APPENDIX A. THE EXEC

In most Common Lisp implementations, there is a "top-level **read-eval-print** loop," which reads an expression, evaluates it, and prints the results. In Xerox Common Lisp, the Exec acts as the top-level loop, but in addition to **read-eval-print**, it also performs a number of other tasks, and allows a much greater range of inputs. This appendix contains information from the Lyric and Medley releases. Medley additions are indicated with revision bars in the right margin.

The Exec is based on concepts from the Interlisp Programmer's Assistant (see the *Interlisp-D Reference Manual*).

The Exec traps all throws, and recovers gracefully. It prints all values resulting from evaluation, on separate lines. When zero values are returned, nothing is printed.

The Exec keeps track of your previous input, in a structure called the history list. A history list is a list of the information associated with each of the individual events that have occurred, where each event corresponds to one input. Associated with each event on the history list is the input, its values, plus other optional information such as side-effects, formatting information, etc.

The following dialogue contains illustrative examples and gives the flavor of the use of the Exec. Be sure to type these examples to an Exec whose ***PACKAGE*** is set to the **XCL-USER** package. The Exec that Lisp starts up with is set to the **XCL-USER** package. Each prompt consists of an event number and a prompt character (">").

12>**(setq foo 5)**
5
13>**(setq foo 10)**
10

14>**undo**cr
SETQ undone**.**
15>**foo**cr
5

This is an example of direct communication with the Exec. You have instructed the Exec to **undo** *the previous event.*

. . .

25>**set(lst1 (a b c))**

(A B C)

26>**(setq lst2 '(d e f))**

(D E F)

27>**(mapc #'(lambda (x) (setf (get x 'myprop) t)) lst1)**

(A B C)

The Exec accepts input both in APPLY format (the **SET***) and EVAL format (the* **SETQ***.) In event 27, the user adds a property MYPROP to the symbols* **A***,* **B***, and* **C***.*

28>**use lst2 for lst1 in 27**cr
NIL

You just instructed the Exec to go back to event number 27, substitute **LST2** *for* **LST1,** *and then re-execute the expression. You could have also used -2 instead of 27, specifying a relative address.*

.
.
.

46>**(setf my-hash-table (make-hash-table))**

```
#<Hash-Table @ 66,114034>
```

47>**(setf (gethash 'foo my-hash-table) (string 'foo))**

```
"FOO"
```

If **STRING** *were computationally expensive (which it is not), then you might be caching its value for later use.*

48>**use fie for foo in string**cr

```
"FIE"
```

You now decide you would like to redo the **SETF** *with a different value. You specify the event using "***IN STRING***" rather than* **SETF***.*

49>**?? use**cr

USE FIE FOR FOO IN STRING
48> **(SETF (GETHASH 'FIE MY-HASH-TABLE)**
(STRING 'FIE))

```
"FIE"
```

Here you ask the Exec (using the ?? command) what it has on its history list for the last input. Since the event corresponds to a command, the Exec displays both the original command and the generated input.

The most common interaction with the Exec occurs at the top level or in the debugger, where you type in expressions for evaluation, and see the values printed out. In this mode, the Exec acts much like a standard Common Lisp top-level loop, except that before attempting to evaluate an input, the Exec first stores it in a new entry on the history list. Thus if the operation is aborted or causes an error, the input is still saved and available for modification and/or re-execution. The Exec also notes new functions and variables to be added to its spelling lists to enable future corrections.

After updating the history list, the Exec executes the computation (i.e., evaluates the form or applies the function to its arguments), saves the value in the entry on the history list corresponding to the input, and prints the result. Finally the Exec displays a prompt to indicate it is again ready for input.

Input Formats

The Exec accepts three forms of input: an expression to be evaluated (EVAL-format), a function-name and arguments to apply it to (APPLY-format), and Exec commands, as follows:

EVAL-format input

If you type a single expression, either followed by a carriage-return, or, in the case of a list, terminated with balanced parenthesis, the expression is evaluated and the value is returned. For example, if the value of the variable **FOO** is the list **(A B C)**:

32>**FOO**cr

```
(A B C)
```

Similarly, if you type a Lisp expression, beginning with a left parenthesis and terminated by a matching right parenthesis, the form is simply passed to **EVAL** for evaluation. Notice that it is not necessary to type a carriage return at the end of such a form; the reader will supply one automatically. If a carriage-return is typed before the final matching right parenthesis or bracket, it is treated the same as a space, and input continues. The following examples are interpreted identically:

123> **(+ 1 (* 2 3))**
7

124> **(+ 1 (*cr**
2 3))
7

APPLY-format input

Often, when typing at the keyboard, you call functions with constant argument values, which would have to be quoted if you typed them in "EVAL-format." For convenience, if you type a symbol immediately followed by a list form, the symbol is **APPLY**ed to the elements within the list, unevaluated. The input is terminated by the matching right parenthesis. For example, typing **LOAD(FOO)** is equivalent to typing **(LOAD 'FOO)**, and **GET(X COLOR)** is equivalent to **(GET 'X 'COLOR)**. As a simple special case, a single right parenthesis is treated as a balanced set of parentheses, e.g.

125>**UNBREAK)**

is equivalent to

125>**UNBREAK()**

The reader will only supply the "carriage return" automatically if no space appears between the initial symbol and the list that follows; if there is a space after the initial symbol on the line and the list that follows, the input is not terminated until a carriage return is explicitly typed.

Note that APPLY-format input cannot be used for macros or special forms.

Exec commands

The Exec recognizes a number of commands, which usually refer to past events on the history list. These commands are treated specially; for example, they may not be put on the history list. The format of a command is always a line beginning with the command name. (The Exec looks up the command name independent of package, so that Exec commands are package independent.) The remainder of the line, if any, is treated as "arguments" to the command. For example,

128>**UNDO**cr
`mapc undone`
129>**UNDO (FOO --)**cr
`foo undone`

are all valid command inputs.

Multiple Execs and the Exec's Type

Multiple Execs

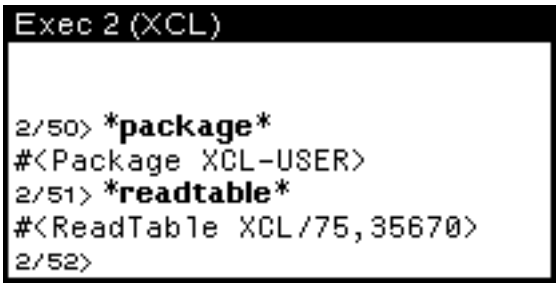
More than one Exec can be active at any one time. New Execs can be created by selecting the Exec menu item in the background pop-up menu. When a prompt is printed for an event in other than the first Exec, the prompt is preceded with the Exec number; for example:

2/50>

might be a prompt in Exec 2. All Execs share the same history list, but each event records which Exec it goes with. That is, although a single global list exists, the Xerox Lisp history system maintains the separate threads of control within each Exec.

Exec type

Several variables are very important to an Exec since they control the format of reading and printing. Together these variables describe a type of exec. Put another way, this is the Exec's mode. To allow easier setting of these modes some standard bindings for the variables have been named. The names provide the user an Exec of the Common Lisp (CL), Interlisp (IL) or Xerox Extended Common Lisp (XCL) type. An Exec's type is usually displayed in the title bar of its window in parentheses:



Event Specification

Exec commands, like **UNDO**, frequently refer to previous events in the session's history. All Exec commands use the same conventions and syntax for indicating which event(s) the command refers to. This section shows you the syntax used to specify previous events.

An event address identifies one event on the history list. For example, the event address **42** refers to the event with event number 42, and **-2** refers to two events back in the *current* Exec. Usually, an event address will contain only one or two commands.

Event addresses can be concatenated. For example, if **FOO** refers to event *N*, **FOO FIE** will refer to the first event before event *N* which contains **FIE**.

The symbols used in event addresses (such as **AND**, **F**, **=**, etc. are compared with **STRING-EQUAL**, so that it does not matter what the current package is when you type an event address symbol to an Exec.

Event addresses are interpreted as follows:

N (an integer)

If *N* is positive, it refers to the event with event number *N* (no matter which Exec the event occurred in.) If *N* is negative, it always refers to the event -*N* events backwards counting *only* events belonging to the *current* Exec.

F

Specifies that the next object in the event address is to be searched for, regardless of what it is. For example, **F -2** looks for an event containing **-2**.

=

Specifies that the next object is to be searched for in the *values* of events, instead of the inputs.

SUCHTHAT *PRED* Specifies an event for which the function *PRED* returns true. *PRED* should be a function of two arguments, the input portion of the event, and the event itself.

PAT Any other event address command specifies an event whose input contains an expression that matches *PAT.* When multiple Execs are active, all events are searched, no matter which Exec they belong to. The pattern can be a simple symbol, or a more complex search pattern.

Note: Specifications used below of the form $EventAddress_i$ refer to event addresses, as described above. Since an event address may contain multiple words, the event address is parsed by searching for the words which delimit it. For example, in $EventAddress_1$ **AND** $EventAddress_2$, the notation $EventAddress_1$ corresponds to all words up to the **AND** in the event specification, and $EventAddress_2$ to all words after the **AND** in the event specification.

FROM *EventAddress* All events since *EventAddress*, inclusive. For example, if there is a single Exec and the current event is number 53, then **FROM 49** specifies events 49, 50, 51, and 52. **FROM** will include events from *all* Execs.

ALL *EventAddress* Specifies all events satisfying *EventAddress*. For example, **ALL LOAD**, **ALL SUCHTHAT FOO-P**.

empty If nothing is specified, it is the same as specifying **-1**, i.e., the last event in the current Exec.

$EventSpec_1$ **AND** $EventSpec_2$ **AND . . . AND** $EventSpec_N$

Each of the $EventSpec_i$ is an event specification. The lists of events are concatenated. For example, **ALL MAPC AND ALL STRING AND 32** specifies all events containing **MAPC**, all containing **STRING**, and also event **32**. Duplicate events are removed.

Exec Commands

All Exec commands are input as lines which begin with the name of the command. The name of an Exec command is not a symbol and therefore is not sensitive to the setting of the current package (the value of ***PACKAGE***).

EventSpec is used to denote an event specification which in most cases will be either a specific event address (e.g., 42) or a relative one (e.g., -3). Unless specified otherwise, omitting *EventSpec* is the same as specifying *EventSpec*=**-1**. For example, **REDO** and **REDO -1** are the same.

REDO *EventSpec* [Exec command]

Redoes the event or events specified by *EventSpec*. For example, **REDO 123** redoes the event numbered 123.

RETRY *EventSpec* [Exec command]

Similar to **REDO** except sets the debugger parameters so that any errors that occur while executing *EventSpec* will cause breaks.

USE *NEW* **[FOR** *OLD* **] [IN** *EventSpec* **]** [Exec command]

Substitutes *NEW* for *OLD* in the events specified by *EventSpec*, and redoes the result. *NEW* and *OLD* can include lists or symbols, etc.

For example, **USE SIN (- X) FOR COS X IN -2 AND -1** will substitute **SIN** for every occurrence of **COS** in the previous two events, and substitute **(- X)** for every occurrence of **X**, and reexecute them. (The substitutions do not change the previous information saved about these events on the history list.)

If **IN** *EventSpec* is omitted, the first member of *OLD* is used to search for the appropriate event. For example, **USE DEFAULTFONT FOR DEFLATFONT** is equivalent to **USE DEFAULTFONT FOR DEFLATFONT IN F DEFLATFONT**. The **F** is inserted to handle correctly the case where the first member of *OLD* could be interpreted as an event address command.

If *OLD* is omitted, substitution is for the "operator" in that command. For example **FBOUNDP(FF)** followed by **USE CALLS** is equivalent to **USE CALLS FOR FBOUNDP IN -1**.

If *OLD* is not found, **USE** will print a question mark, several spaces and the pattern that was not found. For example, if you specified USE Y FOR X IN 104 and X was not found, "X ?" is printed to the Exec.

You can also specify more than one substitution simultaneously as follows:

USE NEW_1 **FOR** OLD_1 **AND** ... **AND** NEW_N **FOR** OLD_N **[IN** *EventSpec* [Exec command]

Note: The **USE** command is parsed by a small finite state parser to distinguish the expressions and arguments. For example, **USE FOR FOR AND AND AND FOR FOR** will be parsed correctly.

Every **USE** command involves three pieces of information: the expressions to be substituted, the arguments to be substituted for, and an event specification that defines the input expression in which the substitution takes place. If the **USE** command has the same number of expressions as arguments, the substitution procedure is straightforward. For example, **USE X Y FOR U V** means substitute **X** for **U** and **Y** for **V**, and is equivalent to **USE X FOR U AND Y FOR V**.

However, the **USE** command also permits distributive substitutions for substituting several expressions for the same argument. For example, **USE A B C FOR X** means first substitute **A** for **X** then substitute **B** for **X** (in a new copy of the expression), then substitute **C** for **X**. The effect is the same as three separate **USE** commands.

Similarly, **USE A B C FOR D AND X Y Z FOR W** is equivalent to **USE A FOR D AND X FOR W**, followed by **USE B FOR D AND Y**

FOR W, followed by **USE C FOR D AND Z FOR W**. **USE A B C FOR D AND X FOR Y** also corresponds to three substitutions, the first with **A** for **D** and **X** for **Y**, the second with **B** for **D**, and **X** for **Y**, and the third with **C** for **D**, and again **X** for **Y**. However, **USE A B C FOR D AND X Y FOR Z** is ambiguous and will cause an error.

Essentially, the **USE** command operates by proceeding from left to right handling each **AND** separately. Whenever the number of expressions exceeds the number of expressions available, multiple **USE** expressions are generated. Thus **USE A B C D FOR E F** means substitute **A** for **E** at the same time as substituting **B** for **F**, then in another copy of the indicated expression, substitute **C** for **E** and **D** for **F**. This is also equivalent to **USE A C FOR E AND B D FOR F**.

Note: The **USE** command correctly handles the situation where one of the old expressions is the same as one of the new ones, **USE X Y FOR Y X**, or **USE X FOR Y AND Y FOR X**.

? &OPTIONAL *NAME* [Exec command]

If *NAME* is not provided describes all available Exec commands by printing the name, argument list, and description of each. With *NAME*, only that command is described.

?? *EventSpec* [Exec command]

Prints the most recent event matching the given *EventSpec*.

CONN *DIRECTORY* [Exec command]

Changes default pathname to *DIRECTORY*.

DA [Exec command]

Returns current date and time.

DIR &OPTIONAL *PATHNAME* &REST *KEYWORDS* [Exec command]

Shows a directory listing for *PATHNAME* or the connected directory. If provided, *KEYWORDS* indicate information to be displayed for each file. Some keywords are: AUTHOR, AU, CREATIONDATE, DA, etc.

DO-EVENTS &REST *INPUTS* &ENVIRONMENT *ENV* [Exec command]

DO-EVENTS is intended as a way of putting together several different events, which can include commands. It executes the multiple *INPUTS* as a single event. The values returned by the **DO-EVENTS** event are the concatenation of the values of the inputs. An input is not an EventSpec, but a call to a function or command. If *ENV* is provided it is a lexical environment in which all evaluations (functions and commands) will take place. Event specification in the *INPUTS* should be explicit, not relative, since referring to the last event will reinvoke the executing **DO-EVENTS** command.

FIX &REST *EventSpec* [Exec command]

Edits the specified event prior to reexecuting it. If the number of characters in the **Fix**ed line is less than the variable **TTYINFIXLIMIT** then it will be edited using TTYIN, otherwise the Lisp editor is called via **EDITE**.

FORGET &REST *EventSpec* [Exec command]

Erases **UNDO** information for the specified events.

NAME *COMMAND-NAME* &OPTIONAL *ARGUMENTS* &REST *EVENT-SPEC* [Exec command]

Defines a new command, *COMMAND-NAME*, and its *ARGUMENTS*, containing the events in *EVENT-SPEC*.

NDIR &OPTIONAL *PATHNAME* &REST *KEYWORDS* [Exec command]

Shows a directory listing for *PATHNAME* or the connected directory in abbreviated format. If provided, *KEYWORDS* indicate information to be displayed for each file. Some keywords are: AUTHOR, AU, CREATIONDATE, DA, etc.

PL *SYMBOL* [Exec command]

Prints the property list of *SYMBOL* in an easy to read format.

REMEMBER &REST *EVENT-SPEC* [Exec command]

Tells File Manager to remember type-in from specified event(s) , *EVENT-SPEC* , as expressions to save.

SHH &REST *LINE* [Exec command]

Executes *LINE* without history list processing.

UNDO &REST *EventSpec* [Exec command]

Undoes the side effects of the specified event (see below under "Undoing").

PP &OPTIONAL *NAME* &REST *TYPES* [Exec command]

Shows (prettyprinted) the definitions for *NAME* specified by *TYPES*.

SEE &REST *FILES* [Exec command]

Prints the contents of *FILES* in the Exec window, hiding comments.

SEE* &REST *FILES* [Exec command]

Prints the contents of *FILES* in the Exec window, showing comments.

TIME *FORM* &KEY *REPEAT* &ENVIRONMENT *ENV* [Exec command]

Times the evaluation of *FORM* in the lexical environment *ENV*, repeating *REPEAT* number of times. Information is displayed in the Exec window.

TY &REST *FILES* [Exec command]

Exactly like the **TYPE** Exec command.

TYPE &REST *FILES* [Exec command]

Prints the contents of *FILES* in the Exec window, hiding comments.

Variables

A number of variables are provided for convenience in the Exec.

IL:IT [Variable]

Whenever an event is completed, the global value of the variable **IT** is reset to the event's value. For example,

```
312>(SQRT 2)
1.414214
313>(SQRT IL:IT)
1.189207
```

Following a **??** command, **IL:IT** is set to the value of the last event printed. The inspector has an option for setting the variable **IL:IT** to the current selection or inspected object, as well. The variable **IL:IT** is global, and is shared among all Execs. **IL:IT** is a convenient mechanism for passing values from one process to another.

Note: **IT** is in the INTERLISP package and these examples are intended for an Exec whose ***PACKAGE*** is set to **XCL-USER**. Thus, **IT** must be package qualified (the **IL:**).

The following variables are maintained independently by each Exec. (When a new Exec is started, the initial values are **NIL**, or, for a nested Exec, the value for the "parent" Exec. However, events executed under a nested Exec will not affect the parent values.)

CL:- [Variable]

CL:+ [Variable]

CL:++ [Variable]

CL:+++ [Variable]

While a form is being evaluated by the Exec, the variable **-** is bound to the form, **CL:+** is bound to the previous form, **CL:++** the one before, etc. If the input is in apply-format rather than eval-format, the value of the respective variable is just the function name.

CL:* [Variable]

CL:** [Variable]

CL:*** [Variable]

While a form is being evaluated by the Exec, the variable **CL:*** is bound to the (first) value returned by the last event, **CL:**** to the event before that, etc. The variable **CL:*** differs from **IT** in that **IT** is global while each separate Exec maintains its own copy of **CL:***, **CL:**** and **CL:***. In addition, the history commands change **IT**, but only inputs which are retained on the history list can change **CL:***.

CL:/ [Variable]

CL:// [Variable]

CL:/// [Variable]

While a form is being evaluated by an Exec, the variable **CL:**/ is bound to a list of the results of the last event in that Exec, **CL:**// to the values of the event before that, etc.

Fonts in the Exec

The Exec can use different fonts for displaying the prompt, user's input, intermediate printout, and the values returned by evaluation. The following variables control the Exec's font use:

PROMPTFONT [Variable]

Font used for printing the event prompt.

INPUTFONT [Variable]

Font used for echoing user's type-in.

PRINTOUTFONT [Variable]

Font used for any intermediate printing caused by execution of a command or evaluation of a form. Initially the same as **DEFAULTFONT**.

VALUEFONT [Variable]

Font used to print the values returned by evaluation of a form. Initially the same as **DEFAULTFONT**.

Changing the Exec

(CHANGESLICE *N HISTORY* —**)** [Function]

Changes the time-slice of the history list *HISTORY* to *N*. If **NIL**, *HISTORY* defaults to the top level history **LISPXHISTORY**.

Note: The effect of *increasing* the time-slice is gradual: the history list is simply allowed to grow to the corresponding length before any events are forgotten. *Decreasing* the time-slice will immediately remove a sufficient number of the older events to bring the history list down to the proper size. However, **CHANGESLICE** is undoable, so that these events are (temporarily) recoverable. Therefore, if you want to recover the storage associated with these events without waiting *N* more events until the **CHANGESLICE** event drops off the history list, you must perform a **FORGET** command.

Defining New Commands

You can define new Exec commands using the **XCL:DEFCOMMAND** macro.

(XCL:DEFCOMMAND *NAME ARGUMENT-LIST &REST BODY***)** [Macro]

XCL:DEFCOMMAND is similar to **XCL:DEFMACRO**, but defines new Exec commands. The *ARGUMENT-LIST* can have keywords, defstructure, and use all of the features of macro argument lists. When *NAME* is subsequently typed to the Exec, the rest of the line is processed like the arguments to a macro, and the *BODY* is executed. **XCL:DEFCOMMAND** is a definer; the File Manager will remember typed-in definitions and allow them to be saved, edited with **EDITDEF**, etc.

There are actually three kinds of commands that can be defined, **:EVAL**, **:QUIET**, and **:INPUT**. Commands can also be marked as only for the debugger, in which case they are labelled as **:DEBUGGER**. The command type is noted by supplying a list for the *NAME* argument to **XCL:DEFCOMMAND**, where the first element of the list is the command name, and the other elements are keyword(s) for the command type and, optionally **:DEBUGGER**.

Note: The documentation string in user defined Exec commands is automatically added to the documentation descriptions by the **CL:DOCUMENTATION** function under the COMMANDS type and can be shown using the **?** Exec command.

:EVAL This is the default. The body of the command just gets executed, and its value is the value of the event. For example (in an XCL Exec),

```
(DEFCOMMAND (LS :EVAL)
(&OPTIONAL (NAMESTRING *DEFAULT-PATHNAME-DEFAULTS*)
&REST DIRECTORY-KEYWORDS)
(MAPC
        #'(LAMBDA (PATHNAME) (FORMAT T "~&~A" (NAMESTRING PATHNAME)))
        (APPLY #'DIRECTORY NAMESTRING DIRECTORY-KEYWORDS))
(VALUES))
```

would define the **LS** command to print out all file names that match the input namestring. The **(VALUES)** means that no value will be printed by the event, only the intermediate output from the **FORMAT**.

:QUIET These commands are evaluated, but neither your input nor the results of the command are stored on the history list. For example, the **??** and **SHH** commands are quiet.

:INPUT These commands work more like macros, in that the result of evaluating the command is treated as a new line of input. The **FIX** command is an input command. The result is treated as a line; a single expression in EVAL-format should be returned as a list of the expression to **EVAL**.

The new Exec now will not consider unparenthesized input with more than one argument to be in apply format. This is the same behavior as the older execs, e.g.:

```
list(1)        ; is apply format (executes after close paren is typed)
list (1)       ; is apply format (second arg is a list, no trailing args
                 given)
list '(1) 2 3 ; is NOT apply format, arguments are evaluated
list 1 2 3     ; is NOT apply format, arguments are evaluated
list 1         ; not legal input: second argument is not a list
```

Undoing

Note: This discussion only applies to undoing under the Exec, Debugger and within the UNDOABLY macro; editors handle undoing in a different fashion.

The **UNDO** facility allows recording of destructive changes such that they can be played back to restore a previous state. There are two kinds of **UNDO**ing: one is done by the Exec, the other is available for use in a programmer's code. Both methods share information about what kind of operations can be undone and where the changes are recorded.

Undoing in the Exec

UNDO *EventSpec* [Exec command]

The Exec's **UNDO** command is implemented by watching the evaluation of forms and requiring undoable operations in that evaluation to save enough information on the history list to reverse their side effects. The Exec simply executes operations, and any undoable changes that occur are automatically saved on the history list by the responsible functions. The **UNDO** command works on itself the same way: it recovers the saved information and performs the corresponding inverses. Thus, **UNDO** is effective on itself, so that you can **UNDO** an **UNDO**, and **UNDO** that, etc.

Only when you attempt to undo an operation does the Exec check to see whether any information has been saved. If none has been saved, and you have specifically named the event you want undone, the Exec types **nothing saved**. (When you just type **UNDO**, the Exec only tries to undo the last operation.)

UNDO watches evaluation using **CL:EVALHOOK** (thus, calling **CL:EVALHOOK** cannot be undone). Each form given to **EVAL** is

examined against the list **LISPXFNS** to see if it has a corresponding undoable version. If an undoable version of a call is found, it is called with the same arguments instead of the original. Therefore, before evaluating all subforms of your input, the Exec substitutes the corresponding undoable call for any destructive operation. For example, if you type **(DEFUN FOO ...)**, undoable versions of the forms that set the definition into the symbol function cell are evaluated. **FOO**'s function definition itself is not made undoable.

Undoing in Programs

There are two ways to make a program undoable. The simplest method is to wrap the program's form in the **UNDOABLY** macro. The other is to call undoable versions of destructive operations directly.

(XCL:UNDOABLY *&REST FORMS***)** [Macro]

Executes the forms in *FORMS* using undoable versions of all destructive operations. This is done by "walking" (see **WALKFORM**) all of the *FORMS* and rewriting them to use the undoable versions of destructive operations (**LISPXFNS** makes the association).

(STOP-UNDOABLY *&REST FORMS* **)** [Macro]

Normally executes as **PROGN**; however, within an **UNDOABLY** form, explicitly causes *FORMS* not to be done undoably. Turns off rewriting of the *FORMS* to be undoable inside an **UNDOABLY** macro.

Undoable Versions of Common Functions

Efficiency and overhead are serious considerations for the execution of a user program. Thus, the programmer may need more control over the saving of undo information than that provided by the **UNDOABLY** macro.

To make a function undoable, you can simply substitute the corresponding undoable function if you want to make a destructive operation in your own program undoable. When the undoable function is called, it will save the undo information in the current event on the history list.

Various operations, most notably **SETF**, have undoable versions. The following undoable macros are initially available:

UNDOABLY-POP
UNDOABLY-PUSH
UNDOABLY-PUSHNEW
UNDOABLY-REMF
UNDOABLY-ROTATEF
UNDOABLY-SHIFTF
UNDOABLY-DECF
UNDOABLY-INCF

UNDOABLY-SET-SYMBOL
UNDOABLY-MAKUNBOUND
UNDOABLY-FMAKUNBOUND
UNDOABLY-SETQ
XCL:UNDOABLY-SETF
UNDOABLY-PSETF
UNDOABLY-SETF-SYMBOL-FUNCTION
UNDOABLY-SETF-MACRO-FUNCTION

Note: Many destructive Common Lisp functions do not currently have undoable versions, e.g., **CL:NREVERSE**, **CL:SORT**, etc. The current list of undoable functions is saved on the association list **LISPXFNS**.

Modifying the UNDO Facility

You will usually wish to extend the **UNDO** facility after creating a form whose side effects it might be desirable to undo, for instance a file renaming function.

An undoable version of the function needs to be written. This can be done by explicitly saving previous state information away, or by renaming calls in the function to their undoable equivalent. Undo information should be saved on the history list using **IL:UNDOSAVE**.

You must then hook the undoable version of the function into the undo facility. You do this by either using the **IL:LISPXFNS** association list, or in the case of a **SETF** modifier, on the **IL:UNDOABLE-SETF-INVERSE** property of the **SETF** function.

LISPXFNS [Variable]

Contains an association list which maps from destructive operations to their undoable form. Initially this list contains:

((CL:POP . UNDOABLY-POP)
(CL:PSETF . NDOABLY-PSETF)
(CL:PUSH . UNDOABLY-PUSH)
(CL:PUSHNEW . UNDOABLY-PUSHNEW)
((CL:REMF) . UNDOABLY-REMF)
(CL:ROTATEF . UNDOABLY-ROTATEF)
(CL:SHIFTF . UNDOABLY-SHIFTF)
(CL:DECF . UNDOABLY-DECF)
(CL:INCF . UNDOABLY-INCF)
(CL:SET . UNDOABLY-SET-SYMBOL)
(CL:MAKUNBOUND . UNDOABLY-MAKUNBOUND)
(CL:FMAKUNBOUND . UNDOABLY-FMAKUNBOUND)
. . . plus the original Interlisp undo associations)

(XCL:UNDOABLY-SETF *PLACE VALUE ...* **)** [Macro]

Like **CL:SETF** but saves information so it may be undone. **UNDOABLY-SETF** uses undoable versions of the setf function located on the **UNDOABLE-SETF-INVERSE** property of the function being **SETF**ed. Initially these **SETF** names have such a property:

CL:SYMBOL-FUNCTION - UNDOABLY-SETF-SYMBOL-FUNCTION

CL:MACRO-FUNCTION - UNDOABLY-SETF-MACRO-FUNCTION

(UNDOABLY-SETQ *&REST FORMS***)** [Function]

Typed-in **SETQ**s (and **SETF**s on symbols) are made undoable by substituting a call to **UNDOABLY-SETQ**. **UNDOABLY-SETQ** operates like **SETQ** on lexical variables or those with dynamic bindings; it only saves information on the history list for changes to global, "top-level" values.

(UNDOSAVE *UNDOFORM HISTENTRY***)** [Function]

Adds the undo information *UNDOFORM* to the **SIDE** property of the history event *HISTENTRY*. If there is no **SIDE** property, one is created. If the value of the **SIDE** property is **NOSAVE**, the information is not saved. *HISTENTRY* specifies an event. If *HISTENTRY*=**NIL**, the value of **LISPXHIST** is used. If both *HISTENTRY* and **LISPXHIST** are **NIL**, **UNDOSAVE** is a no-op.

The form of *UNDOFORM* is **(***FN* **.** *ARGS***)**. Undoing is done by performing **(APPLY (CAR** *UNDOFORM***) (CDR** *UNDOFORM***))**.

\#UNDOSAVES [Variable]

The value of **\#UNDOSAVES** is the maximum number of *UNDOFORM*s to be saved for a single event. When the count of *UNDOFORM*s reaches this number, **UNDOSAVE** prints the message **CONTINUE SAVING?**, asking if you want to continue saving. If you answer **NO** or default, **UNDOSAVE** discards the previously saved information for this event, and makes **NOSAVE** be the value of the property **SIDE**, which disables any further saving for this event. If you answer **YES**, **UNDOSAVE** changes the count to -1, which is then never incremented, and continues saving. The purpose of this feature is to avoid tying up large quantities of storage for operations that will never need to be undone.

If **\#UNDOSAVES** is negative, then when the count reaches **(ABS \#UNDOSAVES)**, **UNDOSAVE** simply stops saving without printing any messages or other interactions. **\#UNDOSAVES**=**NIL** is equivalent to **\#UNDOSAVES**=infinity. **\#UNDOSAVES** is initially **NIL**.

The configuration described here has been found to be a very satisfactory one. You pay a very small price for the ability to undo what you type in, since the interpreted evaluation is simply watched for destructive operations, or if you wish to protect yourself from malfunctioning in your own programs, you can explicitly call, or have your program rewritten to explicitly call, undoable functions.

Undoing Out of Order

UNDOABLY-SETF operates undoably by saving (on the history list) the cell that is to be changed and its original contents. Undoing an **UNDOABLY-SETF** restores the saved contents.

This implementation can produce unexpected results when multiple modifications are made to the same piece of storage and then undone out of order. For example, if you type **(SETF (CAR FOO) 1)**, followed by **(SETF (CAR FOO) 2)**, then undo both events by undoing the most recent event first, then undoing the older event, **FOO** will be restored to its state before either event operated. However if you undo the first event, *then* the second event, **(CAR FOO)** will be **1**, since this is what was in **CAR** of **FOO** before **(UNDOABLY-SETF (CAR FOO) 2)** was executed. Similarly, if you type **(NCONC FOO '(1))**, followed by **(NCONC FOO '(2))**, undoing just **(NCONC FOO '(1))** will remove both **1** and **2** from **FOO**. The problem in both cases is that the two operations are not independent.

In general, operations are always independent if they affect different lists or different sublists of the same list. Undoing in reverse order of execution, or undoing independent operations, is always guaranteed to do the right thing. However, undoing dependent operations out of order may not always have the predicted effect.

Format and Use of the History List

LISPXHISTORY [Variable]

The Exec currently uses one primary history list, **LISPXHISTORY** for the storing events.

The history list is in the form **(***EVENTS EVENT# SIZE MOD***)**, where *EVENTS* is a list of events with the most recent event first, *EVENT#* is the event number for the most recent event on *EVENTS*, *SIZE* is the the maximum length *EVENTS* is allowed to grow. *MOD* is is the maximum event number to use, after which event numbers roll over. **LISPXHISTORY** is initialized to **(NIL 0 100 1000)**.

The history list has a maximum length, called its time-slice. As new events occur, existing events are aged, and the oldest events are forgotten. The time-slice can be changed with the function **CHANGESLICE**. Larger time-slices enable longer memory spans, but tie up correspondingly greater amounts of storage. Since a user seldom needs really ancient history, a relatively small time-slice such as 30 events is usually adequate, although some users prefer to set the time-slice as large as 200 events.

Each individual event on *EVENTS* is a list of the form (*INPUT ID VALUE . PROPS*). For Exec events, *ID* is a list (*EVENT-NUMBER EXEC-ID*). The *EVENT-NUMBER* is the number of the event, while the *EXEC-ID* is a string that uniquely identifies the Exec. (The *EXEC-ID* is used to identify which events belong to the "same" Exec.) *VALUE* is the (first) value of the event. *PROPS* is a property list used to associate other information with the event (described below).

INPUT is the input sequence for the event. Normally, this is just the input that the user typed-in. For an APPLY-format input this is a list consisting of two expressions; for an EVAL-format input, this is a list of just one expression; for an input entered as list of atoms, *INPUT* is simply that list. For example,

User Input	*INPUT* is:
LIST(1 2)	**(LIST (1 2))**
(LIST 1 1)	**((LIST 1 1))**
DIR "{DSK}<LISPFILES>"^{cr}	**(DIR "{DSK}<LISPFILES>")**

If you type in an Exec command that executes other events (**REDO**, **USE**, etc.), several events might result. When there is more than one input, they are wrapped together into one invocation of the **DO-EVENTS** command.

The same convention is used for representing multiple inputs when a **USE** command involves sequential substitutions. For example, if you type **FBOUNDP(FOO)** and then **USE FIE FUM FOR FOO**, the input sequence that will be constructed is **DO-EVENTS (EVENT FBOUNDP (FIE)) (EVENT FBOUNDP (FUM))**, which is the result of substituting **FIE** for **FOO** in **(FBOUNDP (FOO))** concatenated with the result of substituting **FUM** for **FOO** in **(FBOUNDP (FOO))**.

PROPS is a property list of the form **(***$PROPERTY_1$ $VALUE_1$ $PROPERTY_2$ $VALUE_2$* **...)**, that can be used to associate arbitrary information with a particular event. Currently, the following properties are used by the Exec:

SIDE A list of the side effects of the event. See **UNDOSAVE**.

LISPXPRINT Used to record calls to **EXEC-FORMAT**, and printed by the **??** command.

Making or Changing an Exec

(XCL:ADD-EXEC &KEY *PROFILE REGION TTY ID***)** [Function]

Creates a new process and window with an Exec running in it. *PROFILE* is the type of the Exec to be created (see below under XCL:SET-EXEC-TYPE). *REGION* optionally gives the shape and location of the window to be used. If not provided the user will be prompted. *TTY* is a flag, which, if true, causes the tty to be given to the new Exec process. *ID* is a string identifier to use for events generated in this exec. *ID* defaults to the number given to the Exec process created.

(XCL:EXEC *&KEY WINDOW PROMPT COMMAND-TABLES ENVIRONMENT PROFILE TOP-LEVEL-P TITLE FUNCTION ID***)** [Function]

This is the main entry to the Exec. The arguments are:

WINDOW defaults to the current TTY display stream, or can be provided a window in which the Exec will run.

PROMPT is the prompt to print.

COMMAND-TABLES is a list of hash-tables for looking up commands (e.g., ***EXEC-COMMAND-TABLE*** or ***DEBUGGER-COMMAND-TABLE***).

ENVIRONMENT is a lexical environment used to evaluate things in.

READTABLE is the default readtable to use (defaults to the "Common Lisp" readtable).

PROFILE is a way to set the Exec's type (see above, "Multiple Execs and the Exec's Type").

TOP-LEVEL-P is a boolean, which should be true if this Exec is at the top level.

TITLE is an identifying title for the window title of the Exec.

FUNCTION is a function used to actually evaluate events, default is **EVAL-INPUT**.

ID is a string identifier to use for events generated in this Exec. *ID* defaults to the number given to the Exec process.

XCL:*PER-EXEC-VARIABLES* [Variable]

A list of pairs of the form **(***VAR INIT***)**. Each time an Exec is entered, the variables in ***PER-EXEC-VARIABLES*** are rebound to the value returned by evaluating *INIT*. The initial value of ***PER-EXEC-VARIABLES*** is:

```
((*PACKAGE* *PACKAGE*)
 (* *)
 (** **)
 (*** ***)
 (+ +)
 (++ ++)
 (+++ +++)
 (- -)
 (/ /)
 (// //)
 (/// ///)
 (HELPFLAG T)
 (*EVALHOOK* NIL)
 (*APPLYHOOK* nil)
 (*ERROR-OUPUT* *TERMINAL-IO*)
 (*READTABLE* *READTABLE*)
 (*package* *package*)
 (*eval-function* *eval-function*)
 (*exec-prompt* *exec-prompt*)
 (*debugger-prompt* *debugger-prompt*))
```

Most of these cause the values to be (re)bound to their current value in any inferior Exec, or to **NIL**, their value at the "top level".

XCL:*EVAL-FUNCTION* [Variable]

Bound to the function used by the Exec to evaluate input. Typically in an INTERLISP Exec this is **IL:EVAL**, and in a Common Lisp Exec, **CL:EVAL**.

XCL:*EXEC-PROMPT* [Variable]

Bound to the string printed by the Exec as a prompt for input. Typically in an INTERLISP Exec this is " ← ", and in a Common Lisp Exec, "> ".

XCL:*DEBUGGER-PROMPT* [Variable]

Bound to the string printed by the debugger Exec as a prompt for input. Typically in an INTERLISP Exec this is " ← : ", and in a Common Lisp Exec, ": ".

(XCL:EXEC-EVAL *FORM &OPTIONAL ENVIRONMENT***)** [Function]

Evaluates *FORM* (using **EVAL**) in the lexical environment *ENVIRONMENT* the same as though it were typed in to **EXEC**, i.e., the event is recorded, and the evaluation is made undoable by substituting the UNDOABLE-functions for the corresponding destructive functions. **XCL:EXEC-EVAL** returns the value(s) of the form, but does not print it, and does not reset the variables *, **, ***, etc.

(XCL:EXEC-FORMAT *CONTROL-STRING &REST ARGUMENTS***)** [Function]

In addition to saving inputs and values, the Exec saves many system messages on the history list. For example, **FILE CREATED ...**, *FN* **redefined**, *VAR* **reset**, output of **TIME**, **BREAKDOWN**, **ROOM**, save their output on the history list, so that when **??** prints the event, the output is also printed. The function **XCL:EXEC-FORMAT** can be used in user code similarly. **XCL:EXEC-FORMAT** performs (APPLY #'CL:FORMAT *TERMINAL-IO* *CONTROL-STRING ARGUMENTS*) and also saves the format string and arguments on the history list associated with the current event.

(XCL:SET-EXEC-TYPE *NAME***)** [Function]

Sets the type of the current Exec to that indicated by *NAME*. This can be used to set up the Exec to your liking. *NAME* may be an atom or string. Possible names are:

INTERLISP, **IL**

READTABLE INTERLISP

PACKAGE INTERLISP

XCL:*DEBUGGER-PROMPT* "←: "

XCL:*EXEC-PROMPT* "←"

XCL:*EVAL-FUNCTION* IL:EVAL

XEROX-COMMON-LISP, **XCL**

READTABLE XCL

PACKAGE XCL-USER

XCL:*DEBUGGER-PROMPT* ": "

XCL:*EXEC-PROMPT* "> "

XCL:*EVAL-FUNCTION* CL:EVAL

COMMON-LISP, **CL**	*READTABLE* LISP
	PACKAGE USER
	XCL:*DEBUGGER-PROMPT* ": "
	XCL:*EXEC-PROMPT* "> "
	XCL:*EVAL-FUNCTION* CL:EVAL
OLD-INTERLISP-T	*READTABLE* OLD-INTERLISP-T
	PACKAGE INTERLISP
	XCL:*DEBUGGER-PROMPT* "←: "
	XCL:*EXEC-PROMPT* ": "
	XCL:*EVAL-FUNCTION* IL:EVAL

(XCL:SET-DEFAULT-EXEC-TYPE *NAME*) [Function]

Like **XCL:SET-EXEC-TYPE** , but sets the type of Execs created by default, as from the background menu. Initially **XCL**. This can be used in your greet file to set default Execs to your liking.

Editing Exec Input

The Exec features an editor for input which provides completion, spelling correction, help facility, and character-level editing. The implementation is borrowed from the Interlisp module **TTYIN**. This section describes the use of the **TTYIN** editor from the perspective of the Exec.

Editing Your Input

Some editing operations can be performed using any of several characters; characters that are interrupts will, of course, not be read, so several alternatives are given. The following characters may be used to edit your input:

CONTROL-A, BACKSPACE	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	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 multiline input).
ESCAPE	Tries to complete the current word from the spelling list **USERWORDS**. In the case of ambiguity, completes as far as is uniquely determined, or beeps.

UNDO key (on 1108 and 1186)
Middle-blank key (on 1132) — 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; when typed in the middle of a line fills in the remaining text from the old line; when typed following CONTROL-Q or CONTROL-W restores what those commands erased.

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.

If you are already at the end of the input and the expression is balanced except for lacking one or more right parentheses, CONTROL-X adds the required right parentheses to balance and returns.

During most kinds of input, lines are 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 will not get carriage returns in your strings unless you explicitly type them.

Using the Mouse

Editing with the mouse during TTYIN input is slightly different than with other modules. 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 type-in will be inserted at the new position.

MIDDLE or *LEFT*+*RIGHT* — 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 inverted; when you let up, the text goes away. If you let up outside the scope of the text, nothing is deleted (this is how to cancel this operation).

If you hold down *MOVE*, *COPY*, *SHIFT* or *CTRL* while pressing the mouse buttons, you instead get secondary selection, move selection or delete selection. The selection is made by holding the appropriate key down while pressing the mouse buttons *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 the selected text is highlighted in a manner indicating what is about to happen. When the selection is complete, release the mouse buttons and then lift up on *MOVE*/*COPY*/*CTRL*/*SHIFT* and the appropriate action will occur:

COPY or *SHIFT* — The selected text is inserted as if it were typed. The text is highlighted with a broken underline during selection.

CTRL — The selected text is deleted. The text is complemented during selection.

MOVE or *CTRL*+*SHIFT* — Combines copy and delete. The selected text is moved to the caret.

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 the UNDO key (on the Xerox 1108/1186/1185) or the Middle-blank key (on the Xerox 1132). This is the same key that retrieves the previous buffer when issued at the end of a line.

Editing Commands

A number of characters have special effects while typing to the Exec. Some of them merely move the caret inside the input stream. While caret positioning can often be done more conveniently with the mouse, some of the commands, such as the case changing commands, can be useful for modifying the input.

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 meta-*CHAR*. The notation $ stands for the ESCAPE/EXPAND key. Most commands can be preceded by numbers or escape (means infinity), only the first of which requires the meta key (or the edit prefix). Some commands also accept negative arguments, but some only look at the magnitude of the argument. Most of these commands are confined to work within one line of text unless otherwise noted.

Cursor Movement Commands

[bs]	Backs up one (or n) characters.
[space]	Moves forward one (or n) characters.
[^]	Moves up one (or n) lines.
[lf]	Moves down one (or n) lines.
[(]	Moves back one (or n) words.
[)]	Moves ahead one (or n) words.
[tab]	Moves to end of line; with an argument moves to nth end of line; [$tab] goes to end of buffer.
[control-L]	Moves to start of line (or nth previous, or start of buffer).
[{] and [}]	Goes to start and end of buffer, respectively (like [$control-L] and [$tab]).
[[ ] (meta-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 "Assorted Flags".)
[ ]] (meta-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].

Buffer Modification Commands

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

[A] or [R] Repeats the last S, B, or Z command, regardless of any intervening input.

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

[cr] When the buffer is empty is the same as undo i.e. restores buffer'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 undo<cr> without the meta 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 odd cases, such as tabs.

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

[L] Puts the current word, or n words on line, in lower case. [$L] puts the rest of the linein lower case; or if given at the end of line puts the entire line in lower case.

[U] Analogous to [L], for putting word, line, or portion of line in upper case.

[C] Capitalizes. If you give it an argument, only the first 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 buffer. No other arguments are handled.

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

Miscellaneous Commands

[P] Prettyprints 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] Refreshes 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 window; in some circumstances, you may need to refresh the line for best results.

[control-Y] Gets an Interlisp Exec.

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

[←] Adds the current word to the spelling list **USERWORDS**. With zero argument, removes word. See **TTYINCOMPLETEFLG** .

Useful Macros

If the event is considered short enough, the Exec command **FIX** will load the buffer with the event's input, rather than calling the structure editor. If you really wanted the Lisp editor for your fix, you can say **FIX** *EVENT* **-** |**TTY:**|.

?= Handler

Typing the characters ?=<cr> displays the arguments to the function currently in progress. Since TTYIN wants you to be able to continue editing the buffer after a ?=, it prints the arguments below your type-in and then puts the cursor back where it was when ?= was typed.

Assorted Flags

These flags control aspects of TTYIN's behavior. Some have already been mentioned. In Interlisp-D, the flags are all initially set to **T**.

?ACTIVATEFLG [Variable]

If true, enables the feature whereby ? lists alternative completions from the current spelling list.

SHOWPARENFLG [Variable]

If true, then whenever you are typing Lisp input and type a right parenthesis, TTYIN will briefly move the cursor to the matching parenthesis, 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 will never notice it).

USERWORDS [Variable]

USERWORDS contains words you mentioned recently: functions you have defined 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 "ED(xx$)") or type a call to it. If there is no completion for the current word from **USERWORDS**, or there is more than one possible completion, TTYIN beeps. If typed when not inside a word, Escape completes to the value of **LASTWORD**, i.e., the last thing you typed that the Exec noticed, except that Escape at the beginning of the line is left alone (it is an Old Interlisp Exec 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 characters.

You may explicitly add words to **USERWORDS** yourself that would not 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 definition 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, if 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 off the end if they haven't been used (they are used if **FIXSPELL** corrects to one, or you use <escape> to complete one).

[This page intentionally left blank]