AT&T Bell Laboratories Technical Journal
Vol. 63, No. 8, October 1984
Printed in U.S.A.

The UNIX *System:*

The Blit: A Multiplexed Graphics Terminal

By R. PIKE*

(Manuscript received August 1, 1983)

The Blit is a programmable bitmap graphics terminal designed specifically to run with the *UNIX*™ operating system. The software in the terminal provides an asynchronous multiwindow environment, and thereby exploits the multiprogramming capabilities of the *UNIX* system, which have been largely under-utilized because of the restrictions of conventional terminals. This paper discusses the design motivation of the Blit, gives an overview of the user interface, mentions some of the novel uses of multiprogramming made possible by the Blit, and describes the implementation of the multiplexing facilities on the host and in the terminal. Because most of the functionality is provided by the terminal, the discussion focuses on the structure of the terminal's software.

I. INTRODUCTION

The Blit[†] is a graphics terminal characterized more by the software it runs than by the hardware itself. The hardware is simple and inexpensive (see Fig. 1): 256K bytes of memory dual-ported between an 800-by-1024-by-1-bit display and a Motorola MC68000 microprocessor, with 24K of ROM, an RS-232 interface, a mouse, and a keyboard. Unlike many graphics terminals, it has no special-purpose graphics hardware; instead, the microprocessor executes all graphical

* AT&T Bell Laboratories.
[†] The name comes from the second syllable of the `bitblt` graphics operator.[1,2] It is not an acronym.

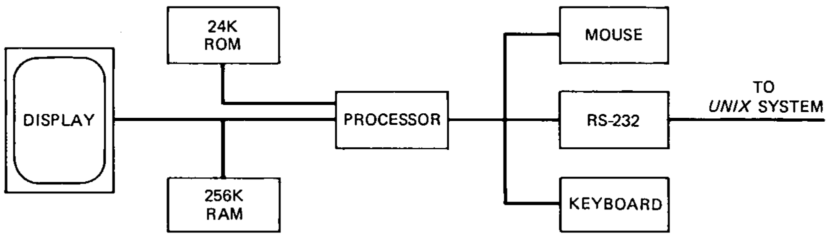


Fig. 1—Hardware overview.

operations in software. The reasons for, and consequences of, this design are discussed elsewhere.[2]

The microprocessor can be loaded from the host with custom applications software, but the terminal is rarely used this way. Instead, a small multiprocess operating system is loaded into the terminal, and the processes under that operating system are then loaded. The operating system is structured around asynchronous overlapping windows, called layers.[3] Layers extend the idea of a bitmap and the `bitblt` operator[1,2] to overlapping areas of the display, so a program may draw in its portion of the screen independently of other programs sharing the display. The Blit screen is therefore much like a set of truly independent, asynchronously updated terminals. This structure nicely complements the multiprogramming capabilities of the *UNIX* system and has led to some new insights about multiprogramming environments.

Programs in the terminal have access to an extensive bitmap graphics library, which is implemented using the `layerop` primitive,[3] and is distinct in its use of abstract data types for geometrical objects and its lack of device independence—the library is closely coupled to the terminal and its programming environment.[2] The programs that have been written for the Blit include a popular text editor with a paucity of commands, a debugger that can be used effectively without reading any documentation, a surfeit of 24-by-80-character terminal emulators, and not nearly enough games. But this paper is not about the programs in the terminal so much as their environment and interrelationships. Reference 3 discusses how to update overlapping windows asynchronously; this paper discusses what to do with them.

The discussion is in three main sections: an overview of the history and motivation behind the terminal, a brief description of the user interface, and some details of the implementation. The reader is assumed to have some familiarity with the *UNIX* operating system, although the details relevant to the Blit will be discussed.

II. HISTORY AND MOTIVATION

The original idea behind the development of the Blit hardware was to provide a graphics machine with about the power of the Xerox Alto,[4] but using 1981 technology (large address space microprocessors, 64K RAMs, and programmed array logic) to keep size, complexity, and, particularly, cost much lower. Too many graphics work stations are so expensive that several people must share one, sometimes using sign-up lists.

Because we refuse to have rotating machinery in our offices, we wanted to build the Blit around a network interface rather than a disc. But after several lengthy discussions we decided that network hardware and software were not yet inexpensive, available, or reliable enough to be the center of a work station (the situation now is hardly better). Rather than compromise our principles, and to keep costs low, we therefore chose to make the Blit a regular terminal with an RS-232 Electronic Industries Association (EIA) port to a time-shared host. Only one integrated circuit is needed to connect the microprocessor to the EIA line, so the electronics fits on a single board, which minimizes cost, size, and packaging complexity—the board mounts inside the monitor cabinet. This decision to use RS-232 limited the high end of the capabilities of the Blit, but it expanded the low end enormously. Blits can be used anywhere 24-by-80 ASCII terminals are used, including each office in our research center.

But perhaps most important (at least to us), Blits are inexpensive, portable, and so easy to communicate with that we can take them home. Researchers in our group have 1200-baud dial-up terminals at home. For the home computing environment to be effective, it must be as similar to the office environment as possible; although 1200 baud is slow (our terminals at work run at 19,200 baud), a Blit at 1200 baud is much better than a regular terminal at 1200 baud. Also, the local processing power of the terminal can make up for some of the reduced bandwidth. So although a high-speed network would be desirable, much of the Blit's success can be attributed to the use of RS-232.

We initially intended to use the Blit to explore interactive graphical environments along the lines of Smalltalk, but soon decided that we had neither the energy nor the inclination to build a complete programming environment. The *UNIX* system has a comfortable set of tools for program development and general programming that would require great effort to reproduce, but that we wanted to use when developing and using the Blit. Also, the *UNIX* system is the framework of all computing done in our group and is not likely to be supplanted easily by something new, no matter how attractive. We therefore began thinking about using the Blit to improve the programming environment, rather than replace or even merely add to it.

One of the distinguishing characteristics of the *UNIX* system is multiprogramming, the ability to run several programs at once. The best known use of multiprogramming is the pipe, an I/O connection between two processes that sends the output from one process to the input of another. The *UNIX* command interpreter, called the shell, has a simple syntax for pipes:

```
who | lpr
```

which sends the output of `who` to the `lpr` command, which spools output for the line printer.

Programs in a pipeline are related by their interconnection, but the *UNIX* system also allows unrelated processes to execute simultaneously. The shell postfix operator `&` runs a command in the background, that is, without waiting for it to finish. For example,

```
cc prog.c &
```

runs the C compiler on the file `prog.c` and immediately returns to the user; normally, the shell would wait for `cc` to complete before reading the next command from the terminal. Background processes have their input disconnected from the terminal, but messages printed on the terminal will appear there, asynchronously with other input and output on the same terminal. This can be annoying if a process using the terminal interactively is maintaining a full-screen image, because output from background processes will modify the screen image without the foreground process's knowledge. For example, error messages from a background `cc` will interfere with a screen editor.

The problem exists because several processes are using a single terminal for their I/O. If the terminal were multiplexed between the processes, their input and output could be kept separate. The "job control" software[5] developed by Jim Kulp at International Institute for Applied Systems Analysis in Vienna and Bill Joy at the University of California at Berkeley allows the user to pass the terminal between processes on the same terminal, essentially by flipping processes from the background to the foreground at the user's signal. But the state of the terminal is not maintained correctly when the user flips between processes—the screen contents and terminal modes are not restored to those of the new foreground process. The problem is resolved by interfacing the editors to the job control mechanism so they can preserve the screen's appearance; but that is far from transparent to the programs.

To provide a better terminal for use by the *UNIX* system, we began thinking about programming the Blit so each process or related set of processes has a reserved portion of the screen, called a window. That way, compiler error messages appear in the window where the compiler is running, and editing can continue undisturbed in another window. If the terminal maintains the state for the various processes and provides an appropriate user interface for creating and switching between windows, the *UNIX* system need not have job control or maintain the state of the screen for the various processes. Instead, the *UNIX* system can treat the windows like individual terminals.

Most window systems permit the user to focus attention on one window at a time, with the other windows maintained statically. Windows on the multiprogrammed *UNIX* system, however, must be updated asynchronously. That is, characters written to a window by a process must appear immediately, regardless of whether the user's keyboard is currently connected to that window. Otherwise, compiler errors would not appear until the user asked for them, which would cancel some of the advantages of multiprogramming. Also, as will develop later, the possibility of conveniently controlling asynchronous processes leads to some innovative computing techniques.

While the Blit hardware was being designed, we experimented with asynchronous windows on a Blit predecessor built by Dave Ditzel. Following the pattern set by "intelligent terminals," we programmed the terminal to interpret escape sequences to create, delete, and switch the host character stream between windows. A program on the *UNIX* system sat between the user programs and the terminal, and inserted escape sequences in the character stream to send data to the correct window. Although this early implementation was clumsy and fragile, it demonstrated the feasibility and power of an asynchronous window terminal and pointed out the issues that must be resolved for a workable multiwindow terminal:

1. Windows must be updated asynchronously. The trial system was primitive but worked well enough to be convincing.

2. The screen is not big enough (regardless of how big it might be). Therefore, windows must overlap. The desires for overlap and asynchronism led to the development of layers, an implementation of overlapping, asynchronously updated windows.

3. The software to generate the incremental control information (escape sequence "switch to window *x*") from high-level requests ("draw these characters in this window") was messy—too much state information was maintained by the terminal and guessed at by the *UNIX* program. The implementation also encouraged attempts to optimize the number of characters sent, which added to the complexity, a situation familiar to authors of screen editors. Putting all data into labeled packets eliminates this confusion and obviates optimization.

4. A simple RS-232 connection is not robust or controllable enough to connect two communicating programs, in this case the *UNIX* system and the code in the terminal. An error-corrected protocol with flow control is required.

5. To draw graphics in the windows, sending escape sequences is traditional but makes poor use of the processing power of the terminal, and requires the terminal to be preprogrammed with all desired

capabilities. Contrary to popular usage, an intelligent terminal is not an idiot savant; it is one that can be educated. If the terminal could be dynamically programmed, the desired functionality could be added on demand. Our solution was to write a small time-shared operating system for the terminal, called `mpxterm` (multiplexed terminal), into which we dynamically load programs from the host, customizing the terminal process running in a layer for the execution of a particular graphics task.

The Blit therefore developed into a programmable graphics multiplexer, distributing the terminal resources—screen, mouse, keyboard, RS-232 interface—between terminal processes connected to independent *UNIX* system processes.

Since the design of the terminal's software was largely dictated by the desired user and programmer interface, the next two sections present the overall user interface and an overview of two programs that run in the multiplexed environment. The subsequent sections outline the implementation of the multiplexing software.

III. WHAT THE USER SEES

After logging in to a *UNIX* system, a Blit user types `mpx` to the shell. The multiplexed terminal code is then down loaded into the terminal, which takes a few seconds at 19,200 baud and about two minutes at 1200 baud. `Mpxterm` includes all the graphics primitives, but since the graphics primitives and interrupt-level I/O drivers execute out of read-only memory, they are not down loaded.

`Mpxterm` is controlled by the mouse. Of course, programs running in the terminal may also be controlled by the mouse, so some rules must decide which mouse events are interpreted by which process in the terminal.

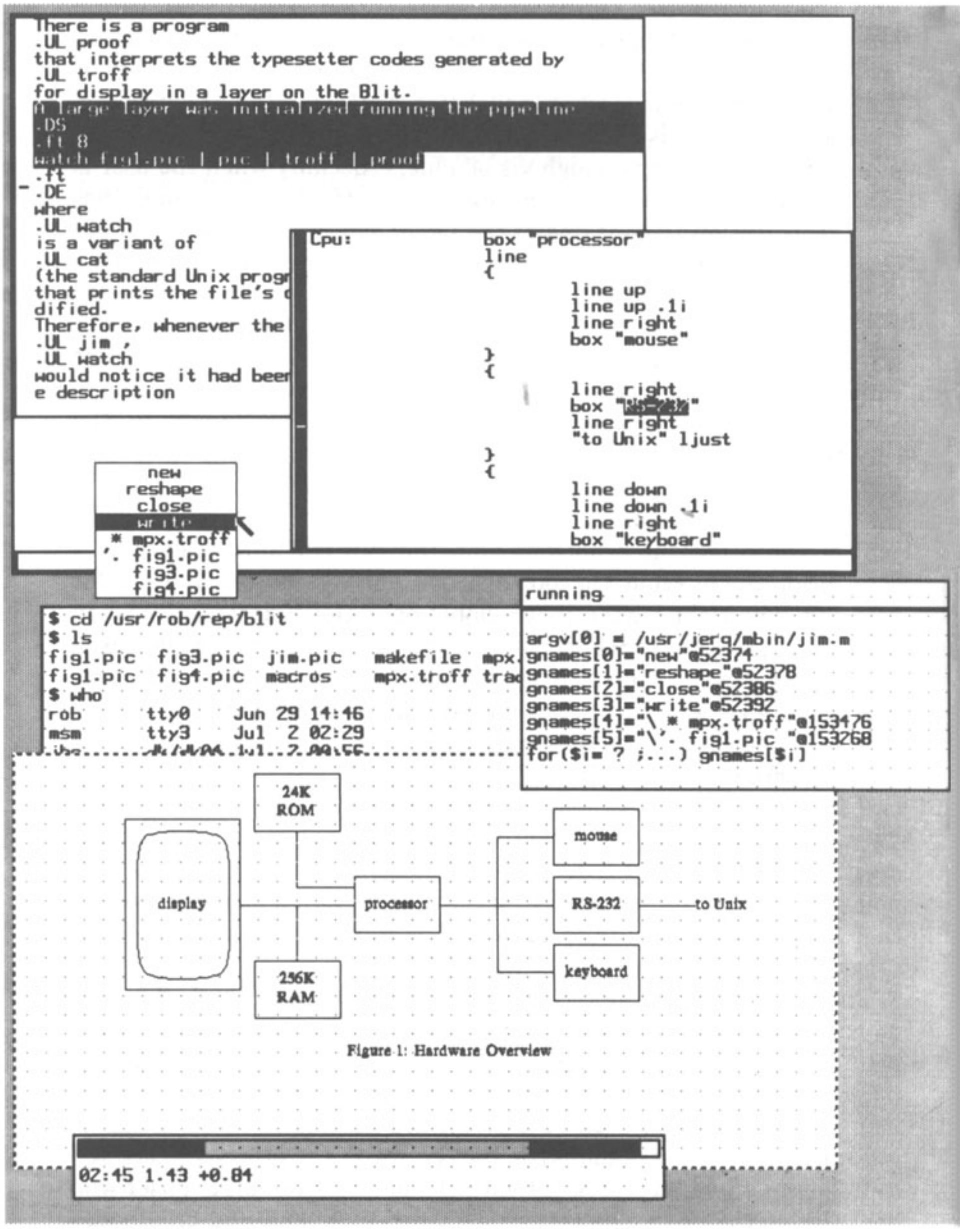
The screen consists of several possibly overlapping layers. Portions of the screen not occupied by layers are "colored" with a distinctive grey texture. Except for internal control and demultiplexer processes of `mpxterm`, terminal processes are one-to-one with layers. Once the first layer has been created, exactly one layer is the current layer, that is, the layer that receives keyboard characters and interprets mouse motion and button hits. The mouse and keyboard come as a pair; all user input is directed at a single process. The control process continually updates the current process's mouse coordinates and button state, and a process may ask to be suspended until it is current. When a button is depressed, the current process receives the event if the mouse cursor is pointing at a visible portion of the process's layer; otherwise, the button hit is interpreted by the `mpxterm` kernel.

To identify the current process, the layers of all noncurrent processes are stippled by a gauzy texture, leaving only the current layer with a clear image* (see Fig. 2). The usual solution to this identification problem is to label the windows, but we elected not to label them because the label takes up useful screen space and either the user or the program must decide what the label is. Neither option is appealing. Another possibility is to distinguish the borders of the layer, but that probably isn't a strong enough visual clue, especially when the user is concentrating on a portion of a large layer. However, we admit that this identification issue is one of the uglier aspects of the system and that our solution is, at best, a small improvement over others. One decision that differs from the usual, but in which we are on firmer ground, is our requirement that a mouse button hit changes the current layer. In most systems, the location of the mouse defines the current window, but when the current window may be partially or even wholly obscured, this is unworkable. (It makes sense, and is common, for the current layer to be obscured: consider typing instructions to a command in one layer based on data displayed on a graph in another large, nearly full-screen, layer.)

The mouse has three buttons, and the Blit software maintains a convention about what the buttons do. The left button is used for pointing. The right button is for global operations, accessed through a menu that appears when the button is depressed and makes a selection when the button is lifted. The middle button is for local operations such as editing. Put simply, the right button changes the position of objects on the screen, and the middle button changes their contents. For example, pointing at a noncurrent layer and clicking the left button makes that layer current. Pointing outside the current layer and pushing the right button presents a menu with entries for creating, deleting, and rearranging layers. Clicking a button while pointing at the current layer invokes whatever function the process in that layer has bound to the button. The next section discusses two programs and how they use the mouse.

The state of mouse input is reflected by the cursor tracked by the mouse as it is moved. Usually, the cursor is an arrow pointing to the pixel at the mouse's location. A program may change the cursor to reflect its state. For example, when the user selects `New` on the `mpxterm` menu, the cursor switches to an outlined rectangle with an arrow, indicating that the user should define the size of the layer to be created by sweeping the screen area out with the mouse. Similarly, a user who has selected the `Exit` menu entry is warned by a skull-and-crossbones

* This practice interferes with noncurrent processes drawing in their layers, but most graphics in the Blit world is done in XOR mode, which commutes with the stippling, and the operating system provides a simple routine to help with graphics that are not XOR.



```
There is a program
.UL proof
that interprets the typesetter codes generated by
.UL troff
for display in a layer on the Blit.
A large layer was initialized running the pipeline
.DS
.ft 8
watch fig1.pic | pic | troff | proof
.ft
.DE
where
.UL watch
is a variant of
.UL cat
(the standard Unix progr
that prints the file's c
dified.
Therefore, whenever the
.UL jim ,
.UL watch
would notice it had beer
e description
```

```
Cpu:                 box "processor"
                     line
                     {
                               line up
                               line up .1i
                               line right
                               box "mouse"
                     }
                     {
                               line right
                               box "RS-232"
                               line right
                               "to Unix" ljust
                     }
                     {
                               line down
                               line down .1i
                               line right
                               box "keyboard"
```

```
$ cd /usr/rob/rep/blit
$ ls
fig1.pic   fig3.pic   jim.pic    makefile    mpx
fig1.pic   fig4.pic   macros     mpx.troff   tra
$ who
rob        tty0     Jun 29 14:46
msm        tty3     Jul  2 02:29
```

```
running

argv[0] = /usr/jerq/mbin/jim.m
gnames[0]="new"@52374
gnames[1]="reshape"@52378
gnames[2]="close"@52386
gnames[3]="write"@52392
gnames[4]="\ * mpx.troff"@153476
gnames[5]="\'. fig1.pic "@153268
for($i= ? ;...) gnames[$i]
```

```
02:45 1.43 +0.84
```

Fig. 2—A representative Blit screen. The small layer at center right is running the debugger `joff`, which is examining the menu data structure in the text editor `jim`, running in the upper layer. `Jim` is the current process — its layer is not freckled — and is editing the files for this paper: `mpx.troff` is the troff input, and the various `fig` files are `pic` descriptions of the illustrations. The lower `jim` window is editing the description for Fig. 1, and when the user selects `write` from the menu, the file will be written and the picture in the typesetter emulation layer at the bottom will asynchronously draw the new picture (see the text). The small layer at the bottom is

cursor that confirmation is required before that potentially dangerous operation will be executed.

IV. TWO APPLICATION PROGRAMS: JIM AND JOFF

A variety of programs have been written for the `mpxterm` environment. As with any graphics terminal, the first few programs were games, which in this case were characterized by being self-playing, at least optionally. On the multiplexed Blit screen, a game program can play itself while the user does putatively useful work in another layer. After the games came a spate of terminal emulators, coinciding with the proliferation of Blits inside our research center and triggered by the desire to promote the programs written for the 24-by-80 displays. This period has passed, and not entirely because a successful emulator has been created. Even strong supporters of the cursor-addressing style of terminal control have accepted the possibilities of a customized terminal program and communications protocol. Many of the 24-by-80 programs have been supplanted by Blit programs that divide the task between the host and terminal. Two programs that divide the labor effectively are `jim`, a text editor, and `joff`, a debugger for `mpxterm` programs. References 6 and 7 describe their user interfaces and the details of their implementation. Here we present an overview of their structure and illustrate how they use the programmability of the terminal.

`Jim` is a multifile screen editor that uses the mouse for all editing tasks and the keyboard only for input of text, including file names and strings such as regular expressions for context search. It is written in two pieces: a *UNIX* program that maintains a copy of the entire file being edited and executes global operations such as context searches on the copy; and a Blit program that does all editing and screen updating. The two programs maintain parallel data structures. The *UNIX* program maintains a complete copy, while the terminal tracks only what is visible on the display. Because the Blit program keeps the visible page locally, screen update can be done entirely inside

running a dynamic *UNIX* system monitor, reporting the current time, average number of *UNIX* processes ready to run, and change in that number in the last minute. The textured bar in the upper portion of the layer adjusts constantly to report the fraction of host CPU time consumed (by all users) in, from left to right, regular user computation, low priority user computation, system overhead, character processing, and idle time. The constantly shifting bars give interesting feedback on the quantity and quality of computation on the host computer. The large obscured layer in the middle is running the *UNIX* shell; the other layers are running down-loaded Blit programs with host support. Note the relationships between the programs: the debugger is examining the editor, but the editor is free to run; the editor and typesetter emulator are asynchronously coupled through the file system; the system monitor runs constantly, and all programs are able to draw on the display at any time, regardless of overlap or user attention.

the terminal; in fact, the *UNIX* program knows nothing about the appearance of the display.

The two programs communicate by a protocol consisting essentially of "insert string" and "delete string" message packets and requests for data, with strings containing arbitrary characters including tabs and newlines. This high-level protocol allows the software to ignore the usual problems of screen update, such as inserting and deleting tab characters and minimizing the length of transmitted strings that update the screen, and makes `jim` efficient in host cycles compared even to line editors. The update algorithm used by the terminal is discussed in Ref. 7. Users want the screen to update quickly, so the protocol is double-buffered for speed and the two programs usually execute asynchronously, with the terminal in control because that permits user input to be handled immediately even with low communications bandwidth.

Unlike most *UNIX* text editors, `jim` has no interactive shell escape to invoke the command interpreter from within the editor, because `mpx` permits the user to create a new layer with a fresh shell at any time. The typical Blit display therefore has a `jim` layer and a shell layer for typing commands such as compilation requests. Conversely, compiler error messages are trivially maintained by the display while a program is being edited.

The `joff` debugger is also controlled mostly by the mouse, although the user interface is substantially different from the user interface of `jim`. The half of `joff` that is a *UNIX* program maintains the large symbol table for the Blit program being debugged, and executes other large-scale tasks such as interpreting C expressions. The code in the terminal displays menus at the user's request, collects typed input, and monitors and probes the target process.

The protocol between these programs falls into two sections: plain text that is displayed in a scrolling region in the debugger's layer, and remote procedure calls that control the debugging, retrieve information about the target process, and build data structures such as menus and breakpoint tables in the `joff` terminal program. The terminal buffers user input such as keyboard characters and mouse button hits, but the host is in control. The menus displayed on a button hit are loaded by the host, and the terminal is not concerned with their contents: all interpretation of user action is done on the *UNIX* system. This structure is significantly simpler than the protocol in `jim`, but results in slower response, which is unimportant in a debugger.

The `joff` debugging program has no direct interface to a text editor (although it displays the text of the source line at a breakpoint), again because the `mpx` environment allows the user to have an editor available at all times.

Both `jim` and `joff` down load about 10K bytes of code to the terminal. The half of `jim` executed on the *UNIX* system is another 20K of VAX-11* code; `joff` is about 70K on the VAX* computer.

V. WHAT DOES IT ALL MEAN?

The Blit application programs, with some noteworthy exceptions, are really not all that interesting. They are fairly ordinary graphics programs, many of them written as playthings by people new to graphics. What is interesting is how the programs work together in the underlying environment. The standard example is compiling a program while editing, with compiler messages appearing in a separate layer without interfering with the editor; but there are more interesting examples.

Our local computing environment contains many minicomputers connected by a local area network, controlled by a cluster of five 24-by-80 terminals, so the person maintaining the network can simultaneously monitor several machines, including those running the network control program. With a Blit, a programmer writing network code can, instead, monitor and debug the distributed processes from a single terminal—and from anywhere there's a Blit, including at home. Similarly, a Blit makes a fine console terminal for a multiprocessor computer.

The graphics capabilities can be used for more than text. Computer-Aided Design (CAD) applications are obvious, although there actually have not been many CAD programs written—certainly fewer than have been asked for. Still, it is valuable to be able to use one's terminal to share graphics and text in separate parts of the screen, for example to edit the textual description of an integrated circuit while inspecting a plot of the circuit in another layer. This extends to looking at separate parts of the same circuit in different layers, or comparing different versions of the same circuit.

These are ordinary uses of multiple window environments, but multiprogramming provides new applications. For example, interactive design programs can be assembled out of existing parts, as is done on the *UNIX* system. The figures in this paper were made with `pic`,[8] and the `pic` source edited with `jim`. There is a program, `proof`, that interprets the typesetter codes generated by `troff` for display in a layer on the Blit. A large layer was initialized running the pipeline

```
watch fig 1. pic|pic|troff|proof
```

where `watch` is a variant of `cat` (the standard *UNIX* program to display a file's contents) that prints the file's contents each time the

* Trademark of Digital Equipment Corporation.

file is modified. Therefore, whenever the `pic` file was written from `jim`, `watch` would notice it had been updated and send the new picture description down the pipeline, without starting a fresh `pic` or `troff` process, for immediate display on the Blit. Syntax errors from `pic` can be redirected to another layer or to a file, which is then watched in another layer. Although this is hardly a real interactive picture-drawing program, it took only a few seconds to assemble and can fill the gap until an interactive program is written.

We discovered an unexpected benefit of asynchronous processes while using `joff`. With the standard system debuggers, the program being debugged is a child process of the debugger, which means, for example, that a program cannot be attacked with the debugger if it was started independently. This is not fundamental to *UNIX*, but rather is a property of the usual terminal environment. The debugger must act as an I/O multiplexer between itself, the user, and the target program. When the terminal does the multiplexing, a debugger can be started at any time and applied to any program, including one that is running—even itself.

A Blit asteroids game had a bug that caused a rock to pass over the spaceship instead of hitting it. The bug was intermittent—perhaps once out of every 100 collisions—so setting a breakpoint was impractical. Instead, `joff` was loaded and applied to an asteroids game, which was then played for about 10 minutes until the bug occurred. Then `joff` was told (by a flick of the wrist and two button clicks) to halt the game. A breakpoint on the collision-testing routine was then set in the asteroids program, and the game resumed. The breakpoint fired and the bug was found easily.

As a second example, consider the following scenario, debugging `joff`. Some changes are made to `joff`, making a new version of `njoff` with bugs. A program with bugs intentionally added, say `Bugs`, is loaded in the Blit as a target for `njoff`. During testing, `njoff` makes a mistake interpreting a data structure in `Bugs`. An instance of `joff` is, therefore, loaded to investigate `njoff` to see where it went wrong, but the correct interpretation of the data structure is unknown, so a second `joff` is called up as a reference source to look at `Bugs`. At this point, there are three debuggers and a target program active on the terminal, but the situation is comfortably under control, although inconceivable in a conventional terminal environment.

There are more mundane uses of the asynchronism. Many of us have mail boxes on remote machines, reachable only through 1200- or even 300-baud phone lines. A mail message could take one minute to print out at 300 baud, but a Blit user need not be idle during that time. The layer with the remote connection will collect the message while the user does something else in another layer, so the user's bandwidth can be much higher. If the phone lines to the remote machine are all busy, the user could type

```
until cu remote-machine
      sleep 600
done
```

to try every ten minutes until the connection is made. The layer with this program will print something like

```
connect failed: line busy
```

every ten minutes. Meanwhile, the user can do anything else on the terminal. Eventually, a line becomes free, the remote machine's login banner pops up, and the user can switch to that layer and log in. No combination of background processes, job control, and static window contexts can achieve this so simply.

VI. MPX: THE HOST PROCESS MULTIPLEXER

The multiplexing is handled by software distributed between the host and terminal. A user-level *UNIX* program, `mpx`, communicates with a small real-time multiprocess operating system, `mpxterm`, running in the terminal (see Fig. 3). The design of `mpx` is sensitive to the details of *UNIX* system Interprocess Communication (IPC) facilities, which vary widely between *UNIX* system versions. `Mpxterm`, on the other hand, is independent of the host except for communication by a simple protocol that it is the job of `mpx` to interpret; all versions of `mpx` speak the same protocol.

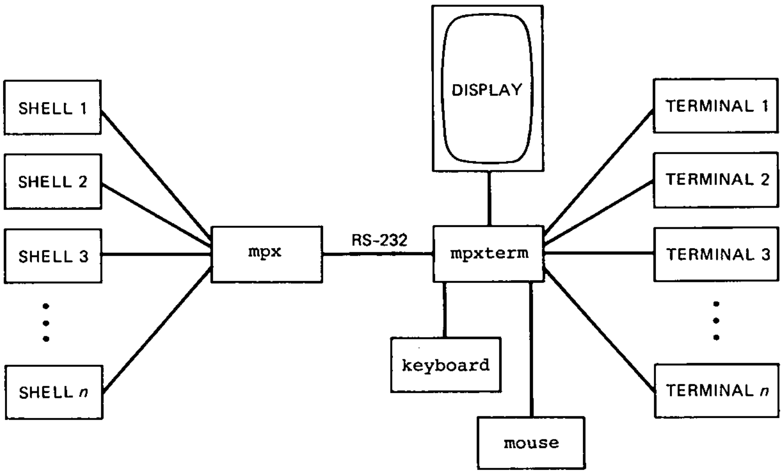


Fig. 3—Overview of `mpx`.

The protocol multiplexes I/O on the single RS-232 cable from the terminal to the host. The multiplexing connects *UNIX* system process groups one-to-one to processes in the terminal. A user on a *UNIX* system with a conventional terminal types instructions to a shell. The shell and the programs it invokes, such as editors and compilers, are members of a single process group, a structure maintained by the kernel. The process group associates processes with a terminal session, mainly to send events such as keyboard interrupts to all processes on the terminal.

The `mpx` program couples each process group to an independent terminal process in the Blit. Four basic capabilities are necessary to implement `mpx`:

1. Dynamic creation and control of several process groups by a single master process (`mpx`)
2. Multiplexing of I/O between the process groups and the master
3. A means to prevent the master from being suspended when it reads data from a process that has no characters available while another has data
4. Ability to distinguish control information (such as setting terminal modes) and data on an interprocess channel.

The original `mpx` was written using Greg Chesson's file multiplexing facilities in the 7th edition *UNIX* system. In *UNIX* System V, the IPC for `mpx` is provided by a kernel driver written by Piers Dick-Lauder. The `mpx` running on the author's machine exploits the user-level IPC in the character I/O system of the 8th edition. Since that version of `mpx` is the closest to hand, it will be described here. It comprises about 1600 lines of code, half of which implement the error-correcting protocol between the host and the terminal. A schematic of the `mpx/mpxterm` pair is in Fig. 3.

Character processing in the 8th edition kernel is done by a sequence of coroutines called line disciplines,[9] each of which is a full-duplex I/O pseudoprocess that performs its portion of the processing and hands the data along to the next line discipline. They are not proper processes because the kernel maintains no call records across scheduling boundaries. They are connected together serially to achieve the desired function, much like a full-duplex shell pipeline. For example, a terminal connected to a user program on our local area network is connected, from the bottom up, to a network driver (essentially half of a line discipline, the other half residing in the network), a line discipline interpreting the network protocol, a standard terminal line discipline that provides services such as character echo and correction of typing mistakes, and another half-discipline to connect to user level.

To connect a terminal, there must be a name in the file system to attach to the associated data structure in the kernel. The directory /

dev/pt contains even-odd pairs of junctor devices, each of which is called a pseudoterminal, or pt. If one process opens an odd-numbered pt file and another opens the corresponding even file, then data written on one file can be read from that file's partner, in symmetrical full-duplex fashion. The odd-numbered member of a pair is the master. Masters and slaves differ only in the rules for opening; I/O is symmetric. Master pt files may be open in at most one process. A process wishing to establish a connection opens an odd-numbered file; then one or more slave processes may open the corresponding even-numbered file and communicate with the master.

Multiplexed I/O is done by a primitive called select. Because I/O can block—if a process reads from a device that has no data available, the process is suspended until data arrive—mpx cannot simply read from the active processes in turn, or it may wait for data from one process while another has data. The select call returns a bit vector indicating which file descriptors have data to be read, or, according to an argument in the call, which file descriptors may be written to without similarly being suspended until the data are read at the other end.

Figure 4 illustrates the interconnection of these components. Following the path from a user process such as a shell, running in a layer, characters enter the kernel and flow through a terminal discipline that does terminal processing for the user process, such as echoing characters typed by the user. The bottom of the terminal discipline connects to the slave side of the pseudoterminal. The characters cross to the master side, where they are passed through a message line discipline out of the kernel to mpx. The message discipline converts all information on the path into data messages, each of which is

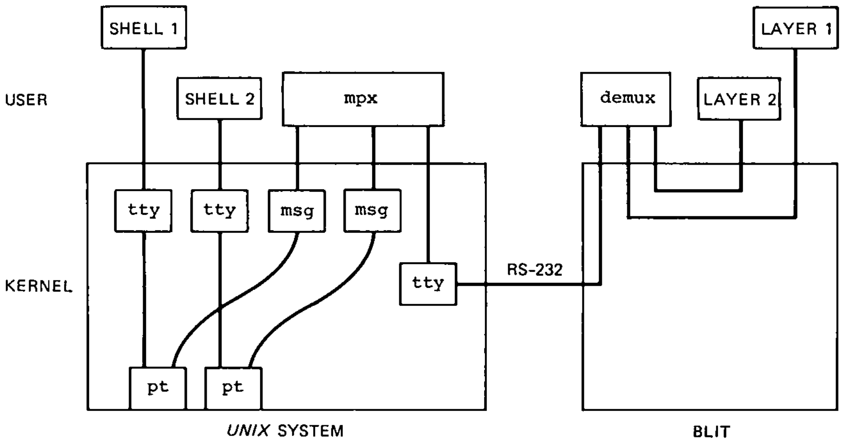


Fig. 4—Interprocess communication in mpx.

prefixed by a header identifying the type of the message. Ordinary characters are tagged DATA, system I/O control requests (`ioctl`) are marked as such, and some other control messages are translated, such as HANGUP, which occurs when the channel shuts down, for example, when the shell exits. These messages are read by `mpx`, which identifies the channel with data using a `select` call. `Mpx` interprets the data, which for ordinary characters merely involves reformatting the message (adding a tag specifying which layer will receive the data and a cyclic redundancy check for error detection and recovery) and sending it down its standard output to the terminal. Data from the process is read from a channel established by `mpx` (see the discussion of layer creation below), while the connection to the Blit is through the standard input and output, because `mpx` is multiplexing its subprocesses onto its terminal, the Blit. On the other hand, the standard input and output of the shell process in a layer are connected to the `mpx` channel for that layer.

On their way from `mpx` to the Blit, the characters enter the kernel again, where they pass through a terminal discipline (the one installed by the login program when the user signed on to the system before running `mpx`; for data transparency this discipline is actually largely disabled) and out to the terminal. In the Blit, the layer identification tag is stripped off, and the data are placed in the input buffer of the terminal process in the appropriate layer. Information flowing in the other direction follows the reverse path.

Although this structure sounds complicated, it is actually fairly clean: the delicate requirements of the interprocess communication are met by connecting together small piece parts with simple interfaces. As a result, the multiplexing does not interfere with other programs, in contrast, for example, with the original `mpx` using multiplexed files, which prohibited running in layers programs that themselves multiplexed. Moreover, because the 8th edition *UNIX* system I/O was written precisely to do this sort of stream processing and interconnection, it is efficient. Perhaps the most brutal test of efficiency is down loading a program into a terminal process: the terminal does almost no processing of the program text, so it is constantly waiting for data from the host. After each 64 bytes of data sent, an acknowledgment packet from the terminal arrives and is processed by `mpx` as part of the communications protocol, so there is frequent scheduling between the down loader and `mpx`. Our *UNIX* system has no assembly language assist for terminal I/O, the hardware generates an interrupt for every character sent or received, and the data from the down loader cross the kernel-user interface twice. Despite this overhead, at 19,200 baud the RS-232 line is almost saturated, delivering over 16,000 user bits per second into the terminal and consuming

70 percent of a VAX-11/750* machine's capability (this implies a maximum of about 400 instructions executed per byte on the VAX system). To our knowledge, no other version of the system on the same hardware can deliver down-loaded programs faster than about 6000 baud.

When the user on the Blit asks to create a new layer, the following events occur. The terminal allocates a layer data structure on the display and creates a terminal process to manage it. It then sends a message on its RS-232 connection, the standard input of `mpx`, stating that a layer has been created and specifying the channel in the communications protocol onto which its data will be multiplexed. Then `mpx` opens an idle master `pt` file, and the channel number (different from the communications channel) returned by the open is the connection of `mpx` to the subprocess about to be created. `Mpx` pushes a message line discipline onto the stream on the master side of the pseudoterminal and forks to create a child process. The child closes all of its file descriptors and opens the slave side of the pseudoteletype, which becomes its standard input and is duplicated to form its standard output and standard error output. It then pushes a terminal line discipline onto the stream and initializes the terminal modes. Finally, it establishes itself as a separate process group and executes a shell. When the shell begins, it prints a prompt on its standard output, which flows through the path outlined above and eventually arrives in the input buffer of the terminal process, which copies it to the display, and the act of creation is complete. The elapsed time is perhaps a half second.

VII. MPXTERM: THE TERMINAL OPERATING SYSTEM

Inside the Blit runs a tiny operating system that provides essentially the same multiprogramming and data transparency as `mpx`. It is basically a mirror image of `mpx`, but with considerably less mechanism, largely because the multiplexing is built into the operating system rather than being constructed at user level. The basic structure of the system is a set of independent processes scheduled round robin that call a primitive queue-based kernel to service I/O requests.

At the time of writing, `mpxterm` is 1627 lines of C, excluding code for the protocol (which uses the same source files as `mpx`) and the graphics primitives, but including all the user interaction and I/O primitives; and 204 lines of assembler. The assembler lines include 11 lines to switch stacks, 108 lines to interface interrupt routines to C code, and 85 repetitive lines to interface to C code after a process traps.

* Trademark of Digital Equipment Corporation.

Process switching is performed only at the process's request; there is no preemptive scheduling. Since the Blit is a terminal, and not a general-purpose computer, the processes all do some form of input or output, whether to read characters from the host or keyboard, or even just display something on the screen. If a process wants a character from, say, the keyboard, but none has been typed, it can suspend itself by executing

```
wait(KBD)
```

which says "wait until a keyboard character becomes available." Because the display is updated at 30 Hz, a display program will usually suspend execution until the screen reflects the change it has made in memory. Therefore, although the programmer must be aware that the CPU is being shared among other processes, the habit of relinquishing the processor fits smoothly into the discipline required for real-time graphics programming. This structure keeps `mpxterm` simple (and easy to debug). Except for the lowest level of I/O, which must protect against device interrupts, there are no semaphores or interlocks in the kernel; the process control part of `mpxterm` was written and debugged in an evening.

The devices—mouse, keyboard, and host RS-232 port—are all interrupt driven. The keyboard and RS-232 port place their characters into queues that are read by server processes running at user level (i.e., with processor interrupts enabled). The mouse buttons generate an interrupt when their state changes, and their value is kept in a global data structure, along with the mouse position. As the mouse moves, the hardware updates two registers in the I/O page but generates no interrupts. Instead, the mouse position on the screen is updated during vertical blanking by a low-priority interrupt routine that runs off a 60-Hz clock coupled to the start of vertical retrace. Because of the 30-Hz display refresh, there is no reason to update it more frequently.

The clock interrupt and mouse button interrupt schedule a control process that multiplexes the mouse among the user processes. At any time, only one user process receives mouse tracking and button hit information from the control process. Any other process attempting to use the mouse is suspended until the user indicates by a button hit, handled by the control process, that the mouse and keyboard should be bound to that process instead.

A second system process, the demultiplexer, reads the characters from the host input queue, unpacks the messages, and executes the error-correcting protocol. Correctly received messages are placed in the input queue of the associated user processes. The error correction is transparent to the processes; as far as they can tell, they have a direct link to a plain RS-232 wire, except that no flow control is necessary on either end (compare this to the control-S/control-Q or NUL-padding flow control necessary with many standard terminals). The demultiplexer occasionally receives control messages, indicating, for example, that a terminal process is to begin executing the download receiving procedure preparatory to loading a new terminal program into a layer.

All resources are shared among the processes in the Blit. Memory allocation occurs through two primitives: `alloc` allocates memory at fixed locations, to store programs, for example; and `gcalloc` allocates relocatable memory in a compacted arena, to store bitmaps and strings. This split structure is imposed by the open addressing of C and the necessity to compact the arena containing dynamically allocated bitmaps. User processes and the kernel allocate using the same code, and each allocated object is tagged with a pointer to the process that owns it, so storage can be reclaimed when a program exits. Storage allocation is simplified by the lack of preemptive scheduling; interlocks during compaction are unnecessary, since allocations are atomic.

Because the hardware does not provide memory management and our C compiler does not generate position-independent code, downloaded programs are relocated in the host to an address returned by `alloc` in the Blit. Relocation is not expensive; the text editor, which is about 10K bytes long, is relocated in three seconds and down loads in about six seconds at 19,200 baud. This is comparable to the initialization time of most conventional screen editors.

The Blit hardware provides one feature for protection. Read or write references to the first eight bytes of the processor's address space generate an interrupt that is caught by the kernel, which halts the offending process. Because a common C programming error is to dereference through a null-valued pointer, this small feature has saved `mpxterm` many times.

For an unprotected system, `mpxterm` is pleasantly robust. It is certainly shut down quietly at the end of a working day far more often than it crashes. Left running, its mean up time is several days, even during periods of program development.

VIII. PROGRAMMING

Processes in the terminal may be loaded, by a procedure analogous to executing a *UNIX* program, to customize the terminal for a particular task. The programmer's interface to `mpxterm` is unaffected by other programs running in the terminal. To a rough approximation, the programming environment is a virtual machine: programs run as though they have a keyboard, mouse, display, and host RS-232 connection all to themselves.

The screen is multiplexed using the idea of a layer,[3] which supports all bitmap operations, especially bitblt, on an extended bitmap data structure that allows overlap. Each Blit process has a global variable called display, which is the layer data structure for the portion of the screen occupied by the process. The display data structure contains the coordinates of the screen rectangle, used to clip graphics operations, and a list of off-screen bitmaps containing obscured contents of the layer. To the programmer, display is like an ordinary bitmap, obscured or not, and by executing graphics primitives on display the process can draw on its screen regardless of overlap, and without communicating with a window manager when the layer configuration changes. As far as the process is concerned, it has its portion of the screen to itself. There is no "window manager" in the conventional sense—bitblt* is the window interface.

Characters arriving from the host are split by the demultiplexer into separate streams and placed in the input queues of the appropriate processes. From a process's point of view, the interface to the host is an ordinary byte stream. The keyboard is handled differently, because the stream of typed characters is directed at a process by the user. Still, the idea is the same: each process sees an ordinary byte stream from the keyboard and is oblivious to characters directed to other processes.

Character I/O in mpxterm is nonblocking. Two routines, kbdchar and hostchar, read characters from the input queues for the process. If no characters are available, they return an error indication but do not block, because typical terminal applications must be ready to receive input from either the host or the keyboard. When a process wants to suspend until characters become available, it calls wait with an argument bit vector stating which resources are of interest. Wait returns a bit vector indicating which queues have data, so the inner loop of a typical terminal program is something like this:

```
int resource;
  while(TRUE) {
          resource = wait(HOST|KBD);
          if(resource & HOST)
                  draw_on_screen(hostchar());
          if(resource & KBD)
                  sendchar(kbdchar());
  }
```

* The lbitblt primitive, discussed in the layers paper,[3] is aliased to bitblt in the mpxterm programming environment, so the distinction between bitmaps and layers vanishes—the programmer treats layers exactly like bitmaps.

`sendchar` sends characters to the host through the error-corrected channel. `Wait` suspends the process, by calling another process that is ready to run, until a character becomes available on either queue and no other process is using the CPU. If no other process is ready, `wait` returns immediately when a character becomes available.

Another system call, `sleep`, suspends a process for a specified number of ticks of the 60-Hz clock, by waiting for a timer set by a nonblocking `ALARM` resource. `Sleep` is roughly:

```
sleep(n)
        int n;
{
        alarm(n);    /* set the timer n ticks in the future */
        wait(ALARM);/* suspend until timer fires */
}
```

but includes protection in case the process has alarms pending. Since the hardware clock is coupled to the vertical retrace, `sleep` is often used to suspend a process until the picture it has placed in memory is visible on the screen.

Each process has a global data structure describing the mouse state—position and button status—that is updated asynchronously whenever the user has assigned the mouse to that process. A process may wait until it owns the mouse by calling

```
wait(MOUSE)
```

Therefore, to wait for a button to be depressed, a process would execute

```
while(mouse.buttons == 0)
        wait(MOUSE);
```

The following code draws line segments connecting mouse positions as the mouse moves:

```
Point p;
p = mouse.xy;     /* first point, where mouse points now */
for (;;) {
        q = mouse.xy;
        segment(&display, p, q, OR);
        p = q;
        sleep(1);    /* wait for mouse and display update */
}
```

The notation `&display` indicates that the address, rather than the value, of the display bitmap structure is passed to `segment`. `OR` specifies that the bit pattern of the line is to be OR'ed into display memory. Line segments are drawn half-open, so adjacent line segments share no points.

As well as I/O, all graphics primitives are implemented as system calls, to interface to the layer code but make everything look like ordinary bitmap graphics. Therefore, the system call interface must be very fast, or system call overhead will dominate graphics performance. Because there is no memory management, processes all live in the same address space, and system calls are indirect subroutine calls through a vector at a known location. The execution penalty is only one extra instruction for a system call compared to an ordinary procedure call. The mapping to the vector is done from C by defining the system calls in a header file, so the mechanism is transparent to the programmer.

Programs are loaded into the Blit from the host computer's disc by a user program that communicates with a special program load process in the terminal. By default, a layer runs a conventional "dumb" terminal emulator. When the *UNIX* program executes a bootstrap `ioctl` request to initiate program loading, `mpx` transmits the request on a reserved communications channel. The Blit demultiplexer process shuts down the terminal emulator and begins the program loader process, which allocates memory, returns to the system the base address of the program, and then copies (asynchronously with the other terminal processes) the relocated program from its `HOST` queue into memory. Since the channel is error corrected, the loading protocol just relocates the program and writes, unformatted, the relocated binary; no checksumming or verification is necessary. When the loading is complete, the program begins executing. If it executes the `exit` system call, the layer remains active but is reinitialized with the dumb terminal emulator.

IX. RETROSPECTION, INTROSPECTION, AND CONCLUSIONS

The Blit has taught us that multiprogramming has been underused. A user is capable of running several related or unrelated programs in parallel if the user interface makes it easy to control their execution. The Blit has also shown the advantages of isolating the issues of user interaction from the operating system. All of the Blit software is user-level code, yet the Blit environment feels naturally coupled to the *UNIX* system. The system really knows nothing about the multiplexing going on; the user is just running more processes than usual. A large part of the Blit's success can probably be attributed to our concentration on the graphics and user interface issues, rather than the development of a new integrated, distributed programming environment. There are a number of things worth noting that were done well on the Blit, and a number that could be improved. To end on an upbeat note, we will discuss the mistakes first.

Although the graphics is fast enough, the hardware is not big enough. That is, memory is tight when working on big programs, and there isn't enough offscreen bitmap storage. The greatest problem, though, is certainly the low bandwidth. Putting aside the issues of availability, simplicity, and portability, RS-232 is not fast enough for file I/O. The text editor must be written in two parts, using the terminal much like a cache. Consider context searches at 1200 baud, which would otherwise require sending the entire file, perhaps hundreds of thousands of characters long, over the phone line. Unfortunately, writing one program in two pieces is much harder than writing two programs. Still, we don't want local disc. The Blit model, using an inexpensive dedicated front-end for high-quality interaction on a traditional time-sharing system, is a powerful one, and we prefer increasing the memory and bandwidth, leaving the basic structure the same, to adding disc and therefore expense, noise, and the proliferation of local copies of software.

`Mpxterm` does not exploit multiprogramming enough itself. Layers and terminal processes are one-to-one, counter to the current fads of message-based systems. There certainly needs to be more terminal IPC so, for example, text in one layer may be copied to another using the `jim` cut and paste operators.

Perhaps most importantly, the current Blit software is tending towards disintegration: *this* layer is an editor and *this* layer is a debugger and *this* layer is a circuit design program. This trend is counter to the uniformity of environments that makes a system easy to use, and misses some obvious simplifications. One obvious change would be to push text editing to a lower level, so text anywhere on the screen, not just in a `jim` layer, could be edited with the mouse. `Mpxterm` is currently being rewritten to support editing of displayed text.

Some things were done well. One of the Blit's competitive advantages was that the two people (Locanthi and Pike) who designed the hardware and software were the people who most wanted to use it. Both understood the hardware and software issues, and the hardware and software were designed together to work together, rather than by competing committees. Particularly in the design of the graphics memory, iterations of the hardware design were punctuated by writing test software to develop a feeling for the hardware/software trade-offs, and where best to resolve them. Finally, the bulk of the software was written by the same two people, and `mpx` and `mpxterm` were written by one (Pike).

Simplicity rules the Blit software. The operating system has no memory management and the simplest process structure possible. The user interface is devoid of the usual frills and bunting that decorate most graphics environments. For example, there is only one type of menu—a list of strings. Many menu styles can be envisioned, and they would certainly be used if implemented, but only one is necessary. The Blit graphics library is about 8K bytes of compiled code, of which over 3K is `bitblt`, `texture`, and the line-drawing primitives. This is a small fraction of the size of most interactive graphics systems.

The Blit is inexpensive. For little more than the cost of replacing the 24-by-80 terminals, everyone in our research center, including the support staff, has a Blit, and several have two. Also, replacing terminals is a simple way to migrate to a new environment. The system underneath is still the same *UNIX* system, in fact—so nothing was left behind, and only new things had to be implemented.

From the user's point of view, the Blit has brought about a far-reaching change in attitude: in conventional environments, even on sophisticated time-sharing systems, the user must often wait for the machine to complete some task such as a compilation. On the Blit, the machine is always ready to do something new—the user is in control, not the machine.

X. ACKNOWLEDGMENTS

Many people helped and influenced the development of the Blit. Most important among them is Bart Locanthi, who designed and built the terminal and much of the underlying graphics software. Piers Dick-Lauder wrote the error-correcting protocol in `mpx` and wrote the eighth edition version of `mpx` itself. Thanks are also due to Sally Browning, Tom Cargill, Greg Chesson, Joe Condon, Dave Ditzel, Steve Johnson, Andrew Hume, John Reiser, and Dennis Ritchie, each of whom provided indispensable assistance and enthusiastic encouragement.

REFERENCES

1. D. H. Ingalls, "The Smalltalk Graphics Kernel," Byte, *6* (August 1981), pp. 168–94.
2. R. Pike, B. N. Locanthi, and J. F. Reiser, "Hardware-Software Tradeoffs for Bitmap Graphics on the Blit," Software—Practice & Experience, *15* (March 1985).
3. R. Pike, "Graphics in Overlapping Bitmap Layers," Trans. Graph., *2*, No. 2 (1983), pp. 135–60.
4. C. P. Thacker et al., "Alto: A Personal Computer," CSL-79-11, August 1979, Xerox Corp.
5. W. N. Joy, R. S. Fabry, and K. Sklower, *UNIX* 4.1BSD Programmer's Manual.
6. T. A. Cargill, "The Blit Debugger," J. Systems and Software, *3*, No. 4 (December 1983), pp. 277–84.
7. R. Pike, unpublished work.
8. B. W. Kernighan, "Pic: A Language for Typesetting Graphics," Software—Practice & Experience, *12* (January 1982), pp. 1–20.
9. D. M. Ritchie, "The *UNIX* System: The Evolution of the *UNIX* Time-sharing System," AT&T Bell Lab. Tech. J., this issue.

AUTHOR

Rob Pike, AT&T Bell Laboratories, 1980—. As a Member of Technical Staff Mr. Pike's best-known work has been as co-developer of the Blit bitmap graphics terminal. His research interests include statistical mechanics and cosmology; his practical interests involve interactive graphics hardware and software.