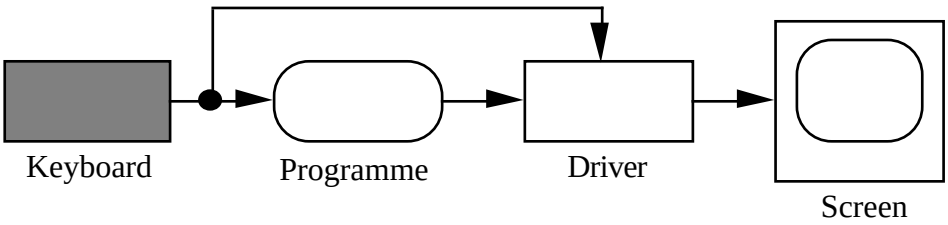
SESSION LOGS AND COMMAND FILES

The consequences of the choice of terminal management technique extend beyond those for the terminal itself. Session logs and command files, both originating in the days of traditional character-based operation but with continuing application in GUI systems, are particularly dependent on the terminal management method chosen. Each of them is closely concerned with one of the streams of information associated with the terminal : the session log is a copy of the stream which flows to the screen from the software which controls it, while a command file in some sense replaces the terminal input stream.

We have composed that last sentence with some care, as what we want is not always the same as what we get. All too often, what we get is something which bears a superficial resemblance to what we want, but is much easier to supply.

The main function of the session log was originally to provide a record of whatever happened on the screen, for use as documentation, to convert into a command file, or as an aid in finding errors. The aim was therefore to produce a file which is as close as practicable to the material which appeared on the screen during the logged period. There is only one point in the system where the required information is readily available : between the terminal itself and the software that drives it. As the diagram shows, at any point more remote from the terminal there are two streams of information which must be correctly coordinated, and attempts to merge the two streams from the keyboard and system output can give bizarre results if type-ahead is used or the system is for some reason running slowly.



(The diagram above is not to be taken too seriously; it is simplified to make the point. In practice, if there is to be any chance of coping with type-ahead and getting a comprehensible screen layout, leaving aside the more difficult question of the log file, then transmission of characters to the driver must not be directly from the keyboard input stream. Instead, the driver must receive input characters as the programme consumes them, not as the keyboard produces them. In some early systems the diagram was a good description of the implementation, and with type-ahead permitted and a slow system it was quite possible for instructions to appear on the screen several lines before their responses.)

What can we say about session logs for GUI systems ? First, that it is not usually very useful to see a blow-by-blow replay of what went on during a session; what we want to know is usually not what happened but what it meant.

Similar problems were experienced even before graphical user interfaces became common. When full screen control became possible, overenthusiastic log software carefully recorded each operation (which could very well be a sequence of control characters meaning "show the character positions from (10, 15) to (10, 24) in reverse video"), so that the log was almost impossible to understand when inspected with an editor. Indeed, even an editor wasn't reliable; unless the editor was clever enough to intercept the control characters, it would faithfully reproduce everything that happened on the screen, usually clearing the screen just after the interesting

bit. The only recourse was to replace all the control characters with some harmless strings as markers, but that was hard enough to be a severe deterrent.

This is a matter of coordination, for we wish to record what the system understood by the sequence of input actions in the context of the original screen pattern – in fact, we usually want to record a sort of reverse-engineered command file which contains the essence of the instructions which were in fact probably given for the most part by mouse movements and clicks.

The session log collects what comes out of a terminal session; the command file controls what goes in. With a textual interface, it is at least fairly easy to decide what we want from the session log, but not even the decision is easy for command files. (With a GUI, it isn't even obvious that there can be such a thing as a command file.) The main point to be determined is just what we want the command files to command. Particularly, do we want to include programme input in the command files, or restrict them to system input, leaving all programme input to the terminal ? Both patterns have their advantages : it is convenient to be able to write a command file to control a complete job, which might well include instructions to the programmes which run in the job, but it is also convenient to be able to test a programme by writing a command file which repeatedly allows us to edit a file used by the programme then to run the programme on the new version of the file. One could also argue that a simple mental model is encouraged if you can think of the command file as an exact substitute for the keyboard. Perhaps the ideal is to be able to do either, so it might seem rather odd that this is only rarely possible. We discuss the reasons below, distinguishing between text and GUI for convenience.

TEXTUAL INTERFACES.

Assuming that the logs and command files are administered by the system, the differences in behaviour are consequences of the changes in accessibility to the system of different parts of these streams with the different schemes for terminal management which we discussed in *TERMINALS AS DEVICES.*

- **The monitor model :** With this type of terminal management, both the terminal and the software are owned by the process. It is difficult or impossible for the system to reach the terminal streams while the process is in control, so both logs and command files are limited. Typically, the logs record only the transactions performed by the system, and not input and output concerned with programmes which take over, and command files can contain only system instructions. This is a particularly clear illustration of the distinction between the two functions of a terminal.

  "Exec" files in CMS were of this sort, though the picture was complicated by the curious architecture of IBM's virtual machine systems. CMS does not in fact run in a real IBM hardware machine; it runs in a virtual machine[IMP21]*, which is emulated by CP ("Control Program"), the real operating system which drives the real hardware. It was therefore really CP which controlled the real terminals, so there was another potential level of intervention which everyone pretended wasn't there except when it became handy to use it – as, for example, with Exec files. By courtesy of CP there was a rather complicated way of storing terminal information to be presented to programmes in a "console stack", which functioned roughly as CP's terminal buffer, and was better described as a deque. To get lines into a programme from an Exec file, therefore, you had to store them all in the console stack before starting the programme, which was a rather odd way to do it, and did not improve the comprehensibility of your Exec files. If you wanted to be really confusing, you loaded the console stack in the simplest way provided – when it really did act as a stack, so you*

had to present the instructions destined for the programme in reverse order. And, yes, this was the early 1980s.

But at least you could do it. MS-DOS batch files are even more primitive; there is no way to get lines from a command file into a programme. The manual says[IMP22] *: "MS-DOS performs these 'batches' of your commands just as if you had typed them from the keyboard"; and, "The result is the same as if the lines in the* .bat *file were entered from the keyboard as individual commands". Well, maybe it depends on how you define "commands".*

But at least you could use batch files. On a Macintosh .... Well, they're here. At last. See below.

- **The supervisor model :** In this case, the terminal is owned by process, while the low-level terminal-driving software is owned by the system. We have a better chance of replacing the terminal uniformly for all processes, though without a UIMS to coordinate things it's difficult to develop a really flexible system without considerable extensions to the interface software, and we don't think many system designers have taken the trouble. The easy way is to provide for the system to replace all terminal input with file input until the end of the file is reached – in effect, simply to unplug the keyboard stream and insert a stream from a disc file in its place. The trouble with the easy way is that it goes to the other extreme; now *all* the input must come from the command file, and there's no way to break out of that if we want to. Of course, provided that we can define what we want to do, we can usually manage it somehow, but every extra feature requires more code in the software. For example, if we want to provide some control sequence which will switch the input from file to file, or file to keyboard, we can do it, but now the software has to inspect the characters as they pass through, not just twiddle the file information blocks.

  In some cases, we can get useful behaviour by trickery. Suppose we want conditional instructions in the command files. (That's just an example – the same method works with other control structures too.) We can do this fairly easily without complicating the software by making **if**, **then**, and **else** (or whatever syntactic form takes your fancy) into ordinary operating system instructions, with the same status as **copy** or **listdirectory**. Then they will be executed when they appear in the file in just the same way as any other ordinary operating instruction.

  Unless, of course, you expect them to work with input to a programme, which will execute them when they appear in the file just as if they were the programme's own instructions, which they are probably not. Now we have conditional instructions which sometimes work. (Perhaps you could say that they work conditionally ?) Welcome back, modes !

  Clearly, if we want to use control structures in our command files in a homogeneous way we must use some software which implements the control features on the input stream from the command file before the stream reaches its destination.

- **The system service model :** The terminal is owned by the system. This is the most flexible organisation. There is a UIMS, already heavily involved in the interface, and it's comparatively easy to build into the software sophisticated control operations which ensure that the command file instructions behave uniformly in all circumstances (so, for example, conditional instructions work both for system and programme input), and control can be exchanged between command file and terminal at will. A command file system (MIC – Macro Interpreted Commands) was available for the Tops-10 system, and was indeed very flexible.

We have remarked that both session logs and command files, traditionally very definitely textual objects, change in nature when we move to a graphical interface, and we have also hinted that they might still be useful. We shall now try to justify this position.

We think that command files would be useful just as they are useful in traditional systems; with a system which can interpret command files, we can encode a sequence of operations once and for all, then execute the whole lot, as often as we want, with a single instruction. Why should computers spend significant chunks of time drawing beautiful (maybe) interfaces, but still force us to plod repetitively though instruction sequences we use every day ?

Except for one purpose, which we shall describe shortly, the logs are perhaps a little less useful. While there have been several occasions when we would have liked to be able to find out what we just did, our own impression is that, perhaps because of changed patterns of work, the sort of detailed record of terminal transactions which was once useful is no longer needed. The exception we mentioned more than makes up for this change in view, though; it is (as with text interfaces) that a log of what you did is a very good starting point for constructing a command file to do it again. It is particularly valuable for a GUI system, because – whatever the language of the command file – it will not be familiar to someone who normally uses a mouse and keyboard.

The difficulty for both log and command file is that, without a language, it is hard to write down what you want, or what you got. The direct parallel to the text command file, which has been implemented in some systems, is a record of sequences of keyboard and mouse actions, so you can *show* the terminal what to do. The defect is obvious : there is no guarantee of success unless you begin with a screen *exactly* like that on which the original demonstration was conducted.

> *An example is the "Recorder" provided with Microsoft Windows 3.1. In some experiments - admittedly, not very extensive - we never managed to record a sequence involving movements of the pointer which did not fail with a "pointer outside window area" error. We did achieve success in recording a sequence of keystrokes, and were able to duplicate a specimen file very efficiently - but only if the sequence started with the window set up exactly as it had been originally. Any change resulted in the duplication of some other file. It is perhaps fortunate that we'd foreseen the possibility and written our sequence to duplicate rather than delete a file.*
>
> *Incidentally, the Recorder is also a bad example of another sort; its icon is a picture of a videorecorder, so we had always assumed that it was something to do with video management, and therefore ignored it.*

Clearly, that's less than satisfactory; you should be able to move an icon without having the whole sequence tumble round your ears. More advanced systems retain the teaching-by-showing idea, but attempt to work out what the actions mean, in effect constructing a command file by recording the operations after they have been interpreted by the screen management software. This should be much more satisfactory, as actions can now be associated with specific items of software, not positions on the screen. It is not clear how either of these approaches can provide for variables.

That gives us a hint on how we should redefine the session log. It is straightforward enough to make a video recording of what happens on the screen and replay it as desired, but in practice that isn't really what you want. It seems likely that what one really needs from a session log is a *description*, rather than a *replay*, of what was supposed to happen – or more precisely, what the system understood was supposed to happen, because a significant proportion of errors are caused by people giving what they mistakenly believe to be correct instructions to perform some task. It is, after all, not

entirely satisfactory to have to repeat a catastrophe in attempting to find out why it occurred.

A system of this sort (called *Applescript*) is available for Macintosh systems[IMP23]. It works some of the time. That isn't Apple's fault – it's a reflection of the greater complexity of a "command language" for a GUI. We saw that it isn't enough just to encode a sequence of screen coordinates of mouse clicks, so the instructions must be presented in some other way. So far, no one has come up with anything to beat text. You might expect that this would make the GUI command file similar to a command file for a textual system, but in fact it emphasises the difference in a very significant way : instead of producing the ordinary input to the programmes from a different source, it produces an equivalent but quite different input, and that is why it only works sometimes. Because the input is different, the programme receiving the input must be equipped with extensions which will understand the script language, and – for obvious reasons – most existing programmes are unable to do so. A programme which can understand the script language is called *scriptable*. (The file produced by the Windows Recorder isn't ordinary text, so you can't inspect it to see what's wrong with it.)

The Applescript system also includes facilities for a session log, intended as a means for recording command files. Again, there are complications; as the meaning of an action in a GUI system can only be determined in the context of the current screen display, and the correct interpretation is only known to the programme managing the part of the screen where the action occurs, the log can only be made with the cooperation of the programmes which will be used when the command file is executed. To give a useful log, then, a programme must be able to produce a textual instruction equivalent to any action performed when it is used. A programme which can do that is called *recordable.* This is quite a bit harder than responding to textual input, so there will doubtless be programmes which are scriptable but not recordable.

Here's an example of an Applescript programme, recorded from the finder. This version was recorded in 1998; the previous version was from 1995, and in 1997 we expressed the optimistic opinion that "it has probably improved now". In fact, the error is still there, but in the intervening years Word has become scriptable and recordable, so now there's a sequence describing the Word operations. The task recorded is that of duplicating a file, dragging the copy to a different folder, changing its name back to that of the original, editing it with Word, then deleting it.

The actions listed describe the events which caused the different parts to be produced. The time sequence is correct; the code listed in a row of the *Programme* column is not generated until after the actions in the *Action* column have been performed. (The intention is to avoid generating lots of code for meaningless operations – so merely selecting a file produces nothing, as you might next select something else without operating on the file first selected.)

Step	*Action*	*Programme*	*Comment*
1	Open the recording session on the Script Editor window.		Starts the recording.

2	Click on the "Discs 1" icon. Press <_-D>.	tell application "Finder" activate select file "Discs 1" of folder "Implementation (bu)" of folder "340 Overhead (bu)" of folder "340 Archive (bu)" of folder "Everything (bu)" of disk "Data" duplicate selection	Where to send the instructions. The full pathname. (The "(bu)" bits are part of the name.) < -D>. The new file is called "Discs 1 copy".
3	Drag the copied file to the "Overhead" folder.	move file "Discs 1 copy" of folder "Implementation (bu)" of folder "340 Overhead (bu)" of folder "340 Archive (bu)" of folder "Everything (bu)" of disk "Data" to folder "Overhead" of folder "340 papers" of folder "Current documents" of folder "Everything" of disk "Data"	
4	Select the file name. Delete the last five characters of the name. (By clicking at the end, followed by five deletes.) Double-click on the file icon to open it. (It's a Word file.)	select file "Discs 1" of folder "Overhead" of folder "340 papers" of folder "Current documents" of folder "Everything" of disk "Data" open selection select item "Discs 1 copy" of folder "Overhead" of folder "340 papers" of folder "Current documents" of folder "Everything" of disk "Data" set name of selection to "Discs 1" end tell	Some confusion. The name change cannot be assumed complete until the double-click, but then there are two things to do, and they appear in the wrong order in the programme. Stop sending to finder.
5	Do some editing. Close the file.	tell application "Microsoft Word" activate do script "LineDown 18" do script "EditClear -14" close window 1 end tell	Send to Word. Nothing specially significant. Stop sending to Word.

6	Drag the icon to the desktop.	tell application "Finder" activate move file "Discs 1" of folder "Overhead" of folder "340 papers" of folder "Current documents" of folder "Everything" of disk "Data" to desktop	Send to finder again.
7	Drag the icon to "Trash".	select file "Discs 1" delete selection	Though implemented as a drag, the operation is recognised as a delete.
8	Close the recording session on the Script Editor window.	end tell	

All the events have been converted into pure descriptions, independent of the geography of the interface. It's wordy, but it's hard to think of another way to do it. There seems to be no provision for parameters at present, so only absolute sequences can be encoded, which makes it not very useful; perhaps this defect will be remedied soon.

The system is not perfect yet. Notice that the first name in step 4 is "Discs 1", but the file selected was in fact "Discs 1 copy". (Proof : you get an error when you run the code; the error disappears when you patch in " copy" in the obvious place.) In fact, the "select file ..." and "open ..." items of step 4 should follow the "select item ..." and "set name …", but both pairs of operations were confirmed at the same time, and the Applescript system chose the wrong order. Don't blame Apple too much; it really is a hard problem.

REFERENCES.

IMP21 : Lane and Mooney[INT3], Section 21.2.

IMP22 : *Microsoft MS-DOS version 3.2 User's reference*, Zenith Data Systems Corporation, 1986.

IMP23 : *Using Applescript* (Apple Computer, 1994 : Documentation with the Applescript package.)

QUESTIONS.

The behaviour of CMS is complicated by its connection with CP, the system which implements virtual machines; but it is much easier for a terminal to communicate with CP than with its "real" operating system, CMS. Why ?

Experiment with Unix. What happens if you run a programme from a shell script. Can you get lines from a shell script into the programme ? Why (not) ? The closest thing to a terminal log is provided by the programme **script**. Try it. How does it work ?

A requirement for nestable command file execution is rather similar to that for conditional instructions. If command file A executes command file B,

what should happen at the end of B ? In a nested system, A will take up again from where it left off. How would you implement this in the various sorts of system ? How does it work in Unix ?

Think about any programme you have written which uses the GUI functions of the Macintosh system. What would it need to make it scriptable ? What would it need to make it recordable ?
