To: Distribution

From: Jim Davis

Date: 01/27/81

Subject: Interim Window System for the Menu Manager - Revised

This document is a re-issue of MTB 462, incorporating changes from review. It is being issued to document the video system, not for purposes of review, although comments are still welcome.

The reader should be familiar with MTB-458 ("Towards a Windowed Video System"), MTB-461 ("A Window Video System Implementation"), with the current input conventions (described in the MPM Communications I/O Manual CC 92), and with the Emacs editor (Emacs Text Editor User's Guide CH27).

send comments to

By continuum (System M):
>udd>m>jrd>mtgs>tv

By the Multics Extended Mail Facility
JRDavis.Multics @ MIT-Multics

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

Multics Project internal working documentation. Not to be distributed outside the Multics Project.

1. OVERVIEW

The Multics Video system is a upwards compatible extension to the I/O system. It provides the user with multiple windows (virtual video terminals), each associated with an I/O switch.

The Video System is divided into two layers, Terminal Control (TC) and Window Management (WM). There is one attachment of TC per terminal attached to the user's process, and one attachment of WM for each window.

TC interprets the video attributes of the TTF for the terminal in use, and sends terminal dependent control sequences to the device. TC does all ring zero I/O, and performs blocking for the user. TC also supports reconnection.

TC is implemented by an I/O module vtty_. Ideally TC should be implemented by tty_, but it is hard to switch tty_'s in mid stream, and users should not be exposed to the intermim window system without asking for it. It is not appropriate to install a new tty_ (with possible bugs), and not necessary. Terminal Control does need to call ring 0 hcs_ tty entries for echo negotiation, so vtty_ will obtain the terminal device index from the ring four tty_ dim.

WM implements a virtual terminal by making calls on the virtual terminal implemented by TC. WM implements output conversion, MORE processing, and the input line editor. WM is implemented by an I/O module named crt_.

Like all I/O modules, these are not called directly by the user. Rather, the user calls iox_ (or ioa_, or language i/o), which in turn calls the appropriate entry in the appropriate I/O module. In addition to the normal iox_ entries, the modules contain many video entries, and these are called by a subroutine (similar to iox_) that transfers to the appropriate entry through a per-attachment transfer vector. This subroutine is called window_, and is the PL/I interface to the window system.

We also provide a command line interface to the window system, the window_call (wdc) command. This command is to window_ what the io_call (io) is to iox_: a way of invoking entries. This command is probably of most value in debugging video programs and the video system.

To the user, the most significant and visible features of the video system are real-time editing and MORE processing. Real-time editing is a way of correcting errors that is incompatible with existing Multics erase and kill processing. MORE processing is an extension to EOP processing that allows unwanted output to be discarded.

2. REAL TIME EDITING

2.1. The Erase Character

The erase character removes the character to the left of the cursor. The cursor moves to the left, and the character is removed. This is significantly different from the current scheme.

First, an erase character immediately deletes a character. Emacs users have grown accustomed to this, but few have seen this in regular Multics usage.

Second, exactly one character is deleted. In the old scheme, an erase character deleted either the complete contents of a single column position (if struck over, or immediately to the left of a column position containing printing characters) or it deleted all white space to the nearest printing character (or beginning of line, which ever came first).

One reason for this change is that real-time erasure is done as soon as the key is struck, before the string is in canonical form. Neither the user nor the system can easily tell what columns hold which characters. Deleting only one character is easy for both.

Secondly, the old scheme required the complete erasure of whitespace because the user could not otherwise tell the resultant column position simply by inspection. Since the new scheme can move the cursor, the new column position will be obvious.

Compatibility would dictate that the old scheme be retained, but there are reasons why the incompatibility should be tolerated: This new way is more like Emacs, which has helped to form people's "video intuition"; the old properties were not, I suspect, widely known or used; the new way is more convenient.

A consequence of real-time erasure is that an erase character cannot be "cancelled". In the old scheme, an erroneous erase character could be overstruck with a graphic character, causing it to obliterate its own (non-empty) column position, rather than whatever lay to its left.

The default erase character is "#", and is settable on a per-window basis by the set_editing_chars control order.

2.2. The Kill Character

The kill character deletes the entire line thus far typed. Again, this happens immediately. The deleted line is saved, and can be recovered. See below. The kill character is settable per-window by the set_editing_chars control order, and defaults to "@".

2.3. The Line Editor

The video system provides more powerful editing than simple erase and kill processing, because this is possible and useful. The interface resembles a very small subset of Emacs. This subset can probably be expanded as resources permit.

The Line Editor is controlled by control characters. Control characters are known throughout the industry, but were unknown to Multics until the rise of Emacs. Control characters are characters in the range 000 to 037. Most terminals generate control characters with a shifting key (called CONTROL) which zeroes the "100" bit of the character typed. For example, if CONTROL is held while "A" (octal 101) is typed, then SOH (octal 001) is generated. For this reason it is common to refer to SOH as CONTROL A, and the notation ^A is used.

In the following list, ESC means the ASCII character ESCAPE (octal 033), not the escape character, and the prefix ^ represents a control character.

DEL — same as the erase character.

ESC DEL — erases one word. A word is defined just as in Emacs, as an unbroken string of upper and lower case alphabetics, numerals, underscores, and the backspace character.

ESC erase — same as ESC DEL.

The Line Editor keeps a ten-deep kill ring, just as Emacs does. Text is saved on the kill ring by the kill character and by the word kill commands. Successive word kills merge the words in the kill ring. The initial contents of the kill ring is the last line returned in the window. This provides editing of the previous command line.

^Y — retrieves deleted text from the kill ring. This is the only way to recover from an erroneous kill

character.

ESC Y — can be typed only after either ^Y or ESC Y. It deletes the text just retrieved, without saving it on the kill ring, rotates the ring (to the next most recently killed text) and retrieves the new item.

The following are also recogninized:

^L — causes the window to be cleared, and the current input line displayed again, starting from home.

^Q — "quotes" the next character, causing it to have no special meaning. This is useful for entering control characters. It serves some of the same purposes as the input escape processing.

No other control characters have meaning. If any are typed, the only action is to cause an audible alarm.

2.4. Input of Control Characters

Any control character (character whose octal value is between 000 and 037) can be input by typing a two character sequence. The first character is called the control character precedence code, and the second is a mnemonic for the control character.

The default control character precedence code is Record Separator (RS, octal 036). This character is also known as Control Circumflex, or "^^", so it has some mnemonic value. Typing the control precedence character has about the same effect on the next character as holding down the CONTROL key does. For example, "^^M" generates CR (or ^M).

The control precedence character is interpreted by the Terminal Control layer of the Video System, not the Window Manager. Thus a control character generated in this way looks just like any other control character, as far as a window is concerned. For example, hitting the BACKSPACE key, typing CONTROL H, and typing ^^H all have the same effect on a window, whereas the escape sequence "\010" does not (the latter inserts a non-canonical backspace into a column position all its own).

The control precedence character can be changed by control order.

The control prefix is interpreted only when the "ctlpfx"

mode (of TC) is on. This mode is on by default.

3. MORE PROCESSING

As lines are displayed in the window, old lines are scrolled off the top of the window or otherwise removed. When output would cause a line to be removed that has been displayed since the most recent input, it is assumed that the user may not have had a chance to read it, and MORE processing occurs. The question "MORE?" appears at the bottom of the screen, and no further output occurs until the user either indicates readiness or that pending output is to be discarded. MORE processing is controlled by the "more" mode, which is enabled by default.

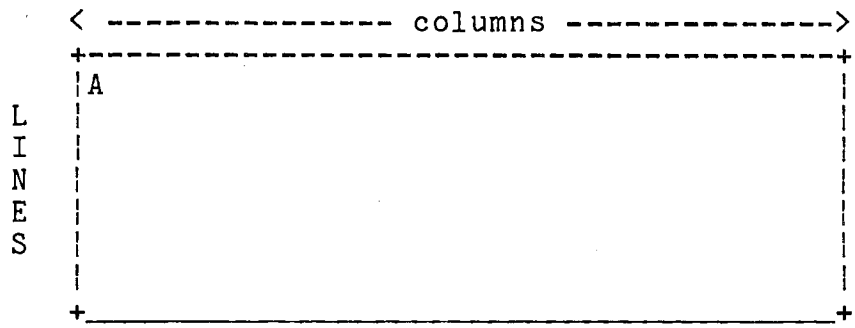
Output resumes if the user strikes CR or FF, and is aborted if the user strikes CONTROL O. The characters used can be set by a control order. Typeahead characters are not seen by MORE processing. The response to MORE must be typed after the prompt appears. All other characters are buffered to be returned later.

When output is discarded, WM simply ignores output until a get_line or get_chars call is made, or a "reset_more" control order call is made, or the window is cleared, or the cursor is homed. Warning: a prompt sent just before a get_line call will not be printed if output is discarded, unless the prompter makes an effort to first reset the discarding.

Name: window_

The window_ subroutine performs I/O to, and controls, a window. A window is a virtual video terminal with an addressable cursor and selectable erase, in a terminal independent manner.

The virtual terminal implemented by window_ is a rectangle of characters, arranged in a variable number of lines (Y) and columns (X). Each screen has a "cursor" which is at some position in the window. I/O is done at the current cursor position. The home position (origin) is in the upper left corner, line 1, column 1.



The window_ subroutine is used in conjunction with the iox_ subroutine to call entry points in an I/O module. The I/O module does the actual work, but is called through window_. The two I/O modules that window_ can be used with are crt_ and vtty_.

The virtual terminal implemented by window_ corresponds closely to common video terminals. The features of the terminal are defined implicitly by the entries below. Not all entries can be supported on all real terminals. The result of calling an unsupported feature is the error code error_table_$no_operation. Programs can determine whether the device in use supports a given operation by making a get_capabilities control order, described in the write-up of crt_, below.

Additional terminals may be supported by defining their video attributes in the Terminal Type File (the TTF). The MPM Communications I/O Manual (CC92) describes the TTF.

Each entry point has an argument denoting the particular I/O switch involved in the operation. The I/O switch pointer is obtained from iox_.

There are several types of arguments accepted by the window_ entry points. Every entry point has at least two arguments, an I/O switch pointer and an error code. Some have additional arguments.

iocbp
: is the first argument to every entry point. Its points to the iocb for the window.

code
: is the last argument to every entry point. It is a standard system error code.

l
: is the line number.

c
: is the column position.

count
: is a count of characters or lines (as appropriate) to be operated on.

region_width
: is the width of a region to be operated on, specified in characters.

region_height
: is the height of a region to be operated on, in characters.

Arguments used only with a specific entry point are documented with that entry point. The calling sequences for all the entry points are in the include file window_dcls.incl.pl1

Entries are listed in categories.

In all entries, an attempt to operate outside the window will result in the error code error_table_$out_of_window, and the

operation will not be performed.

Cursor Motion

Entry: window_$position_cursor

This entry will move the cursor to any requested position on the screen.

usage

```
dcl window_$position_cursor entry (ptr, fixed bin, fixed
      bin, fixed bin (35);

call window_$position_cursor (iocbp, l, c, code);
```

Entry: window_$position_cursor_rel

The entry moves the cursor relative to the current location.

usage

```
dcl window_$position_cursor_rel entry (ptr, fixed bin, fixed
      bin, fixed bin (35));

call window_$position_cursor_rel (iocbp, l_change, c_change,
      code);
```

where:

l_change (input)
is the change in line number.

c_change (input)
is the change in column position.

Entry: window_$home

This entry moves the cursor home. The home position is the upper left hand corner of the screen, (1, 1).

usage

```
dcl window_$home entry (ptr, fixed bin (35));

call window_$home (iocbp, code);
```

Entries: window_$cursor_left
window_$cursor_right
window_$cursor_up
window_$cursor_down

These entries move the cursor one position in the direction indicated by the name.

usage

```
dcl window_$cursor_left entry (ptr, fixed bin (35));

call window_$cursor_left (iocbp, code);
```

Selective Erasure

Entry: window_$clear_window

This entry clears the entire window to spaces, and leaves the cursor at home.

usage

```
dcl window_$clear_window entry (ptr, fixed bin (35));
```

```
call window_$clear_window (iocbp, code);
```

Entry: window_$clear_to_end_of_window

This entry clears all of the window between the cursor and the end of the window. This includes all space to the right of the cursor on the current line, and all lines below the cursor. The position of the cursor is not changed.

usage

```
dcl window_$clear_to_end_of_window entry (ptr,
      fixed bin (35));

call window_$clear_to_end_of_window (iocbp, code);
```

Entry: window_$clear_to_end_of_lines

This entry clears all space to the right of the cursor on the current line to spaces. Positions to the left of the cursor are not affected. The cursor is not moved. It can also clear the complete contents of suceeding lines.

usage

```
dcl window_$clear_to_end_of_lines entry (ptr, fixed bin,
      fixed bin (35));

call window_$clear_to_end_of_lines (iocbp, count, code);
```

where:

count (input)
is the count of lines to clear. It must be greater than zero. If it is 1, then only the current line is cleared. If greater than one, (say N) then the next N-1 lines are completely cleared.

Entry: window_$clear_region

This entry clears the region specified, and leaves the cursor at the upper left hand corner of the region. The region is defined by giving the upper left hand corner (line and column), and the width and height of the region.

usage

```
dcl window_$clear_region entry (ptr, fixed bin, fixed bin,
      fixed bin, fixed bin, fixed bin (35));

call window_$clear_region (iocbp, l, c, region_width,
      region_height, code);
```

Scrolling

Entry: window_$scroll_window

This entry scrolls the window a given number of lines. The window may be scrolled up or down. A positive scroll count scrolls the window up - lines are deleted from the top of the window, and new, blank lines are added to the bottom. The cursor is left at the beginning of the first new blank line. A negative count scrolls the window down - deleting lines from the bottom, and adding lines to the top. The cursor is left at home. It is an error to call this entry if the terminal does not support either scrolling or insert and delete lines.

usage

```
dcl window_$scroll_window entry (ptr, fixed bin, fixed bin
      (35));

call window_$scroll_window (iocbp, count, code);
```

Entry: window_$scroll_region

This entry scrolls a region a specified number of lines up or down, in the same way that window_$scroll_window scrolls the entire window.

usage

```
dcl window_$scroll_region entry (ptr, fixed bin, fixed bin,
      fixed bin, fixed bin, fixed bin, fixed bin (35));

call window_$scroll_region (iocbp, l, c, region_width,
      region_height, count, code);
```

Detailed Alteration of the Screen

Entry: window_$insert_text

This entry inserts text at the current cursor position. Text already on the window at or to the right of the cursor is shifted to the right to accommodate the new text. It is an error to call this entry if the terminal does not support the insertion of text.

usage

```
dcl window_$insert_text entry (ptr, char (*), fixed bin
      (35));

call window_$insert_text (iocbp, text, code);
```

where:

text (input)
: is the character string to be written. Each character in this string must, when output-converted, occupy exactly one print position. The length of this string must be such that characters moved to the right will still stay on the current line in the window. If these conditions are not met, the result is undefined. The cursor is set after the last character inserted..

Entry: window_$delete_lines

This entry deletes one or more lines from the window by moving the cursor to the beginning of the current line, and deleting the contents of that line by moving all lines below up one position. This is repeated for as many lines as requested. The cursor is left at the beginning of the line. It is an error to call this entry point if the terminal does not support the delete lines operation.

usage

```
dcl window_$delete_lines entry (ptr, fixed bin,
      fixed bin (35));

call window_$delete_lines (iocbp,  count, code);
```

Entry: window_$insert_lines

This entry inserts lines at the current line of the window by moving the cursor to the beginning of the current line, and moving all lines below the current line down on position. The newly created line is clear. The bottom line(s) must be clear, so that text is not moved off the bottom of the window. If not, the result is undefined. It is an error to call this entry point if the terminal does not support the insert lines function.

usage

```
dcl window_$insert_lines entry (ptr, fixed bin,
      fixed bin (35));

call window_$insert_lines (iocbp, count, code);
```

Entry: window_$delete_chars

This entry deletes characters on the current line. Any characters to the right of the current cursor position on the same line are moved to the left. Character positions opened up on the right margin are filled with spaces. It is an error to call this entry point if the terminal does not support the delete chars operation.

usage

```
dcl window_$delete_chars entry (ptr, fixed bin,
      fixed bin (35));

call window_$delete_chars (iocbp, count, code);
```

Miscellaneous Functions

Entry: window_$assert_cursor_position

This entry can be used when the caller of window_ has done something non-standard to the window such that window_ may no longer know the true cursor position.

usage

```
dcl window_$assert_cursor_position entry (ptr, fixed bin,
      fixed bin, fixed bin (35));

call window_$assert_cursor_position (iocbp, l, c, code);
```

Entry: window_$get_cursor_position

This entry is used to return the current window co-ordinates of the cursor, which is always maintained by window_.

usage

```
dcl window_$get_cursor_position entry (ptr, fixed bin,
      fixed bin,fixed bin (35));

call window_$get_cursor_position entry (iocbp, l, c, code);
```

where x and y are output arguments of the current position.

Entry: window_$undefine_cursor_position

This entry may be used after the caller of window_ has performed some non-standard operation which has left the cursor in an unknown place. Calling this entry will disable any optimizations that window_ might make on its next call based on knowing the cursor position. The user should inform window_ of the new cursor position by calling window_$assert_cursor_position, or by an absolute cursor position.

usage

```
dcl window_$undefine_cursor_position entry (ptr,
      fixed bin (35));

call window_$undefine_cursor_position (iocbp, code);
```

Entry: window_$output_raw_chars

This entry is used to output a terminal dependent sequence. If the sequence moves the cursor, it is the callers responsibility to inform window_ of the new position. If the new cursor position is unknown, it is the callers responsibility to so inform window_. The sequence is sent to the terminal, but results are undefined and terminal dependent.

usage

```
dcl window_$output_raw_chars entry (ptr, char (*),
      fixed bin (35));

call window_$output_raw_chars (iocb, sequence, code);
```

Entry: window_$bell

This entry rings the terminal bell.

usage

```
dcl window_$bell entry (ptr, fixed bin (35));

call window_$bell (iocbp, code);
```

Internal Entries

The following entries are internal to the window system. They are implemented by TC for use by WM only.

Entry: window_$get_chars_echo

This entry reads up to a requested number of characters, echoing them as read, until either the caller supplied buffer is full, or a "break character" is read. Echo Negotiation is described in MTB-418.

usage

```
dcl window_$get_chars_echo entry (ptr, char (*),
      fixed bin,fixed bin (21), char (1), fixed bin (35));

call window_$get_chars_echo (iocbp, buffer, columns,
      chars_read, break_char, code);
```

where:

buffer (input)
is a caller supplied buffer to hold characters returned.

columns (input)
is the number of columns between the cursor and the end of the window. At most this many characters will be returned.

chars_read (output)
is the number of characters returned. Each character was echoed.

break_char (output)
is the character that caused the echoing to stop. This character has not been echoed. It is either a break character, or was typed after all columns were filled.

The window_$get_chars_echo entry point uses Echo Negotiation to obtain and echo characters. This entry can only be used when the cursor is at the end of a line. The break table must be setup before hand, by a set_break_table control order to TC.

Entry: window_$write_text

This entry writes text on the window in the current cursor location.

usage

```
dcl window_$write_text entry (ptr, char (*), fixed bin (35));

call window_$write_text (iocbp, text, code);
```

where:

text (input)
is the character string to be written. This string should consist of only ASCII graphics (octal codes 040 thru 176 inclusive), and should not be longer than the space remaining on the current line.

Entry: window_$write

This entry flushes TC's buffer, if any.

usage

```
dcl  window_$write entry (ptr, fixed bin (35);
call window_$write (iocbp, code);
```

Name: crt_

The crt_ I/O module supports I/O to a window - a virtual screen located on a real screen. In addition to the usual iox_ entries, the module provides terminal independent access to special video terminal features such as a movable cursor, selective erasure, and scrolling of regions. The module provides a real-time input line editor, does output conversion and "MORE" processing.

The crt_ module implements the Window Management layer of the Multics Window Video System. Like all I/O modules, it is not directly called by users; rather the module is accessed through the I/O system.

Attach Description

```
crt_ SWITCH {FIRST_LINE {HEIGHT {COLUMN_ORIGIN {WIDTH}}}}
```

where

SWITCH
: is the name of the I/O switch implementing Terminal Control. This switch must be attached through vtty_.

FIRST_LINE
: is the line number of the line where the window is to begin. If omitted, the first line is used.

HEIGHT
: is the number of lines in the window. It cannot be supplied if FIRST_LINE is omitted. The default is to use all lines to the end of the screen.

COLUMN_ORIGIN
: is the column number on the terminal of the first column of the window. It cannot be supplied if SIZE is omitted. If omitted, the default is 1. This value must not be greater than the number of columns available.

WIDTH
: is the width of the window. It cannot be supplied if COLUMN_ORIGIN is omitted. The default width of the window is the number of columns between the

COLUMN_ORIGIN and the right edge of the terminal.

The attach args must specify a region which lies wholy in the containing window. If not, the attachment is not made, and the error_table_$out_of_window is returned.

When the window is attached it is cleared and the cursor is left at home.

Opening

The following opening modes are supported: stream_input, stream_output, stream_input_output.

Editing

On input via get_line, lines are edited. The user may strike the erase character and the kill character to delete characters and the entire line. Characters deleted are removed from the screen.

Get Chars Operation

This operation returned exactly one character, unechoed, regardless of the size of the callers buffer. The line editor is not invoked by this call.

Get Line Operation

The get_line operation invokes the real-time input line editor, and returns a complete line typed by the user. A description of the typing conventions is given above.

Control Operation

The following orders are supported:

get_window_info
returns information about the position and extent of the window. The info ptr points to the following structure (declared in window_control_info.incl.pl1)

```
dcl 1 window_position_info,
      2 version fixed bin,
      2 origin,
        3 line fixed bin,
        3 col fixed bin,
      2 extent,
        3 width fixed bin,
        3 height fixed bin;
```

where:

version
is the version number of this structure. It must be window_position_info_version.

line
is the line number of the upper left hand corner of the window on the containing screen.

col
is the column of the upper left hand corner of the window on the containing screen.

width
is the width of the window.

height
is the height of the window.

set_window_info
causes the window to be relocated or to change size (or both). The info ptr points the same structure used in the "get_window_info" control order, and the values have the same meaning, but are new value for the window to assume. It is an error to cause the window to exceed the bounds of the screen. The

results of overlapping another window are not defined.

get_capabilities
returns information about the generic capabilities of the terminal. The info ptr should point to the following structure (declared in window_control_info.incl.pl1)

```
dcl 1 capabilities_info based
        (capabilities_info_ptr),
      2 version fixed bin,
      2 screensize,
        3 columns fixed bin,
        3 rows fixed bin,
      2 flags,
        3 insert_lines bit (1) unal,
        3 delete_lines bit (1) unal,
        3 insert_chars bit (1) unal,
        3 delete_chars bit (1) unal,
        3 scroll bit (1) unal,
        3 true_windows bit (1) unal,
        3 tabs bit (1) unal,
        3 overprint bit (1) unal,
        3 pad bit (28) unal,
      2 line_speed       fixed bin,
      2 lines_per_scroll fixed bin;
```

where:

version
is the version number of this structure and must be capabilities_info_version.

columns
is the number of columns on the terminal

rows
is the number of rows (lines) on the terminal.

insert_lines
is true if the insert_lines function is supported.

delete_lines
is true if the deletre lines function is supported".

insert_chars
is true if the insert_chars function is supported.

delete_chars
is true if the delete_chars function is supported.

scroll
is true if the terminal is capable of scrolling. This will be true for terminals with insert and delete lines, or with direct scroll features.

true_windows
is true if the terminal supports windows. If this bit is not true, then the above functions will fail on a window if it is not full-width (vertical).

tabs
is true if the terminal can move the cursor to a tab stop without erasing characters the cursor passes over.

overprint
is true if sucessive characters at the same location overprint.

line_speed
is the speed of the line to the terminal, in characters per second.

lines_per_scroll
if the terminal can scroll, this gives the number of lines moved per scroll operation. This is usually 1, but may be greater.

reset_more
causes MORE Processing to be reset. All lines on the window may be freely discarded without querying the user.

get_editing_chars
is identical to the operation supported by tty_, which see.

set_editing_chars
is identical to the operation supported by tty_, which see.

get_more_responses
returns information about the responses to MORE processing. The info pointer should point to the following structure (decared in window_control_info.incl.pl1)

```
dcl 1 more_responses_info aligned based
(more_responses_info_ptr),
      2 version fixed bin,
      2 n_yeses fixed bin,
      2 n_noes  fixed bin,
      2 yeses   char (32),
      2 noes    char (32),
```

where:

version (input)
is the version number of this structure and must be set to more_responses_info_version_1, also declared in the include file.

n_yeses (output)
is the number of different affirmative responses, from zero to 32.

n_noes (output)
is the number of different negatives.

yeses (output)
is the concatenation of all the affirmatives. Only the first "n_yeses" are valid.

noes (output)
is the concatenation of all negatives. Only the first "n_noes" are valid.

set_more_responses
Sets the responses. The data structure is the same one used for the "get_more_responses" order. At most 32 yeses and 32 noes may be supplied. It is highly recommended that there be at least one yes, so that output may continue. The "yes" and "no" characters must be distinct, otherwise error_table_$overlapping_more_responses is returned,

and the responses are not changed.

Modes Operation

The modes operation is supported. The recognized modes are listed below. Some modes have a complement indicated by the circumflex character (^) that turns the mode off (e.g. ^more). For these modes, the complement is displayed with that mode. Some modes specify a parameter that can take on a value (e.g. more_mode). These modes are specified as MODE=VALUE, where MODE is the name of the mode and VALUE is the value it is to be set to. Parameterized modes are indicated by the notation (P) in the description below.

more, ^more
Turns MORE processing on. Default is on.

more_mode (P)
controls behavior when a line must be removed to make room for another. Values are:

scroll lines are scrolled off the top of the window. This is the default for all terminals capable of scrolling.

wrap output resumes at the first line. With each new line, the contents of the old line are erased, to make room for new characters. This is the default for all other terminal types.

clear the window is cleared, and output starts at home.

more_prompt (P)
is the string printed to prompt the user when MORE processing occurs.

vertsp, ^vertsp
is only effective when the more mode is on. When vertsp mode is on, output of a FF or VT will cause an immediate MORE query. The default is ^vertsp.

rawo, ^rawo
causes following characters to be output with no processing whatsoever. The result of output in this mode is undefined.

can, ^can
causes input lines to be canonicalized before they are returned. The default is on.

erkl, ^erkl
controls the editing functions of get_line. The default is on, which allows erase and kill processing, and the additional line editor functions.

esc, ^esc
controls input escape processing. The default is on.

rawi, ^rawi
acts as a master control for can, erkl, and esc. If this mode is off, none of the input conventions are provided. The default is on, and the presence of a component of input conventions is controlled by a single mode.

ll (P)
is the width of the window, in characters. Changing the width may also require changes to the"more_mode".

pl (P)
is the height of the window, in characters.

red, ^red
controls interpretation of red shift and black shift characters on output. The default is ^red, which ignores them. In red mode, the character sequence given in the TTF is output. The effect is undefined and terminal-specific. In some cases, "red shifted" output appears in inverse video, but this is not guarenteed.

ctl_char,^ctl_char
specifies that ASCII control characters other than format effectors are to be accepted as input, except for the NUL character. If this mode is off, all such characters are discarded, and the bell (if any) is rung. Note that several of the control characters are used for editing. The default is off.

Control Operations from Command Level

Those control operations which require no info_ptr may be performed from command level using the io_call command, as follows:

io_call control switch_name order_arg
where:

switch_name
is the name of the I/O switch.

order_arg
can be any control order described above under "Control Operation" that can accept a null info_ptr.

The io_call active function is not supported.

Name: vtty_

The vtty_ I/O module supports the Terminal Control layer of the Multics Window Video System.

Entry points in this module are not called directly by users; rather the module is accessed through the I/O system.

Attach Description

vtty_ switch

Where switch is the I/O switch attached to tty_. (This will normally be user_i/o.) The attachment of this switch is moved to a newly created dummy switch, and the target switch is attached to the dummy switch through syn_. These operations are undone when the vtty_ module is detached. The dummy switch's name is "tty_i/o" followed by 15 unique characters.

Opening

The module is opened for stream_input_output when it is attached. The open operation is not supported.

Buffering

This module maintains input and output buffers to reduce calls into ring 0. The output buffer is written out when full or by explicit request.

Get Chars Operation

The get_chars operation always returns exactly one character, unechoed. This module calls the ring zero tty_ primitives, and performs blocking for the user if no character is available.

Get Line Operation

The get_line operation is not supported.

Control Operation

The following control orders are supported.

get_capabilities
returns information about the capabilities of the terminal. The info structure is described in the description of the "get_capabilities" control order in the crt_ module.

get_break_table
returns the current break table. The info ptr should point at a break table, declared as (0:127) bit (1) unal. The i'th bit is set if the character whose rank (in the Multics character set) is i is a break character.

set_break_table
sets the break table. The info pointer should point to a break table as defined by the get_break_table order, above.

get_control_precedence
returns the current control precedence character. The info pointer should point to a structure declared as follows (control_precedence.incl.pl1)

```
dcl 1 control_precedence_info based,
      2 version fixed bin,
      2 precedent char (1) unal;
```

version
must be set by the caller to control_precedence_version_1.

precedent
is the control precedence character.

set_control_precedence
sets the control precedence character. The structure is the same one used in the get_control_precedence control order.

Modes Operation

The following modes are supported. Some modes have a complement, indicated by the circumflex character (^). For these modes, the complement is displayed with the mode.

buffered, ^buffered
In buffered mode, characters are stored by vtty_ until the buffer fills or otherwise deemed necessary.

In unbuffered mode, are characters are output as soon as possible.

ctlpfx, ^ctlpfx
When the ctlpfx mode is on (which it is, by default) one character, the control precedence character, is specially interpreted to allow control characters to be typed in even when the terminal does not support control characters. See above.

Window System Usage

There should be one attachment of vtty_ for each terminal in the process. In general, the vtty_ I/O module is not to be called by users, rather, they should call crt_. The vtty_ module implements only a subset of the full window_ interface, sufficient for crt_, as well as several internal window_ entries. Furthermore, it does not check for errors.

window_call (wdc)

Name: window_call

The window_call command allows the user to perform window video operations on a given window from command level.

Syntax as a Command

window_call FUNCTION {switch} {args}

where:

FUNCTION
is the name of the operation to be performed. It may be chosen from the list below.

switch
is the I/O switch for the window. If omitted, user_i/o is used.

args
are arguments required by the function chosen. If the switch is omitted, no arguments may be given.

List of Supported Functions

In the list below, L represents a line number; C a column number; W a width; (of a region); H a height (of a region). A string is represented by STRING, and must be quoted if it contains spaces, and N represents a count. Finally, DL and DC represent change in line and column, respectively.

position_cursor	L C
position_cursor_rel	DL DC
home	
cursor_left	
cursor_right	
cursor_up	
cursor_down	
clear_window	
clear_to_end_of_window	
clear_to_end_of_lines	N
clear_region	L C W H

window_call (wdc)

scroll_window	N
scroll_region	L C W H N
bell	
delete_line	
delete_lines	N
insert_line	
insert_lines	N
delete_char	
delete_chars	N
insert_text	STRING
output_raw_chars	STRING
assert_cursor_pos	L C
set_window	LH C W

Syntax as an Active Function

```
[window_call FUNCTION {switch} ]
```

where:

FUNCTION
: is the name of the operation to be performed. It must be chosen from the list below.

switch
: is the I/O switch for the window. If omitted, user_i/o is used.

List of Supported Functions

get_cursor_line
get_cursor_col
get_origin_line
get_origin_col
get_width
get_height