14. ERRORS AND DEBUGGING

Occasionally, while a program is running, an error occurs which stops the computation. Errors can be caused in different ways. A coding mistake may have caused the wrong arguments to be passed to a function, or caused the function to attempt something illegal. For example, `PLUS` will cause an error if its arguments are not numbers. It is also possible to interrupt a computation by typing one of the "interrupt characters," such as Control-D or Control-E (Medley interrupt characters are listed in Chapter 30). Finally, you can specify that certain functions automatically cause an error whenever they are entered (see Chapter 15). This facilitates debugging by allowing you to examine the context within the computation.

When an error occurs, the system can either reset and unwind the stack, or go into a "break", and attempt to debug the program. You can modify the mechanism that decides whether to unwind the stack or break, and is described in the Controlling When to Break section in this chapter. Within a break, Medley offers an extensive set of "break commands".

This chapter explains what happens when errors occur. It also tells you how to handle program errors using breaks and break commands. The debugging capabilities of the break window facility are described, as well as the variables that control its operation. Finally, advanced facilities for modifying and extending the error mechanism are presented.

Breaks

One of the most useful debugging facilities in Medley is the ability to put the system into a "break", stopping a computation at any point, allowing you to interrogate the state of the world and affect the course of the computation. When a break occurs, a "break window" (see the Break Windows section below) is brought up near the TTY window of the broken process. The break window looks like a top-level executive window, except that the prompt character is **":"** instead of "←" as in the top-level executive. A break saves the environment where the break occurred, so that you may evaluate variables and expressions in the borken environment. In addition, the break program recognizes a number of useful "break commands", providing an easy way to interrogate the state of the broken computation.

Breaks may be entered in several ways. Some interrupt characters (Chapter 30) automatically cause a break whenever you type them. Function errors may also cause a break, depending on the depth of the computation (see Controlling When to Break below). Finally, Medley provides facilities which make it easy to "break" suspect functions so that they always cause a break whenever they are entered.

Within a break you have access to all of the power of Medley; you can do anything you can do at the top-level executive. For example, you can evaluate an expression, call the editor, change the function, and evaluate the expression again, all without leaving the break. You can also type in commands like `REDO`, and `UNDO` (Chapter 13), to redo or undo previously executed events, including break commands.

Similarly, you can prettyprint functions, define new functions or redefine old ones, load a file, compile functions, time a computation, etc. In addition, you can examine the stack (see Chapter 11), and even force a return back to some higher function via the functions `RETFROM` or `RETEVAL`.

Once a break occurs, *you* are in complete control of the flow of the computation, and the computation will not proceed without specific instruction from you. If you type in an expression whose evaluation causes an error, the break is maintained. Similarly if you abort a computation initiated from within the break (by typing Control-E), the break is maintained. Only if you give one of the commands that exits from the break, or evaluates a form which does a RETFROM or RETEVAL out of BREAK1, will the computation continue. Also, BREAK1 does not "turn off" Control-D, so a Control-D will force an immediate return to the top level.

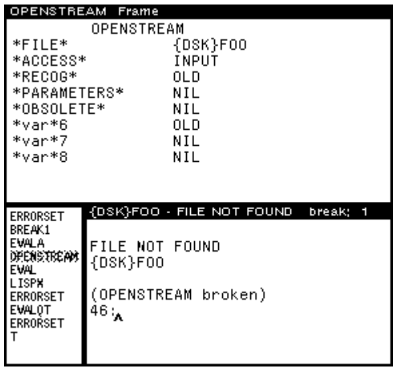
Break Windows

When a break occurs, a break window is brought up near the TTY window of the borken process and the terminal stream switched to it. The title of the break window is changed to the name of the broken function and the reason for the break. If a break occurs under a previous break, a new break window is created.

If a break is caused by a storage full error, the display break package will not try to open a new break window, since this would cause an infinite loop.

While in a break window, clicking the middle button brings up a menu of break commands: EVAL, EDIT, revert, ↑, OK, BT, BT!, and ?=. Clicking on these commands is equivalent to typing the corresponding break commandm, except BT and BT! which behave differently from the typed-in commands (see Break Commands below).

The BT and BT! menu commands 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 the backtrace menu, it is grayed 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 appear as *var* *N*). The "backtrace frame" window is an inspect window (see Chapter 26). In this window, the left button is used to select the name of the function, the names of the variables or the values of the variables. For example, below is a picture of a break window with a backtrace menu created by BT. The OPENSTREAM stack frame has been selected, so its variables are shown in an inspect window on top of the break window:



```
OPENSTREAM  Frame
           OPENSTREAM
*FILE*              {DSK}FOO
*ACCESS*            INPUT
*RECOG*             OLD
*PARAMETERS*        NIL
*OBSOLETE*          NIL
*var*6              OLD
*var*7              NIL
*var*8              NIL
```

```
{DSK}FOO - FILE NOT FOUND   break: 1

FILE NOT FOUND
{DSK}FOO

(OPENSTREAM broken)
46:
```

After selecting an item, the middle button brings up a menu of commands that apply to the selected item. If the function name is selected, you are given a choice of editing the function or seeing the compiled code with INSPECTCODE (Chapter 26). If you edit the function in this way, the editor is called in the broken process, so variables evaluated in the editor are in the broken process.

If a variable name is selected, the command SET is 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. Exercise caution in setting variables in other than your own code.

If a value is selected, the inspector is called on the selected value.

The internal break variable LASTPOS (see the section below) is set to the selected backtrace menu frame 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 typed-in break commands BT, BTV, etc. use the value of LASTPOS to determine where to start listing the stack, so selecting a stack frame name in the backtrace menu affects these commands.

Break Commands

The basic function of the break package is BREAK1. BREAK1 is just another Interlisp function, not a special system feature like the interpreter or the garbage collector. It has arguments, and returns a value, the same as any other function. For more information on the function BREAK1, see Creating Breaks with BREAK1 below.

The value returned by BREAK1 is called "the value of the break." You can specify this value explicitly by using the RETURN break command (see below). But in most cases, the value of a break is given implicitly, via a GO or OK command, and is the result of evaluating "the break expression." The break expression, stored in the variable BRKEXP, is an expression equivalent to the computation that would have taken place had no break occurred. For example, if you break on the function FOO, the break expression is the body of the definition of FOO. When you type OK or GO, the body of FOO is evaluated, and its value returned as the value of the break, i.e., to whatever function called FOO. BRKEXP is set up by the function that created the call to BREAK1. For functions broken with BREAK or TRACE, BRKEXP is equivalent to the body of the definition of the broken function (see Chapter 15). For functions broken with BREAKIN, using BEFORE or AFTER, BRKEXP is NIL. For BREAKIN AROUND, BRKEXP is the indicated expression (see Chapter 15).

BREAK1 recognizes a large set of break commands. These are typed in *without* parentheses. In order to facilitate debugging of programs that perform input operations, the carriage return that is typed to complete the GO, OK, EVAL, etc. commands is discarded by BREAK1, so that it will not be part of the input stream after the break.

GO [Break Command]

Evaluates BRKEXP, prints its value, and returns it as the value of the break. Releases the break and allows the computation to proceed.

OK [Break Command]

Same as GO except that the value of BRKEXP is not printed.

EVAL [Break Command]

Same as OK except that the break is maintained after the evaluation. The value of EVAL is bound to the local variable !VALUE, which you can interrogate. Typing GO or OK following EVAL will not cause BRKEXP to be reevaluated, but simply returns the value of !VALUE as the value of the break. Typing another EVAL will cause reevaluation. EVAL is useful when you are not sure whether the break will produce the correct value and want to examine it before continuing with the computation.

RETURN *FORM* [Break Command]

FORM is evaluated, and returned as the value of the break. For example, one could use the EVAL command and follow this with RETURN (REVERSE !VALUE).

↑ [Break Command]

Calls ERROR! and aborts the break, making it "go away" without returning a value. This is a useful way to unwind to a higher level break. All other errors, including those encountered while executing the GO, OK, EVAL, and RETURN commands, maintain the break.

The following four commands refer to "the broken function", whose name is stored in the BREAK1 argument BRKFN.

!GO [Break Command]

The broken function is unbroken, the break expression is evaluated, the function is rebroken, and then the break is exited with the value printed.

!OK [Break Command]

The broken function is unbroken, the break expression is evaluated, the function is rebroken, and then the break is exited without the value printed.

UB [Break Command]

Unbreaks the broken function.

@ [Break Command]

Resets the variable LASTPOS, which establishes a context for the commands ?=, ARGS, BT, BTV, BTV*, EDIT, and IN? described below. LASTPOS is the position of a function call on the stack. It is initialized to the function just before the call to BREAK1, i.e., (STKNTH -1 'BREAK1).

When control passes from BREAK1, e.g. as a result of an EVAL, OK, GO, REVERT, ↑ command, or via a RETFROM or RETEVAL you type in, (RELSTK LASTPOS) is executed to release this stack pointer.

@ treats the rest of the teletype line as its argument(s). It first resets LASTPOS to (STKNTH -1 'BREAK1) and then for each atom on the line, @ searches down the stack for a call to that atom. The following atoms are treated specially:

`@`	Do not reset `LASTPOS` to `(STKNTH -1 'BREAK1)` but leave it as it was, and continue searching from that point.
a number *`N`*	If negative, move `LASTPOS` down the stack *`N`* frames. If positive, move `LASTPOS` up the stack *`N`* frames.
`/`	The next atom on the line (which should be a number) specifies that the *previous* atom should be searched for that many times. For example, "`@ FOO / 3`" is equivalent to "`@ FOO FOO FOO`".
`=`	Resets `LASTPOS` to the *value* of the next expression, e.g., if the value of `FOO` is a stack pointer, "`@ = FOO FIE`" will search for `FIE` in the environment specified by (the value of) `FOO`.

For example, if the push-down stack looks like:

```
[9]   BREAK1
[8]   FOO
[7]   COND
[6]   FIE
[5]   COND
[4]   FIE
[3]   COND
[2]   FIE
[1]   FUM
```

then "`@ FIE COND`" will set `LASTPOS` to the position corresponding to *`[5]`*; "`@ @ COND`" will then set `LASTPOS` to *`[3]`*; and "`@ FIE / 3 -1`" to *`[1]`*.

If `@` cannot successfully complete a search for function *`FN`*, it searches the stack again from that point looking for a call to a function whose name is a possible misspelling of *`FN`* (see spelling correction in Chapter 20). If the search is still unsuccessful, `@` types `(`*`FN`*` NOT FOUND)`, and then aborts.

When `@` finishes, it types the name of the function at `LASTPOS`, i.e., `(STKNAME LASTPOS)`.

`@` can be used on `BRKCOMS` (see Creating Breaks with BREAK1 below). In this case, the *next* command on `BRKCOMS` is treated the same as the rest of the teletype line.

?= [Break Command]

This is a multi-purpose command. Its most common use is to interrogate the value(s) of the arguments of the broken function. For example, if `FOO` has three arguments `(X Y Z)`, then typing `?=` to a break on `FOO` will produce:

```
:?=
X = value of X
Y = value of Y
Z = value of Z
:
```

?= operates on the rest of the teletype line as its arguments. If the line is empty, as in the above case, it operates on all of the arguments of the broken function. If the you type ?= X (CAR Y), you will see the value of X, and the value of (CAR Y). The difference between using ?= and typing X and (CAR Y) directly to BREAK1 is that ?= evaluates its inputs as of the stack frame LASTPOS, i.e., it uses STKEVAL. This provides a way of examining variables or performing computations *as of a particular point on the stack.* For example, @ FOO / 2 followed by ?= X will allow you to examine the value of X in the previous call to FOO, etc.

?= also recognizes numbers as referring to the correspondingly numbered argument, i.e., it uses STKARG in this case. Thus

```
:@ FIE
FIE
:?= 2
```

will print the name and value of the second argument of FIE.

?= can also be used on BRKCOMS (see Creating Breaks with BREAK1 below), in which case the next command on BRKCOMS is treated as the rest of the teletype line. For example, if BRKCOMS is (EVAL ?= (X Y) GO), BRKEXP is evaluated, the values of X and Y printed, and then the function exited with its value being printed.

?= prints variable values using the function SHOWPRINT (see Chapter 25), so that if SYSPRETTYFLG = T, the value is prettyprinted.

?= is a universal mnemonic for displaying argument names and their corresponding values. In addition to being a break command, ?= is an edit macro that prints the argument names and values for the current expression (see Chapter 16), and a read macro (actually ? is the read macro character) which does the same for the current level list being read.

PB [Break Command]

Prints the bindings of a given variable. Similar to ?=, except ascends the stack starting from LASTPOS, and, for each frame in which the given variable is bound, prints the frame name and value of the variable (with PRINTLEVEL reset to (2 . 3)), e.g.

```
:PB FOO
@  FN1:  3
@  FN2:  10
@  TOP:  NOBIND
```

PB is also a programmer’s assistant command (see Chapter 13) that can be used when not in a break. PB is implemented via the function PRINTBINDINGS.

BT [Break Command]

Prints a backtrace of function names starting at LASTPOS. The value of LASTPOS is changed by selecting an item from the backtrace menu (see the Break Window Variables section below) or by the @ command. The several nested calls in system packages such as break, edit, and the top level executive appear as the single entries **BREAK**, **EDITOR**, and **TOP** respectively.

BTV [Break Command]

Prints a backtrace of function names *with* variables beginning at LASTPOS.

The value of each variable is printed with the function SHOWPRINT (see Chapter 25), so that if SYSPRETTYFLG = T, the value is prettyprinted.

BTV+ [Break Command]

Same as BTV except also prints local variables and arguments to SUBRs.

BTV* [Break Command]

Same as BTV except prints arguments to local variables.

BTV! [Break Command]

Same as BTV except prints *everything* on the stack.

BT, BTV, BTV+, BTV*, and BTV! all take optional functional arguments. Use these arguments to choose functions to be *skipped* on the backtrace. As the backtrace scans down the stack, the name of each stack frame is passed to each of the arguments of the backtrace command. If any of these functions returns a non-NIL value, then that frame is skipped, and not shown in the backtrace. For example, BT EXPRP will skip all functions definied by expr definitions, BTV (LAMBDA (X) (NOT (MEMB X FOOFNS))) will skip all but those functions on FOOFNS. If used on BRKCOMS (see Creating Breaks with BREAK1 below) the functional argument is no longer optional, i.e., the next element on BRKCOMS must either be a list of functional arguments, or NIL if no functional argument is to be applied.

For BT, BTV, BTV+, BTV*, and BTV!, if Control-P is used to change a printlevel during the backtrace, the printlevel is restored after the backtrace is completed.

The value of BREAKDELIMITER, initially the carriage return character, is printed to delimit the output of ?= and backtrace commands. You can reset it (e.g. to a comma) for more linear output.

ARGS [Break Command]

Prints the names of the variables bound at LASTPOS, i.e., (VARIABLES LASTPOS) (see Chapter 11). For most cases, these are the arguments to the function entered at that position, i.e., (ARGLIST (STKNAME LASTPOS)).

REVERT [Break Command]

Goes back to position LASTPOS on stack and reenters the function called at that point with the arguments found on the stack. If the function is not already broken, REVERT first breaks it, and then unbreaks it after it is reentered.

REVERT can be given the position using the conventions described for @, e.g., REVERT FOO -1 is equivalent to @ FOO -1 followed by REVERT.

REVERT is useful for restarting a computation in the situation where a bug is discovered at some point *below* where the problem actually occurred. REVERT essentially says "go back there and start over in a break." REVERT will work correctly if the names or arguments to the function, or even its function type, have been changed.

ORIGINAL [Break Command]

For use in conjunction with BREAKMACROS (see Creating Breaks with BREAK1 below). Form is (ORIGINAL . *COMS*). *COMS* are executed without regard for BREAKMACROS. Useful for redefining a break command in terms of itself.

EDIT [Break Command]

Designed for use in conjunction with breaks caused by errors. Facilitates editing the expression causing the break:

```
NON-NUMERIC ARG
NIL
(IPLUS BROKEN)
:EDIT
IN FOO...
(IPLUS X Z)
EDIT
*(3 Y)
*OK
FOO
:
```

and you can continue by typing OK, EVAL, etc.

This command is very simple conceptually, but its implementation is complicated by all of the exceptional cases involving interactions with compiled functions, breaks on user functions, error breaks, breaks within breaks, et al. Therefore, we shall give the following simplified explanation which will account for 90% of the situations arising in actual usage. For those others, EDIT will print an appropriate failure message and return to the break.

EDIT begins by searching up the stack beginning at LASTPOS (set by @ command, initially position of the break) looking for a form, i.e., an internal call to EVAL. Then EDIT continues from that point looking for a call to an interpreted function, or to EVAL. It then calls the editor on either the EXPR or the argument to EVAL in such a way as to look for an expression EQ to the form that it first found. It then prints the form, and permits interactive editing to begin. You can then type successive 0's to the editor to see the chain of superforms for this computation.

If you exit from the edit with an OK, the break expression is reset, if possible, so that you can continue with the computation by simply typing OK. (Evaluating the new BRKEXP will involve reevaluating the form that causes the break, so that if (PUTD (QUOTE (FOO)) *BIG-COMPUTATION*) were handled by EDIT, *BIG-COMPUTATION* would be reevaluated.) However, in some situations, the break expression cannot be reset. For example, if a compiled function FOO incorrectly called PUTD and caused the error Arg not atom followed by a break on PUTD, EDIT might be able to find the form headed by FOO, and also find *that* form in some higher interpreted function. But after you corrected the problem in the FOO-form, if any, you would still not have informed EDIT what to do about the immediate problem, i.e., the incorrect call to PUTD. However, if FOO were *interpreted*, EDIT would find the PUTD form itself, so that when you corrected that form, EDIT could use the new corrected form to reset the break expression.

IN? [Break Command]

Similar to EDIT, but just prints parent form, and superform, but does not call the editor, e.g.,

```
ATTEMPT TO RPLAC NIL
T
(RPLACD BROKEN)
:IN?
FOO:  (RPLACD X Z)
```

Although EDIT and IN? were designed for error breaks, they can also be useful for user breaks. For example, if upon reaching a break on his function FOO, you determine that there is a problem in the *call* to FOO, you can edit the calling form and reset the break expression with one operation by using EDIT.

Controlling When to Break

When an error occurs, the system has to decide whether to reset and unwind the stack, or go into a break. In the middle of a complex computation, it is usually helpful to go into a break, so that you may examine the state of the computation. However, if the computation has only proceeded a little when the error occurs, such as when you mistype a function name, you would normally just terminate a break, and it would be more convenient for the system to simply cause an error and unwind the stack in this situatuation. The decision over whether or not to induce a break depends on the depth of computation, and the amount of time invested in the computation. The actual algorithm is described in detail below; suffice it to say that the parameters affecting this decision have been adjusted empirically so that trivial type-in errors do not cause breaks, but deep errors do.

(**BREAKCHECK** *ERRORPOS ERXN*) [Function]

BREAKCHECK is called by the error routine to decide whether or not to induce a break when a error occurs. *ERRORPOS* is the stack position at which the error occurred; *ERXN* is the error number. Returns T if a break should occur; NIL otherwise.

BREAKCHECK returns T (and a break occurs) if the “computation depth” is greater than or equal to HELPDEPTH. HELPDEPTH is initially set to 7, arrived at empirically by taking into account the overhead due to LISPX or BREAK.

If the depth of the computation is less than HELPDEPTH, BREAKCHECK next calculates the length of time spent in the computation. If this time is greater than HELPTIME milliseconds, initially set to 1000, then BREAKCHECK returns T (and a break occurs), otherwise NIL.

BREAKCHECK determines the “computation depth” by searching back up the stack looking for an ERRORSET frame (ERRORSETs indicate how far back unwinding is to take place when an error occurs, see the Catching Errors section below). At the same time, it counts the number of internal calls to EVAL. As soon as the number of calls to EVAL exceeds HELPDEPTH, BREAKCHECK immediately stops searching for an ERRORSET and returns T. Otherwise, BREAKCHECK continues searching until either an ERRORSET is found or the top of the stack is reached. (If the second argument to ERRORSET is INTERNAL, the ERRORSET is ignored by BREAKCHECK during this search.) BREAKCHECK then counts the number of function calls between the error and the last ERRORSET, or the top of the stack.

The number of function calls plus the number of calls to EVAL (already counted) is used as the "computation depth".

BREAKCHECK determines the computation time by subtracting the value of the variable HELPCLOCK from the value of (CLOCK 2), the number of milliseconds of compute time (see Chapter 12). HELPCLOCK is rebound to the current value of (CLOCK 2) for each computation typed in to LISPX or to a break. The time criterion for breaking can be suppressed by setting HELPTIME to NIL (or a very big number), or by setting HELPCLOCK to NIL. Setting HELPCLOCK to NIL will not have any effect beyond the current computation, because HELPCLOCK is rebound for each computation typed in to LISPX and BREAK.

You can suppress all error breaks by setting the top level binding of the variable HELPFLAG to NIL using SETTOPVAL (HELPFLAG is bound as a local variable in LISPX, and reset to the global value of HELPFLAG on every LISPX line, so just SETQing it will not work.) If HELPFLAG = T (the initial value), the decision whether to cause an error or break is decided based on the computation time and the computation depth, as described above. Finally, if HELPFLAG = BREAK!, a break will always occur following an error.

Break Window Variables

The appearance and use of break windows is controlled by the following variables:

(**WBREAK** *ONFLG*) [Function]

If *ONFLG* is non-NIL, break windows and trace windows are enabled. If *ONFLG* is NIL, break windows are disabled (break windows do not appear, but the executive prompt is changed to "**:**" to indicate that the system is in a break). WBREAK returns T if break windows are currently enabled; NIL otherwise.

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]

This variable controls when and what kind of backtrace menu is automatically brought up. The value of AUTOBACKTRACEFLG can be one of the following:

NIL	The backtrace menu is not automatically brought up (the default).
T	On error breaks the BT menu is brought up.
BT!	On error breaks the BT! menu is brought up.
ALWAYS	The BT menu is brought up on both error breaks and user breaks (calls to functions broken by BREAK).
ALWAYS!	On both error breaks and user breaks the BT! menu is brought up.

BACKTRACEFONT [Variable]

The backtrace menu is printed in the font BACKTRACEFONT.

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. In this case, you 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 (see Chapter 27) whose LEFT and BOTTOM fields are an offset from the LEFT and BOTTOM of the TTY window. The WIDTH and HEIGHT fields 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 you close it. TRACEWINDOW can be set to a particular window to cause the tracing formation to print there.

TRACEREGION [Variable]

The trace window is first created in the region TRACEREGION.

Creating Breaks with BREAK1

The basic function of the break package is BREAK1, which creates a break. A break appears to be a regular executive, with the prompt "**:**", but BREAK1 also detects and interpretes break commands (see the Break Commands section above).

(**BREAK1** *BRKEXP BRKWHEN BRKFN BRKCOMS BRKTYPE ERRORN*) [NLambda Function]

If *BRKWHEN* (evaluated) is non-NIL, a break occurs and commands are then taken from *BRKCOMS* or the terminal and interpreted. All inputs not recognized by BREAK1 are simply passed on to the programmer's assistant.

If *BRKWHEN* is NIL, *BRKEXP* is evaluated and returned as the value of BREAK1, without causing a break.

When a break occurs, if *ERRORN* is a list whose CAR is a number, ERRORMESS (see the Signalling Errors section below) is called to print an identifying message. If *ERRORN* is a list whose CAR is not a number, ERRORMESS1 (see the Signalling Errors section below) is called. Otherwise, no preliminary message is printed. Following this, the message (*BRKFN* broken) is printed.

Since BREAK1 itself calls functions, when one of these is broken, an infinite loop would occur. BREAK1 detects this situation, and prints Break within a break on *FN*, and then simply calls the function without going into a break.

The commands GO, !GO, OK, !OK, RETURN and ↑ are the only ways to leave BREAK1. The command EVAL causes *BRKEXP* to be evaluated, and saves the value on the variable !VALUE. Other commands can be defined for BREAK1 via BREAKMACROS (below).

BRKTYPE is NIL for user breaks, INTERRUPT for Control-H breaks, and ERRORX for error breaks. For breaks when *BRKTYPE* is not NIL, BREAK1 will clear and save the input buffer. If the break returns a value (i.e., is not aborted via ↑ or Control-D) the input buffer is restored.

The fourth argument to BREAK1 is *BRKCOMS*, a list of break commands that BREAK1 interprets and executes as though they were keyboard input. One can think of *BRKCOMS* as another input file which always has priority over the keyboard. Whenever *BRKCOM*S = NIL, BREAK1 reads its next command from the keyboard. Whenever *BRKCOMS* is non-NIL, BREAK1 takes (CAR BRKCOMS) as its next command and sets *BRKCOMS* to (CDR BRKCOMS). For example, suppose you wished to see the value of the variable X *after* a function was evaluated. You could set up a break with *BRKCOMS* = (EVAL (PRINT X) OK), which would have the desired effect. If *BRKCOMS* is non-NIL, the value of a break command is not printed. If you desire to see a value, you must print it yourself, as in the above example. The function TRACE (see Chapter 15) uses *BRKCOMS*: it sets up a break with two commands; the first one prints the arguments of the function, or whatever you specify, and the second is the command GO, which causes the function to be evaluated and its value printed.

Note: If an error occurs while interpreting the *BRKCOMS* commands, *BRKCOMS* is set to NIL, and a full interactive break occurs.

The break package has a facility for redirecting ouput to a file. All output resulting from *BRKCOMS* is output to the value of the variable BRKFILE, which should be the name of an open file. Output due to user type-in is not affected, and will always go to the terminal. BRKFILE is initially T.

BREAKMACROS [Variable]

BREAKMACROS is a list of the form ((*NAME*$_1$ *COM1*$_1$ ... *COM1*$_n$)(*NAME*$_2$ *COM2*$_1$ ... *COM2*$_n$)...). Whenever an atomic command is given to BREAK1, it first searches the list BREAKMACROS for the command. If the command is equal to *NAME*$_i$, BREAK1 simply appends the corresponding commands to the front of *BRKCOMS*, and goes on. If the command is not found on BREAKMACROS, BREAK1 then checks to see if it is one of the built in commands, and finally, treats it as a function or variable as before.

If the command is not the name of a defined function, bound variable, or LISPX command, BREAK1 will attempt spelling correction using BREAKCOMSLST as a spelling list. If spelling correction is unsuccessful, BREAK1 will go ahead and call LISPX anyway, since the atom may also be a misspelled history command.

For example, the command ARGS could be defined by including on BREAKMACROS the form:

```
(ARGS (PRINT (VARIABLES LASTPOS T)))
```

(**BREAKREAD** *TYPE*) [Function]

Useful within BREAKMACROS for reading arguments. If BRKCOMS is non-NIL (the command in which the call to BREAKREAD appears was not typed in), returns the next break command from BRKCOMS, and sets BRKCOMS to (CDR BRKCOMS).

If BRKCOMS is NIL (the command was typed in), then BREAKREAD returns either the rest of the commands on the line as a list (if *TYPE* = LINE) or just the next command on the line (if *TYPE* is not LINE).

For example, the BT command is defined as (BAKTRACE LASTPOS NIL (BREAKREAD 'LINE) 0 T). Thus, if you type BT, the third argument to BAKTRACE is NIL. If you type BT SUBRP, the third argument is (SUBRP).

BREAKRESETFORMS [Variable]

If you are developing programs that change the way a user and Medley normally interact (e.g., change or disable the interrupt or line-editing characters, turn off echoing, etc.), debugging them by breaking or tracing may be difficult, because Medley might be in a "funny" state at the time of the break. BREAKRESETFORMS is designed to solve this problem. You put in BREAKRESETFORMS expressions suitable for use in conjunction with RESETFORM or RESETSAVE (see Changing and Restoring System State below). When a break occurs, BREAK1 evaluates each expression on BREAKRESETFORMS *before* any interaction with the terminal, and saves the values. When the break expression is evaluated via an EVAL, OK, or GO, BREAK1 first restores the state of the system with respect to the various expressions on BREAKRESETFORMS. When control returns to BREAK1, the expressions on BREAKRESETFORMS are *again* evaluated, and their values saved. When the break is exited with an OK, GO, RETURN, or ↑ command, by typing Control-D, or by a RETFROM or RETEVAL you type in, BREAK1 again restores state. Thus the net effect is to make the break invisible with respect to your programs, but nevertheless allow you to interact in the break in the normal fashion.

All user type-in is scanned to make the operations undoable, as described in Chapter 13. At this point, RETFROMs and RETEVALs are also noticed. However, if you type in an expression which calls a function that then does a RETFROM, this RETFROM will not be noticed, and the effects of BREAKRESETFORMS will *not* be reversed.

As mentioned earlier, BREAK1 detects "Break within a break" situations, and avoids infinite loops. If the loop occurs because of an error, BREAK1 simply rebinds BREAKRESETFORMS to NIL, and calls HELP. This situation most frequently occurs when there is a bug in a function called by BREAKRESETFORMS.

SETQ expressions can also be included on BREAKRESETFORMS for saving and restoring system parameters, e.g. (SETQ LISPXHISTORY NIL), (SETQ DWIMFLG NIL), etc. These are handled specially by BREAK1 in that the current value of the variable is saved before the SETQ is executed, and upon restoration, the variable is set back to this value.

Signalling Errors

With the Medley release, Interlisp errors use the Xerox Common Lisp (XCL) error system. Most of the functions still exist for compatibility with previous releases, but the underlying machinery has changed. There are some incompatible differences, especially with respect to error numbers. All errors are now handled by signalling an object of type XCL:CONDITION. This means the error numbers generated are different from the old Interlisp method of registered numbers for well-known errors and error messages for all other errors. The mapping from Interlisp erors to Lisp error conditions is listed in the Error List sections below. The obsolete error numbers still generate error messages, but they are useless.

(**ERRORX** *ERXM*) [Function]

Calls CL:ERROR after first converting *ERXM* into a condition. If *ERXM* is NIL the value of *LAST-CONDITION* is used. If *ERXM* is an Interlisp error descriptor, it is first converted to a condition. If *ERXM* is already a condition, it is passed along unchanged. ERRORX also sets a proceed case for XCL:PROCEED, which will attempt to re-evaluate the caller of ERRORX, much as OK did in older versions of the break package.

(**ERROR** *MESS*$_1$ *MESS*$_2$ *NOBREAK*) [Function]

Prints *MESS*$_1$ (using PRIN1), followed by a space if *MESS*$_1$ is an atom, otherwise a carriage return. Then *MESS*$_2$ is printed (using PRIN1 if *MESS*$_2$ is a string; otherwise PRINT). For example, (ERROR “NON-NUMERIC ARG” T) prints

```
NON-NUMERIC ARG
T
```

and (ERROR 'FOO "NOT A FUNCTION") prints FOO NOT A FUNCTION. If both *MESS*$_1$ and *MESS*$_2$ are NIL, the message printed is simply ERROR.

If *NOBREAK* = T, ERROR prints its message and then calls ERROR! (below). Otherwise it calls (ERRORX '(17 (*MESS*$_1$. *MESS*$_2$))), i.e., generates error number 17, in which case the decision as to whether to break, and whether to print a message, is handled as any other error.

If the value of HELPFLAG (see the Controlling When to Break section above) is BREAK!, a break will always occur, irregardless of the value of *NOBREAK*.

If ERROR causes a break, the “break expression“ is (ERROR MESS$_1$ MESS$_2$ NOBREAK). Using the GO, OK, , or EVAL break commands (see the Break Commands section above) will simply call ERROR again. It is sometimes helpful to design programs that call ERROR such that if the call to ERROR returns (as the result of using the RETURN break command), the operation is tried again. This lrts you fix any problems within the break environment, and try to continue the operation.

(**HELP** *MESS*$_1$ *MESS*$_2$ *BRKTYPE*) [Function]

Prints *MESS*$_1$ and *MESS*$_2$ similar to ERROR, and then calls BREAK1 passing *BRKTYPE* as the BRKTYPE argument. If both *MESS*$_1$ and *MESS*$_2$ are NIL, Help! is used for the message. HELP is a convenient way to program a default condition, or to terminate some portion of a program which the computation is theoretically never supposed to reach.

(**SHOULDNT** *MESS*) [Function]

Useful in situations when a program detects a condition that should never occur. Calls HELP with the message arguments *MESS* and "Shouldn't happen!" and a BRKTYPE argument of 'ERRORX.

(**ERROR!**) [Function]

Equivalent to XCL:ABORT, except that if no ERRORSET or XCL:CATCH-ABORT isa found, it unwinds to the top of the process.

(**RESET**) [Function]

Programmable Control-D; immediately returns to the top level.

LAST-CONDITION [Variable]

Value is the condition object most recently signaled.

(**SETERRORN** *NUM MESS*) [Function]

Converts its arguments into a condition, then sets the value of *LAST-CONDITION* to the result.

(**ERRORMESS** *U*) [Function]

Prints message corresponding to its first argument. For example, (ERRORMESS '(17 T)) would print: T is not a LIST

(**ERRORMESS1** $MESS_1$ $MESS_2$ $MESS_3$) [Function]

Prints the message corresponding to a HELP or ERROR break.

(**ERRORSTRING** *X*) [Function]

Returns as a new string the message corresponding to error number *X*, e.g., (ERRORSTRING 10) = "NON-NUMERIC ARG".

Catching Errors

All error conditions are not caused by program bugs. For some programs, it is reasonable for some errors to occur (such as file not found errors) and it is possible for the program to handle the error itself. There are a number of functions that allow a program to "catch" errors, rather than abort the computation or cause a break.

(**ERRORSET** *FORM FLAG*) [Function]

Performs (EVAL *FORM*). If no error occurs in the evaluation of *FORM*, the value of ERRORSET is a list containing one element, the value of (EVAL *FORM*). If an error did occur, the value of ERRORSET is NIL.

ERRORSET is a lambda function, so its arguments are evaluated *before* it is entered, i.e., (ERRORSET X) means EVAL is called with the *value* of X. In most cases, ERSETQ and NLSETQ (below) are more useful.

Note: Beginning with the Medley release, there are no longer frames named ERRORSET on the stack and any programs that explicity look for them must be changed.

Performance Note: When a call to ERSETQ or NLSETQ is compiled, the form to be evaluated is compiled as a separate function. However, compiling a call to ERRORSET does not compile *FORM*. Therefore, if *FORM* performs a lengthy computation, using ERSETQ or NLSETQ can be much more efficient than using ERRORSET.

The argument *FLAG* controls the printing of error messages if an error occurs. If a *break* occurs below an ERRORSET, the message is printed regardless of the value of *FLAG*.

If *FLAG* = T, the error message is printed; if *FLAG* = NIL, the error message is not printed (unless NLSETQGAG is NIL, see below).

If *FLAG* = INTERNAL, this ERRORSET is ignored for the purpose of deciding whether or not to break or print a message (see the Controlling When to Break section above). However, the ERRORSET is in effect for the purpose of flow of control, i.e., if an error occurs, this ERRORSET returns NIL.

If *FLAG* = NOBREAK, no break will occur, even if the time criterion for breaking is met (the Controlling When to Break section above). *FLAG* = NOBREAK will *not* prevent a break from occurring if the error occurs more than HELPDEPTH function calls below the errorset, since BREAKCHECK will stop searching before it reaches the ERRORSET. To guarantee that no break occurs, you would also either have to reset HELPDEPTH or HELPFLAG.

(**ERSETQ** *FORM*) [NLambda Function]

Evaluates *FORM*, letting a break happen if an error occurs, but 9^ brings you back to the ERSETQ. Performs (ERRORSET '*FORM* T), printing error messages.

(**NLSETQ** *FORM*) [NLambda Function]

Evaluates *FORM*, witout breaking, returning NIL if an error occurs or a list containing *FORM* if no error occurs. Performs (ERRORSET '*FORM* NIL), without printing error messages.

NLSETQGAG [Variable]

If NLSETQGAG is NIL, error messages will print, regardless of the *FLAG* argument of ERRORSET. NLSETQGAG effectively changes all NLSETQs to ERSETQs. NLSETQGAG is initially T.

Changing and Restoring System State

In Medley, a computation can be interrupted / aborted at any point due to an error, or more forcefully, because a Control-D was typed, causing return to the top level. This situation creates problems for programs that need to perform a computation with the system in a "different state", e.g., different radix, input file, readtable, etc. but want to be able to restore the state when the computation has completed. While program errors and Control-E are "caught" by ERRORSETs, Control-D is not. The program could redefine Control-D as a user interrupt (see Chapter 30), check for it, reenable it, and call RESET or something similar. Thus the system may be left in its changed state as a result of the computation being aborted. The following functions address this problem.

These functions cannot handle the situation where their environment is exited via anything other than a normal return, an error, or a reset. Therefore, a RETEVAL, RETFROM, RESUME, etc., will never be seen.

(**RESETLST** *FORM*$_1$ . . . *FORM*$_N$) [NLambda NoSpread Function]

RESETLST evaluates its arguments in order, after setting up an ERRORSET so that any reset operations performed by RESETSAVE (see below) are restored when the forms have been evaluated (or an error occurs, or a Control-D is typed). If no error occurs, the value of RESETLST is the value of *FORM*$_N$, otherwise RESETLST generates an error (after performing the necessary restorations).

RESETLST compiles open.

(**RESETSAVE** *X Y*) [NLambda NoSpread Function]

RESETSAVE is used within a call to RESETLST to change the system state by calling a function or setting a variable, while specifying how to restore the original system state when the RESETLST is exited (normally, or with an error or Control-D).

If *X* is atomic, resets the top level value of *X* to the value of *Y*. For example, (RESETSAVE LISPXHISTORY EDITHISTORY) resets the value of LISPXHISTORY to the value of EDITHISTORY, and provides for the original value of LISPXHISTORY to be restored when the RESETLST completes operation, (or an error occurs, or a Control-D is typed).

Note: If the variable is simply rebound, the RESETSAVE will not affect the most recent binding but will change only the top level value, and therefore probably not have the intended effect.

If *X* is not atomic, it is a form that is evaluated. If *Y* is NIL, *X* must return as its value its "former state", so that the effect of evaluating the form can be reversed, and the system state can be restored, by applying CAR of *X* to the value of *X*. For example, (RESETSAVE (RADIX 8)) performs (RADIX 8), and provides for RADIX to be reset to its original value when the RESETLST completes by applying RADIX to the value returned by (RADIX 8).

In the special case that CAR of *X* is SETQ, the SETQ is transparent for the purposes of RESETSAVE, i.e. you could also have written (RESETSAVE (SETQ X (RADIX 8))), and restoration would be performed by applying RADIX, not SETQ, to the previous value of RADIX.

If *Y* is not NIL, it is evaluated (before *X*), and its *value* is used as the restoring expression. This is useful for functions which do not return their "previous setting". For example,

```
[RESETSAVE (SETBRK ...) (LIST 'SETBRK (GETBRK]
```

will restore the break characters by applying SETBRK to the value returned by (GETBRK), which was computed before the (SETBRK ...) expression was evaluated. The restoration expression is "evaluated" by *applying* its CAR to its CDR. This insures that the "arguments" in the CDR are not evaluated again.

If *X* is NIL, *Y* is still treated as a restoration expression. Therefore,

```
(RESETSAVE NIL (LIST 'CLOSEF FILE))
```

will cause FILE to be closed when the RESETLST that the RESETSAVE is under completes (or an error occurs or a Control-D is typed).

RESETSAVE can be called when *not* under a RESETLST. In this case, the restoration is performed at the next RESET, i.e., Control-D or call to RESET. In other words, there is an "implicit" RESETLST at the top-level executive.

RESETSAVE compiles open. Its value is not a "useful" quantity.

(**RESETVAR** *VAR NEWVALUE FORM*) [NLambda Function]

Simplified form of RESETLST and RESETSAVE for resetting and restoring global variables. Equivalent to (RESETLST (RESETSAVE *VAR NEWVALUE*) *FORM*). For example, (RESETVAR LISPXHISTORY EDITHISTORY (FOO)) resets LISPXHISTORY to the value of EDITHISTORY while evaluating (FOO). RESETVAR compiles open. If no error occurs, its value is the value of *FORM*.

(**RESETVARS** *VARSLST* E_1 E_2 . . . E_N) [NLambda NoSpread Function]

Similar to PROG, except that the variables in *VARSLST* are global variables. In a deep bound system (like Medley), each variable is "rebound" using RESETSAVE.

In a shallow bound system (like Interlisp-10) RESETVARS and PROG are identical, except that the compiler insures that variables bound in a RESETVARS are declared as SPECVARS (see Chapter 18).

RESETVARS, like GETATOMVAL and SETATOMVAL (see Chapter 2), is provided to permit compatibility (i.e. transportablility) between a shallow bound and deep bound system with respect to conceptually global variables.

Note: Like PROG, RESETVARS returns NIL unless a RETURN statement is executed.

(**RESETFORM** *RESETFORM* $FORM_1$ $FORM_2$ . . . $FORM_N$) [NLambda NoSpread Function]

Simplified form of RESETLST and RESETSAVE for resetting a system state when the corresponding function returns as its value the "previous setting." Equivalent to (RESETLST (RESETSAVE *RESETFORM*) $FORM_1$ $FORM_2$ ... $FORM_N$). For example, (RESETFORM (RADIX 8) (FOO)). RESETFORM compiles open. If no error occurs, it returns the value returned by $FORM_N$.

For some applications, the restoration operation must be different depending on whether the computation completed successfully or was aborted somehow (e.g., by an error or by typing Control-D). To facilitate this, while the restoration operation is being performed, the value of RESETSTATE is bound to NIL, ERROR, RESET, or HARDRESET depending on whether the exit was normal, due to an error, due to a reset (i.e., Control-D), or due to call to HARDRESET (see Chapter 23). As an example of the use of RESETSTATE,

```
(RESETLST
  (RESETSAVE (INFILE X)
              (LIST '[LAMBDA (FL)
                     (COND ((EQ RESETSTATE 'RESET)
                            (CLOSEF FL)
                             DELFILE FL]
                                   X))
                                   FORMS)
```

will cause X to be closed and deleted only if a Control-D was typed during the execution of *FORMS*.

When specifying complicated restoring expressions, it is often necessary to use the old value of the saving expression. For example, the following expression will set the primary input file (to FL) and execute some forms, but reset the primary input file only if an error or Control-D occurs.

```
(RESETLST
         (SETQ TEM (INPUT FL))
         (RESETSAVE NIL
            (LIST '(LAMBDA (X) (AND RESETSTATE (INPUT X)))
                  TEM))
         FORMS)
```

So that you will not have to explicitely save the old value, the variable OLDVALUE is bound at the time the restoring operation is performed to the value of the saving expression. Using this, the previous example could be recoded as:

```
(RESETLST
         (RESETSAVE (INPUT FL)
            '(AND RESETSTATE (INPUT OLDVALUE)))
         FORMS)
```

As mentioned earlier, restoring is performed by applying CAR of the restoring expression to the CDR, so RESETSTATE and (INPUT OLDVALUE) will not be evaluated by the APPLY. This particular example works because AND is an nlambda function that explicitly evaluates its arguments, so APPLYing AND to (RESETSTATE (INPUT OLDVALUE)) is the same as EVALing (AND RESETSTATE (INPUT OLDVALUE)). PROGN also has this property, so you can use a lambda function as a restoring form by enclosing it within a PROGN.

The function RESETUNDO (see Chapter 13) can be used in conjunction with RESETLST and RESETSAVE to provide a way of specifying that the system be restored to its prior state by *undoing* the side effects of the computations performed under the RESETLST.

Error List

There are currently fifty-plus types of errors in Medley. Some of these errors are implementation dependent, i.e., appear in Medley but may not appear in other Interlisp systems. The error number is set internally by the code that detects the error before it calls the error handling functions, and is used by ERRORMESS for printing error messages.

Most errors will print the offending expression as part of the error message. Error number 18 (Control-B) always causes a break (unless HELPFLAG is NIL). All other errors cause breaks if BREAKCHECK returns T (see Controlling When to Break above).

The folloing error messages are arranged numerically with the printed message next to the error number. *X* is the offending expression in each error message. The obsolete error numbers still generate error messags, but they aren't particularly useful. For information on how to use the Common Lisp error conditions in your own programs, see *Common Lisp: the Language* by Steele.

0 Obsolete.

1 Obsolete.

2 **Stack Overflow**

Occurs when computation is too deep, either with respect to number of function calls, or number of variable bindings. Usually because of a non-terminating recursive computation, i.e., a bug. Condition type: STACK-OVERFLOW.

3 **RETURN to nonexistant block: *X***

Call to RETURN when not inside of an interpreted PROG. Condition type: ILLEGAL-RETURN.

4 ***X* is not a LIST**

RPLACA called on a non-list. Condition type: XCL:SIMPLE-TYPE-ERROR *culprit* :EXPECTED-TYPE 'LIST

5 **Device error: *X***

An error with the local disk drive. Condition type: XCL:SIMPLE-DEVICE-ERROR *message*

6 **Serious condition XCL:ATTEMPT-TO-CHANGE-CONSTANT occured.**

Via SET or SETQ. Condition type: XCL:ATTEMPT-TO-CHANGE-CONSTANT

7 **Attempt to rplac NIL with *X***

Attempt either to RPLACA or to RPLACD NIL with something other than NIL. Condition type: XCL:ATTEMPT-TO-RPLAC-NIL *message*

8 **GO to a nonexistant tag: *X*.**

GO when not inside of a PROG, or GO to nonexistent label. Condition type: ILLEGAL-GO *tag*

9 **File won't open: *X***

From OPENSTREAM (see Chapter 24). Condition type: XCL:FILE-WONT-OPEN *pathname*

10 ***X* is not a NUMBER**

A numeric function e.g., PLUS, TIMES, GREATERP, expected a number and didn't get one. Condition type: XCL:SIMPLE-TYPE-ERROR *culprit* :EXPECTED TYPE 'CL:NUMBER

11 **Symbol name too long**

Attempted to create a symbol (via PACK, or typing one in, or reading from a file) with too many characters. In Medley, the maximum number of characters in a symbol is 255. Condition type: XCL:SYMBOL-NAME-TOO-LONG

12 **Symbol hash table full**

No room for any more (new) atoms. Condition type: XCL:SYMBOL-HT-FULL

13 **Stream not open: *X***

From an I/O function, e.g., READ, PRINT, CLOSEF. Condition type: XCL:STREAM-NOT-OPEN *stream*

14 ***X* is not a SYMBOL.**

SETQ, PUTPROP, GETTOPVAL, etc., given a non-atomic argument. Condition type: XCL:SMPLE-TYPE-ERROR *culprit* :EXPECTED-TYPE 'CL:SYMBOL

15 Obsolete

16 **End of file *X***

From an input function, e.g., READ, READC, RATOM. After the error occurs, the file will still be left open. Condition type: END-OF-FILE *stream*

17 ***X varying messages.***

Call to ERROR (see Signalling Errors above). Condition type: INTERLISP-ERROR MESSAGE

18 Obsolete

19 **Illegal stack arg: *X***

A stack function expected a stack position and was given something else. This might occur if the arguments to a stack function are reversed. Also occurs if you specified a stack position with a function name, and that function was not found on the stack (see Chapter 11). Condition type: ILLEGAL-STACK-ARG *arg*.

20 Obsolete

21 **Array space full**

System will first initiate a garbage collection of array space, and if no array space is reclaimed, will then generate this error. Condition type: XCL:ARRAY-SPACE-FULL.

22 **File system resources exceeded**: ***X***

Includes no more disk space, disk quota exceeded, directory full, etc. Condition type: XCL:FS-RESOURCE-EXCEEDED

23 **File not found**

File name does not correspond to a file in the corresponding directory. Can also occur if file name is ambiguous. Condition type: XCL:FILE-NOT-FOUND *pathname*

24 Obsolete

25 **Invalid argument: *X***

A form ends in a non-list other than NIL, e.g., (CONS T . 3). Condition type: INVALID-ARGUMENT-LIST *argument*

26 **Hash table full**: ***X***

See hash array functions, Chapter 6. Condition type: XCL:HASH-TABLE-FULL *table*

27 **Invalid argument: *X***

Catch-all error. Currently used by PUTD, EVALA, ARG, FUNARG, etc. Condition type: INVALID-ARGUMENT-LIST *argument*

28 **X is not a ARRAYP.**

ELT or SETA given an argument that is not a legal array (see Chapter 5). Condition type: XCL:SIMPLE-TYPE-ERROR *culprit* :EXPECTED-TYPE 'ARRAYP

29 Obsolete

30 **Stack ptr ahs been released NOBIND**

A released stack pointer was supplied as a stack descriptor for a purpose other than as a stack pointer to be re-used (see Chapter 11). Condition type: STACK-POINTER-REALEASED *name*

31 **Serious condition XCL:STORAGE-EXHAUSTED occured.**

Following a garbage collection, if not enough words have been collected, and there is no unallocated space left in the system, this error is generated. Condition type: XCL:STORAGE-EXHAUSTED

32 Obsolete

33 Obsolete

34 **No more data types available**

All available user data types have been allocated (see Chapter 8). Condition type: XCL:DATA-TYPES-EXHAUSTED

35 **Serious condition XCL:ATTEMPT-TO-CHANGE-CONSTANT occured.**

In a PROG or LAMBDA expression. Condition type: XCL:ATTEMPT-TO-CHANGE-CONSTANT

36 Obsolete

37 Obsolete

38 ***X* is not a READTABLEP.**

The argument was expected to be a valid read table (see Chapter 25). Condition type: XCL:SIMPLE-TYPE-ERROR *culprit* :EXPECTED-TYPE 'READTABLEP

39 ***X* is not a TERMTABLEP.**

The argument was expected to be a valid terminal table (see Chapter 30). Condition type: XCL:SIMPLE-TYPE-ERROR *culprit* :EXPECTED-TYPE 'TERMTABLEP

40 Obsolete

41 **Protection violation**: ***X***

Attempt to open a file that you do not have access to. Also reference to unassigned device. Condition type: XCL:FS-PROTECTION-VIOLATION

42 **Invalid pathname: *X***

Illegal character in file specification, illegal syntax, e.g. two ;'s etc. Condition type: XCL:INVALID-PATHNAME *pathname*

43 Obsolete

44 ***X* is an unbound variable**

This occurs when a variable (symbol) was used which had neither a stack binding (wasn't an argument to a function nor a PROG variable) nor a top level value. The "culprit" ((CADR ERRORMESS)) is the symbol. If DWIM corrects the error, no error occurs and the error number is not set. However, if an error is going to occur, whether or not it will cause a break, the error number will be set. Condition type: UNBOUND-VARIABLE *name*

45 **Serious condition UNDEFINED-CAR-OF-FORM occured.**

Undefined function error. This occurs when a form is evaluated whose function position (CAR) does not have a definition as a function. Condition type: UNDEFINE-CAR-OF FORM *function*

46 ***X varying messages*.**

This error is generated if APPLY is given an undefined function. Culprit is *(LIST FN ARGS)* Condition type: UNDEFINED-FUNCTION-IN-APPLY

47 **CONTROL E**

Control-E was typed. Condition type: XCL:CONTROL-E-INTERRUPT

48 **Floating point underflow.**

Underflow during floating-point operation. Condition type: XCL:FLOATING-UNDERFLOW

49 **Floating point overflow.**

Overflow during floating-point operation. Condition type: XCL:OVERFLOW

50 Obsolete

51 ***X* is not a HASH-TABLE**

Hash array operations given an argument that is not a hash array. Condition type: XCL:SIMPLE-TYPE-ERROR *culprit* :EXPECTED-TYPE 'CL:HASH-TABLE

52 **Too many arguments to *X***

Too many arguments given to a lambda-spread, lambda-nospread, or nlambda-spread function.

Medley does not cause an error if more arguments are passed to a function than it is defined with. This argument occurs when more individual arguments are passed to a function than Medley can store on the stack at once. The limit is currently 80 arguments.

In addition, many system functions, e.g., DEFINE, ARGLIST, ADVISE, LOG, EXPT, etc, also generate errors with appropriate messages by calling ERROR (see Signalling Errors above) which causes error number 17. Condition type: TOO-MANY-ARGUMENTS *callee* :MAXIMUM CL:CALL-ARGUMENTS-LIMIT

[This page intentionally left blank]