WHY MEMORY ?

The reason for our questioning the need for more than one sort of storage is that it didn't appear in our analysis; we wanted somewhere to store data over the long term, when it wasn't in use by a programme, and noticed that if it was required for computation then the processor would have to be able to get at it. The obvious solution is to find some way to give the processor access to the existing storage. Why do we need anything else ?

The answer is purely to do with the implementation, and the nature of the devices we have at our disposal : the file stores we usually use now are far too slow to be used directly as working storage when executing programmes. That's why we have to invent short-term storage, which we call *memory*, which we can use to hold data while it is in use by a programme. If we really want to emphasise the distinction, we shall call it *primary memory* or *primary storage*, referring to storage using discs or other external media as *secondary* <whatever>. Because of this connection with use by the programme, we assume that primary memory is randomly addressable, in that any memory unit can be addressed individually and equally easily.

Extending the same argument, we shall have to put the programme code into memory as well as the data when the programme is running. Can we use the same memory, or should we have two ? If you want to make absolutely sure that your code isn't going to be altered in any way while it runs, two memories wouldn't be a bad idea – then you could make one of them such that the processor was unable to write into it, and guarantee your code's safety. In practice we don't do that, perhaps because in the earliest days of computing it was very common to write programmes which did change their own code, and this possibility was regarded as one of the powerful features of electronic computers. We've rather changed our minds since then, but we don't know of anyone who's revisited the question of providing separate code and data memories.

We really are ***obliged*** *to invent memory, as is demonstrated by an old but instructive argument. Consider an old electromechanical calculator. To perform a calculation, we have to put in some data at a keyboard, enter an instruction, usually at a different part of the keyboard, wait until the calculation is finished, and copy down the result. Entering the instruction, a single key depression, is the fastest of these components, and even that takes a significant fraction of a second. Suppose for argument's sake that each of the four components takes one second. If we use electronic means, the same operations have to be carried out – data must be moved into the processor, the instruction must be found, the computation must be performed, and the result stored – and each of these can be completed in the order of a microsecond or less. But to get the millionfold increase in speed,* ***all*** *operations must be accelerated; if only three of the four are accelerated, the increase in speed is only fourfold.*

A common pattern of work will therefore be to begin by moving the encoded programme from the file store to memory. We can then start the programme, and while it runs we shall commonly have to do move data between memory and file store as the programme requires.

This might come as no surprise to you. It is, after all, how almost everybody has used computers ever since file stores first became available – and, in a sense, even before that, for computers had fast memory even before they had file stores. We have spelt it out in detail to emphasise that this pattern of computation is only an accident – or, from another point of view, an implementation detail forced upon us by the need to get the computing done reasonably quickly. If we could get the same results acceptably quickly by working only on the file store, we would, because it would cut out a great deal of complex fiddling about which we are at present unable to avoid just to move material back and forth between file store and memory. If it were simply copying, it might be no more than time-consuming, but for complex structures it can amount to recoding in a different notation, requiring more or less complicated processing for both reading and writing, and consequently introducing more possibility of error. (We commented, censoriously, on this practice in *PROPERTIES OF FILES*, arguing that a proper design would make provision for the universal requirement to store structures, and we saw in analysing requirements for storage – *HOW PROGRAMMES USE STORAGE* – that structures were required, but generally useful means of storing structures in files are not available.) From this point of view, memory is not merely inconvenient – it's a nuisance, and should be eliminated.

We think it's important to make the point because we are approaching a time when fast memory will be cheap enough to build a file store of respectable size. (One could argue that the time is already here; with eight megabytes of internal memory a commonplace, even small computers have more space in memory than many large early machines had in their file stores. Why do we still feel constrained ?) We shall say more about this view when discussing how to implement files in the chapter *DISC FILE SYSTEMS* – and a somewhat more subtle consequence of the use of both files and memory will appear in our discussion of *VIRTUAL MEMORY*.