1. Introduction
This report describes briefly a set of display primitives that we have developed at PARC to extend the capabilities of InterLisp[1]. These primitives are designed to operate a raster-scanned display, and concentrate on facilities for placing text carefully on the display and for moving chunks of an already-created display.

The primitives are deliberately designed to provide a low-level interface to the display. A display output primitive will cause a specific change to appear on the display by changing the contents of a frame buffer (or some other memory) that is used to refresh the raster-scanned image. The primitives make no assumptions about the sorts of data structures for describing the display that an application program may wish to build.

Our implementation of these primitives involves two computers: InterLisp is executed on MAXC, and communicates with a program called Chat which maintains the frame buffer that drives a 808 by 606 point raster display. Although the communications link ably provides the bandwidth necessary to achieve rapid screen changes, it nonetheless requires special treatment of synchronization. If these primitives were to be implemented on a single computer that both executes InterLisp and performs the display modifications, the synchronization problems could be ignored.

The design of this system is complicated by the need to accommodate on the display ordinary "teletype" output generated by InterLisp and Tenex, as well as carefully-constructed graphic displays. The primitives resolve this problem in a reasonably effective way: the unformatted character output may be directed to a specific region of the screen under complete control of the LISP program. However, we are forced to acknowledge the existence of two independent sources of information for Chat: a stream of teletype characters emerging from Tenex and a separate graphics connection that carries the characters and graphics protocol generated by LISP. As with the Network Graphics Protocol[2], it is important that these connections be kept separate: the graphics connection must transmit highly structured protocol messages that cannot suffer interference from system messages and other uncontrolled teletype transmissions.

2. The ADIS functions
This report does not cover the detailed implementation of the system, but concentrates on a description of the collection of LISP functions that are provided, and their intended effects. Each function is named ADISXXX; the actual names of the functions use all upper-case
letters, even though the description below does not.

**ResetSave.** Several of the functions for setting state variables use argument conventions that permit use with ResetSave. They have the following properties: (1) called with no arguments, they return a "current state;" (2) when called with legitimate arguments, they return the "current state" before the alteration induced by the arguments; (3) called with a "current state" (either a list or a number) as argument, they reset their state. Functions with these properties are flagged (ResetSave). Note that the value returned by such a function is usually a pointer to an internal (ADIS) data structure. If the calling program wishes to make any use of the value, it should be copied.

### 2.1 Initializing the Chat display

**ADISInit[n]**

Initializes a graphics connection to Chat for all graphics output. If a connection already exists, it is first closed cleanly, and then re-opened. If the connection can be established, the result of this function will be a LISP file; otherwise, the result will be NIL.

The global variable ADISFILE is set to the file that results, or NIL if no connection is opened. If other ADISxxx routines find ADISFILE null, they return without transmitting protocol over the (non-existent) connection. If you detach, or if the Chat program crashes, or if the message 10 DATA ERROR appears on your LISP job and you are forced to re-enter, before calling any other ADIS routines you should call ADISCheck which carefully interrogates the state of the connection and sets ADISFILE according to the actual state of the connection.

The parameter n specifies how many text lines at the bottom of the Chat display should be reserved for the initial "teletype simulation area," in which characters typed by MAXC over the teletype connection will be displayed. Note that the size and location of this area can be altered with ADIS functions after the connection is initialized. ADIS requires one special character code (currently 3, control C) for communication with Chat over the teletype connection. ADISInit informs Tenex that this character is to be passed on unmolested; user programs should not alter this setting.

ADISInit will set four global variables which give the dimensions of the available Chat screen. The coordinate system is (SCREENXMIN, SCREENYMIN) at the lower left corner, and (SCREENXMAX, SCREENYMAX) at the upper right. The global variable ADISNREGIONS is set to the number of regions available; ADISNFONTS to the number of character fonts available.

**ADISCheck[ ]**

Checks the state of the display connection, and closes it if something is found to be out of order. If the connection is closed, ADISFILE is set to NIL.

**ADISClose[ ]**

Closes the connection cleanly, and returns the entire Chat display to "teletype simulation."

**ADISFlush[wait]**

This function causes any partially-filled output buffers to be transmitted to Chat. Most of the ADISxxx functions given below do not flush output buffers because further commands may be issued. Those functions which flush output are marked (Flush).
If the "wait" argument is non-NIL, the ADISFlush function will not return until
Chat has successfully processed all the protocol and characters already shipped
from MAXC. Waiting requires an end-to-end exchange of protocol.

ADISInputAvail[
This function returns T if some form of graphical input from Chat is waiting for
the LISP job, NIL otherwise. Note that this function does not look for "teletype
input," but rather for input events or timeouts (see below).

ADISFlushInput[]
This function discards all input waiting on the graphics connection.

ADISOFIELD
This global variable is set to the LISP file used for display protocol output to Chat;
it is NIL if no connection is open. ASCII characters may be displayed in the
"current region" simply by copying them into this file. (The handling of
characters is discussed in more detail below.)

ADISrOUT[n]
This function transmits the 8-bit byte n over the ADISOFIELD connection to Chat.
This function is provided so that programs may transmit escape sequences easily.
Programmers should be warned that if n> 127, Chat will interpret the byte as a
protocol command, with disastrous results.

2.2 Display output

Display output for Chat is always directed at some region. Each region has state associated
with it, and ADISxxx functions for changing the state:

ADISLimits: The limits are coordinates of the four edges of a rectangular box which
surrounds the region. If information destined to be displayed in the region lies
outside these limits, it is clipped off.

ADISSetX, ADISSetY (or ADISSetXY): A "current position" within the window. This
position is used to determine where the next character printed in the window
will be displayed: the left edge of the character will be placed at the current x
position, and the baseline of the character will be at the current y position.
Displaying the character will cause the current x position to be increased by the
width of the character.

ADISBold, ADISItalic, ADISSetCR, ADISSetLF, ADISSetTab: These state variables govern
the display of characters.

Chat provides a number of independent regions. All functions for changing region state
operate on one "current region," as set by ADISRegion. Because Chat stores the basic region
state internally, it will often not be necessary to change region variables often but simply to
switch regions using ADISRegion. ADISInit initializes all the regions with reasonable
defaults, sets the limits of region 0 to a teletype simulation area at the bottom of the screen,
sets the limits of other regions to the entire screen, and makes region 1 the "current region."

In the functions that follow, a REGION is defined by (RECORD REGION (X Y WIDTH
HEIGHT)), and defines a rectangular area of the screen X≤X≤X+WIDTH-1 and
Y≤Y≤Y+HEIGHT-1.

ADISRegion(n) (ResetSave)
Sets the current region to \( n \), \( 0 \leq n < \text{ADISNREGIONS} \). It generates an error if no such region exists.

\textbf{ADISLimits}(\text{region}) \text{ -or- ADISLimits}(l,r,b,t) \text{ (ResetSave)}

Sets the limits of the current region to be the rectangle defined by the argument (a \text{REGION} record). The four-argument version is an alternative for setting left, right, bottom or top limits individually: if an argument is a number, it is assumed to be a new value.

\textbf{ADISSetX}[x] \textbf{ADISSetY}[y] \textbf{ADISSetXY}[x;y]

These functions set the "current" \( x,y \) position of the current region. \textbf{ADISSetX} and \textbf{ADISSetY} return their arguments.

\textbf{ADISPRINI}[\text{obj}] \text{-and- ADISPRIN2}[\text{obj}]

These functions are exactly like \text{PRINI} and \text{PRIN2}, except the characters will appear on the Chat display. The first character is displayed at the current \( (x,y) \) position in the current region (i.e., its baseline is aligned with the \( y \) setting, and its left edge is aligned with the \( x \) setting), and the current position is advanced to \( (x+\text{width},y) \), where width is the width of the character. This process is repeated for all characters in the \text{PRINI} form of \text{obj}. (Note: Because \text{ADISInit} returns a \textit{file}, you are free to treat it as any other file in the LISP system. However, \text{ADISPRINI} is provided so that (1) if the LISP program is not running under Chat, no error is generated, and (2) for documentation in your program.) Bear in mind that characters will not be displayed until appropriate buffers in \text{MAXC} are flushed (see \text{ADISFlush}, above). See also "Escape Sequences," below.

\textbf{ADISBackup}[x]

This function permits "backspacing" from the current \( (x,y) \) position, and erasing the intervening region. The argument \( x \) specifies how much to back up, and is usually derived from a font width table in order to erase a character just displayed.

\textbf{ADISFont}[n] \text{ (ResetSave)}

Sets the font of the current region to \( n \), \( 0 \leq n < \text{ADISNFONTS} \); all characters subsequently typed to this connection will appear in the specified font. The available fonts are either declared to Chat when it is initialized or read in with \text{ADISReadFont}.

\textbf{ADISBold}[h] \textbf{ADISItalic}[h] \text{ (ResetSave)}

These functions turn on and off "bold" and "italic" features that will affect each character typed. If \( h \) is 'ON or T, subsequent characters will be bold or italic; otherwise the feature is turned off (but \( h=\text{NIL} \) will not turn it off because of \text{ResetSave} conventions).

\textbf{ADISSetCR}[x] \textbf{ADISSetLF}[\text{deltay}] \textbf{ADISSetTab}[\text{tabx}] \text{ (ResetSave)}

These functions can be used to control the interpretation of carriage-return, line-feed and tab characters. When a carriage-return is "displayed," the only effect is to set the current \( x \) to the value last given in \text{ADISSetCR}. When a line-feed is displayed, the value \( \text{deltay} \) is added to the current \( y \) coordinate. If the value of \( \text{tabx} \) is not zero, it is interpreted as a tab grid, relative to the carriage-return position (\( x \)). Receipt of a tab character will move the horizontal position to the next tab stop. If \( \text{tabx}=0 \), tab characters will not be subject to special interpretation, but will be "displayed." (Defaults; \( x=0; \text{deltay}=-12, \text{tabx}=0 \))

\textbf{ADISPrintMode}[\text{CharDisplayOp;ClearColor;Scroll}] \text{ (ResetSave)}

Sets additional details pertaining to character display. \text{CharDisplayOp} is the "directive" (see \text{ADISRegionOp}, below) to use for displaying characters. It defaults
to 1, which causes characters to be "painted" from the font description onto the screen.

If Scroll is T or ON or EXPAND, receipt of a line-feed will cause all information in the region to be scrolled: if the line-feed would make new information lie partly below the region, the entire region is scrolled up (by the amount set by ADISSetLF); if the line-feed would not cause the new line to lie off the bottom, scrolling moves information below the current position down and thereby opens up a new line.

If scrolling is enabled, a small region will need to be "cleared" after scrolling; the value of ClearColor (same conventions as gray in ADISRegionOp, below) governs how the cleared area should appear.

If Scroll is set to EXPAND, a line of text will be "expanded" before a new character is displayed in it. This operation involves first translating all information to the right of the current position (within the region) to the right by the width of the character, and then clearing out the small area as described above (the small area will be just wide enough for the character). After this "horizontal scroll," the new character will be displayed. This feature allows simple text "inserts" in a line.

The arguments are individually defaulted: numbers for CharDisplayOp and ClearColor are taken as new values; non-nil Scroll is taken as a new value.

ADISLineTo[xy;width]
Causes a line to be drawn in the current region from the current position to (x,y); the current position is then set to (x,y). The optional parameter width controls the width of the line. If width<0, the line will be exclusive or'ed with the information already on the display: this is can be used to erase an existing line.

ADISRegionOp[region;directive;altregion;gray]
This function causes region to be altered in one of several ways, governed by the values of directive, altregion and gray. Roughly speaking, directive tells what to do, and altregion and/or gray are arguments that together specify another region, the source region. The directive is itself the sum of two parts: a number that specifies the operation to perform, and a number that specifies the source-type. Thus directive= operation + source-type.

The operations are:

0 Replace. The source region is stored in the region.
1 Paint. The source region is "or"ed into the region.
2 Invert. The source region is "xor"ed with the region, and stored in the region.
3 Erase. The complement of the source region is "and"ed with the region.

The source-type specifies how the source region (mentioned in the above list of operations) is to be computed from the arguments:

0 The source is the altregion region of the screen.
4 The source is the complement of the altregion region of the screen.
8 The source region is the logical "and" of the altregion and the gray-region.
12 The gray-region is used.
Gray-region is a pseudo-region that covers the entire screen, and is filled uniformly with a pattern of 1's and 0's specified by the gray parameter (a 16-bit constant). The pattern is governed by the constant:

Simple cases:
0 White throughout
-1 Black throughout

General case:
The 16-bit constant is viewed as 4 4-bit bytes which define a 4x4 bit square pattern that is repeated throughout the entire screen. The first byte is for the top scan-line of the 4x4 square, the second for the second, etc.

For source-types 0 and 4 the source:REGION.WIDTH and source:REGION.HEIGHT are ignored and a simple transfer between equally-sized rectangles is performed. For source-types 0-4, the gray argument may be NIL; for type 12, the altregion argument may be NIL.

ADisScroll[region;delta;y;gray]
This command is a simplified form of the general region operations given below. It causes region to be scrolled up or down by the amount delta. The information "scrolled off" the region is lost. The gap at the bottom or top of the region will be set to the color specified by gray (see discussion above). Default gray=white.

ADisGetXY[]
This function returns (CONs currentX currentY) for the current region. The function is quite slow, as it must interrogate Chat for the current state of the region.

2.3 Input functions
Chat is capable of reporting interactions with various graphical input devices to the LISP program. The devices used in this way are a mouse, which has three keys and additionally steers the cursor, a five-finger keyset, and 7 uninterpreted keys on the keyboard. Normal "typing" on the keyboard is not noticed by these functions, but is instead transmitted to MAXC to serve as "teletype input" to the LISP program.

The functions deal with a single notion of "event," described by a LISP record: (RECORD EVENT (X Y BUTTONCHANGES BUTTONS OTHERBUTTONS ELAPSED)); where X and Y give the cursor location at the time of the event; BUTTONCHANGES is an 8-bit number that has bits on corresponding to the buttons that changed to cause the event (200Q is the left handset button, 100Q the next, etc., down to 10Q the rightmost; 4Q is the top or left mouse button; 1Q the middle, 2Q the bottom or right); BUTTONS gives the state of the buttons after the change that caused the event (a bit on implies that the button is depressed); OTHERBUTTONS is the present state of the various spare buttons on the keyboard (200Q=lock, 100Q=left shift, 40Q=ctrl, 10Q=right shift, 4Q=B3, 2Q=B1, 1Q=B2); ELAPSED gives the time that elapsed since the previous event (units of 1/27 second; maximum value is 255).

ADisButtonEnable[EventEnables;TimerStartEnables;TimerStopEnables;TimerInterval]
(Flush, ResetSave)
This function governs subsequent interpretation of button pushes. An "enable" is a 16-bit mask that defines whether action should be taken on down or up transitions of the buttons: it is (DownMask)*256+(UpMask), where each mask uses
the same bit assignments as for the **BUTTONS** entry in the **EVENT** record. For example, to enable for down transitions of all mouse buttons, **EventEnables** would be \(7 \times 2^8 + 0\).

In order to detect double clicks of keys, there is a timer facility. The idea is that **TimerStartEnables** describes what button transitions should start the timer. **TimerStopEnables** describes what button transitions should stop the timer. When the timer is started, it will run for \(\text{TimerInterval}/2^7\) seconds and then expire if it is not stopped in the interim. If the timer ever expires, a "timeout" event is generated, and the timer is stopped. (See **ADIEvent** for a discussion of timeout events.)

Null arguments except **EventEnables** default to 0.

**ADISStartTimer**[[timeout]]  (Flush)
This function starts the timer under program control. The argument is the amount of time (in units of \(1/2^7\) second) to wait. When that time expires (unless some other event resets the timer), a timeout event is generated.

**ADIEvent**[[wait;activateon;typein;oldevent]]
This function returns a button event, or **NIL** if there is none waiting. If **wait** is not **NIL**, the function will hang until an event is generated. The function returns an **EVENT** record describing the event, or **NIL** if a timeout event occurs. If **oldevent** is provided, it is an **EVENT** record that is smashed when reporting the new event.

If **activateon;typein** and **wait** are both non-**NIL**, the **LISP** job will hang waiting for either (1) an event to occur, in which case the **EVENT** record described above is returned, or (2) the user types some character on the keyboard, in which case **ADIEvent** returns **T**. (Note that there is a potential race, because **ADIEvent** must inform the Chat program to indeed activate if a character is typed. The character may be typed before the information is received by Chat, in which case the wakeup will not occur for that character.)

**ADISTypeOnEvent**[[c1;c2]]  (Flush, ResetSave)
When an event is generated by Chat, it is sometimes preferable if it signals the **LISP** program by typing a character in addition to sending the event description via protocol. This permits the **LISP** job to be blocked for terminal input; the character typed when an event occurs can activate a **READ** macro that calls **ADIEvent** to read the event details.

The arguments **c1** and **c2** are character codes for up to two characters that will be typed. If a character code is 0, it will not be typed. Null arguments except **c1** default to 0. Thus **ADISTypeOnEvent**[0] resets the feature.

**ADISReadState**[]  (Flush)
This function records the current state of the mouse and buttons, and returns an **EVENT** structure that describes it. It is absolutely independent of which buttons are enabled for events.

2.4 Miscellaneous functions

**ADISTTYRegion**[n]  (Flush)
Sets the "TTY region" to **n**. All characters transmitted from **MAXC** over the normal
teletype connection will be displayed in the specified region; ADISInit does
ADISTTYRegion(0). Because this is a region like any other, parameters such as
margin, font, limits, can all be set by ADISxxx functions. ADIS functions that
modify the TTY region are carefully synchronized with the network Telnet
connection so that the appearance of the TTY region can be carefully controlled.

**ADISReadFont[n;name]** (Flush)
This function tries to replace font n with a font of the given name read from the
Chat disk. Storage for the old font n, if any, is reclaimed. ADISReadFont updates
the data structure of available fonts (ADISNFONTS) appropriately. The function
returns T if the font appears to have been successfully read, otherwise NIL.

**ADISFontWidths[n;array]** (Flush)
This function pokes Chat to discover the parameters of the font numbered n. The
function returns NIL if the Chat program has no such font. Otherwise, the array
is filled with font widths, in units of screen resolution, and the function returns
(CONS height baseline), where height is the spacing, in scan-lines, between
baselines of consecutive lines of text in this font, and baseline is the height, above
the baseline, of the tallest character.

**ADISSync[]** (Flush)
This function completes a protocol exchange that ensures that Chat and the LISP
program are synchronized.

**ADISCursor[array;x;y;invertFlag]** (Flush, ResetSave)
This function sets the bit pattern in the cursor that follows the mouse position.
Array is an array of 16 16-bit numbers; the first is the pattern for the top
scan-line of the cursor, the second for the second, etc.

The optional parameters x and y are the location of the "point" of the cursor
within the 16x16 square (lower left corner is 0,0). These numbers are used for
two purposes: (1) to arrange that the "point" of the new cursor will lie precisely
where the "point" of the old cursor was at the instant the change was made; and
(2) to offset the X and Y coordinates reported in the EVENT record so that these
coordinates refer the the location of the "point" of the cursor.

The optional parameter invertFlag, if non-NIL, will cause the cursor bit pattern to
be set from array, but with the bits inverted.

**ADISCursorMove[dx;dy]** (Flush)
Nudges the current cursor position by (dx,dy).

**ADISCaret[region;rate;array;x;y]**
This function sets up a blinking caret at the current x,y position of the region
specified (only one caret is available -- putting it in a region will remove it from
any previous region). As the current x,y position changes, so will the location of
the caret. The appearance of the caret is controlled by the three arguments array,
x, and y; these have the same interpretation as for ADISCursor. Rate is the amount
of time (in 1/27th seconds) that is allowed to elapse before the caret state is
flipped (i.e., 2*Rate is the time for a complete cycle). Defaults: rate=13, x=0, y=0,

**ADISCaret[]** will disable the caret entirely.

**ADISData[region;array]** (Flush)
This function is available for writing arbitrary patterns into a region of the
display. The region argument specifies where the data should go (Restriction: X
and WIDTH must be multiples of 16). The array is interpreted as 16-bit numbers; the first WIDTH/16 elements of the array correspond to the top scan-line of data; the first word corresponds to the left-most word of the top scan-line.

3. Conclusion

These functions are still evolving. The design has been worked out in conjunction with Warren Teitelman, who has been extending the InterLisp user interface (DWIM, MASTERSCOPE, HELPSYS, HISTORY, etc.) to take maximum advantage of an interactive display terminal. The necessity of interposing a communications system between the interactive program (InterLisp) and the display and input devices has generated some of the unpleasant aspects of the design: when necessary, we have chosen solutions that make a cleaner user interface and a happier user rather than a cleaner programming interface and a happier programmer.

References


4. Chat operation

This document describes Chat version 1005P.6T.13D.

When running Chat, you must specify that display protocol is to be enabled. This can be accomplished by saying Chat/P or Chat/D to the command processor, or by putting the line "DISPLAYPROTOCOL: ON" in your User.Cm with other Chat entries. To exit Chat, use the <shift><Swat> convention.

Fonts are declared in User.Cm as follows: a line of the form "DISPLAY-FONT: FileName" is a font declaration. Numbers are associated with the fonts by the order in the file: the first is font 0, the second font 1, etc. The fonts must be in "strike" format; several fonts in this format are saved on the <ALTOFONTS> directory with extension .STRIKE.

The number of regions available to Chat can be altered by including a line of the form "DISPLAY-REGIONS: 6" in User.Cm.

The LISP functions documented here are contained in <SPROULL>ADIS.COM; the symbolics (should you need them) in <SPROULL>ADIS.

WARNING: Be very careful when attaching and detaching jobs that have Chat display connections open. If you re-attach to a LISP job that previously had connections open, and CONTINUE your LISP job, the connections are no longer usable because the Pup executive has timed them out. ADISCheck should be called to verify the state of the connection. After this call, it may be necessary to invoke ADISInit again. If this procedure is not followed, you may get traps with "IO Data Error" or some such message coming out of your LISP program!

Two last functions, dear to my heart:

ADISPress[file] (Flush)
This function writes a one-page Press file of the given name (on your Alto). The page contains a bit-map for the current contents of the Chat display area. WARNING: This function requires considerable quantities of disk space (about 130 pages per file), and may lead to errors while writing the file. Best use it only when your state is safe.

ADISPressMaxc[file;scaleFactor] (Flush)
This function is similar to ADISPress, but the file will be written on the connected MAXC directory. The scaleFactor defaults to 1.0, but can be set to any fraction. It will cause the Press file to contain directives to reduce the size of the image of the screen when it is printed. (Note that only the Slot/3100 will print Press files with scaleFactor other than 1.0.)

4.1 Efficiency and space

The ADIS protocol operations cost a certain amount in LISP function call and Tenex JSYS overhead; they also have a cost determined by the number of bytes of protocol commands that are sent to Chat. Thus we can express the communication cost in terms of the number of "characters" we could display by transmitting the same number of bits. Here are approximate numbers:
| ADlSRegion       | 4   |
| ADlSLimits       | 16  |
| ADlSSetX,ADlSSetY,ADlSFont | 5   |
| ADlSBold,ADlSItalic,ADlSSetCR,ADlSSetLF | 5   |
| ADlSLnTo         | 6   |
| ADlSRegionOp     | 13 or 21 (two-region variety) |
| ADlSScroll       | 34 in most cases |
| ADlSButtonEnable | 16  |
| ADlSTypeOnEvent  | 4   |
| ADlSCursor       | 43  |
| ADlSCursorMove   | 7   |

Space in the Alto is at a premium. At present, about 6700 words must be shared among all fonts and region descriptions. Sizes are:

<table>
<thead>
<tr>
<th>Region</th>
<th>34 words (always)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Helvetica8.Strike</td>
<td>570 words</td>
</tr>
<tr>
<td>Helvetica10.Strike</td>
<td>630 words</td>
</tr>
</tbody>
</table>