To: Distribution

From: Bernie Greenberg and Jim Davis

Date: 10/02/80

Subject: A Window Video System Implementation

1. INTRODUCTION

This MTB proposes an implementation of a window video system on Multics. A previous MTB (458) described the purpose and general nature of a window video system, and must be read and mastered first. This MTB shows how such a thing can be done.

Unfortunately, there are still some problems to be solved. A subsequent MTB (462) proposes an interim system, sufficient for the needs of the menu system, which also sidesteps the unsolved problems.

Acknowledgement:

We would like to thank Olin Sibert and Lyman Hazelton for contributing to many of the ideas expressed herein. We also owe quite a bit to the ITS and Lisp Machine output systems, and the ideas contained within CRTSTY on ITS.

Send comments to:

the System M continuum >udd>m>jrd>mtgs>tv

or by Multics mail, on MIT or System M, to
Greenberg.Multics or JRDavis.Multics

or by phone
Greenberg: (617)-492-9330 HVN 261-9330
Davis: (617)-492-9382 HVN 261-9382

2. OVERVIEW OF THE PROPOSAL

The essense of the proposal is to implement windows by an upward compatible extension of iox_ I/O switches. Each window will correspond to a switch. Normal iox_ stream I/O will work as it does on a terminal, and additional features will be accessed by a new subroutine similar to iox_.

Once the switches are set up (by a subsystem or by initialization of the process environment) for the required windows, all that need be done is to route different outputs through different io switches - the outputs will appear in separate windows. After the initial setup, all that is required is a call to ioa_$switch. Routing various outputs through different switches is a small modification to subsystems which buys back great utility in a video-managed environment. On printing terminals, "syn"ing all these switches to user_output as today retains today's functionality with no loss of generality.

The Window system will be divided into two levels, each implemented by an iox_ module. The first level, Window Management (WM) makes terminal - independent calls to the lower level to implement a single window for a user application program. Each attachment of WM maintains a single window. Attachments of WM are ignorant of other attachments, if any. The second level, Terminal Control (TC) contains all terminal-specific support, and is called only by WM.

Normal user_output will be routed through a WM attachment. Subsystems which can use a video screen to advantage, as opposed to those that manage one as their primary function, can use WM attachments to keep various forms of output visible simultaneously. Ordinary ioa_ calls (ioa_$switch) will suffice to output to windows in this way.

Applications can manage their screens by calls to WM. Typical calls might create a window, position the cursor to some location in the window, and output some characters in the window.

To conform to practical limitations, we propose to limit windows to non-overlapping, non-nested windows.

3. BASIC WINDOW MANAGEMENT

A program that wishes to use a window obtains one via iox_ attachment. Just as for any other use of iox_, a new IOCB is created and returned.

The usual iox_ operations are also defined for windows. Characters may be output by iox_$put_chars, and (after output conversion) appear on the screen, replacing whatever used to be there. The window system will be controlled by iox_$modes and iox_$control calls. A call to iox_$get_line returns a line (but see Input Line Editting below).

3.1. Line Wrapping

Overlength lines will be wrapped, just as today. As much of the line as will fit on the current terminal line is printed, with the remainder of the line placed on the next line, preceeded by the character sequence "\c". There has been no demand for any other sort of processing, although plausible options include a user-settable "wrap character"; placing the wrap character at the end of the line, instead of the begining of the new; and truncating lines, rather than wrapping them. There is no particular problem in adding these, if need ever arises.

At all times, the window virtual cursor should be at the row and column where the next character will go. Printing a character in column N also moves the cursor to column N+1. (Some characters occupy multiple columns, since they print as escape sequences, and others may occupy no column, but that doesn't matter here.) But where should the cursor go when a character is printed in the last column? (assume 80, for this discussion.) The next character will be printed on the next line, so perhaps an implicit CR-LF should be done? But this is wrong, since if the user ends an 80 character string with a NEWLINE (as is proper) this will cause an extra blank line. Instead, we propose that when a character is printed in the last column (say, 80), the window cursor advances to a "phantom" column (81). If no cursor motion commands are received, line wrapping will be done when the next character is received. This allows the use of the full width of the window.

The phantom column will be treated like any other column for the purposes of cursor motion, and the user who calls the window system to ask for the position of the cursor must be prepared to be told it is in column 81.

This scheme is a little tricky to implement on todays terminals, because they do not behave uniformly when a character is printed in the last column. In all other columns, printing a character causes the cursor to move to the right, into the column where the next character will go. When a character is printed in the last column, some terminals put the cursor in a "phantom" column (the VIP 7801 and the VIP 7200). The cursor may be moved at this point (usually to the next line, by a CR-LF from the host), but if it is not, an implicit CR-LF is done when the next printing character is recieved. Other terminals (DELTA DATA 4000) do the implicit CR-LF when the 80th column is used.

Emacs solves this problem by using windows that are 79 characters wide (on an 80 character terminal), so that there will always be a "next column" that will be safe to move into. I will solve it by adding information about end of line behavior to the TTF. Before this can be made specific, additional terminals must be studied. The scheme we propose will work fine on the two VIPs mentioned above, and other terminals will work if used with a shorter line length.

3.2. Character Conversion

Output conversion will be done in accordance with the terminal type file conversion tables, just as today. These conversion tables must be expanded to cover the full range of the Multics (9-bit) Character Set.

There are some changes to the conversion tables that are appropriate for video devices, for the ASCII format effectors. Some, such as SPACE, TAB, and LF have obvious translations into cursor movement. WM will scan output for successive use of these characters and (where possible) output a single cursor positioning command. The other format effectors (BS, VT, FF, and CR) will be displayed as escape sequences.

3.3. MORE Processing

In our proposal, The user will strike SPACE to continue output, or strike RETURN to abort output. Any other character will cause the terminal bell to ring.

If the response to MORE is negative, program_interrupt will be signalled. Multics commands which handle program_interrupt (subsystems) usually do so by resuming their input loop. The Multics command processor must be changed to handle the condition for the sake of those commands which do not handle pi now - this should be equivalent to the command_abort condition. The user has indicated that the remaining output of the command is uninteresting, so the command should return.

More processing will be determined by the "more" mode, which is on by default, except on printing terminals.

Further control of more processing might include user settable prompt string for the MORE processor and user settable characters for the two responses. A subsystem might want to handle more processing itself. This could be done by returning an error code from the iox_$put_chars call or signalling a condition, or perhaps by calling a user supplied entry point. We make no proposal to supply this now.

3.4. End of Window Processing

We will support three modes of operation for end of window processing. The user or application selects the desired option by an iox_ mode call. We define a new mode, called "more_mode", which can take on one of three values, "scroll", "wrap", or "clear". The notion of an iox mode taking on values is not new, since the "ll" mode does that now, as does the audit_trigger mode. We will use the syntax convention established by the audit_ dim, and write "more_mode=wrap" in the mode string. A subroutine has been designed to parse mode strings in this syntax, and will be useful to other io modules, as well as any command that has a mode string argument.

The default mode will be more_mode=scroll for terminals with insert-delete lines, and more_mode=wrap for all others.

An attempt to select scroll mode for a terminal without insert-delete lines will be in error. (The TTF tells whether a terminal supports insert-delete lines, and a control order is provided to allow the user to enquire.)

3.5. Handling QUIT

MORE processing offers a new way to discard output of programs, and will serve one of the purposes of QUIT. But programs must still be interrupted, so QUIT is still needed. The behavior of QUIT on Multics causes several problems today. The standard Multics handler for quit eventually passes iox_ a resetwrite order, which in turn is passed into ring 0, and out to the FNP.

The first problem is that the program executing when a QUIT (ips) signal occurs may not be the one that produced the output that prompted the user to strike QUIT. Nevertheless the resetwrite discards all pending output, even that of the "innocent" program. There is no concept in the io system of what "program" produced which "output" (except for use of dedicated switches, now possible via the window system). The worst case of this now is when message output is lost.

A second problem is that when a "resetwrite" is passed to the FNP it stops its output immediately. It is possible to stop output while in the middle of a multiple character terminal control sequence. The resulting terminal state is undefined. Users of the Honeywell VIP 7801 may have noticed that the terminal sometimes beeps, and flashes "INVALID COMMAND", refusing to do anything until reset. Other terminals have similar problems. Furthermore, there is no way for ring 0 or ring 4 to know just how many characters were actually sent to the terminal, so the screen contents and cursor location become undefined as well.

A third problem with resetwrites is that occasionally the resetwrite also resets characters written after the reset order itself. Emacs users who have hit QUIT may have noticed that the screen is not always cleared before entering a new command level. This is because the screen-clearing character is discarded. This problem is due to an MCS design problem.

The grave lack of MCS resources makes it appropriate to fix these problems in ring four.

Until the FNP can be made aware of cursor position and the indivisibility of certain character sequences, it is imperative that resetwrite orders not be sent to ring 0. This means that there is no way for the user to stop output that is already "on route" to the terminal. Fortunately, WM will never send more than one window full of output at a given time.

Some printing terminals also have output escape sequences. Normally, printing terminals are run without MORE processing. We can choose to send the resetwrite (thus possibly interrupting a multi-character escape sequence), or not send it (and all pending output in ring 0 will still be sent. We don't know which choice is better (per terminal).

3.6. Control of Asynchronous Output

A problem with the current output system is that output can appear while the user is blocked for output. For example, during the printing of a lengthy segment, the user may go blocked for output. An IPC wakeup can arrive at this point (e.g. a message) and the output will appear in the middle of the segment being printed.

A second kind of asynchronous event is the IPS signal. This includes QUIT handling (printing "QUIT") and "alrm" and "cput" timers. Output from IPS programs (e.g. memo, blip) is harder to handle because it can come at any time, not just when the process is blocked. The window system can be interrupted at any time, possibly with the window data in an inconsistent state. IPS interrupts could be masked, but that might impose performance problems.

We do not know how to handle asynchronous output at this time. A particularly hard problem is what to do when output arrives during MORE processing. We will try avoiding this by masking ipc_ during MORE waits.

3.7. User Input

Programs can get input from the window system in a variety of ways. The simplest, and most common, is to call iox_$get_line. This will invoke the line editor (see below).

A call to iox_$get_chars will always return exactly one character. Since the system operates in breakall mode, there is no need to wait for a newline. Characters returned are not echoed.

An important feature of the get_chars call is that it never returns more than one character. This is essential to allow type-ahead to work. Otherwise, if the the user were typing

ahead, it would be possible for the caller to inadvertantly "gobble" input intended for some other program. As it is, type-ahead will not be echoed ahead. Characters will be echoed when they are read. The alternative is to echo characters "blindly", and thus occasionally in the wrong place or in the wrong way. We prefer the conservative approach, and most operating systems providing video support make the same choice.

At some point we might consider a scheme allowing "wrong" echoing to be "rolled back" upon notification from ring four that the "naive" echo-ahead was incorrect.

3.8. Input Line Editing

WM will provide line-editing input for all callers of iox_$get_line. WM will get characters from TC, wrap or scroll or "\c" as lines pass the right margin, and so forth. WM will perform all erase and kill processing in real time, via echo-negotiation. When the user strikes newline, the line is returned to the caller.

The current echo-negotiation interface allows for simple # (rubout) processing to be done by ring zero or the FNP- although not currently implemented, this again can be done as resources permit.

The BS character will echo as an escape sequence. The editor will canonicalize the line before returning it.

The line editor will keep track of the line being edited. The input line may extend across several screen lines. If the cursor is at the left edge of a screen line, and the erase character is struck, the cursor must move to the right edge of the preceeding line.

There are some unsolved problems with the editor:

It is likely that users may want to supply their own editors. This should be possible.

It isn't clear what to do if the input line can't fit on the whole window.

As mentioned above, the editor uses echo negotiation. Users may want to use echo negotiation as well, for example, to do command completion (where you need only type enough of the name of a command to make it unique, strike some key, and the system completes the name, or tells you that the name wasn't unique). It should be possible to save and restore the table of break

characters for echo negotiated input.

4. ADVANCED WINDOW USAGE

All the feature thus far described are available to all callers of the window system, even if they are unaware of calling it. But for those programs that will benefit from a video interface, we provide facilities for manipulating the screen itself. A window is a superset of a sequential input/output device, and additional operations are provided for windows that do not correspond to current iox_ calls. These operations work on the virtual screen provided by each window.

We propose an interface, very much like that of iox_. An array of entry variables (like iox_ entry variables) will be accesible from the attach data of the IOCB for the window attachment. Also like iox_, these entries will be callable through a transfer vector, which we call dctl_. The array of entry variables is called a Video Control Block (vcb).

These calls could be added to iox_ itself, but since they apply only to video terminals we feel it is more appropriate to access them through a different name. It would also be possible to access these functions by iox_ control orders, but this would require the programmer to allocate and fill structures for each call. This would be far less convenient than the subroutine interface, and less efficient as well.

We don't give all the subroutine interfaces here, only enough to give the "flavor" of the interface. All the subroutines have a first argument which is the IOCB for the window switch, and a last argument which is a standard system error code.

```
dctl_$position_cursor (iocbp, xpos, ypos, code);
dctl_$home (iocbp, code);
dctl_$clear_to_end_of_line (iocbp, code);
dctl_$clear_window (iocbp, code);
dctl_$clear_to_end_of_window (iocbp, code);
dctl_$scroll_window (iocbp, scroll_count, code);
dctl_$output_raw_chars (iocbp, "string", code);
```

The next MTB gives full and specific details for a set of window operations.

Each attachment of WM maintains its own 'logical' cursor within its window. WM can be called explicitly to move the cursor, and the cursor also moves implicitly as characters are

typed, deleted, etc. WM in turn calls TC to position the terminal cursor to some new absolute position. WM knows the position of its window on the screen, and translates cursor coordinates from the relative (the virtual screen it provides) to the absolute. The lowest level (TC) will relocate the cursor by the fastest motion - this is quite terminal dependent.

The logical window cursor may not be where the physical terminal cursor is, since the most recent I/O may have been in some other window. WM must be aware of this possibility. Suppose an application positions the cursor in one window, then does some I/O in a different window, the some output in the first window. WM must cause the cursor to come back to where it thinks it should be. The alternative is to have the caller of WM keep track of the window containing the cursor, but this is an undue burden, is not modular (and may be impossible for the user to fulfill). It is easy for WM to do this. All attachments of WM will share a common static pointer to the last IOCB that window i/o was done on. If the IOCB is not the same as the current IOCB, then an absolute position is done before further operations.

5. TERMINAL CONTROL

TC deals with issues of padding, whitespace optimization, and choice of optimal cursor movement. TC performs blocking on behalf of ring 0 buffer management. TC also has the responsibility for interface with Echo negotiation, and ultimately, with that help, has the responsibility for producing terminal input and echoing it.

WM calls TC as if it were another WM. Unlike WM, TC will output terminal-specific escape sequences, not dctl_ calls.

The tty_ dim will implement TC. Ring zero tty modules will be run in raw mode, as all conversion will already have been done by WM.

Like WM, TC will have an array of entry points implementing terminal control operations. For most reasonable terminals, these entry points will be into tty_, which will interpret the TTF, and output the appropriate escape sequences. Each "unreasonable" terminal (such as the DELTA DATA 4000, or the ADDS 980) will require a special module (a Terminal Control Module, or TCM), analogous to an Emacs CTL.

The Multics SUPDUP USER will interpret SUPDUP TD codes recieved from a foreign video program, and call TC to perform screen manipulations on a Multics terminal.

Emacs itself will call TC, thus freeing Emacs users from writing Lisp-coded CTLs. A more ambitious WM is required if Emacs is to call WM.

TC maintains knowledge of the physical terminal's cursor position, which can be read via the dctl $read_cursorpos call. TC performs relative and absolute cursor motion via the two calls dctl_$position_cursor and dctl_$position_cursor_rel. TC determines based on its own criteria of optimality and its knowledge of the cursor position whether the terminal's absolute or relative positioning should be used, or tabs, carriage returns, or any or what combination of some or all of these should be used.

The TC level is intended, we reiterate, to be called only by totally video-intensive programs (real-time video editors, user SUPDUP, etc.), and the WM level. It does not fit well within the functionality of iox_; like the 3270 dim, it supports neither get_line, or any PL/I-file-like operations; its model is a virtual video terminal, as opposed to a PL/I file, and the two have little in common: iox_ serves only to manage instantiations of terminal and I/O-controlling entities.

5.1. Buffered Output

Under the normal mode of operation, each iox_ or dctl operation will be sent out through ring 0 as calls are made. This may have performance implications. Since in many cases the caller of TC will make several calls in sucession, with only the ultimate window state of interest, it is worthwhile to be able to operate TC in a buffered mode, where requests are stored until a write order is received, or the buffer fills, or some other event occurs which makes it necessary to dump the buffer. Buffered mode gives permission to TC to buffer output, but does not impose an obligation on WM to do so.

There will be order calls to turn buffered mode on and off, and a dctl_ call to write the buffer.

Buffering will be done by TC. Each attachment of WM will cause the TC buffer to be flushed if it may contain characters buffered on behalf of some other WM attachment.

6. DEVICE INDEPENDENCE

All device independent sequences are stored in the System Terminal Type File, and interpreted by TC. WM must have some information about the generic capabilities of the terminal, although not about how the capabilities are invoked.

WM must know if the terminal is able to overprint, to scroll a window, to erase specific positions. It must know how the terminal acts in the last column.

Some terminals will be so complex that they cannot be described by the video info in the TTF. We expect to support these terminals with special io modules (analagous to CTLs used by Emacs), but haven't defined these further.

7. PRINTING TERMINALS

7.1. Line Editing on a Printing Terminal

Printing terminals differ from video terminals in that (among other things) they can usually overprint, but can't delete characters. This means that the line editor will not be able to remove deleted characters from the line. Instead, a deleted character will be overstruck with the erase character. The cursor will remain over the deleted character. If further erase characters are struck, then more characters will be overstruck. If a printing character is hit, the terminal will linefeed, and characters will be echoed under the character they replace. This is similar to Emacs use on a printing terminal now.

A kill character will echo as @, followed by a newline.

BS will echo as cursor motion.

8. QUESTIONS

How do you use audit_ with the window system?

How does this fit with the ANSI X3.64 standard for video control?

How can control and escape sequences be reliably sent and received over networks?