MAKING DEVICES WORK

We now address the question of moving information between device buffers and the devices themselves. This is what we need to support the software end of the device interface.

We won't spend much time on very low level details of communications between devices and computers for two reasons :

- The machinery used is below the level at which the operating system has anything to contribute, just as the machinery used to fetch data from memory isn't itself of concern to the system.

- While small systems might still deal directly with individual devices, larger systems have for a long time been pushing out the fiddly details to *device controllers*, which look after all the nasty bits and present a clean and orderly interface to the operating system. We've already met disc controllers (*DISCS - THE HARDWARE VIEW*), and we'll say a bit more about the controllers later (*DEVICE CONTROL HARDWARE*).

There's a short example of a simple low-level device driver in *DEVICE CONTROL SOFTWARE* just to show how it all fits in.

That second reason is an example of a fairly common phenomenon : once we learn enough about an operating system function to be confident that we can do it well under all circumstances, then, provided that it makes economic sense, we implement that in a self-contained subsystem. This reduces the load on the operating system proper, and usually makes it simpler. The separate subsystem might be implemented in software, or in hardware within the processor (virtual memory, hardware dispatchers), or as a separate processor (device controllers), but the principle is the same.

Our account will be concentrated on the problem of imposing order on the manipulations required to control a set of quite different devices, typically individually designed, with different interfacing requirements, and possibly not even in existence when we design our operating system. We think that the only way to make any progress in such circumstances is to study what is required to control devices in general, and then to use our findings to lay down a convention which defines some sort of interface with which all devices and their software must comply. Obviously, we make sure that all the devices we know of *can* conform to our new standard – then we hope that all new ones will conform too.

Yes, we have come back to device independence again. We've already mentioned this idea from time to time, but now at last we have to make it work. Even if we don't manage complete device independence, we need to be able to think about devices as quite simple things as a part of our system mental model, and the more idiosyncrasies we have to take into account, the less simple our system mental model will be. Notice, though, that we're now talking about a lower level of model than we did at the beginning of the course; we are, by definition, now working at the interface between operating system and device where very few will venture. (Where angels fear to tread, perhaps ?) Even so, the idea of a mental model is still worth maintaining as a principle, for any complication avoided is one less mistake to be made.

It is by no means easy to devise a coherent solution to the problem, and for quite a long time there wasn't one. In the early systems, devices were regarded as the responsibility of the programmes which used them, with the system giving little or no help. (The *monitor model* which we introduced when discussing terminals is an example.) Here's a review of some of the difficulties which must be overcome.

- All devices have **different hardware interfaces**, and therefore must be dealt with in different ways – but they must be made easy to use if we want a system which helps people to do their work. The diversity of devices doesn't make it easy to present a uniform interface either at the programming level or at the user interface.

- The devices operate at **different speeds**, but we don't want that to interfere in any way with programme execution. Parallel operation of device and programme might therefore be desirable.

- The devices are **independent machines**, which run at their own paces independent of anything done by the processor. This is not just a matter of their being private devices; the system operates as though they were highly specialised other processors – which, indeed, they are.

- The devices must be kept **under system control** for safety, so the devices must be separated from the processes which use them.

That perhaps gives some indication of the difficulties which must be solved if we are to achieve even an approximation to device independence – but concentrating on the differences isn't going to make it easier. To make any progress, it is likely to be more fruitful to consider the similarities between the devices, because we shall have to build on these common features if we are to achieve any solution at all.

SOME PATTERNS.

If we look at transactions between computers and devices, we find that most will fit more or less well into quite a simple pattern :

- Preliminaries, to establish communication. Initiative can lie with the device (interrupts) or with the computer.

- Transmission, to achieve communication. The transmission can be in either direction – from the computer's point of view, input or output.

- Tidying up, to break communication. This can also be managed in either direction : "message received" from the receiver or "end of transmission" from the sender (or both). Also, sometimes (ideally always), something from the device to tell the computer what has happened – did all go well ? if something went wrong, what ?

There is a lot of variation in detail between different examples, and sometimes a component of the pattern is missing, but any pattern is better than none, so we'll clutch at it. We can further classify the transactions by observing the directions of the components; as hinted in the list above, each component can be directed in either way. Notice that the pattern can be nested, as when a request to transmit a block of information is implemented as a sequence of individual requests for smaller units. ("Get a line of text from the keyboard.")

By looking at what happens in each of those steps from the computer's viewpoint, we can find another pattern, rather more important for our preoccupation with the operating system. We want to know what the operating system will have to do to make all these things happen. We've already asked this question in our analysis of the system requirements (*STREAMS IN PROGRAMMES*); if we now consider again our description of what is being moved, and which direction it's going, we can classify almost all the cases into these operation classes :

	computer to device	device to computer
Move data	Put	Get
Move administrative information	Control	Status

Physically, the data and administration streams may be merged (which is why many control characters are defined in conventional representations such as ASCII), or they may be separated (which is why RS232 connections include Request to Send and Clear to Send connections as well as data connections).

That scheme covers *almost* all the cases because we have considered only actions initiated by the computer, and in normal use these handle most of the device operation. There are two exceptions : interrupts, and direct memory access transfers. We'll defer interrupts for a while, and won't talk about direct memory access in any detail at all, because the operating system has nothing to say about it.

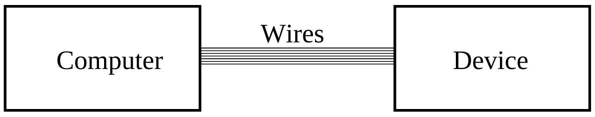
Notice that the low level is important if you want to build a truly general method. The alternative is a long list, which might begin **seek**, **open**, **close**, ... There are two reasons why that won't do : first, it's hard to give a complete list, because it depends on what devices you're using; and, second, it can't possibly cater for all future eventualities, so it's a bad basis on which to build a stable system.

Of the four operations we have identified, the pure data transfer pair are (or, more precisely, can be made) essentially independent of the device concerned. They vary little, if at all, between devices, and are fully defined by the device identifier, a memory address, and the quantity of data concerned. That works because any other details – which is to say, those specific to the device – have been sorted out in one way or another before the **put** and **get** are used. For example, if the device is a disc, a complete specification of a data transfer must include a disc address of some sort, but this can be previously established by a disc-specific **seek** control operation, or (if a file is being used sequentially) by keeping track of the disc address in the device-handling software.

More generally, all the variability must be dealt with by administrative operations. For example, the **seek** operation for the disc would be implemented as a **control** with parameters recognisable by the disc software as defining the required operation. Special instructions for other devices are similarly concerned with the peculiar properties of the devices themselves – operations like "skip to a new page" are only significant with certain sorts of device. This seems to be an effective division of labour, and something like this scheme is quite widespread in operating systems.

A SINGLE SYSTEM CALL.

We can take the systematisation a step further. We know that we want to do four different things with the devices, and it might also be useful for the operating system to know that at a higher level. Down here in the operating system mud, though, the view is restricted, and all the transactions look very much the same : in each case, the computer sends something to the device, and waits for something to come back again. It might even do the sending and receiving using the same wires, whether we think we're doing data transfer or control – we've already commented on the possible merging of control and data when using ASCII transmission. The real communication operations, therefore, are not necessarily distinguished by type of operation, so to maintain a distinction in our description could be misleading. Removing the distinction gives us yet another pattern, and this time it's a very simple one :



For this level, then, it makes sense to combine all the requests into one system call. We shall call it **doio**, just as we predicted in the *DEVICES* chapter, and use it to manage all the communication between system and device.

Unfortunately, finding a reason to call everything by the same name doesn't solve any problems. The universal communications procedure still has to be able to send the right signals for all the different sorts of communication we want, so we have to give it a fair bit of information through parameters. In fact, we have to tell it four things :

Which wires to use : something to identify the device.

What sort of signal to send : is it **put**, **get**, **control**, or **status** ?

What else to send or receive : a specification of the data to be sent, or where to put the data received.

What to do with its reply : where to put its report of what happened.

The **doio** supervisor call will therefore look something like this :

doio(stream, action, data, result).

stream might appear to be a step up to a higher level; perhaps it is, and we should really have **device** in this position, but **stream** will be appropriate if we can really manage device independence – and naming the stream immediately identifies the device through the file information block. In practice, it is probably rather more realistic to restrict this function to the device level, largely because it's difficult to implement the same set of control functions on different devices, but we'll stick to **stream** for the moment. (Bear in mind that this is a system call, not intended to be used by people writing conventional programmes; it is therefore rather less important than it might otherwise be to avoid technical detail, though it's usually a good principle to keep things as general as possible.)

action is one of the four operations we identified above. This parameter determines what sort of action is caused by **doio**, and also determines the meanings of some of the other parameters. That's inevitable, because, while **get** and **put** are fairly tame variants, **control** and **status** have to manage communication of arbitrary complexity with the devices. In these cases, therefore, the **data** parameters can be quite complex.

data says where to find the message to be sent, or where the answer is to be put. It is commonly given as the address of some sort of structure, but the structure might include more than just a data buffer. Particularly for control functions, it is helpful to transmit a more structured set of information. The details obviously depend on the device.

result identifies some structure which will be given a value by the system call to show what happened during the operation. This might be a simple variable, sometimes called a *result code*, or again a more complicated structure might be used.

REMARK ON DATA TYPES : We have written ***doio( )*** *as a procedure to emphasise that all its variants have essentially the same form; but – as we have pointed out in the comments – the last two "parameters" might have to be of different types in different circumstances. You might or might not find that disturbing, depending on your view of data types. The difficulty is that, though the job we wish* *to carry out is very precisely defined, few typed languages provide the sort of flexibility we need to write it down. We would like to use the values of* ***stream*** *and* ***action*** *to select the procedure we really want to use, then to pass to that procedure values of the appropriate types for* ***data*** *and* ***result****. If we wish to make it respectable, we can call it a polymorphic procedure.*

Here's an example to illustrate how **doio** might be used. Suppose that a programme is using a disc file XYZ. The programme executes an instruction which means "WRITE 67 TO BYTE 3376 IN FILE XYZ". What happens ?

Two operations are necessary. First, there must be a **control** action to find the correct position in the file, then his must be followed by a **put** operation to transfer the data. The compiler must therefore compile the original instruction into something like this :

```
doio( XYZ, control, { seek byte 3376 }, result );
doio( XYZ, put, 67, result );
```

where the parenthesised parameter in the first instruction illustrates, by a very simple example, how the requirement for quite complicated structures can arise.

When the instruction is executed, the first request is handled by the disc device's **control** procedure, which must be able to understand the third parameter. Because of the disc structure, it must respond by executing a sequence of instructions something like this :

```
calculate the sector number for byte 3376;
if this sector isn't in the sector buffer
then begin
     if another sector is in the buffer
     then if it has been changed
          then put the resident sector back to the disc;
     get the new sector;
     end;
```

Once this operation is complete, the second request is handled by the device's **put** procedure. It inserts the value given (67) into the current file position as determined by the preceding control operation.

That leaves the change to the file in the file buffer, but doesn't write it to the disc. It will eventually reach the disc either when some other sector has to be brought into the buffer, or when the file is close – notice that the **close** procedure must take care of this. That is the usual practice, because writing to the disc takes time, but if the system should break down at this point you are likely to lose the change. If you want to be very careful you can (sometimes) write the buffer back to disc with what is usually called a *flush* instruction :

```
doio( XYZ, control, ( flush buffer }, result );
```

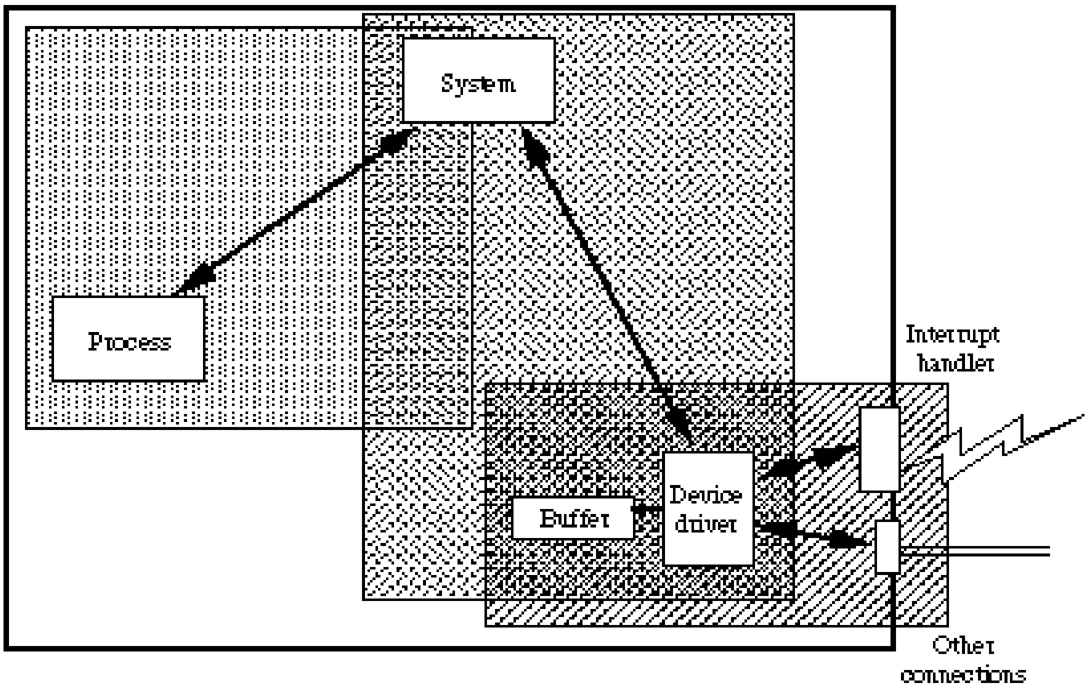
HOW TO MAKE IT WORK.

We've designed a software interface; now we have to make the hardware act accordingly, and to do so we have to dig even lower. What lies behind **doio** ?

In some cases, **doio** can satisfy the requests it receives by reference to the device table, and no further software levels are necessary. This is sometimes so for requests for information on the device state, and will usually be so for badly formed requests, in response to which **doio** will return an appropriate error indication in the result code.

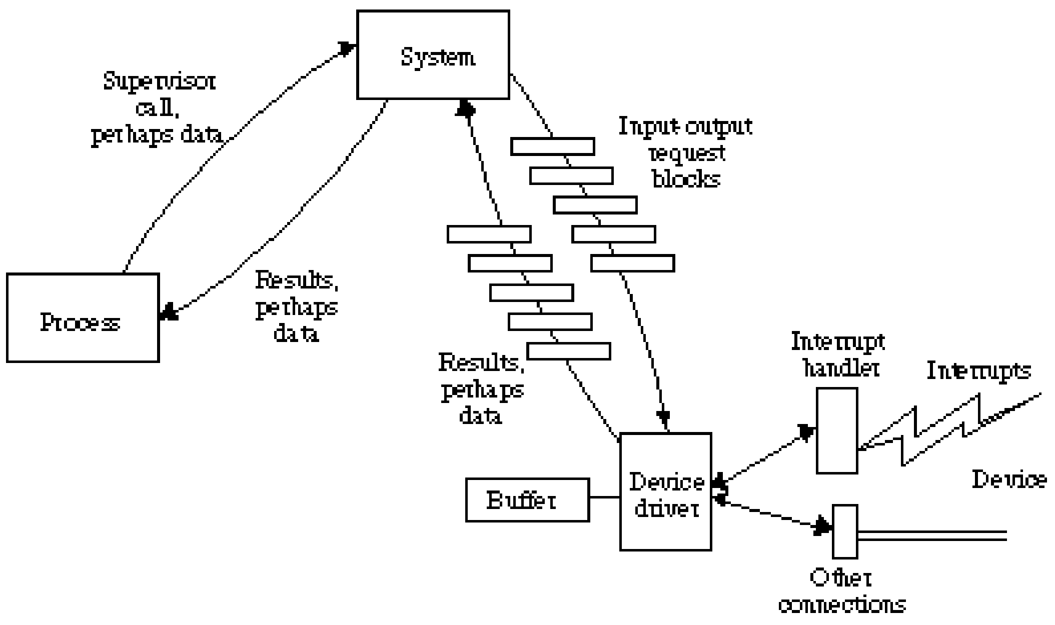
More commonly, **doio** can only satisfy a request by reference to the device itself. As usual, just what is needed depends on the device, but typically there are two layers of software : one primarily concerned with dealing with the requests as they come from **doio**, and another to handle interrupts from the device. We shall call these the *device driver* and *interrupt handler* – once again, there is no standard terminology. As you would expect, these two components, though logically separate, work closely together, and typically communicate through a shared buffer and various interlocking primitives.

The upper diagram on the next page shows a possible organisation which satisfies the requirements. The shading on the diagram is supposed to suggest regions within which different sorts of communications are confined. The process communicates with the system, to provide device independence, and also because we don't want the process to be able to see the device directly for safety and security reasons – and because, with a shared device, no single process is in a position to decide what should be done next. The system acts as intermediary between process and device driver, passing on such information as might be necessary in both directions. The device driver acts as an intermediary between the system and the device itself, so that the system need not know about the peculiarities of individual devices.



The diagram below shows more detail. In action, the sequence begins when the process executes a stream instruction - **read**, **open**, etc. - which will have been translated by a compiler into a **doio** supervisor call. **doio** identifies the device from the file control block, and refers to the device table (*DEVICES*) to find out how to handle it. If it can resolve the request using information in the table, it does so; otherwise, it uses the device's procedure appropriate to the **action** parameter to send a request to the device driver. The request cannot simply be implemented as a procedure call, because the device driver might have to handle requests from several different processes, and the requests could conflict. (Recall the queues of requests for disc operations which we mentioned while discussing disc scheduling.) Instead, there is a queue of requests; so the device descriptor in the device table must contain information about its queue. The record queued, which might be called something like an *input-output request block*, is rather like the information given to **doio**, but with one difference : as we have now moved from the process side of the system to the device side, the stream or device identifier is replaced by the process identification.

On receiving the function request, the device driver interprets it, and performs as instructed – or, if it can't, doesn't. Information may be returned to the calling process by following the outward route back, and some result code should always be returned. The operation is now complete. The communications involved are shown in this diagram :



The device driver must also be able to report faults effectively. In the simple case, the device identifies a fault of some sort and reports it by sending an administrative message to the computer. The device driver can then receive the message in the ordinary way and take action; a message of this sort is defined in the device description, and provision for handling it will be incorporated in the device driver. Other events might be more difficult to deal with – for example, the device driver might receive spurious signals from electrical interference or because the wrong device has been connected, or the connection might be broken so that the device effectively vanishes. Any incomprehensible message should be reported as an error; a broken connection can be detected in many cases by occasionally polling the device to request some status report, when the absence of a reply indicates that the connection is lost. This is a very common activity in communication networks, where accurate knowledge of the states of all connections is very important.

SYNCHRONOUS AND ASYNCHRONOUS OPERATION.

The process executes the **doio** instruction, and at some later time a result code should return from the device driver. What does the process do in the meantime ?

In the simplest case, the process waits for the result before continuing. This is called *synchronous* operation. This is the easy way to do it, because the process gets the result when it knows what it's doing; the context hasn't changed, and it can handle the result in a straightforward way.

But you might recall that this habit of waiting for input and output operations to complete before continuing was one of the time-wasting characteristics of the early operating systems which we have been trying to get rid of since about 1950. We avoided the worst consequences by inventing multiprogramming, but still this answer leaves something to be desired. The alternative is obviously to carry on with the process while waiting for the input or output to happen, which is called *asynchronous* operation – but then what do we do about the result ? We can usually expect that all will be well, but if something does go wrong we want to know.

*You will recall that we have used these terms before
(in COMMUNICATION BETWEEN PROCESSES),
where we promised further startling revelations.
Here they are. We asked you to think about the
meanings of the two words, so you will be aware that*

"synchronous" means "at the same time", with "asynchronous" its opposite. That being so, it is surely obtuse to call operations running at different times "synchronous", and overlapped operations "asynchronous".

The simplest solution is to carry on, but check the result from time to time to make sure that nothing untoward has happened. That works, but it's a nuisance, as it places the onus of checking at the right time on the programmer, who might not know everything about the device requirements. A neater solution is found in (among others) the Macintosh system software. Many of the provided subroutines return a result code as a function value, but also accept a *completion routine* as parameter. A zero result code means that all is completed and satisfactory; a negative result code means that a fault has already been detected (perhaps the parameters were unacceptable); but a positive result code means that all is well so far, but execution is proceeding. If anything should subsequently go wrong, the completion routine is executed, and that can take any action desired by the programmer. (And that sounds easy enough, but if you read about it in *Inside Macintosh* you will find that it's harder than you expect.)

In the diagram which follows[IMP32], we have tried to show the relationship between the various parts of the device control system. The vertical axis corresponds roughly to passage of time; the bottom of the diagram is later than the top. We have expressed the operations in terms of three semaphores (*interrupt*, *device*, and *complete*), but other synchronisation methods could be used.

In the two left-hand columns, the process identity is known and the device must be explicitly identified in any message where its identity is important; in the two right-hand columns, the opposite is the case.

We have combined the two modes of operation (synchronous and asynchronous) in the same diagram to show how they are related. Most of the items – written in normal type – are common to both modes. Items shown in *italic* type are specific to asynchronous operations, with the assumption that the device operation takes longer than the computing to be done; the underlined bits are specific to synchronous operations.

USER	SYSTEM		
Process	doio	device driver	interrupt handler
Process known, identify device if needed.		Device known, identify process if needed.	
doio(stream, action, data, result)			

	use the file table to determine the device;
	check for errors;
	put IORB in queue;
	signal(device);		

	return or
	wait(complete)
		wait(device)
		get IORB;	

(do things)
......
		start operation;
		wait(interrupt)	

......
......
wait(complete)
			......
			identify the device;
			save input in buffer;
			signal(interrupt)

		check for errors;
		move data;
		return result code;
		signal(complete);	

	return	start again	
check result;			
(continue)			

There is a significant difficulty with asynchronous operation. If you just carry on with your programme after requesting a device operation, how do you know that there will be anything there to catch the result when it returns ? The most obvious example is the process which requests an operation, then finishes before the result comes back, but even within a programme it is possible to leave the environment in which the request was made before receiving an answer. We've papered over the problem in the example by holding up the process using a semaphore, but unless there's a lot of work we can do under the heading of "*do things*" we can end up not much better off than we were with only synchronous operations.

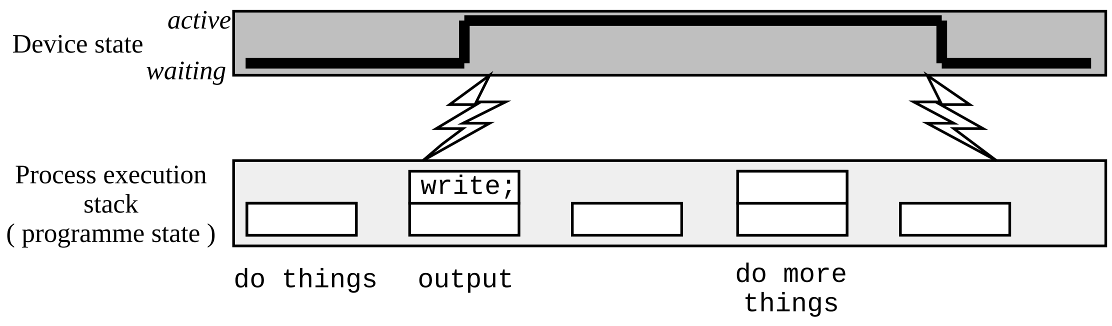
Here's an example. It's obviously contrived, but we think it's not unrealistic. The programme alternately performs input and output, and does some computation in between :

```
start programme :
     repeat for a while
          do things;
          output;
          do more things;
          input;
     end repeat;
end programme.

output procedure :
     prepare data;
     write;
end procedure.
```

(We've concentrated on the output operation because that's the more severe case; we usually have to await the input at some point before proceeding, while if all is well we don't need to wait for anything after an output operation.)

It's a nicely structured programme. We've supposed that the details of input, output, and "do more things" are encoded in their own procedures, in accordance with widely accepted practice. Because of that, though, the environment of the `write` instruction is very likely to have disappeared from the execution stack before the operation is complete :



If the device is just a bit slower (or `do more things` takes less time) we might even find that input and output overlap. According to our previous arguments, this is excellent, because the more things that happen at once the more efficient the processing, but it doesn't simplify the problems of coping with the results in the programme.

There is a very simple solution, if your system will let you do it : use separate threads for input, processing, and output. Then each can maintain its own state, and is ready and available to deal with any results which might turn up. Not many systems will let you do that unless you burrow down into deep system code. The consequence of this problem for the Macintosh implementation was that the completion routine couldn't live in the same address space as the rest of the programme in case the programme died before the operation was completed. This made communication hard.

(LATE NEWS : it seems that Apple are doing things to improve this part of their system. This is supposed to be, or to be about to be, improved now, but we're not sure just how.)

COMPARE :

Lane and Mooney[INT3] : Chapters 8 and 9, and (for some individual devices) Appendix B; Silberschatz and Galvin[INT4] : Sections 2.1 and 2.2.

REFERENCES.

IMP32 : Acknowledgments to A.M. Lister, *Fundamentals of operating systems* (Macmillan, second edition, 1979), pages 66 and 68

IMP33 : *Inside Macintosh*, vol. 2 (Addison-Wesley, 1985).

QUESTIONS.

The diagrams in this chapter hint at necessary synchronising operations within the device control software. What are the full requirements ? How would you implement them ?

Inspect the table showing the sequence of actions of the four cooperating components. Convince yourself that you know what it means by laying it out in proper time sequence. What are the various components doing in the times during which you haven't specified their actions ?

What sort of actions can sensibly be taken in a completion routine ?

We can manage device independence by constraining all input and output operations to follow a form something like

doio(stream, action, data, result)

but to achieve the requisite variety of actions we have to make the target of the data pointer quite a complicated structure. Could we do it any better ?

One possible line of attack would be to think of the **doio** as a sort of CISC instruction – does a lot of work, but inevitably complicated. Would a set of RISC equivalents be better ? If so, which set ?

You are implementing a disc file system incorporating the tree-structured method of providing file generations described in the *FILE GENERATIONS* chapter. You wish to provide ordinary file access, without generations, as well. Would it be sensible to implement the generations technique as a separate "device" in the device table ? How would the **get** and **put** procedures work ?

Why is the Macintosh implementation of completion routines so complicated ?
