CHAPTER 20

INTERLISP-D DISPLAY-ORIENTED TOOLS

One of the greatest strengths of Interlisp-D is the window display system. Using this system, a number of the existing Interlisp tools have been extended, and some new ones developed. This chapter describes some of these tools.

20.1 DEDIT

DEdit is a structure oriented, modeless, display based editor for objects represented as list structures, such as functions, property lists, data values, etc. DEdit is an integral part of the standard Interlisp-D environment.

20.1.1 General Comments

DEdit is designed to be the user's primary editor for programs and data. To that end, it has incorporated the interfaces of the (older) teletype oriented Interlisp editor so the two can be used interchangeably. In addition, the full power of the teletype editor, and indeed the full Interlisp system itself, is easily accessible from within DEdit.

DEdit is structure, rather than character, oriented to facilitate selecting and operating on pieces of structure as objects in their own right, rather than as collections of characters. However, for the occasional situation when character oriented editing is appropriate, DEdit provides access to the Interlisp-D text editing facilities. DEdit is modeless, in that all commands operate on previously selected arguments, rather than causing the behavior of the interface to change during argument specification.

20.1.2 Operation

DEdit is normally called through of the following functions:

`(DF FN)` [NLambda NoSpread Function]
Calls DEdit on the definition of the function `FN`.

`(DV VAR)` [NLambda NoSpread Function]
Calls DEdit on the value of the variable `VAR`.

`(DP NAME PROP)` [NLambda NoSpread Function]
Calls DEdit on the property `PROP` of the atom `NAME`. If `PROP` is not given, the whole property list of `NAME` is edited.

(DC FILE) [NLambda NoSpread Function]
Calls DEdit on the file commands for the file FILE.

DEdit is normally installed as the default editor for all editing operations, including those invoked by other subsystems, such as the Programmer's Assistant and Masterscope. DEdit provides functions EF, EV and EP (analogous to the corresponding Dx functions) for conveniently accessing the teletype editor from within a DEdit context, e.g. from under a call to DEdit or if DEdit is installed as the default editor.

The default editor may be set with EDITMODE:

(EDITMODE NEWMODE) [Function]
If NEWMODE is non-NIL, sets the default editor to be DEdit (if NEWMODE is DISPLAY), or the teletype editor (if NEWMODE is TELETYPE). Returns the previous setting.

DEdit operates by providing an alternative, plug compatible definition of EDITL (DEDITL). The normal user entries operate by redefining EDITL and then calling the corresponding Edit function (i.e., DF calls EDITF etc). Thus, the normal Edit file package, spelling correction, etc. behavior is obtained.

If Edit commands are specified in a call to DEDITL (e.g., in calls to the editor from Masterscope), DEDITL will pass those commands to EDITL, after having placed a TTY: entry on EDITMACROS which will cause DEdit to be invoked if any interaction with the user is called for. In this way, automatic edits can be made completely under program control, yet DEdit's interactive interface is available for direct user interaction.

(RESETDEDIT) [Function]
Completely reinitializes DEdit. Closes all DEdit windows, so that the user must specify the window the next time DEdit is envoked. RESETDEDIT is also used to make DEdit recognize the new values of variables such as DEDITTYPEINCOMS, when the user changes them.

20.1.3 Interactive Operation

When DEdit is called for the first time, it prompts for an edit window, which is preserved and reused for later DEdits, and pretty prints the expression to be edited therein. (Note: The pretty printer ignores user PRETTYPRINTMACROS because they do not provide enough structural information during printing to enable selection.) A standard Interlisp-D scroll bar is set up on the left edge of the window and an edit command menu, which remains active throughout the edit, on the right edge. DEdit then goes into a select, command, execute loop, during which it yields control so that background activities, such as mouse commands in other windows, continue to be performed.

20.1.3.1 Selection

Selection in a DEdit window is as follows: the LEFT button selects the object being directly pointed at; the MIDDLE button selects the containing list; and the RIGHT button extends the current selection to the lowest common ancestor of that selection and the current position. The only things that may be pointed at are atomic objects (literal atoms, numbers, etc) and parentheses, which are considered to represent the list they delimit. White space is not selectable or editable.

When a selection is made, it is pushed on a selection stack which will be the source of operands for

DEdit commands. As each new selection pushes down the selections made before it, this stack can grow arbitrarily deep, so only the top two selections on the stack are highlighted on the screen. This highlighting is done by underscoring the topmost (most recent) selection with a solid black line and the second topmost selection with a dashed line. The patterns used were chosen so that their overlappings would be both visible and distinct, since selecting a sub-part of another selection is quite common.

Because one can invoke DEdit recursively, there may be several DEdit windows active on the screen at once. This is often useful when transferring material from one object to another (as when reallocating functionality within a set of programs). Selections may be made in any active DEdit window, in any order. When there is more than one DEdit window, the edit command menu (and the type-in buffer, see below) will attach itself to the most recently opened (or current) DEdit window.

20.1.3.2 Typein

Characters may be typed at the keyboard at any time. This will create a type-in buffer window which will position itself under the current DEdit window and do a `LISPXREAD` (which must be terminated by a right parenthesis or a return) from the keyboard. During the read, any character editing subsystem (such as `TTYIN`) that is loaded can be used to do character level editing on the typein. When the read is complete, the typein will become the current selection (top of stack) and be available as an operand for the next command. Once the read is complete, objects displayed in the type-in buffer can be selected from, scrolled, or even edited, just like those in the main window.

One can also give some editing commands directly into the typein buffer. Typing control-Z will interpret the rest of the line as a teletype editor command which will be interpreted when the line is closed. Likewise, ''control-S `OLD NEW` '' will substitute `NEW` for `OLD` and ''control-F `X`'' will find the next occurrence of `X`.

20.1.3.3 Shift-Selection

Often, significant pieces of what one wishes to type can be found in an active DEdit window. To aid in transferring the keystrokes that these objects represent into the typein buffer, DEdit supports shift-selection. Whenever a selection is made in the DEdit window with the left shift key down, the selection made is not pushed on the selection stack, but is instead *unread* into the keyboard input (and hence shows up in the typein buffer). A characteristically different highlighting is used to indicate when shift (as opposed to normal) selection is taking place.

Note that shift-selection remains active even when DEdit is not. Thus one can unread particularly choice pieces of text from DEdit windows into the typescript window.

20.1.3.4 Commands

A DEdit command is invoked by selecting an item from the edit command menu. This can be done either directly, using the `LEFT` mouse button in the usual way, or by selecting a subcommand. Subcommands are less frequently used commands than those on the main edit command menu and are grouped together in submenus ''under'' the command on the main menu to which they are most closely related. For example, the teletype editor defines six commands for adding and removing parentheses (defined in terms of transformations on the underlying list structure). Of these six commands, only two (inserting and

removing parentheses as a pair) are commonly used, so DEdit provides the other four as subcommands of the common two. The subcommands of a command are accessed by selecting the command from the commands menu with the MIDDLE button. This will bring up a menu of the subcommand options from which a choice can be made. Subcommands are agged in the list below with the name of the top level command of which they are options.

If one has a large DEdit window, or several DEdit windows active at once, the edit command window may be far away from the area of the screen in which one is operating. To solve this problem, the DEdit command window is a ''snuggle up'' menu. Whenever the TAB key is depressed, the command window will move over to the current cursor position and stay there as long as either the TAB key remains down or the cursor is in the command window. Thus, one can ''pull'' the command window over, slide the cursor into it and then release the TAB key (or not) while one makes a command selection in the normal way. This eliminates a great deal of mouse movement.

Whenever a change is made, the prettyprinter reprints until the printing stablizes. As the standard pretty print algorithm is used and as it leaves no information behind on how it makes its choices, this is a somewhat heuristic process. The Reprint command can be used to tidy the result up if it is not, in fact, ''pretty''.

All commands take their operands from the selection stack, and may push a result back on. In general, the rule is to select *target* selections rst and *source* selections second. Thus, a Replace command is done by selecting the thing to be replaced, selecting (or typing) the new material, and then buttoning the Replace command in the command menu. Using TOP to denote the topmost (most recent) element of the stack and NXT the second element, the DEdit commands are:

`After` [DEdit Command]
Inserts a copy of TOP after NXT .

`Before` [DEdit Command]
Inserts a copy of TOP before NXT .

`Delete` [DEdit Command]
Deletes TOP from the structure being edited. (A copy of) TOP remains on the stack and will appear, selected, in the edit bu er.

`Replace` [DEdit Command]
Replaces NXT with a copy of TOP obtained by substituting a copy of NXT wherever the value of the atom EDITEMBEDTOKEN (initially, the & character) appears in TOP . This provides an MBD facility, see Idioms below.

`Switch` [DEdit Command]
Exchanges TOP and NXT in the structure being edited.

`()` [DEdit Command]
Puts parentheses around TOP and NXT (which can, of course, be the same element).

`( in` [DEdit Command]
Subcommand of (). Inserts (before TOP (like the LI Edit command)

`) in` [DEdit Command]
Subcommand of (). Inserts) after TOP (like the RI Edit command)

`() out` [DEdit Command]
Removes parentheses from `TOP`.

`( out` [DEdit Command]
Subcommand of `() out`. Removes `(` from before `TOP` (like the `LO` Edit command)

`) out` [DEdit Command]
Subcommand of `() out`. Removes `)` from after `TOP` (like the `RO` Edit command)

`Undo` [DEdit Command]
Undoes last command.

`!Undo` [DEdit Command]
Subcommand of `Undo`. Undoes all changes since the start of this call on DEdit.

`?Undo` [DEdit Command]
`&Undo` [DEdit Command]
Subcommands of `Undo`. Allows selective undoing of other than the last command. Both of these commands bring up a menu of all the commands issued during this call on DEdit. When the user selects an item from this menu, the corresponding command (and if `&Undo`, all commands since that point) will be undone.

`Find` [DEdit Command]
Selects, in place of `TOP`, the first place after `TOP` which matches `NXT`. Uses the Edit subsystem's search routine, so supports the full wildcarding conventions of Edit.

`Swap` [DEdit Command]
Exchanges `TOP` and `NXT` on the stack, i.e. the stack is changed, the structure being edited isn't.

The following set of commands are grouped together as subcommands of `Swap` because they all affect the stack and the selections, rather than the structure being edited.

`Center` [DEdit Command]
Subcommand of `Swap`. Scrolls until `TOP` is visible in its window.

`Clear` [DEdit Command]
Subcommand of `Swap`. Discards all selections (i.e., ''clears'' the stack).

`Copy` [DEdit Command]
Subcommand of `Swap`. Puts a copy of `TOP` into the edit buffer and makes it the new `TOP`.

`Pop` [DEdit Command]
Subcommand of `Swap`. Pops `TOP` off the selection stack.

`Reprint` [DEdit Command]
Reprints `TOP`.

`Edit` [DEdit Command]
Runs DEdit on the definition of the atom `TOP` (or `CAR` of list `TOP`). Uses `TYPESOF` to determine what definitions exist for `TOP` and, if there is more than one, asks

the user, via menu, which one to use. (Note: DEdit caches each subordinate edit window in the window from which it was entered, for as long as the higher window is active. Thus, multiple DEdit commands do not incur the cost of repeatedly allocating a new window.) If TOP is de ned and is a non-list, calls INSPECT on that value. Edit also has a variety of subcommands which allow choice of editor (DEdit, Edit, TEdit, etc.) and whether to invoke that editor on the de nition of TOP or the form itself.

EditCom [DEdit Command]
Allows one to run arbitrary Edit commands on the structure being DEdited (there are far too many of these for them all to appear on the main menu). TOP should be an Edit command, which will be applied to NXT as the current Edit expression. On return to DEdit, the (possibly changed) current Edit expression will be selected as the new TOP. Thus, selecting some expression, typing (R FOO BAZ), and buttoning EditCom will cause FOO to be replaced with BAZ in the expression selected.

In addition, a variety of common Edit commands are available as subcommands of EditCom. Currently, these include ?=, GETD, CL, DW, REPACK, CAP, RAISE, and LOWER.

Break [DEdit Command]
Does a BREAKIN AROUND the current expression TOP. (See page 10.5.)

Eval [DEdit Command]
Evaluates TOP, whose value is pushed onto the stack in place of TOP, and which will therefore appear, selected, in the edit bu er.

Exit [DEdit Command]
Exits from DEdit (equivalent to Edit OK).

OK [DEdit Command]
Stop [DEdit Command]
Subcommands of Exit. OK exits without an error; STOP exits with an error. Equivalent to the Edit commands with the same names.

20.1.3.5 Multiple Commands

It is occasionally useful to be able to give several commands at once - either because one thinks of them as a unit or because the intervening reprettyprinting is distracting. The stack architecture of DEdit makes such multiple commands easy to construct - one just pushes whatever arguments are required for the complete suite of commands one has in mind. Multiple commands are speci ed by holding down the CONTROL key during command selection. As long as the CONTROL key is down, commands selected will not be executed, but merely saved on a list. Finally, when a command is selected without the CONTROL key down, the command sequence is terminated with that command being the last one in the sequence.

One rarely constructs long sequences of commands in this fashion, because the feedback of being able to inspect the intermediate results is usually worthwhile. Typically, just two or three step idioms are composed in this fashion. Some common examples are given in the next section.

20.1.3.6 Idioms

As with any interactive system, there are certain common idioms on which experienced users depend heavily. Not only is discovering the idioms of a new system tiresome, but in places the designer may have assumed familiarity with one or more of them, so not knowing them can make life quite unbearable. In the case of DEdit, many of these idioms concern easy ways to achieve the effects of specific commands from the Edit system, with which many users are already familiar. The DEdit idioms described below are the result of the experience of the early users of the system and are by no means exhaustive. In addition to those that each user will develop to fit his or her own particular style, there are many more to be discovered and you are encouraged to share your discoveries.

Because of the novel argument specification technique (postfix; target first) many of the DEdit idioms are very simple, but opaque until one has absorbed the ''target-source-command'' way of looking at the world. Thus, one selects where typein is to go before touching the keyboard. After typing, the target will be selected second and the typein selected on top, so that an `After`, `Before` or `Replace` will have the desired effect. If the order is switched, the command will try to change the typein (which may or may not succeed), or will require tiresome `Swap`ping or reselection. Although this discipline seems strange at first, it comes easily with practice.

Segment selection and manipulation are handled in DEdit by first making them into a sublist, so they can be handled in the usual way. Thus, if one wants to remove the three elements between `A` and `E` in the list `(A B C D E)`, one selects `B`, then `D` (either order), then makes them into a sublist with the ''`()`'' command (pronounced ''both in''). This will leave the sublist `(B C D)` selected, so a subsequent `Delete` will remove it. This can be issued as a single ''`(); Delete`'' command using multiple command selection, as described above, in which case the intermediate state of `(A (B C D) E)` will not show on the screen.

Inserting a segment proceeds in a similar fashion. Once the location of the insertion is selected, the segment to be inserted is typed as a list (if it is a list of atoms, they can be typed without parentheses and the `READ` will make them into a list, as one would expect). Then, the command sequence ''`After` (or `Before` or `Replace`); `() out`'' (given either as a multiple command or as two separate commands) will insert the typein and splice it in by removing its parentheses.

Moving an expression to another place in the structure being edited is easily accomplished by a delete followed by an insert. Select the location where the moved expression is to go to; select the expression to be moved; then give the command sequence ''`Delete`; `After` (or `Before` or `Replace`)''. The expression will first be deleted into the edit buffer where it will remain selected. The subsequent insertion will insert it back into the structure at the selected location.

Embedding and extracting are done with the `Replace` command. Extraction is simply a special case of replacing something with a subpiece of itself: select the thing to be replaced; select the subpart that is to replace it; `Replace`. Embedding also uses `Replace`, in conjunction with the ''embed token'' (the value of `EDITEMBEDTOKEN`, initially the single character atom `&`). Thus, to embed some expression in a `PROG`, select the expression; type ''`(PROG VARSLST &)`''; `Replace`.

`Switch` can also be used to generate a whole variety of complex moves and embeds. For example, switching an expression with typein not only replaces that expression with the typein, but provides a copy of the expression in the buffer, from where it can be edited or moved to somewhere else.

Finally, one can exploit the stack structure on selections to queue multiple arguments for a sequence of commands. Thus, to replace several expressions by one common replacement, select each of the

expressions to be replaced (any number), then the replacing expression. Now hit the Replace command as many times as there are replacements to be done. Each Replace will pop one selection o the stack, leaving the most recently replaced expression selected. As the latter is now a copy of the original source, the next Replace will have the desired e ect, and so on.

20.1.4 DEdit Parameters

There are several global variables that can be used to a ect various aspects of DEdit's operation. Although most have been alluded to above, they are summarized here for reference.

EDITEMBEDTOKEN [Variable]

Initially &. Used in both DEdit and the teletype editor to indicate the special atom used as the ''embed token''.

DEditLinger [Variable]

Initially T. The default behavior of the topmost DEdit window is to remain active on the screen when exited. This is occasionally inconvenient for programs that call DEdit directly, so it can be made to close automatically when exited by setting this variable to NIL.

DEDITTYPEINCOMS [Variable]

De nes the control characters recognized as commands during DEdit typein. Only accessed when DEdit is initialized, so DEdit should be reinitialized with (RESETDEDIT) if this is changed.

20.2 INTERACTIVE BITMAP EDITING

One important concept of the Interlisp- D display system is the idea of a bitmap, a rectangular array of bits. While working with the display system, it is extremely useful to be able to manipulate bitmaps, textures, and character bitmaps. The following functions provide an easy-to-use interactive editing facility for various types of bitmaps.

(EDITBM BITMAP) [Function]

If BITMAP is a bitmap, it is edited. If BITMAP is an atom whose value is a bitmap, its value is edited. If BITMAP is NIL, EDITBM asks for dimensions and creates a bitmap. If BITMAP is a region, that portion of (SCREENBITMAP) is used. If BITMAP is a window, it is brought to the top and its contents edited.

EDITBM sets up the bitmap being edited in an editing window. The editing window has two major areas: a gridded edit area in the lower part of the window and a display area in the upper left part. In the edit area, the left button will add points, the middle button will erase points. The right button provides access to the normal window commands to reposition and reshape the window. The actual size bitmap is shown in the display area.

If the bitmap is too large to t in the edit area, only a portion will be editable. This portion can be changed by scrolling both up and down in the left margin and left and right in the bottom margin. Pressing the middle button while in the display area will bring up a menu that allows global placement of

the portion of the bitmap being edited. To allow more of the bitmap to be editing at once, the window can be reshaped to make it larger or the GridSize_ command described below can be used to reduce the size of a bit in the edit area.

Pressing the middle button while not in either the edit area or the display area (i.e. while in the grey area in the upper right or in the title) will bring up a command menu. There are commands to stop editing, to restore the bitmap to its initial state and to clear the bitmap. Holding the middle button down over a command will result in an explanatory message being printed in the prompt window. The commands are described below:

OK — Copies the changed image into the original bitmap, stops the bitmap editor and closes the edit windows. The changes the bitmap editor makes during the interaction occur on a copy of the original bitmap. Unless the bitmap editor is exited via OK, no changes are made in the original.

Stop — Stops the bitmap editor without making any changes to the original bitmap.

Clear — Sets all or part of the bitmap to 0. Another menu will appear giving a choice between clearing the entire bitmap or just the portion that is in the edit area. The second menu also acts as a con rmation, since not selecting one of the choices on this menu results in no action being taken.

Reset — Sets all or part of the bitmap to the contents it had when EDITBM was called. As with the Clear command, another menu gives a choice between resetting the entire bitmap or just the portion that is in the edit area.

GridSize_ — Allows speci cation of the size of the editing grid. Another menu will appear giving a choice of several sizes. If one is selected, the editing portion of the bitmap editor will be redrawn using the selected grid size, allowing more or less of the bitmap to be edited without scrolling. The original size is chosen hueristically and is typically about 8. It is particularly useful when editing large bitmaps to set the edit grid size smaller than the original.

ShowAsTile — Tesselates the current bitmap in the upper part of the window. This is useful for determining how a bitmap will look if it were made the background (using the function CHANGEBACKGROUND). Note: The tiled display will not automatically change as the bitmap changes; to update it, use the ShowAsTile command again.

Paint — Puts the current bitmap into a window and call the window PAINT command on it. The PAINT command implements drawing with various brush sizes and shapes but only on an actual sized bitmap. The PAINT mode is left by pressing the RIGHT button and selecting the QUIT command from the menu. At this point, you will be given a choice of whether or not the changes you made while in PAINT mode should be made to the current bitmap.

CURSOR_ — Makes the lower left part of the bitmap become the cursor and will prompt you for the ‘‘hot spot’’.

The bitmap editing window can be reshaped to provide more or less room for editing. When this happens, the space allocated to the editing area will be changed to t in the new region.

Whenever the left or middle button is down and the cursor is not in the edit area, the section of the

display of the bitmap that is currently in the edit area is complemented. Pressing the left button while not in the edit region will put the lower left 16 x 16 section of the bitmap into the cursor for as long as the left button is held down.

(EDITSHADE SHADE) [Function]

Opens a window that allows the user to edit small textures (4 by 4) patterns. In the edit area, the left button adds bits to the shade and the middle button erases bits from the shade. The top part of the window is painted with the current texture whenever all mouse keys are released. Thus it is possible to directly compare two textures that di er by more than one pixel by holding a mouse key down until all changes are made.

If SHADE is a texture object, EDITSHADE starts with it, otherwise, it starts with white.

(EDITCHAR CHAR CODE FONT) [Function]

Calls the bitmap editor (EDITBM) on the bitmap image of the character CHAR CODE in the font FONT . CHAR CODE can be a character code (as returned by CHCON1) or an atom or string, in which case the rst character of CHAR CODE is used.

20.3 DISPLAY BREAK PACKAGE

The display break package allows easier access to the information available during a break, by modifying the function BREAK1 to use the window system. It is turned on in the standard system but can be turned o with the following function:

(WBREAK ONFL G) [Function]

If ONFL G is non-NIL, installs the display break package. If ONFL G is NIL, it uninstalls the display break package, which makes BREAK1 behave as in Interlisp-10. WBREAK returns T if the display break package was previously installed; NIL otherwise.

The display break package maintains a trace window and as many break windows as necessary. When a break occurs, a break window is brought up near the tty window of the process that broke and the terminal stream switched to it. The title of the break window is changed to give the name of the broken function, the reason for the break, and the depth of the break recursions. If a break occurs under a previous break, a new break window is created.

While in a break window, the middle button brings up a menu of break commands (EVAL, EVAL!, EDIT, revert, ^, OK, BT, BT!, and ?=). The commands BT and BT! bring up a backtrace menu beside the break window showing the frames on the stack. BT shows frames for which REALFRAMEP is T; BT! shows all frames. When one of the frames is selected from this menu, it is greyed and the function name and the variables bound in that frame (including local variables and PROG variables) are printed in the ‘‘backtrace frame’’ window. If the left button is used for the selection, only named variables are printed. If the middle button is used, all variables are printed (variables without names will appear as *var*N). The ‘‘backtrace frame’’ window is an inspect window (see page 20.12). In this window, the left button can be used to select the name of the function, the names of the variables or the values of the variables.

After selecting an item, the middle button brings up a command menu of commands that apply to the

selected item. If the function name is selected, a choice of editing the function or seeing the compiled code with INSPECTCODE will be given. If a variable name is selected, the command SET will be offered. Selecting SET will READ a value and set the selected to the value read. (Note: The inspector will only allow the setting of named variables. Even with this restriction it is still possible to crash the system by setting variables inside system frames. It is recommended that you exercise caution in setting variables in other than your own code.) If the item selected is a value, the inspector will be called on the selected value.

The internal break variable LASTPOS is set to the selected frame of the backtrace menu so that the normal break commands EDIT, revert, and ?= work on the currently selected frame. The commands EVAL, revert, ^, OK, and ?= in the break menu cause the corresponding commands to be ''typed in.'' This means that these break commands will not have the intended effect if characters have already been typed in.

The operation of the display break package is controlled by the following variables:

MaxBkMenuWidth [Variable]
MaxBkMenuHeight [Variable]
: The variables MaxBkMenuWidth (default 125) and MaxBkMenuHeight (default 300) control the maximum size of the backtrace menu. If this menu is too small to contain all of the frames in the backtrace, it is made scrollable in both vertical and horizontal directions.

AUTOBACKTRACEFLG [Variable]
: If the variable AUTOBACKTRACEFLG is non-NIL (default is NIL), then on error breaks the command BT is executed automatically.

BACKTRACEFONT [Variable]
: The backtrace menu is printed in the font BACKTRACEFONT, which is initially Gacha 8.

CLOSEBREAKWINDOWFLG [Variable]
: The system normally closes break windows after the break is exited. If CLOSEBREAKWINDOWFLG is NIL, break windows will not be closed on exit. Note: In this case, the user must close all break windows.

BREAKREGIONSPEC [Variable]
: Break windows are positioned near the tty window of the broken process, as determined by the variable BREAKREGIONSPEC. The value of this variable is a region whose LEFT and BOTTOM are an offset from the LEFT and BOTTOM of the tty window. The WIDTH and HEIGHT of BREAKREGIONSPEC determine the size of the break window.

TRACEWINDOW [Variable]
: The trace window, TRACEWINDOW, is used for tracing functions. It is brought up when the first tracing occurs and stays up until the user closes it. TRACEWINDOW can be set to a particular window to cause the tracing formation to print out there.

TRACEREGION [Variable]
: The trace window is first created in the region TRACEREGION.

20.4 THE INSPECTOR

The Inspector provides a display-oriented facility for looking at and changing arbitrary Interlisp-D data structures. The inspector can be used to inspect all user datatypes and many system datatypes (although some objects such as numbers have no inspectable structure). The inspector displays the field names and values of an arbitrary object in a window that allows setting of the properties and further inspection of the values. This latter feature makes it possible to ''walk'' around all of the data structures in the system at the touch of a button. In addition, the inspector is integrated with the break package to allow inspection of any object on the stack and with the display and teletype structural editors to allow the editors to be used to ''inspect'' list structures and the inspector to ''edit'' datatypes.

The underlying mechanisms of the data inspector have been factored to allow their use as specialized editors in user applications. This functionality is described at the end of this section.

Note: Currently, the inspector does *not* have UNDOing. Also, variables whose values are changed will not be marked as such.

20.4.1 Inspect Windows

An inspect window displays two columns of values. The lefthand column lists the property names of the structure being inspected. The righthand column contains the values of the properties named on the left. For variable length data such as lists and arrays, the ''property names'' are numbers from 1 to the length of the inspected item and the values are the corresponding elements. For arrays, the property names are the array element numbers and the values are the corresponding elements of the array.

For large lists or arrays, or datatypes with many fields, the initial window may be too small to contain all of them. In these cases, the unseen elements can be scrolled into view (from the bottom) or the window can be reshaped to increase its size.

In an inspect window, the LEFT button is used to select things, the MIDDLE button to invoke commands that apply to the selected item. Any property or value can be selected by pointing the cursor directly at the text representing it, and clicking the LEFT button. There is one selected item per window and it is marked by having its surrounding box inverted.

The commands offered by the MIDDLE button depend on whether the selection is a property or a value. If the selected item is a value, the commands provide different ways of inspecting the selected structure. The exact commands that are given depend on the type of the value. If the value is a litatom, the commands are the types for which the atom has definitions as determined by HASDEF. Some typical commands are:

FNS	Edit the definition of the selected litatom.
VARS	Inspect the value.
PROPS	Inspect the property list.

If the value is a list, there will be choice of how to inspect the list:

`Inspect`	Opens an inspect window in which the properties are numbers and the values are the elements of the list.
`TtyEdit`	Calls the teletype structural editor on the list.
`DisplayEdit`	Calls the display editor on the list.
`AsPList`	(If the list is in P-list form) Inspects the list as a property list.
`AsAList`	(If the list is in `ASSOC` list form) Inspects the list as an association-list.
`AsRecord`	Brings up a submenu with all of the `RECORD`s in the system and inspect the list with the one chosen.
`"a record type"`	(If the `CAR` is the name of a `TYPERECORD`) Inspects the list as the record of the type named in its `CAR`.

If the value is neither a litatom or a list, the only command is `Inspect`, which opens an inspector window onto the selected value.

If the selected item is a property, the user will be asked for a new value and the selected property will be set to the result of evaluating the read form. The evaluation of the read form and the replacement of the selected item property will appear as their own history events and are individually undoable. Properties of system datatypes cannot be set. (There are often consistency requirements which can be inadvertently violated in ways that crash the system. This may be true of some user datatypes as well.)

20.4.2 Calling the Inspector

The inspector can be called directly, by using the function `INSPECT`:

`(INSPECT OBJECT ASTYPE WHERE )` [Function]

Creates an inspect window onto `OBJECT` . If `ASTYPE` is given, it will be taken as the record type of `OBJECT` . This allows records to be inspected with their property names. If `ASTYPE` is `NIL`, the data type of `OBJECT` will be used to determine its property names in the inspect window.

`WHERE` speci es the location of the inspect window. If `WHERE` is `NIL`, the user will be prompted for a location. If `WHERE` is a window, it will be used as the inspect window. If `WHERE` is a region, the inspect window will be created in that region of the screen. If `WHERE` is a position, the inspect window will have its lower left corner at that position on the screen.

`INSPECT` returns the inspect window onto `OBJECT` , or `NIL` if no inspection took place.

There are several ways to open an inspect window onto an object. In addition to calling `INSPECT` directly, the inspector can also be called by buttoning an `Inspect` command inside an existing inspector window. Finally, if a non-list is edited with `EDITV`, the inspector is called. This also causes the inspector to be called by the `Dedit` command from the display editor or the `EV` command from the standard editor if the selected piece of structure is a non-list.

(INSPECTCODE FN) [Function]
Opens a window and displays the compiled code of the function FN using PRINTCODE. The window is scrollable.

20.4.3 Choices Before Inspection

For some datatypes there is more than one aspect that is of interest or more than one method of inspecting the object. In these cases, the inspector will bring up a menu of the possibilities and wait for the user to select one.

For litatoms, the choice includes inspecting its value, its de nition, its property list, its MACRO or any other aspect returned from TYPESOF. For BITMAPs, the choice is between inspecting the bitmap's contents with the bitmap editor (EDITBM) or inspecting the bitmap's elds. For LISTPs, the choice is how to inspect it and is between a one level inspector, the teletype editor (EDITE) or the display editor (DEDIT).

20.4.4 Redisplaying an Inspect Window

An inspect window is *not* automatically updated when the structure it is inspecting is changed. The inspect window can be updated by selecting the ''redisplay'' command from the menu brought up by pressing the MIDDLE button in the title of the window. The ''redisplay'' command will cause the values of the properties to be re-fetched from the structure and redisplayed.

20.4.5 Interaction With the Display Break Package

The display break package knows about the inspector in the sense that the backtrace frame window is an inspect window onto the frame selected from the back trace menu during a break. Thus you can call the inspector on an object that is bound on the stack by selecting its frame in the back trace menu, selecting its value with the LEFT button in the back trace frame window, and selecting the inspect command with the MIDDLE button in the back trace frame window. The values of variables in frames can be set by selecting the variable name with the LEFT button and then the ''Set'' command with the MIDDLE button.

Note: The inspector will only allow the setting of named variables. Even with this restriction it is still possible to crash the system by setting variables inside system frames. Exercise caution in setting variables in other than your own code.

20.4.6 Controlling the Amount Displayed During Inspection

The amount of information displayed during inspection can be controlled using the following variables:

MAXINSPECTCDRLEVEL [Variable]
The inspector prints only the rst MAXINSPECTCDRLEVEL elements of a long list, and will make the tail containing the unprinted elements the last item. The last item can be inspected to see further elements. Initially 50.

MAXINSPECTARRAYLEVEL [Variable]

The inspector prints only the first MAXINSPECTARRAYLEVEL elements of an array. The remaining elements can be inspected by calling the function (INSPECT/ARRAY ARRAY BEGINOFFSET) which inspects the BEGINOFFSET through the BEGINOFFSET + MAXINSPECTARRAYLEVEL elements of ARRAY. Initially 300.

INSPECTALLFIELDSFLG [Variable]

If INSPECTALLFIELDSFLG is T, the inspector will show computed fields (ACCESSFNS) as well as regular fields for structures that have a record definition. Initially T.

20.4.7 Inspect Macros

The Inspector can be extended to inspect new structures and datatypes by adding entries to the list INSPECTMACROS. An entry should be of the form (OBJECTTYPE . INSPECTINFO). OBJECTTYPE is used to determine the types of objects that are inspected with this macro. If OBJECTTYPE is a litatom, the INSPECTINFO will be used to inspect items whose type name is OBJECTTYPE. If OBJECTTYPE is a LIST of the form (FUNCTION DATUM-PREDICATE), DATUM-PREDICATE will be APPLYed to the item and if it returns non-NIL, the INSPECTINFO will be used to inspect the item.

INSPECTINFO can be one of two forms. If INSPECTINFO is a litatom, it should be a function that will be applied to three arguments (the item being inspected, OBJECTTYPE, and the value of WHERE passed to INSPECT) that should do the inspection. If INSPECTINFO is not a litatom, it should be a list of (PROPERTIES FETCHFN STOREFN PROPCOMMANDFN VALUECOMMANDFN TITLECOMMANDFN TITLE SELECTIONFN WHERE PROPPRINTFN) where the elements of this list are the arguments for INSPECTW.CREATE, described below. From this list, the WHERE argument will be evaluated; the others will not. If WHERE is NIL, the value of WHERE that was passed to INSPECT will be used.

Examples:

The entry ((FUNCTION MYATOMP) PROPNAMES GETPROP PUTPROP) on INSPECTMACROS would cause all objects satisfying the predicate MYATOMP to have their properties inspected with GETPROP and PUTPROP. In this example, MYATOMP should make sure the object is a litatom.

The entry (MYDATATYPE . MYINSPECTFN) on INSPECTMACROS would cause all datatypes of type MYDATATYPE to be passed to the function MYINSPECTFN.

20.4.8 INSPECTWs

The inspector is built on the abstraction of an INSPECTW. An INSPECTW is a window with certain window properties that display an object and respond to selections of the object's parts. It is characterized by an object and its list of properties. An INSPECTW displays the object in two columns with the property names on the left and the values of those properties on the right. An INSPECTW supports the protocol that the LEFT mouse button can be used to select any property name or property value and the MIDDLE button calls a user provided function on the selected value or property. For the Inspector application, this function puts up a menu of the alternative ways of inspecting values or of the ways of setting properties. INSPECTWs are created with the following function:

(INSPECTW.CREATE DATUM PROPERTIES FETCHFN STOREFN PROPCOMMANDFN VALUECOMMANDFN TITLECOMMANDFN TITLE SELECTIONFN WHERE PROPPRINTFN) [Function]

Creates an INSPECTW that views the object DATUM . If PROPERTIES is a LISTP, it is taken as the list of properties of DATUM to display. If PROPERTIES is an ATOM, it is APPLYed to DATUM and the result is used as the list of properties to display.

FETCHFN is a function of two arguments (OBJECT PROPERTY) that should return the value of the PROPERTY property of OBJECT . The result of this function will be printed (with PRIN2) in the INSPECTW as the value.

STOREFN is a function of three arguments (OBJECT PROPERTY NEWVALUE) that changes the PROPERTY property of OBJECT to NEWVALUE . It is used by the default PROPCOMMANDFN and VALUECOMMANDFN to change the value of a property and also by the function INSPECTW.REPLACE (described below). This can be NIL if the user provides command functions which do not call INSPECTW.REPLACE . Each replace action will be a separate event on the history list. Users are encouraged to provide UNDOable STOREFNs.

PROPCOMMANDFN is a function of three arguments (PROPERTY OBJECT INSPECTW) which gets called when the user presses the MIDDLE button and the selected item in the INSPECTW is a property name. PROPERTY will be the name of the selected property, OBJECT will be the datum being viewed, and INSPECTW will be the window. If PROPCOMMANDFN is a string, it will get printed in the PROMPTWINDOW when the MIDDLE button is pressed. This provides a convenient way to notify the user about disabled commands on the properties. DEFAULT.INSPECTW.PROPCOMMANDFN , the default PROPCOMMANDFN , will present a menu with the single command Set on it. If selected, the Set command will read a value from the user and set the selected property to the result of EVALuating this read value.

VALUECOMMANDFN is a function of four arguments (VALUE PROPERTY OBJECT INSPECTW) that gets called when the user presses the MIDDLE button and the selected item in the INSPECTW is a property value. VALUE will be the selected value (as returned by FETCHFN), PROPERTY will be the name of the property VALUE is the value of, OBJECT will be the datum being viewed, and INSPECTW will be the INSPECTW window. DEFAULT.INSPECTW.VALUECOMMANDFN , the default VALUECOMMANDFN , will present a menu of possible ways of inspecting the value and create a new Inspect window if one of the menu items is selected.

TITLECOMMANDFN is a function of two arguments (INSPECTW OBJECT) which gets called when the user presses the MIDDLE button and the cursor is in the title or border of the inspect window INSPECTW . This command function is provided so that users can implement commands that apply to the entire object. The default TITLECOMMANDFN (DEFAULT.INSPECTW.TITLECOMMANDFN) presents a menu with the single command Redisplay and, if it is selected, redisplays INSPECTW (using INSPECTW.REDISPLAY , described below).

TITLE specifies the title of the window. If TITLE is NIL, the title of the window will be the printed form of DATUM followed by the string ‘‘ Inspector’’. If TITLE is the litatom DON'T, the inspect window will not have a title. If TITLE is any other litatom, it will be applyed to the DATUM and the potential inspect window (if it is known). If this result is the litatom DON'T, the inspect window will not have a title; otherwise the result will be used as a title. If TITLE is not a litatom, it will be used as the title.

SELECTIONFN is a function of three arguments (PROPERTY VALUEFLG INSPECTW) which gets called when the user releases the left button and the cursor is on one of the items. The SELECTIONFN allows a program to take action on the user’s selection of an item in the inspect window. At the time this function is called, the selected item has been ‘‘selected’’. The function INSPECTW.SELECTITEM (described below) can be used to turn off this selection. PROPERTY will be the name of the property of the selected item.

VALUEFLG will be NIL if the selected item is the property name; T if the selected item is the property value.

WHERE indicates where the inspect window should go. Its interpretation is described in INSPECT (page 20.13).

If non-NIL, PROPPRINTFN is a function of two arguments (PROPERTY DATUM) which gets called to determine what to print in the property place for the property PROPERTY. If PROPPRINTFN returns NIL, no property name will be printed and the value will be printed to the left of the other values.

An inspect window uses the following window property names to hold information: DATUM, FETCHFN, STOREFN, PROPCOMMANDFN, VALUECOMMANDFN, SELECTIONFN, PROPPRINTFN, INSPECTWTITLE, PROPERTIES, CURRENTITEM and SELECTABLEITEMS.

(INSPECTW.REDISPLAY INSPECTW PROPERTY _) [Function]

Updates the display of the objects being inspected in INSPECTW. If PROPERTY is a property name or a list of property names, only those properties are updated. If PROPERTY is NIL, all properties are redisplayed. This function is provided because inspect windows do not automatically update their display when the object they are showing changes.

This function is called by the Redisplay command in the title command menu of an INSPECTW.

(INSPECTW.REPLACE INSPECTW PROPERTY NEWVALUE) [Function]

Uses the STOREFN of the inspect window INSPECTW to change the property named PROPERTY to the value NEWVALUE and updates the display of PROPERTY's value in the display. This provides a functional interface for user PROPCOMMANDFNs.

(INSPECTW.SELECTITEM INSPECTW PROPERTY VALUEFLG) [Function]

Sets the selected item in an inspect window. The item is inverted on the display and put on the window property CURRENTITEM of INSPECTW. If INSPECTW has a CURRENTITEM, it is deselected. PROPERTY is the name of the property of the selected item. VALUEFLG is NIL if the selected item is the property name; T if the selected item is the property value. If PROPERTY is NIL, no item will be selected. (This provides a way of deselecting items.)

20.5 CHAT

CHAT is a ''remote terminal'' facility, that allows one to communicate with other machines while inside Interlisp-D. The function CHAT sets up a ''Chat connection'' to a remote machine, so that everything you type is sent to the a remote machine, and everything the remote machine prints is displayed in a ''Chat window''. The remote machine must support the Pup Telnet protocol.

Multiple simultaneous Chat connections are possible. To switch between typing to different Chat connections, simply button within the Chat window you want to use. CHAT prompts for a new window for each new connection, except that it saves the first window to reuse once the connection in that window is closed (other windows just go away when their connections are closed).

CHAT behaves as if its Chat window is a Datamedia- 2500 terminal of the dimensions determined by the size of the window. Hence, you can talk to hosts that supply Datamedia service and expect something reasonable to happen. If the host does not pay attention to the CHAT terminal speci cation protocol, or you go through that host to another host, you may need to inform the host of the dimensions of your ‘‘screen’’; these are given in the title bar of the chat window. The font should be Gacha10 or other xed- width font for proper Datamedia emulation.

(CHAT HOST LOGOPTION INITSTREAM WINDO W _) [Function]

Opens a Chat connection to HOST , or to the value of DEFAULTCHATHOST. If HOST requires login, as determined by whether it responds to the ‘‘where is user’’ protocol, CHAT supplies a login sequence, or if it determines that you have a single detached job, an attach sequence. If you have more than one detached job, it simply performs a WHEREIS command for you and allows you to select the job. You may alternatively specify one of the following values for LOGOPTION :

LOGIN	Always perform a login.
ATTACH	Always perform an attach. This will fail if you do not have exactly one detached job.
GUEST	Login as user GUEST, password GUEST.
NONE	Do not attempt to login or attach.

If INITSTREAM is supplied, it is either a string or the name of a le whose contents will be read as typein. When the string/ le is exhausted, input is taken from T.

If WINDO W is supplied, it is a window to use for the connection; otherwise, the user is prompted for a window.

While CHAT is in control, all Lisp interrupts are turned o , so that control characters can be transmitted to the remote host.

Commands can be given to an active Chat connection by bugging the MIDDLE button in the Chat window to get a command menu. Current commands are:

Close	Close this connection. Once the connection is closed, control is handed over to the main tty window. Closes the window unless this is the primary Chat window.
Suspend	Same as Close, but always leaves the window open.
New	Closes the current connection and prompts for a new host to which to open a connection in the same window.
Freeze	Hold typeout from this Chat window. Bugging the window in any way releases the hold. This is most useful if you want to switch to another, overlapping window and there is typeout in this window that would compete for screen space.
Dribble	Open a typescript le for this Chat connection (closing any previous dribble le for the window). The user is prompted for a le name; a name of NIL just closes the old dribble le.
Input	Prompts for a le to take input from. When the end of the le is reached, input

reverts to T.

Clear — Clears the window and resets the simulated terminal to its default state. This is useful if undesired terminal commands have been received from the remote host that place the simulated terminal into a funny state.

In an inactive Chat window, the MIDDLE button brings up a menu of one item, ReConnect, whose selection reopens a connection to the same host as was last in the window. This is the primary motivation for the Suspend menu command. A new Chat connection can also be opened from the Background menu.

The mouse button LEFT, when inside CHAT, holds output as long as the button is down. Holding down MIDDLE coincidentally does this, too, but not on purpose: since the menu handler does not yield control to other processes, it is possible to kill the connection by keeping the menu up too long.

Chat windows are a little bit knowledgable about window operations. If you reshape a Chat window, Chat informs your partner of the new dimensions. And if you close the window, the connection is also closed.

The following variables control aspects of Chat's behavior:

CHAT.DISPLAYTYPE [Variable]
The type of display (a number) that Chat should tell the remote host the user is on. If Datamedia emulation is desired, this variable should be set to the number corresponding to the terminal type Datamedia for the remote host. If the remote host does not respond to the terminal type protocol in Pup Telnet, this variable is irrelevant.

CHAT.ALLHOSTS [Variable]
A list of host names, as uppercase litatoms, that the user desires to Chat to. Chatting to a host not on the list adds it to the list. These names are placed in the menu that the background Chat command prompts with.

CLOSECHATWINDOWFLG [Variable]
If true, every Chat window is closed on exit. If NIL, the initial setting, then the primary Chat window is not closed.

DEFAULTCHATHOST [Variable]
The host to which CHAT connects when it is called with no HOST argument.

CHAT.FONT [Variable]
If non-NIL, the font that Chat windows are created with. If CHAT.FONT is NIL, Chat windows are created with (DEFAULTFONT 'DISPLAY).

20.6 THE TEDIT TEXT EDITOR

TEdit is a window-based, modeless text editor, capable of handling fonts and some rudimentary formatting. Text is selected with the mouse, and all editor operations act on the current selection.

The top-level entry to TEdit is:

(TEDIT TEXT WINDOW DONTSPAWN PROPS) [Function]

TEXT may be a (litatom) file name, an open STREAM, a string, or an arbitrary [MKSTRING-able] Lisp object. The text is displayed in an editing window, and may be edited there. If TEXT is other than a file name, a STREAM, or a string, TEDIT will call MKSTRING on it, and let you edit the result.

If WINDOW is NIL, you will be prompted to create a window. If WINDOW is non-NIL, TEDIT will use it as the window to edit in. If WINDOW has a title, TEDIT will preserve it; otherwise, TEDIT will provide a descriptive title for the window.

TEDIT will normally spawn a new process to run the edit, so you can edit in parallel with other work; indeed, it is possible to have several editing windows active on the screen. To prevent a new process from being created, call TEDIT with DONTSPAWN set to T.

PROPS is a prop-list-like collection of properties which control the editing session. The following options are possible:

FONT	The default font to be used in the edit window.
QUITFN	A function to call when the user Quits.
LOOPFN	A function to be called each time thru the character-read loop.
CHARFN	A function to be called for each character typed in.
SELFN	A function to be called each time a mouse selection is made in this edit window.
TERMSA	If you want characters displayed other than TEdit's default way, set this to a character table.
READONLY	If this atom is present *anywhere* in the list of PROPS, then the edit window will be read-only, i.e., you can only shift-select in it.
SEL	Tells what text should be selected initially. This can be a SELECTION (see below) describing the selected text, or a character number, or a two-element list of first character number and number of characters to select.
MENU	Describes the menu to be displayed when the MIDDLE mouse button is pressed in the edit window's title region. If it is a MENU, that menu will appear. If it is a list of menu items, a new menu will be constructed.
AFTERQUITFN	A function to be called *after* TEdit has quit. This can be used for cleanup of side-effects by TEdit client programs.

REGION	A window-relative region; TEdit will use only that portion of the window to display text &c. This is for people who want TEdit for lling in forms, etc.
TITLEMENUFN	A function to get called instead of bringing up the usual TEdit command menu when the user LEFT- or MIDDLE-buttons in the edit window's title region.

20.6.1 Selecting Text

TEdit works by operating on ''selected'' pieces of text. Selected text is highlighted in some way, and may have a caret ashing at one end. Insertions go where the caret is; deletion and other operations are applied to the currently selected text.

Text is selected using the mouse. There are two regions within an edit window: The area containing text, and a ''line bar'' just inside the left edge of the window. While the mouse is inside the text region, the cursor is the normal up-and-left pointing arrow. When the cursor moves into the line bar, it changes to an up-and-right pointing arrow. Which region the mouse is in determines what kind of selection happens:

The LEFT mouse button always selects the smallest things. In the text region, it selects the character you're pointing at; in the line bar, it selects the single line you're pointing at.

The MIDDLE mouse button selects larger things. In the text region, it selects the word the cursor is over, and in the line bar it selects the paragraph the cursor is next to.

The RIGHT button always extends a selection. The current selection is extended to include the character/word/line/paragraph you are now pointing at. For example, if the existing selection was a whole-word selection, the extended selection will also consist of whole words.

There are special ways of selecting text which carry an implicit command with them:

If you hold the CTRL key down while selecting text, the text will be shown white-on-black. When you release the CTRL key, the selected text will be deleted. You can abort a CTRL-selection: Hold down a mouse button, and release the CTRL key. Then release the mouse button.

Holding the SHIFT key down while making a selection causes it to be a ''copy-source'' selection. A copy source is marked with a dashed underline. Whatever is selected as a copy source when the SHIFT key is released will be copied to where the caret is. This even works to copy text from one edit window to another. You can abort a copy: Hold down a mouse button, and release the SHIFT key. Then release the mouse button.

Holding the CTRL and SHIFT keys down while making a selection causes it to be a ''move'' selection, which is marked by making it veverse video. Whatever is selected as a ''move'' source when the CTRL and SHIFT keys are released will be moved to where the caret is. This even works to move text from one edit window to another. You can abort a move: Hold down a mouse button, and release the CTRL and SHIFT keys. Then release the mouse button. If the variable TEDIT.BLUE.PENDING.DELETE is non-NIL, extending a selection will display the selection as white-on-black. The next time something is typed, the selected text will be deleted rst.

20.6.2 Editing Operations

Inserting text: Except for command characters, whatever is typed on the keyboard gets inserted where the caret is. The BS key and control-A both act as a backspace, deleting the character just before the caret. Control-W is the backspace-word command.

Deleting Text: Hitting the DEL key causes the currently-selected text to be deleted. Alternatively, you can use the CTRL-selection method described above.

Copying Text: Use SHIFT-selection, as described above.

Moving Text: Use CTRL-SHIFT-selection.

Undoing an edit operation: The top blank key is the Undo key. It will undo the most recent edit command. Undo is itself undo-able, so you can never back up more than a single command.

Redoing an edit operation: The ESC key is the Redo key. It will redo the most recent edit command on the current selection. For example, if you insert some text, then select elsewhere, hitting ESC will insert a copy of the text in the new place also. If the last command was a delete, Redo will delete the currently-selected text; if it was a font change, the same change will be applied to the current selection.

The command menu: You can get command menus by moving into the edit window's title region and hitting the RIGHT or MIDDLE mouse buttons. RIGHT gets the usual menu of window commands. MIDDLE gets a menu of editor commands:

Put	Causes an updated version of the file to be written. Tedit will ask you for a file name, offering the existing name (if any) as the default.
Get	Lets you read in a new file to edit, *without saving the one you were working on.* You'll be asked for a file name in the prompt window.
Include	Lets you copy the contents of a file into the edit window, inserting it where the caret is.
Quit	Causes the editor to stop without updating the file you're editing. If you haven't saved your changes, you'll be asked to confirm this.
Find	Asks for a search string, then hunts from the caret toward the end of document for a match. Selects the first match found; if there is none, nothing happens.
Substitute	Asks for a search string and a replacement string. Within the current selection, all instances of the search string ware replaced by the replacement string. If you wish, TEdit will ask you to confirm each replacement before actually doing it.
Looks	Changes the character looks of the selected characters: The font, character size, and face (bold, italic, etc.). Three menus will pop up in sequence: One to select the font name, one to select the face, and one to select the size. You may select an option in each menu. If, for example, you want to leave the character size alone, just click the mouse outside the size menu. In general, any aspect of the character looks that you don't change will remain the same.
Hardcopy	Prints the document to your default press or InterPress printer, with 1 inch margins

all around. The function PRINTERMODE controls which kind of printer TEdit will send to.

Press File — Creates a Press or InterPress file of the document, with 1 inch margins all around. The file format is also controlled by PRINTERMODE.

20.6.3 TEdit Functional Interface

The Text Stream

TEdit keeps a STREAM which describes the current state of the text you're editing. You can use most of the usual stream operations on that stream: BIN, SETFILEPTR, GETFILEPTR, GETEOFPTR, BACKBIN, and PEEKBIN do the usual things. BOUT inserts a character in the stream just in front of the next character you'd read if you BINned. You can get the stream by (WINDOWPROP EDITWINDOW 'TEXTSTREAM).

If you need to save the state of an edit, you can save this stream. Calling TEDIT with the stream as the TEXT argument will let you continue from where you left off.

The ''Text Object''

TEdit keeps a variety of other information about each edit window, in a data structure called a TEXTOBJ. Field F3 of a text STREAM points to the associated TEXTOBJ, which contains these fields of interest:

\WINDOW — The edit window which contains the text. If this is NIL, there is no edit window for this text.

SEL — The most recent selection made in this text.

SCRATCHSEL — A scratch SELECTION, used by the mouse handler for the edit window, but otherwise available for scratch use.

TEXTLEN — The current length of the edited text.

STREAMHINT — Points to the text STREAM which describes the text.

EDITFINISHEDFLG — If this is non-NIL, TEdit will halt after the next time through the keyboard polling loop. No check will be made for unsaved changes. Unless it it T, the value of EDITFINISHEDFLG will be returned as the result of TEdit.

Selections

The selected text is described by an object of type SELECTION, whose fields are as follows:

CH# — The character number of the first character in the selection. The first character in the text being edited is numbered 1.

CHLIM — The character number of the last character in the selection. Must be CH#.

DCH — The number of characters in the selection. If DCH is zero, then no characters are selected, and the Selection can be used only to describe a place to insert text.

ONFLG	Tells whether the Selection is indicated in the edit window. If T, it is; if NIL, it's not.
\TEXTOBJ	The TEXTOBJ that describes the selected text. You can use this to get to the Stream itself.
X0	The X position (edit-window-relative) of the left edge of the first selected character.
Y0	The Y position of the bottom of the first selected character (not the character's base line, the bottom of its descent).
XLIM	The X position of the right edge of the last character selected. If DCH is zero (a ''point'' selection), XLIM= X0.
YLIM	The bottom of the last character in the selection.
DX	The width of the selection. If DCH is zero, this will be also.
SELOBJ	This is for a future object-oriented editing interface.
POINT	Tells which side of the selection the caret should appear on. It will be one of the atoms LEFT and RIGHT.
SET	T if this selection is currently valid, NIL if it is obsolete or has never been set.
SELKIND	What kind of selection this is. One of the atoms CHAR, WORD, LINE, or PARA.
HOW	A TEXTURE, which will be used to highlight the selecton.
HOWHEIGHT	How high the highlighting is to extend. A selection's highlight starts at the bottom of the lowest descender, and extends upward for HOWHEIGHT pixels. To always get highlighting a full line tall, set this to 16384.
HASCARET	T if this selection should have a caret flashing next to it, NIL otherwise.

20.6.3.1 TEdit Interface Functions

TEdit exports the following functions for use in custom interfaces:

(OPENTEXTSTREAM TEXT WINDOW START END PROPS) [Function]

Creates a text STREAM describing TEXT, and returns it. If WINDOW is specified, the text will be displayed there, and any changes to the text will be reflected there as they happen. You will also be able to scroll the window and select things there as usual. TEXT may be an existing TEXTOBJ or text STREAM. If START and END are given, then only the section of TEXT delimited is edited. PROPS is the same as for TEDIT.

Given the STREAM, you can use a number of functions to change the text in an edit window, under program control. The edit window gets updated as the text is changed.

(TEDIT.SETSEL STREAM CH/SEL LEN POINT) [Function]

Sets the selection in STREAM. If CH/SEL is a SELECTION, it is used as-is. Otherwise, CH/SEL is the first character in the selection, and LEN is the number of characters to select (zero is allowed, and gives just an insertion point). POINT tells which side of the selection the caret should come on. It must be one of the atoms LEFT or RIGHT.

(TEDIT.GETSEL STREAM) [Function]

Returns the SELECTION which describes the current selection in the edit window described by STREAM.

(TEDIT.SHOWSEL STREAM ONFLG SEL) [Function]

Lets you turn the highlighting of the selection SEL on and off. If ONFLG is T, the selection SEL in STREAM will be highlit in the edit window; if NIL, any highlighting will be turned off. If SEL is NIL, it defaults to the current selection in STREAM.

(TEDIT.INSERT STREAM TEXT CH/SEL) [Function]

Inserts the string TEXT into STREAM, as though it had been typed in. CH/SEL tells where to insert the text: If it's NIL, the text goes in where the caret is. If it's a FIXP, the text is inserted in front of the corresponding character (The first character in the stream is numbered 1). If it's a SELECTION, the text is inserted accordingly.

(TEDIT.DELETE STREAM CH/SEL LEN) [Function]

Deletes text from STREAM. If CH/SEL is a SELECTION, the text it describes will be deleted; if CH/SEL is a FIXP, it is the character number of the first character to delete. In that case, LEN must also be present; it is the number of characters to be deleted.

(TEDIT.FIND STREAM TEXT CH/SEL) [Function]

Searches for the next occurence of TEXT inside STREAM. If CH/SEL is present, the search starts there; otherwise, the search starts from the caret. If it finds a match, TEDIT.FIND returns the character number of the first character in the matching text. If no match is found, it returns NIL.

(TEDIT.HARDCOPY STREAM FILE DONTSEND BREAKPAGETITLE) [Function]

Sends the text contained in STREAM to the printer. If a file name is given in FILE, the press file will be left there for you to use. If DONTSEND is non-NIL, the file will not be sent to the printer; use this if you only want to create a press file for later use.

If BREAKPAGETITLE is non-NIL, it is used as the title on the ''break page'' printed before the text.

(TEDIT.LOOKS STREAM NEWLOOKS SELORCH/SEL LEN) [Function]

Changes the character looks of selected characters, e.g., the font, character size, etc. SELORCH/SEL can be a SELECTION, an integer, or NIL. If SELORCH/SEL is a SELECTION, the text it describes will be changed; if it is a FIXP, it is the character number of the first character to changed. In that case, LEN must also be present; it is the number of characters to be changed.

NEWLOOKS is a property-list-like description of the changes to be made. The property names tell what to change, and the property values describe the change. Any property which isn't changed explicitly retains its old value. Thus, it is possible to make a piece of text all bold without changing the fonts the text is in. The possible list entries are as follows:

FAMILY	The name of the font family. All the selected text is changed to be in that font.
FACE	The face for the new font. This may be in either of the two forms acceptable to FONTCREATE: a list such as (BOLD ITALIC REGULAR), or an atom such as MRR.
SIZE	The new point size.
UNDERLINE	The value for this property must be one of the atoms ON or OFF. The text will be underscored or not, accordingly.
OVERLINE	The value for this property must be one of the atoms ON or OFF. The text will be overscored or not, accordingly.
STRIKEOUT	The value for this property must be one of the atoms ON or OFF. The text will be struck through with a single line or not, accordingly.
SUPERSCRIPT	A distance, in points. The text will be raised above the normal baseline by that amount. This is mutually exclusive with SUBSCRIPT.
SUBSCRIPT	A distance, in points. The text will be raised above the normal baseline by that amount. This is mutually exclusive with SUPERSCRIPT.
PROTECTED	The value for this property must be one of the atoms ON or OFF. If it is ON, the text will be protected from mouse selection and from deletion.
SELECTPOINT	The value for this property must be one of the atoms ON or OFF. If a character has this property, the user can make a point selection just after it, even if the character is also PROTECTED.

(TEDIT.QUIT STREAM VALUE) [Function]

STREAM must be the text stream associated with a running TEdit. TEDIT.QUIT causes the editing session to end. If VALUE is given, it is returned as TEdit's result; otherwise, TEdit will return the usual result. The user is not asked to confirm his desire to stop editing.

(TEDIT.ADD.MENUITEM MENU ITEM) [Function]

Adds a menu ITEM to MENU. This will update the menu's image so that the newly-added item will appear the next time the menu pops up. This is only guaranteed to work right with pop-up menus which aren't visible.

(TEDIT.REMOVE.MENUITEM MENU ITEM) [Function]
Removes a menu ITEM from MENU. This will update the menu's image so that the newly-added item will appear the next time the menu pops up. This is only guaranteed to work right with pop-up menus which aren't visible. ITEM may be either the whole menu item, or just the indicator which appears in the menu's image.

20.6.3.2 User-function ''Hooks'' in TEdit

TEdit provides a number of hooks where a user-supplied function can be called. To supply a function, attach it to the edit window under the appropriate indicator, using WINDOWPROP. Every user-supplied function is APPLYed to the text STREAM which describes the text. Some of these functions can also be supplied using the PROPS argument to TEDIT or OPENTEXTSTREAM; the descriptions below contain the details.

TEDIT.QUITFN [Window Property]
A function to be called whenever the user ends an editing session. This may do anything; if it returns the atom DON'T, TEdit will not terminate. Any other result permits TEdit to do its normal cleanup and termination. This can also be supplied using the PROPS argument to TEDIT or OPENTEXTSTREAM.

TEDIT.AFTERQUITFN [Window Property]
A function to be called after the user ends an editing session. This may perform any cleanup of side effects that you desire. This can also be supplied using the PROPS argument to TEDIT or OPENTEXTSTREAM.

TEDIT.CMD.LOOPFN [Window Property]
A function that gets called, for effect only, each time through TEdit's main command loop. This can also be supplied using the PROPS argument to TEDIT or OPENTEXTSTREAM.

TEDIT.CMD.CHARFN [Window Property]
A function that gets called, for effect only, once for each character typed into TEdit. The character code is passed to the function as its second argument. This can also be supplied using the PROPS argument to TEDIT or OPENTEXTSTREAM.

TEDIT.CMD.SELFN [Window Property]
A function that gets called, for effect only, each time the user selects something with the mouse. The new SELECTION is passed as the function's second argument, and an atom describing the kind of selection (one of NORMAL, COPY, MOVE, or DELETE) as the third. This can also be supplied using the PROPS argument to TEDIT or OPENTEXTSTREAM.

TEDIT.PRESCROLLFN [Window Property]
Called just before TEdit scrolls the edit window.

TEDIT.POSTSCROLLFN [Window Property]
Called just after TEdit scrolls the edit window.

TEDIT.OVERFLOWFN [Window Property]
Called when TEdit is about to move some text off-screen. This function may

handle the text over ow itself (say by reshaping the window), or it may let TEdit take its normal course. If the function handles the problem, it must return a non-NIL result. If TEdit is to handle the over ow, the value returned must be NIL.

TEDIT.TITLEMENUFN [Window Property]
Called whenever the user presses the LEFT or MIDDLE mouse button in the edit window's title region. Can also be supplied using the PROPS argument to TEDIT or OPENTEXTSTREAM. Normally, this is the function TEDIT.DEFAULT.MENUFN, which brings up the usual TEdit command menu.

TEdit also saves pointers to its data structures on each edit window. They are available for any user function's use.

TEXTOBJ [Window Property]
The TEXTOBJ which describes the current editing session.

TEXTSTREAM [Window Property]
The text STREAM which describes the text of the document.

20.6.3.3 Changing the TEdit Command Menu

You may replace the MIDDLE-button command menu with one of your own. When you press the MIDDLE button inside an edit window's title region, TEDIT calls the value of the TEDIT.TITLMENUFN window property with the window as its argument. Normally, what gets called is TEDIT.DEFAULT.MENUFN, but you may change it to anything you like.

TEDIT.DEFAULT.MENUFN brings up a menu of commands. If the edit window has a property TEDIT.MENU, that menu is used. If not, TEdit looks for the window property TEDIT.MENU.COMMANDS (a list of menu items) and constructs a menu from that. Failing that, it uses TEDIT.DEFAULT.MENU.

This means that you can control the command menu by setting the appropriate window properties. Alternatively, you may add your own menu buttons to the default menu, TEDIT.DEFAULT.MENU.

```
(TEDIT.ADD.MENUITEM TEDIT.DEFAULT.MENU ITEM)
```

will add ITEM to the TEdit menu. Menu items should be in the form (NAME FUNCTION), where NAME is what appears in the menu, and FUNCTION will be applied to the text stream, and can perform any operation you desire.

Finally, you may *remove* menu items from the default menu, by doing

```
(TEDIT.REMOVE.MENUITEM TEDIT.DEFAULT.MENU ITEM)
```

ITEM can be either a complete menu item, or just the text that appears in the menu; either will do the job.

20.6.3.4 Variables Which Control TEdit

There are a number of global variables which control TEdit, or which contain state information for editing

sessions in progress:

TEDIT.BLUE.PENDING.DELETE [Variable]
If this is non-NIL, extending a selection makes it into a pending-delete selection. See the selection section.

TEDIT.DEFAULT.FONT [Variable]
A FONTDESCRIPTOR. This is the font for displaying TEdit documents which don't specify their own font information.

TEDIT.DEFAULT.FMTSPEC [Variable]
A paragraph-looks description. This contains the default looks for a paragraph.

TEDIT.SELECTION [Variable]
A SELECTION. This is the most recent regular selection made in *any* TEdit window.

TEDIT.SHIFTEDSELECTION [Variable]
A SELECTION. This is the most recent SHIFT-selection made in *any* TEdit window.

TEDIT.MOVESELECTION [Variable]
A SELECTION. This is the most recent CTRL-SHIFT-selection made in *any* TEdit window.

TEDIT.READTABLE [Variable]
A read table, this is used to translate typed-in characters into TEdit commands. See the section on TEdit readtables.

TEDIT.WORDBOUND.READTABLE [Variable]
The read table which controls TEdit's concept of word boundaries. The syntax classes in this table aslo determine which characters TEdit thinks are white space (which gets deleted by control-W along with the preceding word).

20.6.4 TEdit's Terminal Table and Readtables

TEdit now pays attention to the system terminal table. Characters with terminal sytax-classes CHARDELETE, WORDDELETE, or LINEDELETE act as follows:

CHARDELETE acts as a character-backspace.

WORDDELETE acts like control-W (in fact, this is how control-W is implemented.)

LINEDELETE acts like DEL.

Since the system terminal table is used to implement these functions, you can assign them to other keys at will.

TEdit also has a Readtable, which it uses to dispatch to commands. The table is named TEDIT.READTABLE, and it is global. You can use the functions TEDIT.SETSYNTAX and TEDIT.GETSYNTAX to read it and make changes:

(TEDIT.SETSYNTAX CHAR CODE CLASS TABLE) [Function]
Sets the readtable syntax of the character whose charcode is CHAR CODE to be

CLASS in the read-table TABLE. The possible syntax classes are listed below.

(TEDIT.GETSYNTAX CHAR CODE TABLE) [Function]
Returns the TEdit syntax class of the character whose charcode is CHAR CODE, according to the read-table TABLE. The possible syntax classes are listed below. An illegal syntax will be returned as NIL.

The allowable syntax classes are:

CHARDELETE Typing this character acts like backspace

WORDDELETE Typing this character acts like controlW

DELETE Typing this character acts like DEL

UNDO Typing this character causes Undo

REDO Typing this character acts like ESC

FN Typing this character calls a specified function (see below)

NONE Typing this character simply inserts it in the document. NIL also has this effect.

You can also cause a keystroke to invoke a function for you. To do so, use the function

(TEDIT.SETFUNCTION CHAR CODE FN TABLE) [Function]
Sets up the TEdit readtable TABLE so that typing the character with charcode CHAR CODE will APPLY FN to the text STREAM and the TEXTOBJ for the document being edited. The function may have arbitrary side-effects.

The abbreviation feature described below is implemented using this function-call facility.

Finally, TEdit uses the read table TEDIT.WORDBOUND.READTABLE to decide where word boundaries are. Whenever two adjacent characters have different syntax classes, there is a word boundary between them. The state of this table can be controlled by the functions

(TEDIT.WORDGET CHAR TABLE) [Function]
Returns the syntax class (a small integer) for a given character. CHAR may be either a character or a charcode; TABLE defaults to TEDIT.WORDBOUND.READTABLE.

(TEDIT.WORDSET CHAR CLASS TABLE) [Function]
Sets the syntax class for a character. Again, CHAR is either a character or a charcode; TABLE defaults to TEDIT.WORDBOUND.READTABLE; CLASS may be either a small integer as returned by TEDIT.WORDGET, or one of the atoms WHITESPACE, TEXT, or PUNCTUATION. Those represent the syntax classes in the default TEDIT.WORDBOUND.READTABLE.

The initial TEDIT.WORDBOUND.READTABLE assigns every character to one of the above classes, along pretty obvious lines. For purposes of control-W, whitespace between the caret and the word being deleted is also removed.

20.6.5 The TEdit Abbreviation Facility

The list TEDIT.ABBREVS is a list of ''abbreviations known to TEdit.'' Each element of the list is a dotted pair of two strings. The first is the abbreviation (case does matter), and the second is what the abbreviation expands to. To expand an abbreviation, select it and type control-X. It will be replaced by its expansion.

You can also expand single-character abbreviations while typing. Hitting control-X when no characters are underlined (i.e., after you have typed something) will expand the *single-character* abbreviation to the left of the caret.

Here is a list of the default abbreviations and their expansions:

b	The bullet ( )
m	The M-dash ( )
n	The figure dash ( )
"	Open double-quotes (``) which can be matched by two normal quotes ('')

20.7 THE TTYIN DISPLAY TYPEIN EDITOR

TTYIN is an Interlisp function for reading input from the terminal. It features altmode completion, spelling correction, help facility, and fancy editing, and can also serve as a glorified free text input function. This document is divided into two major sections: how to use TTYIN from the user's point of view, and from the programmer's.

TTYIN exists in implementations for Interlisp-10 and Interlisp-D. The two are substantially compatible, but the capabilities of the two systems differ (Interlisp-D has a more powerful display and allows greater access to the system primitives needed to control it effectively; it also has a mouse, greatly reducing the need for keyboard-oriented editing commands). Descriptions of both are included in this document for completeness, but Interlisp-D users may find large sections irrelevant.

20.7.1 Entering Input With TTYIN

There are two major ways of using TTYIN: (1) set LISPXREADFN to TTYIN, so the LISPX executive uses it to obtain input, and (2) call TTYIN from within a program to gather text input. Mostly the same rules apply to both; places where it makes a difference are mentioned below.

The following characters may be used to edit your input, independent of what kind of terminal you are on. The more TTYIN knows about your terminal, of course, the nicer some of these will behave. Some functions are performed by one of several characters; any character that you happen to have assigned as an interrupt character will, of couse, not be read by TTYIN. There is a (somewhat inelegant) way of changing which characters perform which functions, described under TTYINREADMACROS later on.

control-A, BS, DEL

Deletes a character. At the start of the second or subsequent lines of your input, deletes the last character of the previous line.

control-W
: Deletes a ''word''. Generally this means back to the last space or parenthesis.

control-Q (control-U for Tops20 users)
: Deletes the current line, or if the current line is blank, deletes the previous line.

control-R
: Refreshes the current line. Two in a row refreshes the whole buffer (when doing multi-line input).

ESC
: Tries to complete the current word from the spelling list provided to `TTYIN`, if any. In the case of ambiguity, completes as far as is uniquely determined, or rings the bell. For `LISPX` input, the spelling list may be `USERWORDS` (see discussion of `TTYINCOMPLETEFLG`, page 20.44).

  Interlisp-10 only: If no spelling list was provided, but the word begins with a ''<'', tries directory name completion (or filename completion if there is already a matching ''>'' in the current word).

?
: If typed in the middle of a word will supply alternative completions from the `SPLST` argument to `TTYIN` (if any). `?ACTIVATEFLG` (page 20.43) must be true to enable this feature.

control-F
: Sumex, Tops20 only: Invokes `GTJFN` for filename completion on the current ''word''.

control-Y
: Escapes to a Lisp userexec, from which you may return by the command `OK`. However, when in READ mode and the buffer is non-empty, control-Y is treated as Lisp's unquote macro instead, so you have to use edit-control-Y (below) to invoke the userexec.

<middle-blank> in Interlisp-D, LF in Interlisp-10
: Retrieves characters from the previous non-empty buffer when it is able to; e.g., when typed at the beginning of the line this command restores the previous line you typed at TTYIN; when typed in the middle of a line fills in the remaining text from the old line; when typed following ^Q or ^W restores what those commands erased.

;
: If typed as the first character of the line means the line is a comment; it is ignored, and TTYIN loops back for more input.

control-X
: Goes to the end of your input (or end of expression if there is an excess right parenthesis) and returns if parentheses are balanced, beeps if not. Currently implemented in Interlisp-D only.

During most kinds of input, TTYIN is in ''auto fill'' mode: if a space is typed near the right margin, a carriage return is simulated to start a new line. In fact, on cursor-addressable displays, lines are always broken, if possible, so that no word straddles the end of the line. The ''pseudo-carriage return'' ending the line is still read as a space, however; i.e., the program keeps track of whether a line ends in a carriage return or is merely broken at some convenient point. You won't get carriage returns in your strings unless you explicitly type them.

20.7.2 Mouse Commands [Interlisp-D Only]

The mouse buttons are interpreted as follows during TTYIN input:

`LEFT` Moves the caret to where the cursor is pointing. As you hold down `LEFT`, the caret moves around with the cursor; after you let up, any typein will be inserted at the new position.

`MIDDLE` Like `LEFT`, but moves only to word boundaries.

`RIGHT` Deletes text from the caret to the cursor, either forward or backward. While you hold down `RIGHT`, the text to be deleted is complemented; when you let up, the text actually goes away. If you let up outside the scope of the text, nothing is killed (this is how to ''cancel'' the command). This is roughly the same as `CTRL-RIGHT` with no initial selection (below).

If you hold down `CTRL` and/or `SHIFT` while pressing the mouse buttons, you instead get secondary selection, move selection or delete selection. You make a selection by bugging `LEFT` (to select a character) or `MIDDLE` (to select a word), and optionally extend the selection either left or right using `RIGHT`. While you are doing this, the caret does not move, but your selected text is highlighted in a manner indicating what is about to happen. When you have made your selection (all mouse buttons up now), lift up on `CTRL` and/or `SHIFT` and the action you have selected will occur, which is:

`SHIFT` The selected text as typein at the caret. The text is highlighted with a broken underline during selection.

`CTRL` Delete the selected text. The text is complemented during selection.

`CTRL-SHIFT`
Combines the above: delete the selected text and insert it at the caret. This is how you move text about.

You can cancel a selection in progress by pressing `LEFT` or `MIDDLE` as if to select, and moving outside the range of the text.

The most recent text deleted by mouse command can be inserted at the caret by typing <middle-blank> (the same key that retrieves the previous buffer when issued at the end of a line).

20.7.3 Display Editing Commands

On edit-key terminals (Datamedia): In Interlisp-10, TTYIN reads from the terminal in binary mode, allowing many more editing commands via the edit key, in the style of TVEDIT commands. Note that due to Tenex's unfortunate way of handling typeahead, it is not possible to type ahead edit commands before TTYIN has started (i.e., before its prompt appears), because the edit bit will be thrown away. Also, since ESCAPE has numerous other meanings in Lisp and even in TTYIN (for completion), ESCAPE is not used as a substitute for the edit key.

In Interlisp-D: Users will probably have little use for most of these commands, as cursor positioning can often be done more conveniently, and certainly more obviously, with the mouse. Nevertheless, some commands, such as the case changing commands, can be useful. The <bottom-blank> key can be used as an edit (meta) key in Chorus and subsequent releases if you perform `(TTYINMETA T)`. This calls `(METASHIFT T)` to enable the meta key, redefines the middle and top blank keys, and informs TTYIN

that you want to use them. Alternatively, you can use the EDITPREFIXCHAR (by default on <top-blank>) as described in the next paragraph.

On edit-keyless display terminals (Heath): If you want to type any of these commands, you need to prefix them with the ''edit prefix'' character. Set the variable EDITPREFIXCHAR to the character code of the desired prefix char. Type the edit prefix twice to give an ''edit-ESCAPE'' command. Some users of the TENEX TVEDIT program like to make ESCAPE (33Q) be the edit prefix, but this makes it somewhat awkward to ever use escape completion.

On edit-keyless hardcopy terminals: You probably want to ignore this section, since you won't be able to see what's going on when you issure edit commands; there is no attempt made to echo anything reasonable.

In the descriptions below, ''current word'' means the word the cursor is under, or if under a space, the previous word. Currently parentheses are treated as spaces, which is usually what you want, but can occasionally cause confusion in the word deletion commands. The notation [CHAR] means edit-CHAR , if you have an edit key, or <editprefixchar> CHAR if you don't; $ = escape. Most commands can be preceded by numbers or escape (means infinity), only the first of which requires the edit key (or the edit prefix). Some commands also accept negative arguments, but some only look at the magnitude of the arg. Most of these commands are taken from the display editors TVEDIT and/or E, and are confined to work within one line of text unless otherwise noted.

Cursor Movement Commands:

[delete], [bs], [<]
: Back up one (or n) characters.

[space], [>]
: Move forward one (or n) characters.

[^] : Moves up one (or n) lines.

[lf] : Moves down one (or n) lines.

[(] : Move back one (or n) words.

[)] : Move ahead one (or n) words.

[tab] : Moves to end of line; with an argument moves to nth end of line; [$tab] goes to end of buffer.

[control-L]
: Moves to start of line (or nth previous, or start of buffer).

[{] and [}]
: Go to start and end of buffer, respectively (like [$control-L] and [$tab]).

[[ ] (edit-left-bracket)
: Moves to beginning of the current list, where cursor is currently under an element of that list or its closing paren. (See also the auto-parenthesis-matching feature below under ''Flags''.)

[ ]] (edit-right-bracket)
: Moves to end of current list.

[Sx] Skips ahead to next (or nth) occurrence of character x, or rings the bell.

[Bx] Backward search, i.e., short for [-S] or [-nS].

Bu er Modi cation Commands:

[Zx] Zaps characters from cursor to next (or nth) occurrence of x. There is no unzap command yet.

[A] or [R]
Repeat the last S, B or Z command, regardless of any intervening input (note this di ers from Tvedit's A command).

[K] Kills the character under the cursor, or n chars starting at the cursor.

[I] Begin inserting. Exit insert with any edit command. Characters you type will be inserted, rather than overwriting the existing text. If EMACSFLG (page 20.43) is true (default in Interlisp-D), you are always in insert mode, and this command is a noop.

Inserting <cr> behaves slightly di erent from in tvedit. The sequence [I<cr>] behaves as in TVEDIT; it inserts a blank line ahead of the cursor. <cr> typed any other time while in insert mode actually inserts a <cr>, behaving somewhat like TVEDIT's [B]. [$I] is the same as [I<cr>].

[cr] When the bu er is empty is the same as <lf>, i.e. restores bu er's previous contents. Otherwise is just like a <cr> (except that it also terminates an insert). Thus, [<cr><cr>] will repeat the previous input (as will <lf><cr> without the edit key).

[O] Does ''Open line'', inserting a crlf after the cursor, i.e., it breaks the line but leaves the cursor where it is.

[T] Transposes the characters before and after the cursor. When typed at the end of a line, transposes the previous two characters. Refuses to handle funny cases, such as tabs.

[G] Grabs the contents of the previous line from the cursor position onward. [nG] grabs the nth previous line.

[L] Lowercases current word, or n words on line. [$L] lowercases the rest of the line, or if given at the end of line lowercases the entire line.

[U] Uppercases analogously.

[C] Capitalize. If you give it an argument, only the rst word is capitalized; the rest are just lowercased.

[control-Q]
Deletes the current line. [$control-Q] deletes from the current cursor position to the end of the bu er. No other arguments are handled.

[control-W]
Deletes the current word, or the previous word if sitting on a space.

[D<del>] and [D<cr>]
Are the same as [control-W] and [control-Q], for approximate compatibility with TVEDIT.

[J] ''Justify'' this line. This will break it if it is too long, or move words up from the next line

if too short. Will not join to an empty line, or one starting with a tab (both of which are interpreted as paragraph breaks). Any new line breaks it introduces are considered spaces, not carriage returns. [nJ] justifies n lines.

The linelength is defined as TTYJUSTLENGTH, ignoring any prompt characters at the margin. If TTYJUSTLENGTH is negative, it is interpreted as relative to the right margin. TTYJUSTLENGTH is initially 80 in Interlisp-D, 72 in Interlisp-10.

[$F] ''Finishes'' the input, regardless of where the cursor is. Specifically, it goes to the end of the input and enters a <cr>, control-Z or '']'', depending on whether normal, REPEAT or READ input is happening. Note that a '']'' won't necessarily end a READ, but it seems likely to in most cases where you would be inclined to use this command, and makes for more predictable behavior.

Miscellaneous Commands:

[P] Interlisp-D: Prettyprint buffer. Clears the buffer and reprints it using prettyprint. If there are not enough right parentheses, it will supply more; if there are too many, any excess remains unprettyprinted at the end of the buffer. May refuse to do anything if there is an unclosed string or other error trying to read the buffer.

[N] Refresh line. Same as control-R. [$N] refreshes the whole buffer; [nN] refreshes n lines. Cursor movement in TTYIN depends on TTYIN being the only source of output to the screen; if you do a control-T, or a system message appears, or line noise occurs, you may need to refresh the line for best results. In Interlisp-10, if for some reason your terminal falls out of binary mode (e.g. can happen when returning to a Lisp running in a lower fork), Edit-<anything> is unreadable, so you'd have to type control-R instead.

[control-Y]
Gets userexec. Thus, this is like regular control-Y, except when doing a READ (when control-Y is a read macro and hence does not invoke this function).

[$control-Y]
Gets a userexec, but first unreads the contents of the buffer from the cursor onward. Thus if you typed at TTYIN something destined for the Lisp executive, you can do [control-L$control-Y] and give it to Lisp.

[_] Adds the current word to the spelling list USERWORDS. With zero arg, removes word. See TTYINCOMPLETEFLG (page 20.44).

Note to Datamedia, Heath users: In addition to simple cursor movement commands and insert/delete, TTYIN uses the display's cursor-addressing capability to optimize cursor movements longer than a few characters, e.g. [tab] to go to the end of the line. In order to be able to address the cursor, TTYIN has to know where it is to begin with. Lisp keeps track of the current print position within the line, but does not keep track of the line on the screen (in fact, it knows precious little about displays, much like Tenex). Thus, TTYIN establishes where it is by forcing the cursor to appear on the last line of the screen. Ordinarily this is the case anyway (except possibly on startup), but if the cursor happens to be only halfway down the screen at the time, there is a possibly unsettling leap of the cursor when TTYIN starts.

20.7.4 Using TTYIN for Lisp Input

When TTYIN is loaded, or a sysout containing TTYIN is started up, the function SETREADFN is called. If the terminal is a display, it sets LISPXREADFN to be TTYINREAD; if the terminal is non-display, SETREADFN will set the variable back to READ. (SETREADFN 'READ) will also set it back to READ.

There are two principal differences between TTYINREAD and READ: (1) parenthesis balancing. The input does not activate on an exactly balancing right paren/bracket unless the input started with a paren/bracket, e.g., ''USE (FOO) FOR (FIE)'' will all be on one line, terminated by <cr>; and (2) read macros.

In Interlisp-10, TTYIN does not use a read table (TTYIN behaves as though using the default initial Lisp terminal input readtable), so read macros and redefinition of syntax characters are not supported; however, '' ' '' (QUOTE) and ''control-Y'' (EVAL) are built in, and a simple implementation of ? and ?= is supplied. Also, the TTYINREADMACROS facility described below can supply some of the functionality of immediate read macros in the editor.

In Interlisp-D, read macros are (mostly) supported. Immediate read macros take effect only if typed at the end of the input (it's not clear what their semantics should be elsewhere).

20.7.5 Useful Macros

There are two useful edit macros that allow you to use TTYIN as a character editor: (1) ED loads the current expression into the ttyin buffer to be edited (this is good for editing comments and strings). Input is terminated in the usual way (by typing a balancing right parenthesis at the end of the input, typing <cr> at the end of an already balanced expression, or control-X anywhere inside the balanced expression). Typing control-E or clearing the buffer aborts ED. (2) EE is like ED but prettyprints the expression into the buffer, and uses its own window. The variable TTYINEDITPROMPT controls what prompt, if any, EE uses; see prompt argument description in next section (the initial setting is no prompt). EE is not yet implemented in Interlisp-10.

The macro BUF loads the current expression into the buffer, preceded by E, to be used as input however desired; as a trivial example, to evaluate the current expression, BUF followed by a <cr> to activate the buffer will perform roughly what the edit macro EVAL does. Of course, you can edit the E to something else to make it an edit command.

BUF is also defined at the executive level as a programmer's assistant command that loads the buffer with the VALUEOF the indicated event, to be edited as desired.

TV is a programmer's assistant command like EV [EDITV] that performs an ED on the value of the variable.

And finally, if the event is considered ''short'' enough, the programmer's assistant command FIX will load the buffer with the event's input, rather than calling the editor. If you really wanted the Interlisp editor for your tx, you could either say FIX EVENT - TTY:, or type control-U (or whatever on tops20) once you got TTYIN's version to force you into the editor.

20.7.6 Programming With TTYIN

(TTYIN PROMPT SPLST HELP OPTIONS ECHOTOFILE TABS UNREADBUF RDTBL) [Function]
TTYIN prints PROMPT, then waits for input. The value returned in the normal case is a list of all atoms on the line, with comma and parens returned as individual atoms; OPTIONS may be used to get a different kind of value back.

PROMPT is an atom or string (anything else is converted to a string). If NIL, the value of DEFAULTPROMPT, initially "** ", will be used. If PROMPT is T, no prompt will be given. PROMPT may also be a dotted pair (PROMPT$_1$. PROMPT$_2$), giving the prompt for the first and subsequent (or overflow) lines, each prompt being a string/atom or NIL to denote absence of prompt. Note that rebinding DEFAULTPROMPT gives a convenient way to affect all the ''ordinary'' prompts in some program module.

SPLST is a spelling list, i.e., a list of atoms or dotted pairs (SYNONYM . ROOT). If supplied, it is used to check and correct user responses, and to provide completion if the user types ESCAPE. If SPLST is one of the Lisp system spelling lists (e.g., USERWORDS or SPELLINGS3), words that are escape-completed get moved to the front, just as if a FIXSPELL had found them. Autocompletion is also performed when user types a break character (cr, space, paren, etc), unless one of the ''no xspell'' options below is selected; i.e., if the word just typed would uniquely complete by ESCAPE, TTYIN behaves as though ESCAPE had been typed.

HELP, if non-NIL, determines what happens when the user types ? or HELP. If HELP = T, program prints back SPLST in suitable form. If HELP is any other atom, or a string containing no spaces, it performs (DISPLAYHELP HELP). Anything else is printed as is. If HELP is NIL, ? and HELP are treated as any other atoms the user types. [DISPLAYHELP is a user-supplied function, initially a noop; systems with a suitable HASH package, for example, have defined it to display a piece of text from a hash file associated with the key HELP.]

OPTIONS is an atom or list of atoms chosen from among the following:

NOFIXSPELL — Uses SPLST for HELP and Escape completion, but does not attempt any FIXSPELLing. Mainly useful if SPLST is incomplete and the caller wants to handle corrections in a more flexible way than a straight FIXSPELL.

MUSTAPPROVE — Does spelling correction, but requires confirmation.

CRCOMPLETE — Requires confirmation on spelling correction, but also does autocompletion on <cr> (i.e. if what user has typed so far uniquely identifies a member of SPLST, completes it). This allows you to have the benefits of autocompletion and still allow new words to be typed.

DIRECTORY — (only if SPLST = NIL) Interprets Escape to mean directory name completion [Interlisp-10 only].

USER — Like DIRECTORY, but does username completion. This is identical to DIRECTORY under Tenex [Interlisp-10 only].

FILE — (only if SPLST = NIL) Interprets Escape to mean filename completion, i.e. does a GTJFN [Sumex and Tops20 only].

FIX — If response is not on, or does not correct to, SPLST, interacts with user until an

acceptable response is entered. A blank line (returning NIL) is always accepted. Note that if you are willing to accept responses that are not on SPLST, you probably should specify one of the options NOXFISPELL, MUSTAPPROVE or CRCOMPLETE, lest the user's new response get FIXSPELLed away without their approval.

STRING — Line is read as a string, rather than list of atoms. Good for free text.

NORAISE — Does not convert lower case letters to upper case.

NOVALUE — For use principally with the ECHOTOFILE arg (below). Does not compute a value, but returns T if user typed anything, NIL if just a blank line.

REPEAT — For multi-line input. Repeatedly prompts until user types control-Z (as in Tenex sndmsg). Returns one long list; with STRING option returns a single string of everything typed, with carriage returns (EOL) included in the string.

TEXT — Implies REPEAT, NORAISE, and NOVALUE. Additionally, input may be terminated with control-V, in which case the global flag CTRLVFLG will be set true (it is set to NIL on any other termination). This flag may be utilized in any way the caller desires.

COMMAND — Only the first word on the line is treated as belonging to SPLST, the remainder of the line being arbitrary text; i.e., ''command format''. If other options are supplied, COMMAND still applies to the first word typed. Basically, it always returns (CMD . REST-OF-INPUT), where REST-OF-INPUT is whatever the other options dictate for the remainder. E.g. COMMAND NOVALUE returns (CMD) or (CMD . T), depending on whether there was further input; COMMAND STRING returns (CMD . "REST-OF-INPUT"). When used with REPEAT, COMMAND is only in effect for the first line typed; furthermore, if the first line consists solely of a command, the REPEAT is ignored, i.e., the entire input is taken to be just the command.

READ — Parens, brackets, and quotes are treated a la READ, rather than being returned as individual atoms. Control characters may be input via the control-Vx notation. Input is terminated roughly along the lines of READ conventions: a balancing or over-balancing right paren/bracket will activate the input, or <cr> when no parenthesis remains unbalanced. READ overrides all other options (except NORAISE).

LISPXREAD — Like READ, but implies that TTYIN should behave even more like READ, i.e., do NORAISE, not be errorset-protected, etc.

NOPROMPT — Interlisp-D only: The prompt argument is treated as usual, except that TTYIN assumes that the prompt for the first line has already been printed by the caller; the prompt for the first line is thus used only when redisplaying the line.

ECHOTOFILE if specified, user's input is copied to this file, i.e., TTYIN can be used as a simple text-to-file routine if NOVALUE is used. If ECHOTOFILE is a list, copies to all files in the list. PROMPT is not included on the file.

TABS is a special addition for tabular input. It is a list of tabstops (numbers). When user types a tab, TTYIN automatically spaces over to the next tabstop (thus the first tabstop is actually the second ''column'' of input). Also treats specially the characters * and '; they echo normally, and then automatically tab

over.

UNREADBUF allows the caller to ''preload'' the TTYIN buffer with a line of input. UNREADBUF is a list, the elements of which are unread into the buffer (i.e., ''the outer parentheses are stripped off'') to be edited further as desired; a simple <cr> (or control-Z for REPEAT input) will thus cause the buffer's contents to be returned unchanged. If doing READ input, the ''PRIN2 names'' of the input list are used, i.e., quotes and %'s will appear as needed; otherwise the buffer will look as though UNREADBUF had been PRIN1'ed. UNREADBUF is treated somewhat like READBUF, so that if it contains a pseudo-carriage return (the value of HISTSTR0), the input line terminates there.

Input can also be unread from a file, using the HISTSTR1 format: UNREADBUF = (<value of HISTSTR1> (FILE START . END)), where START and END are file byte pointers. This makes TTYIN a miniature text file editor.

RDTBL [Interlisp-D only] is the read table to use for READing the input when one of the READ options is given. A lot of character interpretations are hardwired into TTYIN, so currently the only effect this has is in the actual READ, and in deciding whether a character typed at the end of the input is an immediate read macro, for purposes of termination.

If the global variable TYPEAHEADFLG is T, or option LISPXREAD is given, TTYIN permits type-ahead; otherwise it clears the buffer before prompting the user.

20.7.7 EE Interface

The following may be useful as a way of outsiders to call TTYIN as an editor. These functions are currently only in Interlisp-D.

(TTYINEDIT EXPRS WINDOW PRINTFN) [Function]
This is the body of EE. Switches the tty to WINDOW, clears it, prettyprints EXPRS, a list of expressions, into it, and leaves you in TTYIN to edit it as Lisp input. Returns a new list of expressions.

If PRINTFN is non-NIL, it is a function of two arguments, EXPRS and FILE, which is called instead of PRETTYPRINT to print the expressions to the window (actually a scratch file). Note that EXPRS is a list, so normally the outer parentheses should not be printed. PRINTFN =T is shorthand for ''unpretty''; use PRIN2 instead of PRETTYPRINT.

TTYINAUTOCLOSEFLG [Variable]
If TTYINAUTOCLOSEFLG is true, TTYINEDIT closes the window on exit.

TTYINEDITWINDOW [Variable]
If the WINDOW arg to TTYINEDIT is NIL, it uses the value of TTYINEDITWINDOW, creating it if it does not yet exist.

TTYINPRINTFN [Variable]
The default value for PRINTFN in EE's call to TTYINEDIT.

(SET.TTYINEDIT.WINDOW WINDOW) [Function]
Called under a RESETLST. Switches the tty to WINDOW (defaulted as in TTYINEDIT) and clears it. The window's position is left so that TTYIN will be

happy with it if you now call TTYIN yourself. Specifically, this means positioning an integral number of lines from the bottom of the window, the way the top-level tty window normally is.

(TTYIN.SCRATCHFILE) [Function]
Returns, possibly creating, the scratch file that TTYIN uses for prettyprinting its input. The file pointer is set to zero. Since TTYIN does use this file, beware of multiple simultaneous use of the file.

20.7.8 ?= Handler

In Interlisp, the ?= read macro displays the arguments to the function currently ''in progress'' in the typein. Since TTYIN wants you to be able to continue editing the buffer after a ?=, it processes this macro specially on its own, printing the arguments below your typein and then putting the cursor back where it was when ?= was typed. For users who want special treatment of ?=, the following hook exists:

TTYIN?=FN [Variable]
The value of this variable, if non-NIL, is a user function of one argument that is called when ?= is typed. The argument is the function that ?= thinks it is inside of. The user function should return one of the following:

NIL Normal ?= processing is performed.

T Nothing is done. Presumably the user function has done something privately, perhaps diddled some other window, or called TTYIN.PRINTARGS (below).

a list (ARGS . STUFF)
Treats STUFF as the argument list of the function in question, and performs the normal ?= processing using it.

anything else
The value is printed in lieu of what ?= normally prints.

At the time that ?= is typed, nothing has been ''read'' yet, so you don't have the normal context you might expect inside a conventional readmacro. If the user function wants to examine the typed-in arguments being passed to the fn, however, it can perform (TTYIN.READ?=ARGS), which bundles up everything between the function and the typing of ?= into a list, which it returns (thus it parallels an arglist; NIL if ?= was typed immediately after the function name).

(TTYIN.PRINTARGS FN ARGS ACTUALS ARGTYPE) [Function]
Does the function/argument printing for ?=. ARGS is an argument list, ACTUALS is a list of actual parameters (from the typein) to match up with args. ARGTYPE is a value of the function ARGTYPE; it defaults to (ARGTYPE FN).

20.7.9 Read Macros

When doing READ input in Interlisp-10, no Lisp-style read macros are available (but the ' and control-Y macros are built in). Principally because of the usefulness of the editor read macros (set by

SETTERMCHARS), and the desire for a way of changing the meanings of the display editing commands, the following exists as a hack:

TTYINREADMACROS [Variable]

Value is a set of shorthand inputs useable during READ input. It is an alist of entries (CHAR CODE . SYNONYM). If the user types the indicated character (edit bit is denoted by the 200Q bit in charcode), TTYIN behaves as though the synonym character had been typed.

Special cases: 0 - the character is ignored; 200Q - pure Edit bit; means to read another char and turn on its edit bit; 400Q - macro quote: read another char and use its original meaning. For example, if you have macros ((33Q . 200Q) (30Q . 33Q)), then Escape (33Q) will behave as an edit prefix, and control-X (30Q) will behave like Escape. Note: currently, synonyms for edit commands are not well-supported, working only when the command is typed with no argument.

Slightly more powerful macros also can be supplied; they are recognized when a character is typed on an empty line, i.e., as the first thing after the prompt. In this case, the TTYINREADMACROS entry is of the form (CHAR CODE T . RESPONSE) or (CHAR CODE CONDITION . RESPONSE), where CONDITION is a list that evaluates true. If RESPONSE is a list, it is EVALed; otherwise it is left unevaluated. The result of this evaluation (or RESPONSE itself) is treated as follows:

NIL The macro is ignored and the character reads normally, i.e., as though TTYINREADMACROS had never existed.

An integer

A character code, treated as above. Special case: -1 is treated like 0, but says that the display may have been altered in the evaluation of the macro, so TTYIN should reset itself appropriately.

Anything else

This TTYIN input is terminated (with a crlf) and returns the value of ‘‘response’’ (turned into a list if necessary). This is the principal use of this facility. The macro character thus stands for the (possibly computed) reponse, terminated if necessary with a crlf. The original character is not echoed.

Interrupt characters, of course, cannot be read macros, as TTYIN never sees them, but any other characters, even non-control chars, are allowed. The ability to return NIL allows you to have conditional macros that only apply in specified situations (e.g., the macro might check the prompt (LISPXID) or other contextual variables). To use this specially to do immediate editor read macros, do the following for each edit command and character you want to invoke it with:

```
(ADDTOVAR TTYINREADMACROS (CHAR CODE 'CHARMACRO? EDITCOM )))
```

For example, (ADDTOVAR TTYINREADMACROS (12Q CHARMACRO? !NX)) will make linefeed do the !NX command. Note that this will only activate linefeed at the beginning of a line, not anywhere in the line. There will probably be a user function to do this in the next release.

Note that putting (12Q T . !NX) on TTYINREADMACROS would also have the effect of returning "!NX" from the READ call so that the editor would do an !NX. However, TTYIN would also return !NX

outside the editor (probably resulting in a u.b.a. error, or convincing DWIM to enter the editor), and also the clearing of the output buffer (performed by CHARMACRO?) would not happen.

20.7.10 Assorted Flags

These flags control aspects of TTYIN's behavior. Some have already been mentioned. Their initial values are all NIL. In Interlisp-D, the flags are all initially T.

TYPEAHEADFLG [Variable]
: If true, TTYIN always permits typeahead; otherwise it clears the buffer for any but LISPXREAD input.

?ACTIVATEFLG [Variable]
: If true, enables the feature whereby ? lists alternative completions from the current spelling list.

EMACSFLG [Variable]
: Affects display editing. When true, TTYIN tries to behave a little more like EMACS (in very simple ways) than TVEDIT. Specifically, it has the following effects currently: (1) all non-edit characters self-insert (i.e. behave as if you're always in Insert mode); (2) [D] is the EMACS delete to end of word command.

SHOWPARENFLG [Variable]
: If true, then whenever you are typing Lisp input and type a right parenthesis/bracket, TTYIN will briefly move the cursor to the matching parenthesis/bracket, assuming it is still on the screen. The cursor stays there for about 1 second, or until you type another character (i.e., if you type fast you'll never notice it). This feature was inspired by a similar EMACS feature, and turned out to be pretty easy to implement.

TTYINBSFLG [Variable]
: Causes TTYIN to always physically backspace, even if you're running on a non-display (not a DM or Heath), rather than print \deletedtext\ (this assumes your hardcopy terminal or glass tty is capable of backspacing). If TTYINBSFLG is LF, then in addition to backspacing, TTYIN x's out the deleted characters as it backs up, and when you stop deleting, it outputs a linefeed to drop to a new, clean line before resuming. To save paper, this linefeed operation is not done when only a single character is deleted, on the grounds that you can probably figure out what you typed anyway.

TTYINRESPONSES [Variable]
: An alist of special responses that will be handled by routines designated by the programmer. See ''Special Responses'', below.

TTYINERRORSETFLG [Variable]
: [Interlisp-D only] If true, non-LISPXREAD inputs are errorset-protected (control-E traps back to the prompt), otherwise errors propagate upwards. Initially NIL.

TTYINMAILFLG [Variable]
: [Tenex only] When true, performs mail checking, etc. before most inputs (except EVALQT inputs, where it is assumed this has already been done, or inputs indented

by more than a few spaces). The MAILWATCH package must be loaded for this.

TTYINCOMPLETEFLG [Variable]

If true, enables Escape completion from USERWORDS during READ inputs. Details below.

USERWORDS (page 15.15) contains words you mentioned recently: functions you have de ned or edited, variables you have set or evaluated at the executive level, etc. This happens to be a very convenient list for context-free escape completion; if you have recently edited a function, chances are good you may want to edit it again (typing ''EF xx$'') or type a call to it. If there is no completion for the current word from USERWORDS, the escape echoes as ''$'', i.e. nothing special happens; if there is more than one possible completion, you get beeped. If typed when not inside a word, Escape completes to the value of LASTWORD, i.e., the last thing you typed that the p.a. ''noticed'' (setting TTYINCOMPLETEFLG to 0 disables this latter feature), except that Escape at the beginning of the line is left alone (it is a p.a. command).

If you really wanted to enter an escape, you can, of course, just quote it with a control-V, like you can other control chars.

You may explicitly add words to USERWORDS yourself that wouldn't get there otherwise. To make this convenient online the edit command [_] means ''add the current atom to USERWORDS'' (you might think of the command as ''pointing out this atom''). For example, you might be entering a function de nition and want to ''point to'' one or more of its arguments or prog variables. Giving an argument of zero to this command will instead remove the indicated atom from USERWORDS.

Note that this feature loses some of its value if the spelling list is too long, for then the completion takes too long computationally and, more important, there are too many alternative completions for you to get by with typing a few characters followed by escape. Lisp's maintenance of the spelling list USERWORDS keeps the ''temporary'' section (which is where everything goes initially unless you say otherwise) limited to #USERWORDS atoms, initially 100. Words fall o the end if they haven't been used (they are ''used'' if FIXSPELL corrects to one, or you use <escape> to complete one).

20.7.11 Special Responses

There is a facility for handling ''special responses'' during any non-READ TTYIN input. This action is independent of the particular call to TTYIN, and exists to allow you to e ectively ''advise'' TTYIN to intercept certain commands. After the command is processed, control returns to the original TTYIN call. The facility is implemented via the list TTYINRESPONSES.

TTYINRESPONSES [Variable]

TTYINRESPONSES is a list of elements, each of the form:

(COMMANDS RESPONSE-FORM OPTION)

COMMANDS is a single atom or list of commands to be recognized; RESPONSE-FORM is EVALed (if a list), or APPLYed (if an atom) to the command and the rest of the line. Within this form one can reference the free variables COMMAND (the command the user typed) and LINE (the rest of the line). If OPTION is the atom LINE, this means to pass the rest of line as a list; if it is STRING, this means to pass it as a string; otherwise, the command is only valid if there is nothing else on the line. If RESPONSE-FORM returns the atom IGNORE, it is not treated as a

special response (i.e. the input is returned normally as the result of TTYIN).

In MYCIN, the COMMENT command is handled this way; any time the user types COMMENT as the first word of input, TTYIN passes the rest of the line to a mycin-defined function which prompts for the text of the comment (recursively using TTYIN with the TEXT option). When control returns, TTYIN goes back and prompts for the original input again. The TTYINRESPONSES entry for this is (COMMENT (GRIPE LINE) LIST); GRIPE is a MYCIN function of one argument (the one-line comment, or NIL for extended comments).

Suggested use: global commands or options can be added to the toplevel value of TTYINRESPONSES. For more specialized commands, rebind TTYINRESPONSES to (APPEND NEWENTRIES TTYINRESPONSES) inside any module where you want to do this sort of special processing.

Special responses are not checked for during READ-style input.

20.7.12 Display Types

[This is not relevant in Interlisp-D]

TTYIN determines the type of display by calling DISPLAYTERMP, which is initially defined to test the value of the GTTYP jsys. It returns either NIL (for printing terminals) or a small number giving TTYIN's internal code for the terminal type. The types TTYIN currently knows about:

0 = glass tty (capable of deleting chars by backspacing, but little else);

1 = Datamedia;

2 = Heath.

Only the Datamedia has full editing power. DISPLAYTERMP has built into it the correct terminal types for Sumex and Stanford campus 20's: Datamedia = 11 on tenex, 5 on tops20; Heath = 18 on Tenex, 25 on tops20. You can override those values by setting the variable DISPLAYTYPES to be an alist associating the GTTYP value with one of these internal codes. For example, Sumex displays correspond to DISPLAYTYPES = ((11 . 1) (18 . 2)) [although this is actually compiled into DISPLAYTERMP for speed]. Any display terminal other than Datamedia and Heath can probably safely be assigned to ''0'' for glass tty.

To add new terminal types, you have to choose a number for it, add new code to TTYIN for it and recompile. The TTYIN code specifies what the capabilities of the terminal are, and how to do the primitive operations: up, down, left, right, address cursor, erase screen, erase to end of line, insert character, etc.

For terminals lacking an Edit key (currently only Datamedias have it), set the variable EDITPREFIXCHAR to the ascii code of an Edit ''prefix'' (i.e. anything typed preceded by the prefix is considered to have the edit bit on). If your EDITPREFIXCHAR is 33Q (Escape), you can type a real Escape by typing 3 of them (2 won't do, since that means ''Edit-Escape'', a legitimate argument to another command). You could also define an Escape synonym with TTYINREADMACROS if you wanted (but currently it doesn't work in filename completion). Setting EDITPREFIXCHAR for a terminal that is not equipped to handle the full range of editing functions (only the Heath and Datamedia are currently so equipped) is not guaranteed to work, i.e. the display will not always be up to date; but if you can keep track of what you're doing, together with an occasional control-R to help out, go right ahead.