USING TERMINALS

A conventional computer terminal (which we take to include screen and keyboard, and any appendages like mice or light pens or whatever) is at least two things : an input device and an output device, which we shall for convenience call keyboard and screen. In normal usage, at least until quite recently, these were quite independent, and were handled by the computer as two separate devices. The only link between them was the *echo* – the computer's normally automatic retransmission to the screen of every character it receives from the associated keyboard. We shall see that newer ways of using a terminal require closer coordination between the input and output sides, so that the separation is becoming less clear; but we shall assume it for a while because it helps to clarify some issues.

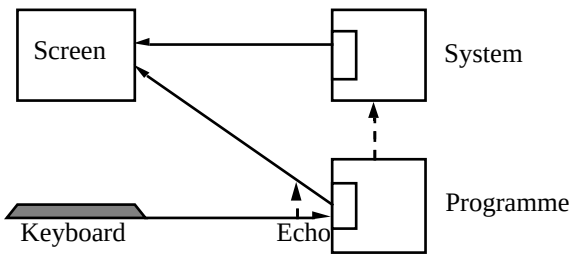
As well as that rather low-level definition, though, a terminal is several other things. In particular, it is the primary communications channel between people using the computer system and the system itself : that distinguishes it from any other device attached to the computer. Other devices are driven by the computer, but the computer must in some respects be driven by instructions from the terminal. Indeed, the terminal has two functions : it is both somebody's interface with a running programme, and an interface with the operating system. Ideally, it should be switchable between these functions at the whim of the person using it – so that, for example, the system can be instructed to stop a programme which is for some reason out of control.

What do we want the terminal to do in these two rôles ? The requirements seem rather simple. Surely we just want the signals from the input devices to be communicated without change to the programme, and the programme's output to be communicated without change to the screen. Don't we ?

No, we don't. Not always. Even with (especially with ?) character terminals, people want systems to provide for programmable function keys, and not many Macintosh programmes need to know about every bit which passes from the mouse to the computer. The tricky task is to define just what we *do* want. In fact, the "definition" in the previous paragraph is couched in language at much too low a level. Ideally, we would like an interface which simply and effectively conveys whatever it is that we want conveyed between ourselves and the computer – we want a terminal which knows what we mean.

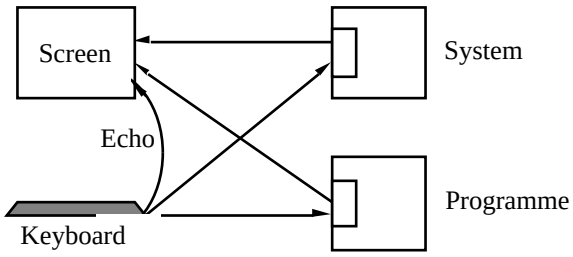
That gets us out of the low level straight away, for the things we really want to get into the computer through the terminal are things that come from our brains, which we shall call ideas. Some of these are matters of straightforward information (numbers, names, etc.), but others are things we want the computer to do (programmes), things we want the screen to do (move or resize windows), or matters of æsthetics (the layout of a document). We are very accustomed to encoding all such material in text of one sort or another, but one of the factors which led to the development of graphical interfaces was the belief that such methods made it much easier to convey some ideas. This is particularly obvious in matters directly connected with terminals, but it spreads more widely too.

We must also worry about the two functions described in the first paragraph. How (if at all) do we provide for access to the operating system while a programme is running (for example, to stop the programme if it's out of control) ? Unfortunately, most older systems give you this :



(The little rectangles within the software boxes represent the terminal communications parts; we'll say much more about them in the *IMPLEMENTATION* section.) When your

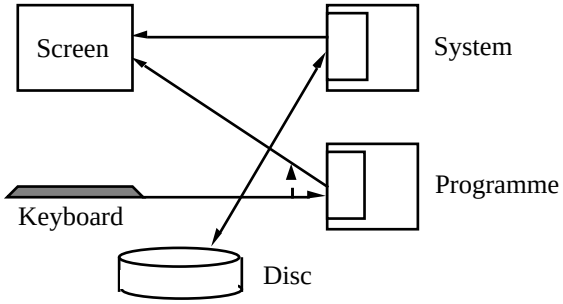
programme is running, the keyboard won't talk to the system at all, except by courtesy of the programme, which is usually not extended. We'd like some signal to switch the destination of input between programme and operating system – and it would also be good to have some means of separating output from your programme from that from the system. Most people who have used interactive systems of the old style know how annoying it is to find a system message appearing in the middle of your nicely formatted output. What we'd like, in fact, is something like this :



We'd like the keyboard to communicate impartially under our control with system and programme, with each returning its output to the screen, and the input echo also appearing on the screen, the whole organised into an appropriate logical sequence. (That comment is added because if your programme takes any length of time to perform calculations between its output operations, it can be easy to enter the next input before the logically preceding output appears. If the echo is displayed immediately, that can give curious results.) There is no great difficulty in making your system work like this, provided that you decide that you want to before writing the software – but it hardly ever happened.

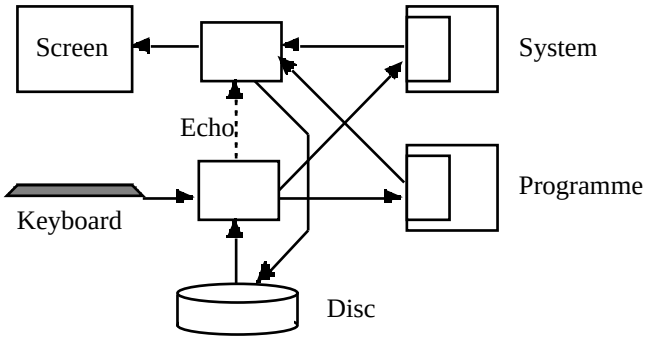
If we push the idea to its limit, we define a function of a terminal which has always been possible, but – until recently – has rarely been exploited : that of controlling several programmes simultaneously. "The" programme mentioned above might be several; with graphical interfaces like the Macintosh's desktop or the Windows system, we can switch from programme to programme at will, and from activity to activity within a programme. That's just what we want – but one might wonder why we never realised it before.

Even without multiple simultaneous programmes, though, the simple picture we have given is far from adequate. Just to give one example of such shortcomings, consider the requirements imposed on the interface by the requirement which we have already mentioned for command files. At first thought, implementing command files appears to be quite simple; all we need is slightly more complicated communications software within the programmes which can read either from the terminal or from a designated disc file. A simple system based on this idea does indeed work quite well so long as the command files are required to work only with the system input – but once you start to run programmes, matters become more complicated. How does the programme's communication software know that it has to use a command file ? – and, if it does know, how does it know which file, and where to start in the file ? Because of these difficulties, many early command file systems were (and an uncomfortable number of systems still are) organised something like this :



The focus of the problems is the requirement for coordinated action of the separate communications procedures in different programmes, and as the requirements imposed on the input and output interfaces become more demanding the problems multiply.

The solution is to separate the software controlling the input and output into a distinct part of the operating system. This new component communicates with the various devices – terminal, disc, and others as required – and with the rest of the software. The new scheme combines the desirable features of the simple interface, which we saw earlier, with the capacity to incorporate other devices and features, such as terminal logs :



Systems of this sort really did work, and were very comfortable to use. We remember with nostalgia a DEC Tops-10 system running on a DEC10 computer, with command file facilities provided by software called MIC. That came quite close to our recommended system above, the main deviation being that the language used by MIC was not quite the same as that used through the terminal. But it was very close ...

We shall not elaborate the details any further at the moment, but the point – once again – is that without careful specification and design from the beginning, it is difficult to produce an operating system which does what you want it to. In this case, we have taken the first steps in the evolution of what is now known as a *user interface management system* (UIMS), and we shall have more to say about it later in the chapter *TERMINALS AS DEVICES*.

Even without going to such lengths, we clearly want some sort of software layer between the running programme and the terminal which does more than just transmit signals. For a character terminal, it might just look for particular character sequences and do something special when they occur – so the function key's escape sequence can be replace by a defined text string, or an escape character might switch control to the operating system. Perhaps that's where the idea of shortcuts originated. (Earlier systems were more likely to provide a small selection of control characters, each of which performed a specific operating system function, rather than a general switch; if you want the more general result, you have to be more sophisticated in representing what's happening in the system.) There are even some circumstances in which the interface software needs to watch both input and output transactions : consider how the password is blanked in the Unix login sequence, and how the significance of a mouse click is interpreted when using a graphical interface.

This software layer can be organised in several ways. The organisation we choose is determined by the facilities we wish to provide, and we shall come back to it later when discussing how the system handles terminals regarded as devices.

QUESTIONS.

Do we need to distinguish between the operating system and a running programme when discussing what terminals do ? Is the distinction real ?
