TERMINALS AS DEVICES

The idea of a terminal is not what it used to be. It was originally called a terminal because it terminated a wire, which ran between it and the computer. That is still true in a sense, but as the computer might now be inside the terminal's box, the name is a little artificial. Unfortunately, there doesn't seem to be a better one. "User interface device" is about the closest to the meaning we want, but it's clumsy, so we'll stay with "terminal", and by that we shall mean the combination of screen, keyboard, mouse, and any other associated bits and pieces which behave as a unit and through which the dialogue between the computer and someone using it is conducted.

THE NEED FOR SEVERAL DEVICES.

We remarked long ago in the chapter *USING TERMINALS* that we want terminals to do two quite separate things – sometimes simultaneously. We want them to be input and output devices of the programmes we run; and we want them to be our operating system control interfaces.

So why don't we have two terminals ? This is a depressingly obvious, simple, and straightforward solution which was rarely, if ever, adopted. We don't really know why. We recall no debate on the question at the time; it was just assumed that you had one terminal which you used for everything. It is also true that no one ever talked about the two different activities of terminals. It is quite possible that nobody thought of it; we have arrived at this view of the function of a terminal by a sort of analysis which was rare in earlier days, and essentially nonexistent among people who wrote operating systems, and who were mainly concerned with making something work somehow. From their (bottom-up) point of view, the available resource which everyone had was one terminal, so that was what they had to use. It is also possible that, despite the multiprogramming operating system, the idea that the *people* might be doing more than one thing at once was not well understood.

Perhaps the closest approach to recognising that there was a question there somewhere was the provision in several systems of means to share the screen (rarely the keyboard) between several streams. Unix offers an example : every Unix process has two predefined output streams, the standard output and the standard error. Both of these are normally directed to the screen.

Of course, there are other reasons. Cost is the most obvious, and was more significant in the early days of timesharing systems than it is now. Space is another : one terminal on your desk is bad enough, and two would be hard to accommodate in most people's work areas. A final reason is seen to be even more pressing when we consider current practice : what do you do when you need yet more terminals ? It is commonplace nowadays to use programmes which open several windows. Even if two terminals are manageable, can you manage ten ?

THE FACT OF ONE DEVICE.

So we have to make do with one terminal, which is why we are treating terminals as special devices. Even so, there is no reason why the programmes which use the terminals should have to worry about that, and plenty of reason why they shouldn't, most of which is that we want programmes to work with more than one sort of terminal. (This is a slightly specialised sort of device independence again.) We need *virtual terminals*, much as we have previously needed virtual devices of other sorts, but in this case we can't use the spooling trick of saving data in a disc file until the real device is ready.

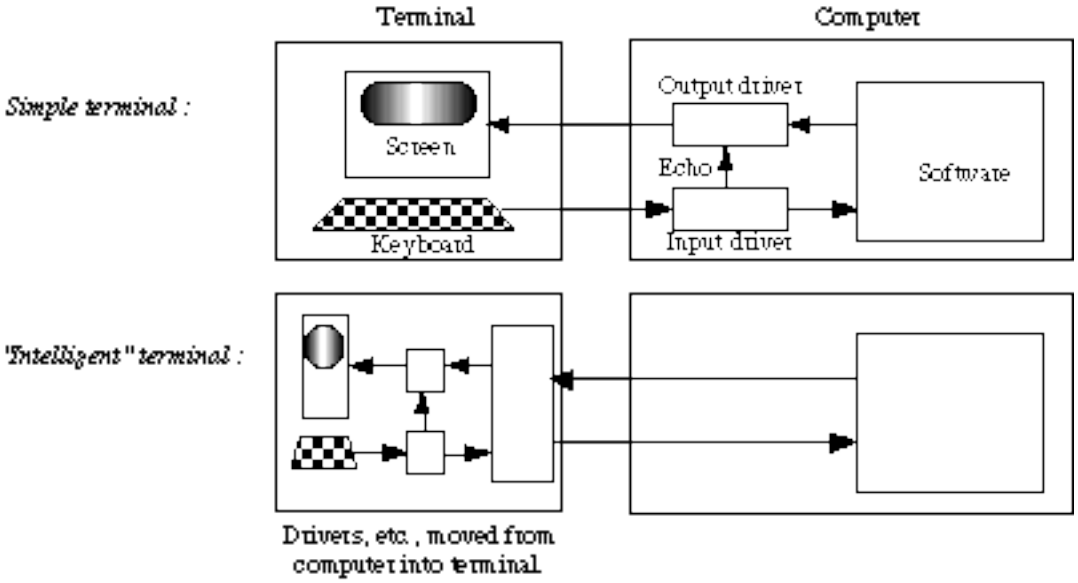
This gets us straight back to the software which we have already seen (in *USING TERMINALS*) that we want. When this is well developed, as is becoming more

common in recent systems, it might be called a *User Interface Management System*, or UIMS; the corresponding software in early systems was less well developed, often fragmented, and rarely dignified with a name. Whatever, if anything, we call it, though, this is a layer of software which comes between the programmes and the terminals, and which is concerned with managing the terminals to best advantage. Here we shall discuss systems with and without a UIMS separately. Notice that the presence or absence of a UIMS has, in principle at least, nothing to do with the nature of the operating system itself; Unix is still widely used both in the traditional way with character terminals, and through graphical interfaces.

WITHOUT A UIMS.

If there is no obvious UIMS, the terminals have to be handled by some combination of system and programme software. The terminals are thought of as being owned by the processes which use them. We shall concentrate on the simplest requirement, with the terminal concerned with only four data streams, providing input and output between terminal and running programme, and input and output between terminal and operating system. (We haven't forgotten the control streams, which are still necessary. In practice, though, the control streams are determined by what we wish to happen in the data streams, and in the case of a terminal are almost always carried by the same channel as the data streams. The logically distinct streams are, physically, merged into one, with the control stream represented as special identifiable characters or character sequences within the data stream. That's why the first 32 characters in the ASCII set are reserved for special purposes.) It's obvious from everyday experience that more complicated systems which can handle several processes at once through separate windows work; they can even be made to work on character terminals, though that does require either character-addressable terminals or the transfer of whole screens of information at a time between terminal and computer. Details, particularly for the addressable terminals, were commonly handled by the programme rather than the operating system.

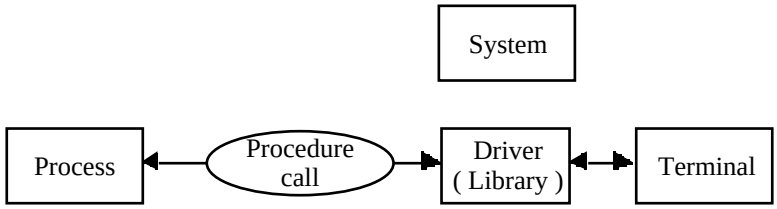
Communication screen by screen is interesting, because it was a very common mode of operation of some timesharing systems for many years. Typically, the computer would display some pattern on the screen, this would be modified more or less extensively through the keyboard without any communication with the computer, then the whole modified pattern would be returned to the computer. (That's why the ENTER key is so called; terminals of this type had both a RETURN key, which did, and an ENTER key, which initiated the transfer to the computer. The keyboard on which this text is being typed has an ENTER key, which RETURNs, and no RETURN key. We don't know why.) Relegating some of the work to the terminal in this way is quite a clever trick, as it reduces the frequency of interruptions to the computer's operations. This is very effective if the nature of work to be done lends itself to this stepwise operation, as it does with the forms interfaces which were, and are, popular and effective in many areas of commercial work. The disadvantage of this method is that it requires more processing in the terminals – which introduced the idea of intelligent terminals – and much faster communications between terminal and computer than was needed with simple character-at-a-time transmission. The difference between the systems is illustrated in this diagram :



We speak from personal experience in adding that the screen-by-screen mode is astonishingly bad for anything except forms or menus; programme development and text preparation and editing are severely impeded by the sudden disappearance of the context in which you're working when you happen to reach the bottom of the screen.

UIMSless terminal management is perhaps the simplest and most obvious organisation, and is found in many early multiuser interactive systems and most simple microcomputer systems, and occasionally encountered in large systems even now. The terminal is closely identified with the running process, and it might therefore be comparatively difficult for the terminal to discharge its other duty by communicating with the operating system while a process is running. It's convenient to distinguish between two subclasses.

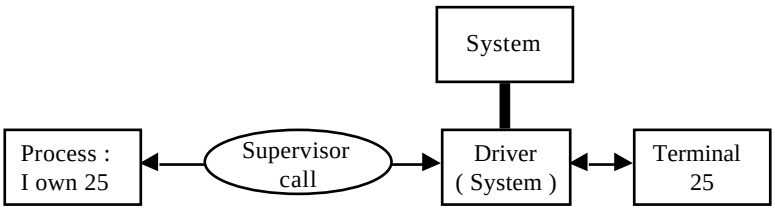
- **The terminal is driven directly by the process.** The programme contains the software which communicates directly with the hardware interface. The system might provide a set of procedures, but they are used as a library just as you might use a mathematical function library, with copies as required running completely in the process's addressing space. We might call this the *monitor model*. This is perhaps the obvious way to drive a terminal if you don't have the benefit of years of development of operating systems for guidance, and is found in primitive monitor systems – including the IBM system CMS (Conversational Monitor System), though that has its own peculiarities which make it a special case. (We shall say a little more about CMS shortly.) Here is a diagram which shows the features of this system architecture *very* schematically; it's intended mainly for comparison with the diagrams of other systems which follow :



With this organisation, any communication with the operating system is at the mercy of the process – so there usually isn't any. (That might not be the process's fault; it is not always easy to talk to the operating system from a programme. Early systems in particular commonly paid no attention to the requirement for an application programmer interface, assuming that all imaginably necessary functions were provided as input and output procedures. Many later systems are not much better.) It has the perhaps less significant, but annoying, disadvantage that, as each process has its own terminal buffer, you might lose characters if you try to "type-

ahead" as one process is being replaced by another. The system can still send information to the terminal using its own output procedures, so error messages or other urgent information can be displayed, but as it has no means of determining what's on the screen these messages are commonly disruptive of any material already displayed, and might be quite hard to read. Apart from that, the process has complete control over the terminal.

- **The lowest level of processing is handled by the system.** The system's involvement is only at a low level, but its procedures handle things like buffering and character translation if needed. There is a permanent layer of operating system software between the hardware and the programme, and the programme's requests for service, now conveyed as supervisor calls, must be dealt with by this layer. We'll call this the *supervisor model*. A free representation :

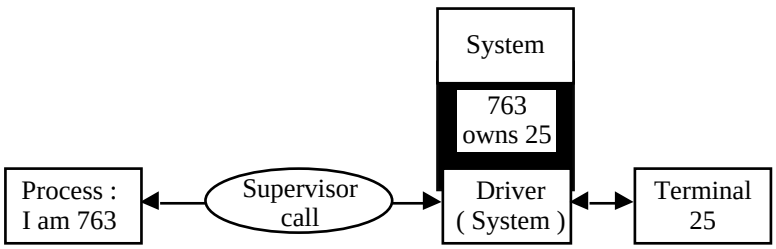


The function of the system's layer has more to do with protection than with constructive assistance with input and output; it appeared when the introduction of shared systems made it necessary to hide the real devices from the programmers, and it does little more than that. Because the system is involved, it can (though it need not) also monitor incoming text for escape characters (or *command heralds*) which can signal messages to the system, and it can respond by getting information from the system, stopping the active process, etc. While this does provide a path to the system, it is rarely possible to use the full system in this way; instead, a few specific operations are provided as an incidental activity of the terminal. Except when engaged on such special tasks, the terminal remains clearly under the process's control. One noticeable difference is that, because the direct control of the terminal is now the responsibility of the system which is there all the time, there need be no dead spaces between programmes, so "type-ahead" might now be possible; if you know exactly what you have to type next, you can type it without waiting for the system to catch up in the confidence that it will be handled properly. This is a useful feature, but it seems to have been lost somewhere along the way to GUI systems.

WITH A UIMS.

With this organisation, all terminal transactions are mediated by the system. There is an association between terminal and process, but this is in principle at the disposal of the system, and may be changed from time to time according to circumstances; the terminals are no longer owned by the processes, but by the system. This could be called the *system service model*. It should be emphasised that there is no hard-and-fast line between this scheme and the second scheme discussed above. The difference is quantitative rather than qualitative, reflecting an increasing involvement of the system software in terminal transactions, but the point at which the terminal and process become independent is a convenient boundary in our present discussion.

The nature of the transaction is now rather different. The process asks the system to do things to the display, or the system receives input from a keyboard or mouse and works out where to send it. A diagram :



With this architecture, the system acts as intermediary between process and terminal, and can be substantially involved in transactions to whatever arbitrary extent is desirable in each case. Before rushing onward, we should pause to ask just what sort of involvement is desirable. We have already noticed that some rather low level interactions are likely to be useful : in *USING TERMINALS*, we remarked on the usefulness of programmable function keys and orderly control of screen layout, while in *MAKING LIFE EASIER* we mentioned command files and session logs. In addition, several systems provide facilities for defining macros (text strings which are automatically translated into other – usually longer – strings whenever they appear) and "wild" characters (usually appearing in file names, and expanded using information available from the system tables). What else should be included ?

There are at least two easy, correct, and rather unhelpful answers to that question. They are : "we don't know"; and (getting right back to our original aims) "whatever makes it easier to use the system productively". Perhaps a better, and more constructive, answer is that, as there are clearly useful sorts of intervention between process and terminal, and terminal technology and usage styles are changing rapidly, it is sensible to make provision for such changes or extensions as might become necessary. This architecture provides access to the terminal streams in a general way, and should therefore be a good base on which to build any additional facilities that might be required.

Its versatility is demonstrated in that the same model works with both traditional text interfaces and GUI systems. In simple cases, the nature of the system's involvement in managing the terminal might make little difference, but its significance is clearer when a terminal can be used with several processes concurrently. Once again, it is convenient to distinguish two cases.

- **The terminal can be switched from process to process.** The DEC operating system Tops-10 illustrated this organisation. A terminal could be detached from a process and attached to another; or you could detach your terminal from a process, go to another terminal, and attach that one to carry on working. While attached to a process, communication with the system was by an escape character as with the second group of terminals owned by the process, but general operating system functions were available to a detached terminal.

- **The terminal can be shared by many processes.** This is the mode of operation of terminals used with windowing systems, and we discussed it a long time ago (*GUI – GRAPHICAL USER INTERFACE*) in dealing with the user interface. While there are various ways of driving windowed screens, the software must usually know which process owns which part of the screen at any moment, so that messages from the process can be displayed in the correct place, and so that messages from the keyboard or mouse can be sent to the correct process.

In both cases, the association of the terminal with several processes introduces a new feature into the system : switches between running processes can now be initiated directly from the terminal. While some process switching from the terminal was sometimes possible with early systems, this was almost always restricted to switches between the running process and the system itself; the novelty is that switches between arbitrary processes are now possible. This is particularly prominent with GUIs, where a process switch is as simple as clicking in a new window – where we again rely on the system's knowledge of the current screen geography to make the implementation possible.

INTERACTION BETWEEN INPUT AND OUTPUT.

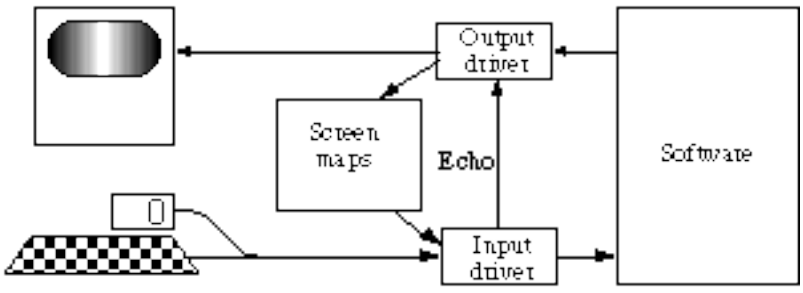
In the two preceding paragraphs, we have noticed actions of a GUI for which the system must maintain information about the configuration of the screen. A special feature of modern practice in using terminals is that the significance of the input is strongly dependent on the current configuration of the screen. This is most obvious in the interpretation of mouse clicks in selecting icons or windows on a GUI screen, but exactly the same principles are involved in choosing a menu item by using the "arrow" keys to

move a cursor. This is not just a curiosity; it is at the root of the effectiveness of the GUI. Recall the notions of limiting the number of modes in an interface which we met in *ABOUT GRAPHICAL USER INTERFACES*. You could make a completely modeless interface if you could provide a keyboard with a separate key for every conceivable operation which would ever be needed. As a practicable proposition, that's obviously ridiculous – but by taking note of the screen context of actions, we are in effect doing something quite like it, as we can define a new "key" on our abstract "keyboard" for any new action we require.

Some earlier systems had simulated this type of interaction, notably in menu and forms interfaces and in screen editors, but little or no support was provided by the operating system; instead, the programme had to maintain an internal copy of the screen, follow the path of a cursor by counting cursor movement characters, and work it out in the obvious way. This worked well most of the time, but it was not unknown for occasional characters to be lost in a busy system, or for screens to be less than accurately written – and this could give most remarkable results from an editing session.

The UIMS must therefore track the mouse movements, moving the pointer accordingly, and know where all current windows are on the screen, and how they overlap. A signal from an input device can then be directed to the intended destination, which might be the process associated with the currently active window, or the operating system if a process switch is required.

What sort of information should the operating system maintain in order to manage a GUI in an acceptable way ? The answer clearly depends on just how we want the screen to behave, and we sketched a few ideas on the subject in the chapter *GUI : GRAPHICAL USER INTERFACE*. It appears from that description that the essential information is the dispositions of the windows on the screen, which includes both their spatial positions and the "front-to-back" stacking order, the identity of the process owning each window, and the current position of the pointer. If we want the interface transactions to interfere as little as possible with the activities of the processes in the system, then we shall also require that the system maintains a record of the contents of each window in the display, so that temporarily hidden windows can be shown again when other items which obscure them are moved. To perform this function reliably, the interface manager must have at its disposal many times the amount of memory needed to store the information for the screen itself; if that is deemed too expensive, there must be provision in the application programmer interface for the manager to ask processes to redraw portions of their windows which have been uncovered. These considerations lead us to a picture of the interface system which is something like this :



DEALING WITH COMMUNICATION.

As well as the data structures required by the GUI, we must discuss the communications between GUI and process. The diagram above shows device drivers, which control the details of the devices. Qualitatively, there is nothing new about these components for a GUI; as with any other device, the process issues instructions which correspond to a certain predefined set of actions, and the device sends items of information at

unpredictable times to the process. We shall discuss general techniques for handling such operations later in this section.

A new feature is the quantity of data which must be transferred from interface to process. To be more precise, it is the rate of arrival which is new, together with the requirement that the data must commonly be dealt with at something approaching real-time speeds. We are not unused to data for a process arriving at rates of several megabytes per second, but until quite recently we could often assume that it would be sufficient to place them in a buffer and work on them when opportunity arose. Data rates from a moving mouse are not quite in the same range (say, a few hundred mouse movement signals per second), but to give an acceptable response it might be necessary to track each one and make some response to change the displayed image. This data stream might be handled as a stream of interrupts or a stream of messages; a common strategy is to organise them into an *event queue*, essentially a stream of messages each reporting one elementary input action – such as a mouse movement, mouse click, key depression, etc. The process is expected to retrieve the events from the queue and deal with them as they arrive. To give reliably good response in such an environment, a programme must ensure that it inspects its event queue very frequently and the operating system must ensure that the programme gets a chance to inspect the queue; in consequence, the event queue might become the feature round which the whole programme structure is built, whatever the real purpose of the programme. A programme constructed for such a system is likely to operate as a collection of threads, one for each window, which spend much of their time waiting for events connected with their respective windows. The result is a significant change in the way we write our programmes – but that's the programmer's problem.

The operating system's job is to generate the required messages, and send them to the appropriate processes. The structure of an event is determined by the constraints which it must satisfy. If a process owns several windows, it must be able to sort out which incoming event corresponds to which window; each window must therefore have some sort of identifier. Similarly, the messages must identify the physical event which caused them, so there must be an event identifier. An event must therefore include at least a window identifier and an event identifier, and will typically contain further information – such as the window coordinates for a mouse click.

The interface can do even better if the operating system provides for threads. A process can then associate a separate thread with each of its windows, so that on receiving notification of a screen event it need not go through its own analysis of which window is where to identify the correct destination for the event.

REMARKS ON THE APPLICATION PROGRAMMER INTERFACE.

There are clearly advantages in having the operating system involved in managing the terminals, but not all is necessarily sweetness and light. This organisation offers a much smoother and more predictable user interface, as the low-level features at least are controlled by the system, not by individual programmes. It is not always true that the application programmer interface is equally satisfactory. As the programmer can no longer control the terminal directly, the operating system's software interface must provide all the control functions which might be needed, or software quality can be affected.

An attempt to provide tutorial software for beginners many years ago was frustrated by a system with a poorly designed software interface. In order to help determine (read "guess") whether a protracted cessation of input was indicative of deep thought or merely the consequence of someone neglecting to press the RETURN key, we wished to be able to determine whether or not a character was in the input buffer. Unfortunately, the interface software provided no means to do so; all we could do was attempt to read from the terminal, and the read

operation would only complete if and when a key was pressed. We had to abandon that part of our attempt to develop a rather early "user-friendly" interface.

To provide an effectively complete set of system calls for a simple text interface is not difficult. Operations to read and write characters, to find out about the interface state ("is there a character ready for reading ?"), and execute control functions (newline, clear screen, etc.) must be available. (Though the control functions are in practice usually implemented by transmitting characters to the terminal, they are not the same sort of operation as ordinary reading and writing, and – at the programming level – different system calls are appropriate). To do the same trick for a GUI is enormously more complex. Before even starting real communication through the interface, the programme has to draw windows, and in each window there are many parts, each of which must be defined. To make window construction as easy as possible, standard sets of procedures are provided for interfaces of different styles. We saw earlier – in *DEFINING A SYSTEM INTERFACE* – that this was desirable to encourage people to maintain consistency in interface design. The result is a very large application programmer interface, which might be consistent, is certainly not simple, and is by no means always helpful. (We point out that in this context we're evaluating the application programmer interface, so we must make our judgments from the point of view of someone who uses that interface – which is to say, in terms of the programmer's view of the interface software. Even so, we would still like to present a small system mental model, but we know of no GUI software for which this aim has been achieved.)

GUI FROM A DIFFERENT STANDPOINT.

Just in case you think all this sounds easy, here's a note on some aspects of GUI systems which might not have occurred to you. When GUIs appeared, they were widely acclaimed as a great leap forward – except by people who couldn't see, but had been managing rather well on the conventional text interfaces. For them it was a great leap backward, and only now are systems becoming available which make the GUI interfaces tolerable for people who have poor, or no, vision. The difficulty is obvious : if the use of an interface depends strongly on the feedback from the visual terminal, and you can't see the terminal, you are in trouble.

The difficulty for a programmer trying to do something to solve the problem is more subtle. The solution must be to collect the information on the screen and present it in some manner which doesn't depend on sight; an audible presentation of some sort is the usual choice. (Tactile methods are possible in principle, but difficult to manage in practice; no one has produced a satisfactory tactile screen yet.) The first task is to collect the information. That isn't too hard if you're writing a specific programme, because the information is all available – but that doesn't provide a general solution, and a general solution is necessary if the interface is to be useful. Unfortunately, in the general case the information is not available : all you have is the screen display, and your software must try to interpret what's there. It is much harder to convert pixels into sense than sense into pixels.

To avoid this problem, a software architecture intended for exactly this purpose has been proposed[IMP19]. It is suggested that a rather specific application programmer interface should be provided as an extension to the X-Windows interface specification. This interface provides facilities for an agent to acquire information about any X object from the X library procedure which converts the information into the graphical equivalent, with the information presented carefully designed to include the semantically useful material which determines the picture. For example, if text is displayed, the original text itself should be available from the interface, while for a picture a brief description could be presented. (Such a description must be provided with the image; it is not proposed to build in picture recognition facilities. Compare the ALT attribute of the HTML IMG element.) By this means, it should be possible to obtain all the information required about a system's screen display, without the difficulty of decoding the screen image.

Presenting the information in audible form is another problem, but at least this technique should make it easier reliably to get the information to be presented.

COMPARE :

Lane and Mooney[INT3] : Sections 4.3, 4.6.

REFERENCES.

IMP19 : W.K. Edwards, E.D. Mynatt : "An architecture for transforming graphical interfaces", *UIST '94 Seventh Annual Symposium on User Interface Software and Technology* (ACM Press, 1994), 39.

QUESTIONS.

We wrote above : "It has the significant disadvantage that, as each process has its own terminal buffer, you might lose characters if you try to type ahead as one process is being replaced by another." Where do the lost characters go ?

- and here's a message sometimes displayed on a Macintosh screen : "Unfortunately, no one is listening to keystrokes at the moment. You might as well stop trying.". What does that mean, and how can it happen ? Is it a good thing ? Is it a good example of user interface design ?

What sort of signals have to be exchanged between the user interface and the underlying software to make a GUI work ? Try "walking through" a typical session on a computer, considering how it would be implemented on both Unix and a Macintosh or Windows system.

How would you codify a set of signals such as those you found in the last question in such a way as to encourage other people to use them ?
