only be established, or at least provided for, when the programme is set up in memory for execution. The first steps must be taken by the compiler (or something of the sort); this is an area where the compiler must conform to conventions established by the operating system, or communication between the two will be impossible. (Or both compiler and operating system must conform to some externally defined standard : that's the idea behind standard application programme interfaces.) This doesn't just happen; to make it happen effectively, there must be careful design beforehand. Consider what has to be done.

DESIGNING THE APPLICATION PROGRAMMER INTERFACE.

The details of the API will depend on what you (in your capacity as system designer) want your file system to do for you. At the very least, though, the first diagram in this chapter suggests that there must be provision for two sorts of stream, one for data and one for control operations. From the programming point of view, experience suggests that data and control operations are dealt with separately, so it is reasonable to separate them in the API. A programmer can then write procedures to combine them together if required (as in a random file access operation, which includes both a positioning operation – control – and a data transfer), while the separate operations are available if required.

We shall therefore require some structure somewhere to implement the "requests and information" side of the communication. This structure contains information about the stream which is needed while the stream is being used – such as the stream's current state, where to put material coming from or going to the stream, and so on. This

repository of active information passes under different names in different systems; to compound the confusion we shall call it the *file information block*. (Once again, we aim to avoid proliferating terminology, but in this case the structure might well contain information about a file.)

The file information block is to be distinguished from the information held in the system's file table, which describes the static properties of the file : where it is, how big it is, its protection codes, etc. We shall call such a collection of static information about a file its *file descriptor*. Observe that the file table and its file descriptors are properly so called; they are concerned with real static files. Because of their transient nature, streams need no such persistent structures.

The file information block could be thought of as a part of the programme, or it could belong to the operating system; where *should* it be kept ? Logically, it goes with the programme – when you change the programme, you want to change the file information blocks. Administratively, though, it is much more the concern of the operating system, and in the interests of safety should not be vulnerable to accidental (or deliberate) change by the programme without some sort of check. Early systems associated it with the programme, but nowadays it is more common to regard it as part of the system. The change was associated with the general movement of control towards the operating system, and as more and more stream operations move into the province of the operating system, it becomes more sensible to keep the stream information – or, at least, some of it – there too. It is interesting, though, that the movement of the file information block towards the operating system has been accompanied by a contrary movement into the programme of responsibility for maintaining the block, with the provision of appropriate management facilities in the application programmers' interface.

For the data side of the operation, other system structures are likely to be necessary. Generally, some sort of buffering is provided, and buffer areas must be reserved for use by the stream. In principle, these are rather uncontroversial, so we shall say little more about them here (there's a short discussion after this part); at the implementation level, though, it can be difficult to handle them efficiently, and we shall have more to say about that question in the *IMPLEMENTATION* section.

A third component of the API must be provided for administration – particularly, for setting up and taking down the links between programme and system. At some point, the structures we have just described must be set up, and links established between the programme code and the devices which will be used when it runs. We shall suppose that a compiler deals with the details, and consider what it has to do to establish the communication, which in turn defines the API functions which must be provided.

The compiler first hears of the stream when reading a source programme. It finds that some identifier is declared as the name of a stream, and that the programme describes certain operations – **open**, **close**, **read**, **write**, **seek**, etc. – on the stream. From the compiler's point of view, the stream itself remains a free variable, which must be bound to some system component when the programme is run. The compiler must produce code that will, first, set up the binding between the stream identifier in the programme and a physical stream in the system, and then cause the required stream operations to happen when the programme is executed. The API must therefore contain a library of procedures which the programme can call to request the establishment of the file information block and such buffers as might be needed to link it to a particular device, which must itself be described, somewhere, in more or less detail. What the API procedures do is determined by the design of the system's structures for stream management.

How the file information block is managed does have a noticeable effect on how you use the system. As an example, consider the different ways in which one might wish to bind the file information. In some case, one might know the required information (for example, the name of the disc file to be associated with a stream) when the programme is written, so very early binding is possible. In other cases, it is more useful to define the file name when the programme is executed, while even later binding might be required if the file name is to be determined through dialogue after the programme has been started. Two examples illustrate the difference :

- If the operating system is primarily responsible for maintaining the file information block, the first case can be handled as part of the programme starting procedure, with the file name provided by the programme code through an API procedure (typically as part of a data structure including other initial properties as well). Given appropriate syntax in the programme execution instruction, the operating system can deal with the second case directly, without requiring any programme code; while the third case requires an API call to change the file name in the file information block. Provided that the required data are all bound when the stream is opened, all is well.

- If the programme is primarily responsible for maintaining the file information block, the API might be simplified, as there is less need for different sorts of procedure to define the file information block. The programme code must now assemble all the required information before the stream is opened, then a single call can be used. In the first and third cases, this makes little difference, except that it is somewhat less convenient to incorporate the file name in the stream declaration, but in the second case it becomes necessary to pass the file name to the programme by some parameter mechanism, leaving it to the programme code to disentangle the parameters and insert the file name into the file information block.

The difference is clearly of degree rather than kind; nothing becomes possible or impossible with the change of method, but actions change in their relative difficulty or convenience. This should be taken into account in designing the system, though it is not clear that it gets much attention in practice. In consequence, circumstantial evidence from systems we have known suggests that system designers have chosen to pass the responsibility to the programme. It is true that this approach makes life easier for the operating system designers, as it simply uses the means for passing parameters into programmes which are likely to be there anyway, but it is not our position that a major function of an operating system is to make life easier for operating system designers. The programmer is left with the task of defining some arbitrary notation for giving the information in the programme's parameters, then retrieving the file information from the parameter list, probably parsing it, and then using the system calls we've already mentioned to set up the required file information.

There is no file information block within the programme in the Unix system; all the administration is done by system calls. Opening a stream returns a Unix "file descriptor" (which is in fact a table index to be associated with a file handle); all other operations on the stream require this number as an argument.

The second case of stream information binding is implemented as the file redirection feature, the mechanism behind the pipes we discussed in the STREAMS chapter. This feature in effect identifies streams in the programme with external streams when the programme is started; it associates an external stream with the table index mentioned above. It is quite general, in that it will work for any stream – provided that you know the table index. Unfortunately, except for a few system-defined streams – notably the standard input and output steams assumed in the standard pipe and redirection syntax – you don't know the table index until you open the file, by which time it's too late. Table slots are allocated serially, so you can work out the index if you know the exact sequence in which files are opened, but that isn't the way we expect to do things in a modern operating system.

Every stream is always regarded as a stream of characters, with basic single-character input and output operations. Even the developers of C realised that this was a bit restricted, so provided the standard C library routines which, in effect, construct an additional file information block within your programme.

In the Macintosh system[SUP9], the difficulties are ingeniously evaded by abolishing the instruction to run a programme – or, at least, by casting it in quite a different form. If you open a document (a data file), the system identifies its associated programme (by finding its "signature" in the document's "creator" attribute), and sends the identity of the file as "Finder information" to the programme. As there's no way to say anything about any other files whatever, that's the limit of what you, or the system, can do, and everything else has to be managed from within the programme. There's a certain amount of support in the form of standard system procedures, but it's still fairly hard work.

RECORDS.

In that discussion, we concentrated on the administrative details of using streams, but we mentioned in passing the question of data movement, and introduced the idea of buffers. As it is rather important to provide facilities for moving the data to and fro, we now return to this topic in a little more detail.

So far as the data transfer is concerned, we require that the programme should be able to request the transfer of some unit quantity of data between stream and memory. We don't know how big this unit of transfer is, so we shall simply call it a *record.* The record usually maintains its identity in the file, either by defining a file attribute which specifies the length of all records in the file, or by associating a length counter or end marker with each record. The order of the records in a stream or file is significant, so we might think of a file as an *array of records*, while a stream is a *sequence of records*.

How big is a record ? We have no idea. The record is a logical entity, with its size determined by the requirements of the programme. The record size used in an operation must be based on information known to the operating system when it effects the operation, and this information can come either from the stream (presumably the file information block) or from the programme. One might hope to derive some guidance from the way programming languages provide for streams; but they are not much help, as they typically either provide for input and output instructions of arbitrary complexity (particularly older languages, such as Fortran and Cobol), or they assume that everything has to be read a byte at a time (C and Prolog). Pascal contrives a sort of mixture of the two. Of course, the operating system doesn't have to cater for every tiny demand of the programming languages – and it would be quite hard to cater for all possibilities simultaneously. We can reasonably aim to satisfy some compromise specification, and require the language software to provide whatever machinery is needed to look after the rest of the administration. We have expressed our compromise in terms of records.

Strictly speaking, all we need at this stage is the idea of the record, and its transfer in some sense when **read** and **write** operations are performed. In practice, it is also helpful to introduce the idea of a *buffer*, which is an area of memory reserved to hold a stream record. We emphasise that these should really be regarded as implementation mechanisms, but we are not aware of any practical system which does not use some form of buffer.

MAKING IT WORK.

These two structures, the file information block and the buffer, are the key to the effective operation of the system. The programme is concerned primarily with transactions between executing code, memory, and the standard abstraction of the outside world embodied in the two data structures. Meanwhile, the operating system must support the abstraction by making sure that the data structures behave in the required way.

In this chapter, we discuss the characteristics of streams in programmes, so we shall only go as far back as the data structures; the maintenance of the abstraction is discussed further in the next chapter. The rest of this chapter is therefore concerned only with the programmes' interactions with the data structures, and does not address the hard work of dealing with files and streams. But that is as it should be, because the point of devising the interface is to make it as easy as possible for the programmer.

Opening the stream.

The idea of opening a stream is common, and we have already used it several times without defining it. While we know that this practice is less than satisfactory, we have to do it sometimes – otherwise the *HISTORY* section would have to come at the end, which would rather defeat its purpose ! However that might be, here, at last, is a sort of definition.

Opening a stream is a matter of making the link between the stream as identified in the programme and the "real" object, which might be a stream associated with an interface or communications line, or a file which lives on some medium external to the programme.

From our preceding discussion, two operations must be carried out in order to construct this link :

- Fill in the file information block using data previously supplied – from the `OPEN` instruction, defined by convention (maybe implied by the file name), or from a file table.
- Reserve buffer space.

Reading and writing.

A **read** operation will make available to the programme a record of information from the stream; a **write** operation transfers to the stream a record of information identified by the programme. Many file systems include the record size, or something equivalent, in the stream attributes; some devices dictate the size of the records which they handle; in other cases, we might require the size to be presented (usually as a procedure parameter) when the operation is requested.

Whatever the local details might be, the **read** operation requires, first, that something should happen in the outside world to refill the buffer with the new record, then (assuming that the process wants to look at this record) that the buffer contents be copied to some other place so that they will not be overwritten by the next **read** instruction. It's also usually necessary to copy the new record because the buffer isn't in the process's address space, for protection reasons already discussed. The operating system is usually so organised that the process can accomplish this with a single supervisor call; we'll discuss the details later. The **write** operation is accomplished similarly, but with obvious changes in direction.

Navigation and control.

Another sort of operation on the stream is sometimes important : we might wish to control the source, destination, or other details of behaviour of the stream from the programme. There are (almost) as many different possibilities here as there are different devices – the interaction between a programme and a robot is very different from the interaction between a programme and a printer. Here we shall restrict our discussion to conventional data processing devices (which still covers a very wide range) because they're the most common, but the requirement for both data exchange and control is

widespread. The common feature of the control operations as implemented for different devices is that such transactions are mediated by the file information block rather than the buffer.

Perhaps the most common example of this requirement is the random access file, where the temporal sequence of records in the stream seen by the programme is not the same as their physical sequence in a permanent file. To deal with this new feature from the programme end is easy, as all we need do is ensure that the file information block contains a field for the record number, and that there are ways for the programme to set the record number (so that we can implement **seek** functions, or accept record numbers as part of **read** and **write** instructions), and leave the rest for the next chapter.

There is a limit to how far this approach will work. If we (the operating system designers) provide for random access files in which the programmes specify the record number, should we also cater for, say, indexed files, where the programme stores and retrieves a filed record by specifying the contents of a key field in the record ? The answer to that question is that there isn't one. There is absolutely no reason why we *should* or *shouldn't* do so – we are not bound by moral considerations, there are no human legal constraints, and no laws of nature which decide one way or the other. It's obviously possible, as we can just follow the example above and reserve a field in the file information block for the key, leaving the rest to the implementation (which in fact isn't particularly hard, unless you want it done efficiently).

We design the system on grounds of expediency. If we expect it to be widely used in environments in which indexed files are common, we might well wish to extend the standard input-output facilities to include such files, but we do so at some cost of added complexity. If we want to extend the system further to cope with tree-structured files, multiple search keys, approximate key matches, and other useful ways of organising files, streams, and access thereto, we can – but in practice it usually just isn't worth the trouble, and a line has to be drawn somewhere. Most operating systems provide a fairly primitive system which copes reasonably well with most common requirements, and rely on you to build your own desirable features on top of it.

The difficulty with that policy is that there are times when you can't build the desirable features from the facilities offered by the operating system. A recent example is the requirement for dealing with very fast streams in real time[SUP10] – typically streams of video information which must be displayed very soon after its arrival, and which must be received fast enough to maintain a continuous moving display. A system which imposes the overhead of a supervisor call for each byte of input is very unlikely to be able to handle such a stream at anything approaching the required speed, and, even if it could, the enormous wastage of time in repetitive, and essentially identical, system calls would be a ridiculous burden. We shall return to this problem later (*REAL-TIME DISC SYSTEMS*); meanwhile, it is an example of the need to identify the system requirements before designing the details.

> *We have laboured this point a little because experience suggests that many people expect to be able to find the "right" way to construct an operating system. We repeat that the criterion of a good operating system is that it should help people to get the computer to do their work effectively. If we miss out bits which most people want, that isn't helpful; if we put in lots of elaboration which very few people want, it's likely to make the system bigger, more prone to error, and more expensive, which is also not helpful. It is, of course, true that though we can't say what's right in an operating system, we can often say what's wrong.*

There's another sort of navigation which pertains more directly to streams. There is no direct equivalent of random access, because it is in the nature of a stream that we have to deal with the records in the order in which they arrive (though it might be appropriate

to provide a buffer which works as a queue and can store a few records). On the other hand, it is sometimes useful to be able to switch between input streams, or to switch between, or control the distribution of records to, output streams.

Input examples are the injection of the contents of one file into the stream of records from another (often possible with compilers – consider the `#include` directive of the C compiler), and the switch of system input between terminal and command files. The most common output example is in cases where it is desired to keep track of the material which passes along a stream, so each stream record is transmitted to its "real" destination and also copied to a disc file. Notice that the asymmetry between input and output is real : it is straightforward to copy an output record to several different destinations, but the corresponding input operation would be to accept identical input records from several sources, for which it is hard to imagine an application in which having this under control of the programme makes sense. Something like this happens in systems designed for high reliability, where all components are (at least) duplicated, and the performance of different streams always checked to ensure that they are identical, but that's rather a different matter.

We have included this sort of stream navigation for completeness, but we know of no operating system which makes direct provision for it. It isn't particularly difficult to make it work for a single programme, but can become rather tricky if you want a more general ability to switch streams. We have implemented versions of it sufficiently often to wish it were a more popular component of operating systems.

Closing the stream.

When we have finished with the stream, we have to make sure that the state of the operating system remains consistent with the new state of things.

If the external entity associated with the stream is a real stream, there might be nothing much to do, as there is usually little reason to record permanent information about a stream. Once you've finished with your end of a stream, it's just like it was when you started, and it will be pretty much the same next time you start. (Think of screens and keyboards.) If there is some device at the far end of the stream, it might be necessary to go through some sort of protocol to end the connection – just what protocol depends on what sort of connection, but it's usually a matter of executing some procedure which goes through a standard routine.

If the stream was linked to a permanent file, there might be more to do to ensure consistency, because the system does keep permanent information about the file in its file table. From our lofty position of abstraction, then, we have to check that any file attributes which might have to be changed are brought into line with the description of the file we have maintained in the file information block.

In practice, because of the ways we construct file systems, there might be more active things to do, and we shall mention them later when we talk about implementation. (If you're worried about tidying up the contents of buffers, that's where it is.) Generally, though, the same principle holds : the aim is still to make sure that the file is properly up to date in the outside world, and – for a persistent file – the file table entry properly represents the state of the file.

COMPARE :

Silberschatz and Galvin[INT4] : Section 10.2.

REFERENCES.

SUP9 : *Inside Macintosh*, volume 2 chapter 2 and volume 3 chapter 1 (Addison-Wesley, 1985).

SUP10 : P. Druschel : "Operating system support for high-speed communication", *Comm.ACM* **39#9**, 41-51 (September, 1996).

QUESTIONS.

Consider our brief discussion on the proper home of the file information block. Would it be sensible to implement this as another "file component", along with the attribute and data components ?

Think of ways to provide file access without allocating buffers.

In an operating system which records nothing about any internal structure of files, and provides operations for transferring records only of one byte in length, consider what additional facilities might usefully be implemented by the programming language software to make it easier to use files in convenient ways. (This question is inspired by the C standard file package.)