

# TABLE OF CONTENTS

---

<b>30. Terminal Input/Output</b>	30.1
<b>30.1. Interrupt Characters</b>	30.1
<b>30.2. Terminal Tables</b>	30.4
<b>30.2.1. Terminal Syntax Classes</b>	30.5
<b>30.2.2. Terminal Control Functions</b>	30.6
<b>30.2.3. Line-Buffering</b>	30.9
<b>30.3. Dribble Files</b>	30.12
<b>30.4. Cursor and Mouse</b>	30.13
<b>30.4.1. Changing the Cursor Image</b>	30.13
<b>30.4.2. Flashing Bars on the Cursor</b>	30.16
<b>30.4.3. Cursor Position</b>	30.17
<b>30.4.4. Mouse Button Testing</b>	30.17
<b>30.4.5. Low Level Mouse Functions</b>	30.18
<b>30.5. Keyboard Interpretation</b>	30.19
<b>30.6. Display Screen</b>	30.22
<b>30.7. Miscellaneous Terminal I/O</b>	30.24

Most input/output operations in Interlisp can be simply modeled as reading or writing on a linear stream of bytes. However, the situation is much more complex when it comes to controlling the user's "terminal," which includes the keyboard, the mouse, and the display screen. For example, Interlisp coordinates the operation of these separate I/O devices so that the cursor on the screen moves as the mouse moves, and any characters typed by the user appear in the window currently containing a flashing cursor. Most of the time, this system works correctly without need for user modification.

The purpose of this chapter is to describe how to access the low-level controls for the terminal I/O devices. It documents the use of interrupt characters, the keyboard characters that generate interrupts. Then, it describes terminal tables, used to determine the meaning of the different editing characters (character delete, line delete, etc.). Then, the "dribble file" facility that allows terminal I/O to be saved onto a file is presented (page 30.12). Finally, the low-level functions that control the mouse and cursor, the keyboard, and the screen are documented.

---

### 30.1 Interrupt Characters

---

Errors and breaks can be caused by errors within functions, or by explicitly breaking a function. The user can also indicate his desire to go into a break while a program is running by typing certain control characters known as "interrupt characters". The following interrupt characters are currently enabled in Interlisp-D:

Note: In Interlisp-D with multiple processes, it is not sufficient to say that "the computation" is broken, aborted, etc; it is necessary to specify which process is being acted upon. Usually, the user wants interrupts to occur in the TTY process, which is the one currently receiving keyboard input. However, sometimes the user wants to interrupt the mouse process, if it is currently busy executing a menu command or waiting for the user to specify a region on the screen. Most of the interrupt characters below take place in the mouse process if it is busy, otherwise the

TTY process. Control-G can be used to break arbitrary processes. For more information, see page 23.14.

Control-B Causes a break within the mouse process (if busy) or the TTY process. Use control-G to break a particular process.

Control-D Aborts the mouse process (if busy) or the TTY process, and unwinds its stack to the top level. Calls **RESET** (page 14.20).

Control-E Aborts the mouse process (if busy) or the TTY process, and unwinds its stack to the last **ERRORSET**. Calls **ERROR!** (page 14.20).

Control-G Pops up a menu listing all of the currently-running processes. Selecting one of the processes will cause a break to take place in that process.

Control-P Changes the **PRINTLEVEL** setting of **PRINTLEVEL** (see page 25.11) in the TTY process. This allows the **PRINTLEVEL** setting to be changed dynamically, even while Interlisp is printing.

When control-P is typed, Interlisp rings the bell, prints "set printlevel to:", and waits for the user to type a series of digits. Input is terminated by a non-digit, after which the program continues.

If the input is terminated by a period or an exclamation point, the **CAR** printlevel is immediately set to this number, and printing continues with the (possibly new) printlevel. If the print routine is currently deeper than the new level, all unfinished lists above that level will be terminated by "--". Thus, if a circular or long list of atoms, is being printed out, typing "control-P0." will cause the list to be terminated immediately.

If the input is terminated by a comma, another number may be typed terminated by a period or exclamation point. The **CAR** printlevel will then be set to the first number, the **CDR** printlevel to the second number.

In either case, if a period is used to terminate the printlevel setting, the printlevel will be returned to its previous setting after the current printout has finished. If an exclamation point is used, the change is permanent and the printlevel is not restored (until it is changed again).

Control-T Prints status information for the TTY process. First it prints "IO wait," "Waiting", or "Running," depending on whether the TTY process is currently in waiting for characters to be typed, waiting for some other reason, or running. Next, it prints the names of the top three frames on the stack, to show what is running. Then, it prints a line describing the percentage of time (since the last control-T) that has been spent running a program, swapping, garbage collecting, doing local disk i/o, etc. For example:

Running in TTWAITFORINPUT in TTBIN in TTYIN1

95% Util, 0% Swap, 4% GC

**DELETE** Clears typeahead in all processes.

The user can disable and/or redefine Interlisp interrupt characters, as well as define new interrupt characters. Interlisp-D is initialized with the following interrupt channels: **RESET** (control-D), **ERROR** (control-E), **BREAK** (control-B), **HELP** (control-G), **PRINTLEVEL** (control-P), **RUBOUT** (DELETE), and **RAID**. Each of these channels independently can be disabled, or have a new interrupt character assigned to it via the function **INTERRUPTCHAR** described below. In addition, the user can enable new interrupt channels, and associate with each channel an interrupt character and an expression to be evaluated when that character is typed.

**(INTERRUPTCHAR CHAR TYP/FORM HARDFLG —)**

[Function]

Defines **CHAR** as an interrupt character. If **CHAR** was previously defined as an interrupt character, that interpretation is disabled.

**CHAR** is either a character or a character code (page 2.12). Note that full sixteen-bit NS characters can be specified as interrupt characters (see page 2.12). **CHAR** can also be a value returned from **INTERRUPTCHAR**, as described below.

If **TYP/FORM = NIL**, **CHAR** is disabled.

If **TYP/FORM = T**, the current state of **CHAR** is returned without changing or disabling it.

If **TYP/FORM** is one of the literal atoms **RESET**, **ERROR**, **BREAK**, **HELP**, **PRINTLEVEL**, **RUBOUT**, or **RAID**, then **INTERRUPTCHAR** assigns **CHAR** to the indicated Interlisp interrupt channel, (reenabling the channel if previously disabled).

If **TYP/FORM** is any other literal atom, **CHAR** is enabled as an interrupt character that when typed causes the atom **TYP/FORM** to be *immediately* set to **T**.

If **TYP/FORM** is a list, **CHAR** is enabled as a user interrupt character, and **TYP/FORM** is the form that is evaluated when **CHAR** is typed. The interrupt will be hard if **HARDFLG=T**, otherwise soft.

**(INTERRUPTCHAR T)** restores all Interlisp channels to their original state, and disables all user interrupts.

**HARDFLG** determines what process the interrupt should run in. If **HARDFLG** is **NIL**, the interrupt will run in the TTY process, which is the process currently receiving keyboard input. If **HARDFLG** is **T**, the interrupt will occur in whichever process happens to be running. If **HARDFLG** is **MOUSE**, the interrupt will happen in the mouse process, if the mouse is busy, otherwise in the TTY process.

**INTERRUPTCHAR** returns a value which, when given as the **CHAR** argument to **INTERRUPTCHAR**, will restore things as they were before the call to **INTERRUPTCHAR**. Therefore, **INTERRUPTCHAR**

can be used in conjunction with **RESETFORM** or **RESETLST** (page 14.26).

**INTERRUPTCHAR** is undoable.

---

**(RESET.INTERRUPTS PERMITTEDINTERRUPTS SAVECURRENT?)** [Function]

**PERMITTEDINTERRUPTS** is a list of interrupt character settings to be performed, each of the form (**CHAR TYP/FORM HARDFLG**). The effect of **RESET.INTERRUPTS** is as if (**INTERRUPTCHAR CHAR TYP/FORM HARDFLG**) were performed for each item on **PERMITTEDINTERRUPTS**, and (**INTERRUPTCHAR OTHERCHAR NIL**) were performed on every other existing interrupt character.

If **SAVECURRENT?** is non-NIL, then **RESET.INTERRUPTS** returns the current state of the interrupts in a form that could be passed to **RESET.INTERRUPTS**, otherwise it returns NIL. This can be used with a **RESET.INTERRUPTS** that appears in a **RESETFORM**, so that the list is built at "entry", but not upon "exit".

---

**(LISPINTERRUPTS)** [Function]

Returns the initial default interrupt character settings for Interlisp-D, as a list that **RESET.INTERRUPTS** would accept.

---

**(INTERRUPTABLE FLAG)** [Function]

if **FLAG=NIL**, turns interrupts off. If **FLAG=T**, turns interrupts on. Value is previous setting. **INTERRUPTABLE** compiles open.

Any interrupt character typed while interrupts are off is treated the same as any other character, i.e. placed in the input buffer, and will not cause an interrupt when interrupts are turned back on.

---

## 30.2 Terminal Tables

---

A read table (page 25.33) contains input/output information that is *media-independent*. For example, the action of parentheses is the same regardless of the device from which the input is being performed. A terminal table is an object that contains information that pertains to *terminal* input/output operations only, such as the character to type to delete the last character or to delete the last line. In addition, terminal tables contain such information as how line-buffering is to be performed, how control characters are to be echoed/printed, whether lower case input is to be converted to upper case, etc.

Using the functions below, the user may change, reset, or copy terminal tables, or create a new terminal table and install it as the primary terminal table via **SETTERMTABLE**. However, unlike

read tables, terminal tables cannot be passed as arguments to input/output functions.

<b>(GETTERMTABLE TTBL)</b>	[Function]
	If <i>TTBL</i> = NIL, returns the primary (i.e., current) terminal table. If <i>TTBL</i> is a terminal table, return <i>TTBL</i> . Otherwise, generates an <b>ILLEGAL TERMINAL TABLE</b> error.
<b>(COPYTERMTABLE TTBL)</b>	[Function]
	Returns a copy of <i>TTBL</i> . <i>TTBL</i> can be a real terminal table, NIL (copies the primary terminal table), or ORIG (returns a copy of the <i>original</i> system terminal table). Note that <b>COPYTERMTABLE</b> is the only function that creates a terminal table.
<b>(SETTERMTABLE TTBL)</b>	[Function]
	Sets the primary terminal table to be <i>TTBL</i> . Returns the previous primary terminal table. Generates an <b>ILLEGAL TERMINAL TABLE</b> error if <i>TTBL</i> is not a real terminal table.
<b>(RESETTERMTABLE TTBL FROM)</b>	[Function]
	Copies (smashes) <i>FROM</i> into <i>TTBL</i> . <i>FROM</i> and <i>TTBL</i> can be NIL or a real terminal table. In addition, <i>FROM</i> can be ORIG, meaning to use the system's original terminal table.
<b>(TERMTABLEP TTBL)</b>	[Function]
	Returns <i>TTBL</i> , if <i>TTBL</i> is a real terminal table, NIL otherwise.

### 30.2.1 Terminal Syntax Classes

A terminal table associates with each character a single "terminal syntax class", one of CHARDELETE, LINEDELETE, WORDDELETE, RETYPE, CTRLV, EOL, and NONE. Unlike read table classes, only one character in a particular terminal table can belong to each of the classes (except for the default class NONE). When a new character is assigned one of these syntax classes by SETSYNTAX (page 25.37), the previous character is disabled (i.e., reassigned the syntax class NONE), and the value of SETSYNTAX is the code for the previous character of that class, if any, otherwise NIL.

The terminal syntax classes are interpreted as follows:

- |                   |  |
|-------------------|--|
| <b>CHARDELETE</b> | (Initially BackSpace and control-A in Interlisp-D) Typing this character deletes the previous character typed. Repeated use of this character deletes successive characters back to the beginning of the line. |
| <b>LINEDELETE</b> | (Initially control-Q in Interlisp-D) Typing this character deletes the whole line; it cannot be used repeatedly.   |

WORDDELETE	(Initially control-W in Interlisp-D) Typing this character deletes the previous "word", i.e., sequence of non-separator characters.
RETYPE	(Initially control-R) Causes the line to be retyped as Interlisp sees it (useful when repeated deletions make it difficult to see what remains).
CTRLV	
CNTRLV	(Initially control-V) When followed by A, B, ... Z, inputs the corresponding control character control-A, control-B, ... control-Z. This allows interrupt characters to be input without causing an interrupt.
EOL	On input from a terminal, the EOL character signals to the line buffering routine to pass the input back to the calling function. It also is used to terminate inputs to READLINE (page 13.36). In general, whenever the phrase carriage-return linefeed is used, what is meant is the character with terminal syntax class EOL.
NONE	The terminal syntax class of all other characters.
	<b>GETSYNTAX</b> , <b>SETSYNTAX</b> , and <b>SYNTAXP</b> all work on terminal tables as well as read tables (see page 25.36). As with read tables, full sixteen-bit NS characters can be specified in terminal tables (see page 2.12). When given NIL as a <i>TABLE</i> argument, <b>GETSYNTAX</b> and <b>SYNTAXP</b> use the primary read table or primary terminal table depending on which table contains the indicated <i>CLASS</i> argument. For example, ( <b>SETSYNTAX CH 'BREAK</b> ) refers to the primary read table, and ( <b>SETSYNTAX CH 'CHARDELETE</b> ) refers to the primary terminal table. In the absence of such information, all three functions default to the primary read table; e.g., ( <b>SETSYNTAX '{ '%'</b> ) refers to the primary read table. If given incompatible <i>CLASS</i> and table arguments, all three functions generate errors. For example, ( <b>SETSYNTAX CH 'BREAK TTBL</b> ), where <i>TTBL</i> is a terminal table, generates an <b>ILLEGAL READTABLE</b> error, and ( <b>GETSYNTAX 'CHARDELETE RDTBL</b> ) generates an <b>ILLEGAL TERMINAL TABLE</b> error.

---

### 30.2.2 Terminal Control Functions

---

**(ECHOCHAR CHARCODE MODE TTBL)** [Function]

**ECHOCHAR** sets the "echo mode" of the character *CHARCODE* to *MODE* in the terminal table *TTBL*. The "echo mode" determines how the character is to be echoed or printed. Note that although the name of this function suggests echoing only, it affects *all* output of the character, both echoing of input and printing of output.

*CHARCODE* should be a character code. *CHARCODE* can also be a list of characters, in which case **ECHOCHAR** is applied to each of

them with arguments *MODE* and *TTBL*. Note that echo modes can be specified for full sixteen-bit NS characters (see page 2.12).

*MODE* should be one of the litatoms **IGNORE**, **REAL**, **SIMULATE**, or **INDICATE** which specify how the character should be echoed or printed:

<b>IGNORE</b>	<i>CHARCODE</i> is never printed.
<b>REAL</b>	<i>CHARCODE</i> itself is printed. Some terminals may respond to certain control and meta characters in interesting ways.
<b>SIMULATE</b>	Output of <i>CHARCODE</i> is simulated. For example, control-l (tab) may be simulated by printing spaces. The simulation is machine-specific and beyond the control of the user.
<b>INDICATE</b>	For control or meta characters, <i>CHARCODE</i> is printed as # and/or ↑ followed by the corresponding alphabetic character. For example, control-A would echo as ↑A, and meta-control-W would echo as #↑W.

The value of **ECHOCHAR** is the previous echo mode for *CHARCODE*. If *MODE* = **NIL**, **ECHOCHAR** returns the current echo mode without changing it.

Warning: In some fonts, control and meta characters may be used for printable characters. If the echemode is set to **INDICATE** for these characters, they will not print out correctly.

**(ECHOCONTROL CHAR MODE TTBL)**

[Function]

**ECHOCONTROL** is an old, limited version of **ECHOCHAR**, that can only specify the echo mode of control characters. *CHAR* is a character or character code. If *CHAR* is an alphabetic character (or code), it refers to the corresponding control character, e.g., (**ECHOCONTROL** 'Z 'INDICATE) if equivalent to (**ECHOCHAR** (*CHARCODE* ↑ Z) 'INDICATE).

**(ECHOMODE FLG TTBL)**

[Function]

If *FLG* = **T**, turns echoing for terminal table *TTBL* on. If *FLG* = **NIL**, turns echoing off. Returns the previous setting.

Note: Unlike **ECHOCHAR**, this only affects echoing of typed-in characters, not printing of characters.

**(GETECHOMODE TTBL)**

[Function]

Returns the current echo mode for *TTBL*.

The following functions manipulate the "raise mode," which determines whether lower case characters are converted to upper case when input from the terminal. There is no "raise mode" for input from files.

<b>(RAISE FLG TTBL)</b>	[Function]
	Sets the RAISE mode for terminal table <i>TTBL</i> . If <i>FLG=NIL</i> , all characters are passed as typed. If <i>FLG=T</i> , input is echoed as typed, but lowercase letters are converted to upper case. If <i>FLG=0</i> , input is converted to upper case <i>before</i> it is echoed. Returns the previous setting.
<b>(GETRAISE TTBL)</b>	[Function]
	Returns the current RAISE mode for <i>TTBL</i> .
<b>(DELETECONTROL TYPE MESSAGE TTBL)</b>	[Function]
	Specifies the output protocol when a <b>CHARDELETE</b> or <b>LINEDELETE</b> is typed, by specifying character strings to print when characters are deleted.  Interlisp-10 (designed for use on hardcopy terminals) echos the characters being deleted, preceding the first by a \ and following the last by a \, so that it is easy to see exactly what was deleted. Note: Interlisp-D is initially set up to physically erase the deleted characters from the display, so the <b>DELETECONTROL</b> strings are initialized to the null string.  The various values of <i>TYPE</i> specify different phases of the deletion, as follows:
<b>1STCHDEL</b>	<i>MESSAGE</i> is the message printed the first time <b>CHARDELETE</b> is typed. Initially "\ in Interlisp-10.
<b>NTHCHDEL</b>	<i>MESSAGE</i> is the message printed when the second and subsequent <b>CHARDELETE</b> characters are typed (without intervening characters). Initially "" in Interlisp-10.
<b>POSTCHDEL</b>	<i>MESSAGE</i> is the message printed when input is resumed following a sequence of one or more <b>CHARDELETE</b> characters. Initially "\ in Interlisp-10.
<b>EMPTYCHDEL</b>	<i>MESSAGE</i> is the message printed when a <b>CHARDELETE</b> is typed and there are no characters in the buffer. Initially "##cr" in Interlisp-10.
<b>ECHO</b>	If <i>TYPE=ECHO</i> , the characters deleted by <b>CHARDELETE</b> are echoed. <i>MESSAGE</i> is ignored.
<b>NOECHO</b>	If <i>TYPE=NOECHO</i> , the characters deleted by <b>CHARDELETE</b> are not echoed. <i>MESSAGE</i> is ignored.
<b>LINEDELETE</b>	<i>MESSAGE</i> is the message printed when the <b>LINEDELETE</b> character is typed. Initially "##cr".  Note: In Interlisp-10, the <b>LINEDELETE</b> , <b>1STCHDEL</b> , <b>NTHCHDEL</b> , <b>POSTCHDEL</b> , and <b>EMPTYCHDEL</b> messages must be 4 characters or fewer in length.  <b>DELETECONTROL</b> returns the previous message as a string. If <i>MESSAGE=NIL</i> , the value returned is the previous message

without changing it. For **TYPE=ECHO** and **NOECHO**, the value of **DELETECONTROL** is the previous echo mode, i.e., **ECHO** or **NOECHO**.

---

**(GETDELETECONTROL TYPE TTBL)**

[Function]

---

Returns the current **DELETECONTROL** mode for **TYPE** in **TTBL**.

---

### 30.2.3 Line-Buffering

Characters typed at the terminal are stored in two buffers before they are passed to an input function. All characters typed in are put into the low-level "system buffer", which allows type-ahead. When an input function is entered, characters are transferred to the "line buffer" until a character with terminal syntax class **EOL** appears (or, for calls from **READ**, when the count of unbalanced open parentheses reaches 0). Note that **PEEKC** is an exception; it returns the character immediately when its second argument is **NIL**. Until this time, the user can delete characters one at a time from the line buffer by typing the current **CHARDELETE** character, or delete the entire line buffer back to the last carriage-return by typing the current **LINEDELETE**.

Note that this line editing is *not* performed by **READ** or **RATOM**, but by Interlisp, i.e., it does not matter (nor is it necessarily known) which function will ultimately process the characters, only that they are still in the Interlisp line buffer. However, the function that is requesting input at the time the buffering starts does determine whether parentheses counting is observed. For example, if a program performs **(PROGN (RATOM) (READ))** and the user types in "A (B C D)", the user must type in the carriage-return following the right parenthesis before any action is taken, because the line buffering is happening under **RATOM**. If the program had performed **(PROGN (READ) (READ))**, the line-buffering would be under **READ**, so that the right parenthesis would terminate line buffering, and no terminating carriage-return would be required.

Once a carriage-return has been typed, the entire line is "available" even if not all of it is processed by the function initiating the request for input. If any characters are "left over", they are returned immediately on the next request for input. For example, **(LIST (RATOM) (READC) (RATOM))** when the input is "A B<sup>cr</sup>" returns the three-element list **(A % B)** and leaves the carriage-return in the buffer.

If a carriage-return is typed when the input under **READ** is not "complete" (the parentheses are not balanced or a string is in progress), line buffering continues, but the lines completed so

far are not available for editing with CHARDELETE or LINEDELETE.

The function CONTROL is available to defeat line-buffering:

(CONTROL MODE TTBL)	[Function]
	If <i>MODE</i> =T, eliminates Interlisp's normal line-buffering for the terminal table <i>TTBL</i> . If <i>MODE</i> =NIL, restores line-buffering (normal). When operating with a terminal table in which (CONTROL T) has been performed, characters are returned to the calling function without line-buffering as described below. CONTROL returns its previous setting.

(GETCONTROL TTBL)	[Function]
	Returns the current control mode for <i>TTBL</i> .

The function that initiates the request for input determines how the line is treated when (CONTROL T) is in effect:

**READ** If the expression being typed is a list, the effect is the same as though done with (CONTROL NIL), i.e., line-buffering continues until a carriage-return or matching parentheses. If the expression being typed is not a list, it is returned as soon as a break or separator character is encountered, e.g., (READ) when the input is "ABC<space>" immediately returns ABC. CHARDELETE and LINEDELETE are available on those characters still in the buffer. Thus, if a program is performing several reads under (CONTROL T), and the user types "NOW IS THE TIME" followed by control-Q, only TIME is deleted, since the rest of the line has already been transmitted to READ and processed.

An exception to the above occurs when the break or separator character is an opening parenthesis, bracket or double-quote, since returning at this point would leave the line buffer in a "funny" state. Thus if the input to (READ) is "ABC(", the ABC is not read until a carriage-return or matching parentheses is encountered. In this case the user could LINEDELETE the entire line, since all of the characters are still in the buffer.

**RATOM** Characters are returned as soon as a break or separator character is encountered. Until then, LINEDELETE and CHARDELETE may be used as with READ. For example, (RATOM) followed by "ABC<control-A><space>" returns AB. (RATOM) followed by "<control-A>" returns ( and types ## indicating that control-A was attempted with nothing in the buffer, since the ( is a break character and would therefore already have been read.

**READC**  
**PEEKC** The character is returned immediately; no line editing is possible. In particular, (READC) is perfectly happy to return the

**CHARDELETE** or **LINEDELETE** characters, or the **ESCAPE** character (%).

The system buffer and line buffer can be directly manipulated using the following functions.

(CLEARBUF FILE FLG)	[Function]
	Clears the input buffer for <i>FILE</i> . If <i>FILE</i> is T and <i>FLG</i> is T, the contents of Interlisp's system buffer and line buffer are saved (and can be obtained via <b>SYSBUF</b> and <b>LINBUF</b> described below). When control-D or control-E is typed, or any of the interrupt characters that require terminal interaction is typed (control-G, or control-P), Interlisp automatically performs ( <b>CLEARBUF T T</b> ). For control-P and, when the break is exited normally, control-H, Interlisp restores the buffer after the interaction. The action of ( <b>CLEARBUF T</b> ), i.e., clearing of typeahead, is also available as the <b>RUBOUT</b> interrupt character, initially assigned to the delete key in Interlisp-D. Note that this interrupt clears both buffers at the time it is <i>typed</i> , whereas the action of the <b>CHARDELETE</b> and <b>LINEDELETE</b> character occur at the time they are <i>read</i> .
(SYSBUF FLG)	[Function]
	If <i>FLG</i> = T, returns the contents of the system buffer (as a string) that was saved at the last ( <b>CLEARBUF T T</b> ). If <i>FLG</i> = NIL, clears this internal buffer.
(LINBUF FLG)	[Function]
	Same as <b>SYSBUF</b> for the line buffer.
	If both the system buffer and Interlisp's line buffer are empty, the internal buffers associated with <b>LINBUF</b> and <b>SYSBUF</b> are not changed by a ( <b>CLEARBUF T T</b> ).
(BKSYSBUF X FLG RDTBL)	[Function]
	<b>BKSYSBUF</b> sets the system buffer to the <b>PRIN1</b> -name of <i>X</i> . The effect is the same as though the user typed <i>X</i> . Some implementations have a limit on the length of <i>X</i> , in which case characters in <i>X</i> beyond the limit are ignored. Returns <i>X</i> . If <i>FLG</i> is T, then the <b>PRIN2</b> -name of <i>X</i> is used, computed with respect to the read table <i>RDTBL</i> . Note that if the user is typing at the same time as the <b>BKSYSBUF</b> is being performed, the relative order of the type-in and the characters of <i>X</i> is unpredictable. Compatibility note: Some implementations of <b>BKSYSBUF</b> (Interlisp-10) use a "system" buffer, from which keyboard

interrupts are also processed. In this case, **BKSYSBUF** of an interrupt character actually invokes the interrupt at some (asynchronous) time after the **BKSYSBUF** is initiated. In other implementations (Interlisp-D), the characters are not processed for interrupts, and it is possible to **BKSYSBUF** characters which would otherwise be impossible to type.

**(BKLINBUF STR)**

[Function]

*STR* is a string. **BKLINBUF** sets Interlisp's line buffer to *STR*. Some implementations have a limit on the length of *STR*, in which case characters in *STR* beyond the limit are ignored. Returns *STR*.

**BKLINBUF**, **BKSYSBUF**, **LINBUF**, and **SYSBUF** provide a way of "undoing" a **CLEARBUF**. Thus to "peek" at various characters in the buffer, one could perform (**CLEARBUF T T**), examine the buffers via **LINBUF** and **SYSBUF**, and then put them back.

The more common use of these functions is in saving and restoring typeahead when a program requires some unanticipated (from the user's standpoint) input. The function **RESETBUFS** provides a convenient way of simply clearing the input buffer, performing an interaction with the user, and then restoring the input buffer.

**(RESETBUFS FORM<sub>1</sub> FORM<sub>2</sub> ... FORM<sub>N</sub>)**

[NLambda NoSpread Function]

Clears any typeahead (ringing the terminal's bell if there was, indeed, typeahead), evaluates *FORM<sub>1</sub>*, *FORM<sub>2</sub>*... *FORM<sub>N</sub>*, then restores the typeahead. Returns the value of *FORM<sub>N</sub>*. Compiles open.

### 30.3 Dribble Files

A dribble file is a "transcript" of all of the input and output on a terminal. In Interlisp-D, **DRIBBLE** opens a dribble file for the current process, recording the terminal input and output for that process. Multiple processes can have separate dribble files open at the same time.

**(DRIBBLE FILE APPENDFLG THAWEDFLG)**

[Function]

Opens *FILE* and begins recording the typescript. Returns the old dribble file if any, otherwise **NIL**. If *APPENDFLG*=**T**, the typescript will be appended to the end of *FILE*. If *THAWEDFLG*=**T**, the file will be opened in "thawed" mode, for those implementations that support it. (**DRIBBLE**) closes the dribble file for the current process. Only one dribble file can be

---

active for each process at any one time, so **(DRIBBLE FILE1)** followed by **(DRIBBLE FILE2)** will cause *FILE1* to be closed.

---

**(DRIBBLEFILE)**

[Function]

---

Returns the name of the current dribble file for the current process, if any, otherwise NIL.

---

Terminal input is echoed to the dribble file a line buffer at a time. Thus, the typescript produced is somewhat neater than that appearing on the user's terminal, because it does *not* show characters that were erased via control-A or control-Q. Note that the typescript file is *not* included in the list of files returned by **(OPENP)**, nor will it be closed by a call to **CLOSEALL** or **CLOSEF**. Only **(DRIBBLE)** closes the typescript file.

---

## 30.4 Cursor and Mouse

---

A mouse is a small box connected to the computer keyboard by a long wire. On the top of the mouse are two or three buttons. On the bottom is a rolling ball or a set of photoreceptors, to detect when the mouse is moved. As the mouse is moved on a surface, a small image on the screen, called the cursor, moves to follow the movement of the mouse. By moving the mouse, the user can cause the cursor to point to any part of the display screen.

The mouse and cursor are an important part of the Interlisp-D user interface. The Interlisp-D window system allows the user to create, move, and reshape windows, and to select items from displayed menus, all by moving the mouse and clicking the mouse buttons. This section describes the low-level functions used to control the mouse and cursor.

---

### 30.4.1 Changing the Cursor Image

---

Interlisp-D maintains the image of the cursor on the screen, moving it as the mouse is moved. The bitmap that becomes visible as the cursor can be accessed by the following function:

**(CURSORBITMAP)**

[Function]

---

Returns the cursor bitmap.

---

---

<b>CURSORWIDTH</b>	[Variable]
--------------------	------------

---

<b>CURSORHEIGHT</b>	[Variable]
---------------------	------------

---

Value is the width and height of the cursor bitmap, respectively.

---

The cursor bitmap can be changed like any other bitmap by **BITBLT**ing into it or pointing a display stream at it and printing or drawing curves. However, for some applications it is necessary to save and restore the cursor, which can be most easily done using **CURSOR** record objects. A **CURSOR** record contains fields **CURSORBITMAP** and **CURSORHOTSPOT**. The value of the **CURSORBITMAP** field is a bitmap that is **CURSORWIDTH** bits wide by **CURSORHEIGHT** high. The value of the **CURSORHOTSPOT** field is the "hot spot" of the cursor, a position in the bitmap interpreted as the point that the cursor is pointing to. **CURSOR** objects can be saved on a file using the file package command **CURSORS**, or the **UGLYVARS** file package command.

---

<b>(CURSORCREATE BITMAP X Y)</b>	[Function]
----------------------------------	------------

---

Returns a cursor object which has **BITMAP** as its image and the location (**X,Y**) as the hot spot. If **X** is a **POSITION**, it is used as the hot spot. If **BITMAP** has dimensions different from **CURSORWIDTH** by **CURSORHEIGHT**, the lesser of the widths and the lesser of the heights are used to determine the bits that actually get copied into the lower left corner of the cursor. If **X** is **NIL**, 0 is used. If **Y** is **NIL**, **CURSORHEIGHT-1** is used. The default cursor is an uparrow with its tip in the upper left corner and its hot spot at (0,**CURSORHEIGHT-1**).

---



---

<b>(CURSOR NEWCURSOR —)</b>	[Function]
-----------------------------	------------

---

Returns a **CURSOR** record instance that contains (a copy of) the current cursor specification. If **NEWCURSOR** is a **CURSOR** record instance, the cursor will be set to the values in **NEWCURSOR**. If **NEWCURSOR** is T, the cursor will be set to the default cursor **DEFAULTCURSOR**, an upward left pointing arrow:




---



---

<b>(SETCURSOR NEWCURSOR —)</b>	[Function]
--------------------------------	------------

---

If **NEWCURSOR** is a **CURSOR** record instance, the cursor will be set to the values in **NEWCURSOR**. This does not return the old cursor, and therefore, provides a way of changing the cursor without using storage.

---



---

<b>(FLIPCURSOR)</b>	[Function]
---------------------	------------

---

Inverts the cursor.

---

The following list describes the cursors used by the Interlisp-D system. Most of them are stored as the values of various variables.



In variable **DEFAULTCURSOR**. This is the default cursor.



In variable **WAITINGCURSOR**. Represents an hourglass. Used during long computations.



In variable **MOUSECONFIRMCURSOR**. Indicates that the system is waiting for the user to confirm an action by pressing the left mouse button, or aborting the action by pressing any other button. Used by the function **MOUSECONFIRM** (page 28.11).



In variable **SYSOUTCURSOR**. Indicates that the system is saving the virtual memory in a sysout file. See **SYSOUT**, page 12.8.



In variable **SAVINGCURSOR**. Indicates that **SAVEVM** has been called automatically to save the virtual memory state after the system is idle for long enough. See **SAVEVMWAIT**, page 12.7.



In variable **CROSSHAIRS**. Used by **GETPOSITION** (page 28.9) to indicate a position.



In variable **BOXCURSOR**. Used by **GETBOXPOSITION** (page 28.9) to indicate where to place the corner of a box.



In variable **FORCEPS**. Used by **GETREGION** (page 28.10) when the user switches corners.



In variable **EXPANDINGBOX**. Used by **GETREGION** (page 28.10) when a box is first displayed.



In variable **UpperRightCursor**.



In variable **LowerRightCursor**.



In variable **UpperLeftCursor**.



In variable **LowerLeftCursor**.

The previous four cursors are used by **GETREGION** (page 28.10) to indicate the four corners of a region.



In variable **VertThumbCursor**. Used during scrolling to indicate thumbing in a vertical scroll bar.



In variable **VertScrollCursor**.



In variable **ScrollUpCursor**.



In variable **ScrollDownCursor**.

The previous four cursors are used by **SCROLL.HANDLER** (page 28.24) during vertical scrolling.



In variable **HorizThumbCursor**. Used during scrolling to indicate thumbing in a horizontal scroll bar.



In variable **HorizScrollCursor**.



In variable **ScrollLeftCursor**.



In variable **ScrollRightCursor**.

The previous four cursors are used by **SCROLL.HANDLER** (page 28.24) during horizontal scrolling.

TELE  
RAID, ↑D, ↑N, CMD  
CA  
Raid, Brk, NFT

These cursors are used by the Teleraid low-level debugger. These cursors are not accessible as standard Interlisp-D cursors.

### 30.4.2 Flashing Bars on the Cursor

The low-level Interlisp-D system uses the cursor to display certain system status information, such as garbage collection or swapping. This is done because changing the cursor image is very quick, and does not require interacting with the window system. Interlisp inverts horizontal bars on the cursor when the system is swapping pages, or doing certain stack operations. Normally, these bars are only inverted for a very short time, so they look like they are flashing. These cursor changes are interpreted as follows:

Inverted cursor:

Whatever image is being displayed as the cursor, whenever Interlisp does a garbage collection, the whole cursor is inverted.

Top bar:

Swap read. On when Interlisp is swapping in a page from the virtual memory file into the real memory. It is also on when Interlisp allocates a new virtual memory page, even though that doesn't involve a disk read. If this is flashing a lot, the system is doing a lot of swapping. This is an indication that the virtual memory working set is fragmented (see page 22.1). Performance may be improved by reloading a clean Interlisp system.

Upper middle bar:

Stack operations. If this is flashing a lot, it suggests that some process is neglecting to release stack pointers in a timely fashion (see page 11.9).

Lowererler middle bar:

Stack operations. On when Interlisp is moving frames on the stack. If the system is slow, and this is flashing a lot, **HARDRESET** (page 23.1) sometimes helps.

Bottom bar:

Swap write. On when Interlisp writes a dirty virtual memory page from the real memory back into the virtual memory file.

### 30.4.3 Cursor Position

The position at which the cursor bitmap is being displayed can be read or set using the following functions:

---

<b>(CURSORPOSITION NEWPOSITION DISPLAYSTREAM OLDPOSITION)</b>	[Function]
---	------------

Returns the location of the cursor in the coordinate system of *DISPLAYSTREAM* (or the current display stream, if *DISPLAYSTREAM* is **NIL**). If *NEWPOSITION* is non-**NIL**, it should be a position and the cursor will be positioned at *NEWPOSITION*. If *NEWPOSITION* is **NIL**, the current position is simple returned.

Note: The current position of the cursor is the position of the "hot spot" of the cursor, not the position of the cursor bitmap.

If *OLDPOSITION* is a **POSITION** object, this object will be changed to point to the location of the cursor and returned, rather of allocating a new **POSITION**. This can improve performance if **CURSORPOSITION** is called repeatedly to track the cursor.

Note: To get the location of the cursor in absolute screen coordinates, use the variables **LASTMOUSEX** and **LASTMOUSEY** (page 30.18).

---



---

<b>(ADJUSTCURSORPOSITION DELTAX DELTAY)</b>	[Function]
---	------------

Moves the cursor *DELTAX* points in the X direction and *DELTAY* points in the Y direction. *DELTAX* and *DELTAY* default to 0.

---

### 30.4.4 Mouse Button Testing

There are two or three keys on the mouse. These keys (also called buttons) are referred to by their location: **LEFT**, **MIDDLE**, or **RIGHT**. The following macros are provided to test the state of the mouse buttons:

---

<b>(MOUSESTATE BUTTONFORM)</b>	[Macro]
--------------------------------	---------

Reads the state of the mouse buttons, and returns **T** if that state is described by *BUTTONFORM*. *BUTTONFORM* can be one of the key indicators **LEFT**, **MIDDLE**, or **RIGHT**; the atom **UP** (indicating all keys are up); the form (**ONLY KEY**); or a form of **AND**, **OR**, or **NOT** applied to any valid button form.

For example: **(MOUSESTATE LEFT)** will be true if the left mouse button is down. **(MOUSESTATE (ONLY LEFT))** will be true if the left mouse button is the only one down. **(MOUSESTATE (OR (NOT LEFT) MIDDLE))** will be true if either the left mouse button is up or the middle mouse button is down.

---

<b>(LASTMOUSESTATE BUTTONFORM)</b>	[Macro]
Similar to <b>MOUSESTATE</b> , but tests the value of <b>LASTMOUSEBUTTONS</b> (below) rather than getting the current state. This is useful for determining which keys caused <b>MOUSESTATE</b> to be true.	
<b>(UNTILMOUSESTATE BUTTONFORM INTERVAL)</b>	[Macro]
<b>BUTTONFORM</b> is as described in <b>MOUSESTATE</b> . Waits until <b>BUTTONFORM</b> is true or until <b>INTERVAL</b> milliseconds have elapsed. The value of <b>UNTILMOUSESTATE</b> is <b>T</b> if <b>BUTTONFORM</b> was satisfied before it timed out, otherwise <b>NIL</b> . If <b>INTERVAL</b> is <b>NIL</b> , it waits indefinitely. This compiles into an open loop that calls the <b>TTY</b> wait background function. This form should not be used inside the <b>TTY</b> wait background function. <b>UNTILMOUSESTATE</b> does not use any storage during its wait loop.	
<b>30.4.5 Low Level Mouse Functions</b>	
This section describes the functions and variables that provide low level access to the mouse and cursor.	
<b>(LASTMOUSEX DISPLAYSTREAM)</b>	[Function]
Returns the value of the cursor's X position in the coordinates of <b>DISPLAYSTREAM</b> (as of the last call to <b>GETMOUSESTATE</b> , below).	
<b>(LASTMOUSEY DISPLAYSTREAM)</b>	[Function]
Returns the value of the cursor's Y position in the coordinates of <b>DISPLAYSTREAM</b> (as of the last call to <b>GETMOUSESTATE</b> , below).	
<b>LASTMOUSEX</b>	[Variable]
Value is the X position of the cursor in absolute screen coordinates (as of the last call to <b>GETMOUSESTATE</b> , below).	
<b>LASTMOUSEY</b>	[Variable]
Value is the Y position of the cursor in absolute screen coordinates (as of the last call to <b>GETMOUSESTATE</b> , below).	
<b>LASTMOUSEBUTTONS</b>	[Variable]
Value is an integer that has bits on corresponding to the mouse buttons that are down (as of the last call to <b>GETMOUSESTATE</b> , below). Bit <b>4Q</b> is the left mouse button, <b>2Q</b> is the right button, <b>1Q</b> is the middle button.	

<b>LASTKEYBOARD</b>	[Variable]
	Value is an integer encoding the state of certain keys on the keyboard (as of the last call to <b>GETMOUSESTATE</b> , below). Bit <b>200Q</b> = lock, <b>100Q</b> = left shift, <b>40Q</b> = ctrl, <b>10Q</b> = right shift, <b>4Q</b> = blank Bottom, <b>2Q</b> = blank Middle, <b>1Q</b> = blank Top. If the key is down, the corresponding bit is on.
<b>(GETMOUSESTATE)</b>	[Function]
	Reads the current state of the mouse and sets the variables <b>LASTMOUSEX</b> , <b>LASTMOUSEY</b> , and <b>LASTMOUSEBUTTONS</b> . In polling mode, the program must remember the previous state and look for changes, such as a key going up or down, or the cursor moving outside a region of interest.
<b>(DECODEBUTTONS BUTTONSTATE)</b>	[Function]
	Returns a list of the mouse buttons that are down in the state <b>BUTTONSTATE</b> . If <b>BUTTONSTATE</b> is not a small integer, the value of <b>LASTMOUSEBUTTONS</b> (above) is used. The button names that can be returned are: <b>LEFT</b> , <b>MIDDLE</b> , <b>RIGHT</b> (the three mouse keys).

## 30.5 Keyboard Interpretation

For each key on the keyboard and mouse there is a corresponding bit in memory that the hardware turns on and off as the key moves up and down. System-level routines decode the meaning of key transitions according to a table of "key actions", which may be to put particular character codes in the sysbuffer, cause interrupts, change the internal shift/control status, or create events to be placed in the mouse buffer.

<b>(KEYDOWNP KEYNAME)</b>	[Function]
	Used to read the instantaneous state of any key, independent of any buffering or pre-assigned key action. Returns T if the key named <b>KEYNAME</b> is down at the moment the function is executed.
	Most keys are named by the characters on the key-top. Therefore, ( <b>KEYDOWNP 'a</b> ) or ( <b>KEYDOWNP 'A</b> ) returns T if the "A" key is down.
	There are a number of keys that do not have standard names printed on them. These can be accessed by special names as follows:
Space	<b>SPACE</b>
Carriage return	<b>CR</b>

Line-feed	LF
Backspace	BS
Tab	TAB
Blank keys on 1132	The 1132 keyboard has three unmarked keys on the right of the normal keyboard. These can be accessed by BLANK-BOTTOM, BLANK-MIDDLE, and BLANK-TOP.
Escape	ESCAPE
Shift keys	LSHIFT for the left shift key, RSHIFT for the right shift key.
Shift lock key	LOCK
Control key	CTRL
Mouse buttons	The state of the mouse buttons can be accessed using LEFT, MIDDLE, and RIGHT.

(SHIFTDOWNP SHIFT)	[Function]
	Returns <i>T</i> if the internal "shift" flag specified by SHIFT is on; NIL otherwise.
	If SHIFT = 1SHIFT, 2SHIFT, LOCK, META, or CTRL, SHIFTDOWNP returns the state of the left shift, right shift, shift lock, control, and meta flags, respectively.
	If SHIFT = SHIFT, SHIFTDOWNP returns T if either the left or right shift flag is on.
	If SHIFT = USERMODE1, USERMODE2, or USERMODE3, SHIFTDOWNP returns the state of one of three user-settable flags that have no other effect on key interpretation. These flags can be set or cleared on character transitions by using KEYACTION (below).

(KEYACTION KEYNAME ACTIONS —)	[Function]
	Changes the internal tables that define the action to be taken when a key transition is detected by the system keyboard handler. KEYNAME is specified as for KEYDOWNP. ACTIONS is a dotted pair of the form (DOWN-ACTION . UP-ACTION), where the acceptable transition actions and their interpretations are:

NIL	
IGNORE	Take no action on this transition (the default for up-transitions on all ordinary characters).
(CHAR SHIFTEDCHAR LOCKFLAG)	If a transition action is a three-element list, CHAR and SHIFTEDCHAR are either character codes or (non-numeric) single-character litatoms standing for their character codes. Note that CHAR and SHIFTEDCHAR can be full sixteen-bit NS characters (see page 2.12). When the transition occurs, CHAR or SHIFTEDCHAR is transmitted to the system buffer, depending on whether either of the two shift keys are down.

**LOCKFLAG** is optional, and may be **LOCKSHIFT** or **NOLOCKSHIFT**. If **LOCKFLAG** is **LOCKSHIFT**, then **SHIFTEDCHAR** will also be transmitted when the **LOCK** shift is down (the alphabetic keys initially specify **LOCKSHIFT**, but the digit keys specify **NOLOCKSHIFT**). For example, (**a A LOCKSHIFT**) and (**61Q ! NOLOCKSHIFT**) are the initial settings for the down transitions of the "a" and "1" keys respectively.

**1SHIFTUP, 1SHIFTDOWN**  
**2SHIFTUP, 2SHIFTDOWN**  
**CTRLUP, CTRLDOWN**  
**METAUP, METADOWN**

Change the status of the internal "shift" flags for the left shift, right shift, control, and meta keys, respectively. These shifts affect the interpretation of ordinary key actions. If either of the shifts is down, then **SHIFTEDCHARs** are transmitted. If the control flag is on, then the seventh bit of the character code is cleared as characters are transmitted. If the meta flag is on, the eighth bit of the character code is set (normally cleared) as characters are transmitted. For example, the initial keyactions for the left shift key is (**1SHIFTDOWN . 1SHIFTUP**).

**LOCKUP, LOCKDOWN, LOCKTOGGLE**

Change the status of the internal "shift" flags for the shift lock key. If the lock flag is down, then **SHIFTEDCHARs** are transmitted if the key action specified **LOCKSHIFT**. **LOCKUP** and **LOCKDOWN** clear and set the shift lock flag, respectively. **LOCKTOGGLE** complements the flag (turning it off if the flag is on; on if the flag is off).

**USERMODE1UP, USERMODE1DOWN, USERMODE1TOGGLE**  
**USERMODE2UP, USERMODE2DOWN, USERMODE2TOGGLE**  
**USERMODE3UP, USERMODE3DOWN, USERMODE3TOGGLE**

Change the status of the three user flags **USERMODE1**, **USERMODE2**, and **USERMODE3**, whose status can be determined by calling **SHIFTDOWNP** (above). These flags have no other effect on key interpretation.

- |              |  |
|--------------|--|
| <b>EVENT</b> | An encoding of the current state of the mouse and selected keys is placed in the mouse-event buffer when this transition is detected.                          |
|              | <b>KEYACTION</b> returns the previous setting for <b>KEYNAME</b> . If <b>ACTIONS</b> is <b>NIL</b> , returns the previous setting without changing the tables. |

**(MODIFY.KEYACTIONS KEYACTIONS SAVECURRENT?)**

[Function]

**KEYACTIONS** is a list of key actions to be set, each of the form (**KEYNAME . ACTIONS**). The effect of **MODIFY.KEYACTIONS** is as if (**KEYACTION KEYNAME ACTIONS**) were performed for each item on **KEYACTIONS**.

If **SAVECURRENT?** is non-**NIL**, then **MODIFY.KEYACTIONS** returns a list of all the results from **KEYACTION**, otherwise it returns **NIL**.

This can be used with a **MODIFY.KEYACTIONS** that appears in a **RESETFORM**, so that the list is built at "entry", but not upon "exit".

**(METASHIFT FLG)**

[NoSpread Function]

If *FLG* is T, changes the keyboard handler (via **KEYACTION**) so as to interpret the "stop" key on the 1108 as a metashift: if a key is struck while the meta is down, it is read with the 200Q bit set. For CHAT users this is a way of getting an "Edit" key on your simulated Datamedia.

If *FLG* is other than NIL or T, it is passed as the *ACTIONS* argument to **KEYACTION**. The reason for this is that if someone has set the "STOP" key to some random behavior, then **(RESETFORM (METASHIFT T) --)** will correctly restore that random behavior.

---

## 30.6 Display Screen

---

Interlisp-D supports a high-resolution bitmap display screen. All printing and drawing operations to the screen are actually performed on a bitmap in memory, which is read by the computer hardware to become visible as the screen. This section describes the functions used to control the appearance of the display screen.

**(SCREENBITMAP)**

[Function]

---

Returns the screen bitmap.

---

**SCREENWIDTH**

[Variable]

**SCREENHEIGHT**

[Variable]

---

Value is the width and height of the screen bitmap, respectively.

---

**WHOLEDISPLAY**

[Variable]

---

Value is a region that is the size of the screen bitmap.

---

The background shade of the display window can be changed using the following function:

**(CHANGEBACKGROUND SHADE —)**

[Function]

Changes the background shade of the window system. *SHADE* determines the pattern of the background. If *SHADE* is a texture, then the background is simply painted with it. If *SHADE* is a **BITMAP**, the background is tessellated (tiled) with it to cover the

screen. If *SHADE* is T, it changes to the original shade, the value of **WINDOWBACKGROUND SHADE**. It returns the previous value of the background.

**(CHANGEBACKGROUND BORDER SHADE —)**

[Function]

On the Xerox 1108, changes the shade of the border of the display to *SHADE*, which should be a texture. It returns the previous texture of the background border. **CHANGEBACKGROUND BORDER** is a no-op on the Xerox 1132.

**WINDOWBACKGROUND SHADE**

[Variable]

Value is the default background shade for the display.

**(VIDEOCOLOR BLACKFLG)**

[NoSpread Function]

Sets the interpretation of the bits in the screen bitmap. If *BLACKFLG* is **NIL**, a 0 bit will be displayed as white, otherwise a 0 bit will be displayed as black. **VIDEOCOLOR** returns the previous setting. If *BLACKFLG* is not given, **VIDEOCOLOR** will return the current setting without changing anything.

**Note:** This function only works on the Xerox 1100 and Xerox 1108.

**(VIDEORATE TYPE)**

[Function]

Sets the rate at which the screen is refreshed. *TYPE* is one of **NORMAL** or **TAPE**. If *TYPE* is **TAPE**, the screen will be refreshed at the same rate as TV (60 cycles per second). This makes the picture look better when video taping the screen. Note: Changing the rate may change the dimensions of the display on the picture tube.

Maintaining the video image on the screen uses cpu cycles, so turning off the display can improve the speed of compute-bound tasks. When the display is off, the screen will be white but any printing or displaying that the program does will be visible when the display is turned back on. Note: Breaks and **PAGEFULLFN** waiting (page 28.30) turn the display on, but users should be aware that it is possible to have the system waiting for a response to a question printed or a menu displayed on a non-visible part of the screen. The functions below are provided to turn the display off.

**Note:** These functions have no effect on the Xerox 1108 display.

**(SETDISPLAYHEIGHT NSCANLINES)**

[Function]

Sets the display to only show the top *NSCANLINES* of the screen. If *NSCANLINES* is T, resets the display to show the full screen. Returns the previous setting.

(DISPLAYDOWN FORM NSCANLINES)	[Function]
Evaluates <i>FORM</i> (with the display set to only show the top <i>NSCANLINES</i> of the screen), and returns the value of <i>FORM</i> . It restores the screen to its previous setting. If <i>NSCANLINES</i> is not given, it defaults to 0.	

---

## 30.7 Miscellaneous Terminal I/O

---

(RINGBELLS <i>N</i> )	[Function]
Flashes (reverse-videos) the screen <i>N</i> times (default 1). On the Xerox 1108, this also beeps through the keyboard speaker.	

(PLAYTUNE <i>Frequency/Duration.pairlist</i> )	[Function]
On the Xerox 1108, <b>PLAYTUNE</b> plays a sequence of notes through the keyboard speaker. <i>Frequency/Duration.pairlist</i> should be a list of dotted pairs ( <i>FREQUENCY</i> . <i>DURATION</i> ). <b>PLAYTUNE</b> maps down its argument, beeping the 1108 keyboard buzzer at each frequency for the specified amount of time. Specifying <b>NIL</b> for a frequency means to turn the beeper off the specified amount of time. The units of time are <b>TICKS</b> (page 12.16), which last about 28.78 microseconds on the Xerox 1108. <b>PLAYTUNE</b> makes no sound on a Xerox 1132. The default "simulate" entry for control-G (ASCII BEL) on the 1108 uses <b>PLAYTUNE</b> to make a short beep.	

**PLAYTUNE** is implemented using **BEEPON** and **BEEPOFF**:

(BEEPON <i>FREQ</i> )	[Function]
On the Xerox 1108, turns on the keyboard speaker playing a note with frequency <i>FREQ</i> , measured in <b>TICKS</b> (page 12.16). The speaker will continue to play the note until <b>BEEPOFF</b> is called.	

(BEEPOFF)	[Function]
Turns off the keyboard speaker on the Xerox 1108.	

(SETMAINTPANEL <i>N</i> )	[Function]
On the Xerox 1108, this sets the four-digit "maintanance panel" display on the front of the computer to display the number <i>N</i> .	