```
        LIST OF TECO COMMANDS, TECO VERSION 508
        Last updated 27 Nov 1976.  Z=161657

(Note that an uparrow followed immediately by another character
signifies a control character.  A command with the uparrow
modifier is represented with a space between the uparrow and the
command, except inside FS flag names, where that is not necessary
because control characters are not allowed anyway.
Also, altmode is always represented by a dollarsign.)

^@:
<n>^@   for nonnegative <n>, is the same as ".,.+<n>".
        For negative <n>, is the same as ".+<n>,.".
        "10^@XA" puts the 10 characters after the pointer
        in a string in qreg A.
<m>,<n>^@
        returns the value <n>-<m>.

^A      inclusive-or (an arithmetic operator).

^B      is illegal as a command.
        Note: ^B inside search string is a
        special char which is matched by any
        delimiter character.  The set of delimiter chars is
        specified by the contents of q-reg ..D;  initially, the
        delimiter characters are precisely the non-squoze
        characters (that is, all except letters, digits, ".",
        "%" and "$").
        ^B may be used in a file specification as the second
        file name to mean that the default second file name
        should be used.

^C      when typed in from console, terminates the
        command string, and starts execution.  If
        the command executes without error, TECO
        returns to its superior without flushing the type-in
        buffer.  When proceded, it will automatically
        redisplay the buffer on display consoles.
        When TECO returns, AC 2 will contain
        the address of the 7-word "buffer block" describing the current
        buffer - see the section "buffer block" at the end.
        To type in a ^C in a TECO command string, use ^]^Q^C,
        which is specially arranged to inhibit the normal
        action of ^C at command string read-in time.
        A ^C encountered as a command is an error.

^F      inserts its string argument, after deleting the last
        thing found with an S search or inserted with I or \
        (won't work if pointer has moved since the S, I or \
        was done).  Precisely, ^F is the same as FKDI.

^G      causes a "quit" by setting FS QUIT$ to nonzero.
        The consequences of that depend on the value
        FS NOQUIT$.  Normally, FS NOQUIT$ is 0;  ^G will then
        stop whatever TECO is doing and return to its top-level
        loop, or to the innermost ^R invocation if any, to
        read more commands (but first TECO will
        redisplay the buffer).  In particular, it will cancel a
        partially typed-in command string.
```

If FS NOQUIT$ is positive,
^G still sets FS QUIT$ but that has no effect. Thus, a program can inhibit quitting temporarily, or quit in its own manner by testing FS QUIT$ itself.
If FS NOQUIT$ is negative, setting FS QUIT$ nonzero causes an ordinary error (whose error code is "QIT"), which may be caught by an errset (:< - >).

^H backspace; it is illegal as a command.

^I tab; self-inserting character.

^J line feed; flushes current value.

^K valrets a string argument to DDT with dollar signs replaced by altmodes.
(To cause a dollar sign to be valretted, use "^]^Q$" (ctl-close ctl-Q dollar)).
If the command string contains an $P command, TECO command execution will continue with the character after the altmode ending the text string of the ^K.
^K causes TECO to believe that the screen has been clobbered, so it will automatically clear the screen and redisplay everything at the next opportunity. To avoid this, use "^ ^K" (if for example you know DDT will not type anything out, and will $P the TECO).
When TECO executes the .VALUE, AC 2 will contain the address of the 7-word "buffer block" describing the current buffer - see the section "buffer block" at the end.

^L form feed; clears screen on displays (when executed, not when typed). See F+ for more details.

^M carriage return; flushes current value.
In step mode (FS STEP$ nonzero), ^M has other actions: it displays the buffer unless there was typeout recently, then reads in a character and acts according to it.
Most characters simply tell ^M to return so that more commands will be executed. However, there are the following special characters:

- ^F quit. Like ^G, but ignores the setting of FS NOQUIT$ and does a real quit.
- ^P end stepping. Zeroes FS STEP$ and then proceeds without stepping.
- ^R enter ^R mode. On return from ^R, another character will be read and decoded.

^N:
<n>^N sets the FS LINES$ flag to <n>. Like "<n>FS LINES$".
FS LINES$ controls the number of lines used for buffer display and, on display terminals, for all other output.
:^N complements the FS TTMODE$ flag (initially 0).
TECO normally displays the buffer on printing terminals only if this flag is set. User buffer display macros should exhibit similar behavior.

:<n>^N like "<n>^N :^N"
Note: ^N in a search string is a special char which is
matched by any char other than the char after
the ^N in the search string.

^O<filename>$
bigprints <filename> on the device open for output.
Note: ^O in a search string is a special character
signifying "OR" i.e., it divides the search string into
two strings either of which will satisfy the search.
Thus, SFOO^OBAR$ will find either FOO or BAR, whichever is
encountered first.

^P alphabetic (ASCII) sort command.
The entire buffer, or the part within the virtual boundaries,
is sorted, after being divided into
sort records (i.e., things to be
sorted) on the basis of the arguments
given to the command in the form of
three TECO command strings following
the ^P, separated by altmodes
(Notes: (1) two successive null args
will result in a premature end of
command input, so use spaces where
needed; (2) a dollar sign in any
arg will be replaced by an altmode;
(3) the three args will be left in q-regs ..0, ..1, ..2).
The three command strings are used
to divide the buffer into sort
records, each of which has a sort key
(which may be any part of the record,
or outside the record). This is done
as follows:
 1. The pointer is moved to the
beginning of the buffer, which is the
beginning of the first sort record.
 2. The first command string is
executed. This should move the
pointer from the beginning of any
record to the beginning of its key.
 3. The second command string is
executed. This should move the
pointer from the beginning of any key
to the end of that key.
 4. The last command string is
executed. This should move the
pointer from the end of any sort
key to the end of the record, i.e.,
the beginning of the next record.
 5. If step 3 or 4 leaves the pointer
at the end of the buffer, or
executes a search which fails (this
will not cause an error; those
steps are done as if inside an
iteration), the creation of sort
records is complete, and the sort
takes place. Otherwise, go back
to step 2.
Sort records and keys may be
variable length. No char (i.e., a

shorter key) sorts before ^@, and keys are
considered left-justified for the comparison.
There is nothing to prevent overlapping records
from being specified; the sort will copy each record
so the overlap region will be duplicated.
Insertion and deletion are allowed but know that
TECO remembers the boundaries of records and keys as character
numbers, so deleting chars from a record already delimited
will shift chars from the next record into it, etc.
The sort is stable. :^P sorts in reverse order.
If FS ^P CASE$ is nonzero, ^P ignores case; that is,
it sorts lowercase letters as if they were the
corresponding uppercase letters.

uparrow-P

^Q ^Q in a search string causes
the next char to be quoted, i.e.,
it is treated as an ordinary char
even if it normally has a special
meaning ("^Q^Q" is a normal ^Q;
^Q works only at execution time, not at command string
read-in time, so rubout cannot be ^Q'd).
This also works inside file name specifications.

^R real time edit feature, intended mainly for display terminals. The position of the pointer is represented by the terminal's hardware cursor, rather than by any printed characters (^R ignores the contents of ..A, except on printing terminals).

All non-control-non-rubout characters are normally self inserting; the others are normally editing commands. The user may redefine any character by means of the FS ^RCMAC$ flag.
In ^R mode echoing is turned off, so typed-in characters manifest themselves only by their effect on the displayed buffer contents (but see FS ^R ECHO$).

Any command may be given a numeric argument, which most commands (including all characters that insert themselves) treat as a repetition count. If no argument is specified, 1 is the default, but commands can distinguish between an explicit 1 and a defaulted 1. The argument is computed as follows: <arg> = <basic arg> * (4 ** <exponent-of-4>) where <basic arg> is the explicit argument, if any, or 1 otherwise. An explicit argument is given with ^V or by control-digits. <expt-of-4> is initially 0 and incremented by ^U. All commands except argument-setting commands discard their arguments even if they don't use the arguments. Three flags contain the argument data:
FS ^R ARG$ contains the explicit argument, if any, else 0;
FS ^R EXPT$ contains the exponent of 4;
FS ^R ARGP$, if zero, indicates that no arg has been specified (neither the explicit arg nor the exponent of 4);
if 1, indicates that only an exponent of 4 has been specified, and the basic arg is still 1;
if 3, indicates that an explicit arg has been specified.
All three are zeroed after any command that doesn't identify itself as an argument setting command by clearing FS ^R LAST$.

Any character may have a program associated with it, using the FS ^RCMACRO$ command. If that is done, when that character is typed, TECO will execute the program instead of inserting the char or using it as a built-in command. The definition of a character may also be treated as a q-register in the "Q", "U", "X", "G", "[", "]", "M" and "FQ" commands; see "Q" for directions. When the program is executed, q-reg ..0 will contain the character being handled.

When errors take place inside ^R, or in macros called from ^R, after printing the error message TECO returns control to the innermost invocation of ^R (unless FS *RSET$ or ..P is nonzero). The same thing happens for quits.

One may wish to have a mode in which most editing commands are disabled, and most characters that are normally editing commands are self-inserting instead. The FS ^RSUPPRESS$ flag, when nonzero, suppresses all built-in commands except rubout and all user defined commands whose definitions do not begin with "W" (since "W" at the beginning of a macro

is a no-op, the only reason to have one there is to
prevent suppression). When a character is suppressed
as a command, it becomes self-inserting. An additional
feature is the FS CTL MTA$ flag; when it is negative,
all control-meta-letters (and ctl-meta-[,], \, ^ and _)
have their definitions suppressed; this mode is useful
when editing TECO commands.

In "replace mode", printing characters overlay a
character instead of making the line longer.
Replace mode is controlled by FS ^R REPLACE$, which
see for more details.

The ^R-mode input dispatch table is actually indexed by
9-bit TV character code. Each 9-bit code can be redefined.
The list of ^R-mode initial definitions that follows
refers to the characters obtainable on non-TV's - in other
words, the 9-bit characters which are the results of
reading in the 14-bit codes 0000 through 0177, which are
precisely the 9-bit characters which are equivalent to
some 7-bit ASCII character.
A subsystem which is not TV oriented need not worry about
the 9-bit character set; by using FI, and FS ^RCMACRO$
always without the uparrow modifier, it can handle ASCII
characters throughout. TECO will automatically do the
conversion to and from 9-bit characters on TV's.
For those who wish to handle the 9-bit character set,
the definitions of all 9-bit characters
are listed in the section "TECO's character sets",
along with the appropriate conversions between character
sets.

One may wish to have some operation (such as filing
the buffer away) performed every so often while in ^R
mode. See "..F" for how to do this using the
"secretary macro" feature. FS ^R DISPLAY$ can be set
to a macro which will be run every time ^R is about
to do nontrivial redisplay.

Although ^R mode is intended for display terminals,
the creation of large macro-systems intended for use
with ^R mode has made it necessary for ^R to work
at least marginally on printing terminals.
Since the physical cursor is not suitable, the ordinary
TECO cursor is used (whatever is in ..A). The buffer
is displayed only when the screen is "cleared", such as by
giving the built-in ^L command.
Also, unless FS ^RECHO$ > 0, characters actually read
by the ^R-mode command loop are typed out, although
echoing is still turned off.
This echoing can be made to happen even on displays
by making FS ^RECHO$ negative (this is unwise to do
if there is no echo area).

Setting FS ^R SCAN$ to nonzero causes ^R commands to
try to imitate printing terminal line editors by
echoing the characters that they insert/delete/move over.
In this case, FS ^R ECHO$ should be set to 1.

Macros and ^R - reducing redisplay:

Whenever control passes from normal TECO to ^R (that is, when a ^R is executed, when a ^ V is executed within a ^R-mode macro, or when a ^R-mode macro returns), ^R must be able to update the screen according to the changes that have been made in the buffer since the last time ^R mode lost control. ^R can do that in a way that makes no assumptions, but that way is slow. If information is still available on what areas of the buffer were changed, that info can be passed to ^R in the form of numeric args, and ^R will save time by assuming the info to be correct. If the info is not correct, the screen will not be properly updated. The options are: no args - the usual case - means assume nothing. One arg means that the buffer has not changed, although the pointer may have moved. The actual value of the arg does not matter in this case. Two args should specify a range of the buffer outside of which nothing was changed. ^R will limit redisplay to that range if possible. ^R also knows what to do about macros that type text out; if Q..H is nonzero when ^R is entered or returned to, ^R will not do any displaying until it has read one character (and executed it, unless it is a space).

If you like ^R mode, try:

```
:I..G EL 90^S ^R$ :I..B Q..H"N 90^S ' ^R$
```

The commands are:

Control-digits
: accumulate a numeric argument for the next command. Thus, control-5 ^N will move down five lines.

^A
: go to beginning of current line (0L). With argument, <arg>-1 L.

^B
: go back over previous character (R)

^C
: complements the state of the comment mode switch. Types "C" for comment or "T" for text at the bottom of the screen, to say what mode you're in. When in comment mode, the ^N and ^P commands begin by going to the end of the line and if the last character is a semicolon, deleting it and any preceding tabs. Then, after moving to the next or previous line, if the line has a semicolon in it the pointer will be left after the semicolon; otherwise the pointer will move to the end of line, and enough tabs will be inserted to move the pointer at least to the specified comment column, followed by a semicolon. Numeric argument is ignored.

^D
: deletes the next character after . (D) If FS RUBCRLF$ is nonzero, ^D before CRLF deletes both the CR and the LF.

^E	moves to end of line (:L). With argument, <arg>:L.
^F	goes forward over the next character (C)
^G	flushes any numeric argument or case-shift, unsets the mark if it had been set, and resets the case-lock. When ^R is actually in control (as opposed to a macro running inside ^R), ^G's quitting action is suppressed, and ^G acts as a command instead. Thus, it does not flush any type-in.
^H	(backspace) inserts itself.
^J	(linefeed) inserts itself.
^K	kills to eol (K). With arg, <arg>K. The text deleted is put in q-reg ..K.
^L	redisplays the screen (used to recover from datapoint lossage). Chooses a new window. A numeric argument specifies the number of lines of buffer to display - useful on printing terminals. On displays, if only a part of the screen is being used at the moment, only that part is cleared.
^M	inserts a carrage return-line feed.
^N	goes to next line (L). With argument, <arg>L.
^O	inserts a CRLF, then backs over it. "^Ofoo" is equivalent to "foo^M" but often requires less redisplay. With argument, inserts <arg> CRLFs and backs over the last. If you want to insert several lines in the middle of a page, try doing ^U^U^O before and ^U^U^K afterward.
^P	goes to previous line (-L). More generally, -<arg>L.
^Q	inserts the folling character directly, regardless of its meaning as a command. If the char isn't already in the input buffer, ^Q will prompt with a "Q" at the bottom of the screen. An argument to ^Q causes it to insert the same character <arg> times. ^Q is not affected by replace mode; the quoted character is always inserted.
^R	causes the column the pointer is at to become the comment column. Argument is ignored.
^S	reads a character and searches for it. "^SA" in ^R mode is the like "SA$" in TECO.
^T	sets the ^R-mode mark at the current pointer position. The mark is really the value of FS ^RMARK$ and is used by the ^X and ^W commands in ^R mode. If FS ^R MARK$ holds -1 there is no

mark; that is the case initially and after any insertion, deletion or quit in ^R mode. Attempting to use the mark when there is none rings the bell.

^U increments the exponent-of-4 for the next command. This usually is the same as repeating it 4 times. Does not use any previous argument, but leaves it around for the next command.

^V sets the basic arg for the next command. The argument is composed of digits optionally preceded by a minus sign, echoed at the bottom of the screen and turned into a number in the current radix (FS IBASE$). The first non-digit terminates the arg and is treated as a command. ^G will flush the argument.

^W kills everything between the current pointer position and the mark, putting the deleted text in q-reg ..K. If there is no mark, nothing is deleted and the bell is rung.

^X sets the mark at the current pointer position, and moves the pointer to where the mark had been; in other words, exchanges the mark and the pointer. Does nothing if there is no mark. Do this several times to see both ends of the range that a ^W command would delete.

^[(altmode) terminates edit

^] reads a q-reg name and executes that q-reg as a macro. The q-reg should contain ordinary TECO commands, not ^R mode commands. The numeric arg to the ^] will be given to the macro which will see it as the value of ^Y (If no argument is specified, ^Y will be 1, but F^X will indicate that the macro had no argument). The macro may return values to ^R telling it which areas of the buffer may need redisplay (see below).
If the macro is to return values, it should end with a space - otherwise, the values might get lost within TECO.
Example: " .,(G..K .) "
gets q-reg ..K and returns 2 values limiting the range of the buffer in which changes took place.

^? (rubout) deletes bacwards (-D). If FS RUBCRLF$ is nonzero, rubout when the pointer is after a CRLF deletes the whole CRLF.

ctl-rubout
deletes backwards like rubout, except that tabs are converted to spaces and the spaces are deleted one at a time.

^S:
<n>^S if <n> is positive, sleep for <n> 30ths of a

second. If <n> is negative, sleep until system run time (what FS UPTIME$ gets) = -<n>.

<n>:^S sleep for at most <n> 30ths of a second, returning immediately if there is any input available. Returns the value of FS LISTEN$ (nonzero if input is available).

^T enters the old printing-terminal real-time edit mode. If ^T is typed as the first character in the input string, real-time edit starts immediately, otherwise it starts when executed. It terminates the command string, like the ^_ command, so it can't really be used in programs. In real-time edit mode, you edit as you type. Any character whose ASCII code is more than 37 octal is entered directly into the line being edited. Control characters are interpreted as commands. There are two enter modes: insert mode and overlay mode. In insert mode, inserted characters are inserted at the pointer. In overlay mode inserted characters are overlayed over the character to the right of the pointer. In either case, the pointer moves after the inserted character. Initial mode is overlay mode. The ^P command changes the mode.

commands are as follows:

^C	move pointer 1 character echoing as character moved over
^D	delete character following pointer echoing as %
^F	help. 1) type '#' 2) type from pointer to first linefeed 3) type from previous linefeed to pointer
^G	quit.
^I	tab. Insert in current mode.
^J	linefeed. Insert in current mode
^L	terminate edit.
^M	carriage return. Delete to next linefeed and terminate edit.
^N	move pointer before 1st space following a nonspace.
^O	delete to after next space, typing % for each deleted character.

```
      ^P     see above.  Changes insert mode.

^U       display in the user-specified manner the directory of
         the current default device.  That is, invoke the user's
         buffer display macro if any;  otherwise on display consoles
         display in the standard manner, but do
         nothing on printing terminals.  These are the same actions
         TECO always takes at the end of any command string whose
         last command was an E-command.
         Note: if ^U is typed as the first character of a command
         string, it is executed immediately when read.
0^U      sets default device to DSK:, then does ^U.
<n>^U    sets default device to dectape # <n>, then does ^U.

^V       pops the "ring buffer of the pointer".  ^V when the
         first character of a command string acts immediately,
         resetting the pointer to the value it had before the
         last time it was moved.  Successive ^V's will undo
         earlier changes of the pointer.  Up to 8 changes are
         remembered to be undone.  Motion caused by the use of
         ^V in this manner does not get saved to be undone.
         ^V not the first character typed is slightly
         different.  It pops the ring buffer into the pointer,
         and returns as its
         value the number that then remains on the top.  If that
         returned value is put in Q..I (which is what gets
         pushed on the ring buffer at the end of the command
         string) you can fool TECO's top level into thinking
         that the pointer was not moved by the command string
         that just finished, so nothing will get pushed back on
         the ring buffer (this is exactly what ^V as the first
         character typed does).  If TECO's top level is not in
         use, the program that is running must be hacked up to
         push explicitly on the ring buffer (using <n>^V)
         in order for anything to appear on it.
         If ^V attempts to jump out of the buffer, the pointer
         is not moved, but the ring buffer is popped.  A "NIB"
         error happens.
:^V      returns the value on the top of the ring buffer,
         without popping it or changing the pointer.
<n>^V    is equivalent to <n>FS PUSHPT$.  It pushes <n> onto
         the ring buffer unless <n> equals whatever is at the
         top of the ring buffer.
<n>:^V   pushes <n> onto the ring buffer unconditionally.

^W       pops all the way to top level,
         exiting from any break-loops and not running the user
         defined error handler in ..P.

^X       only defined inside macro.  Its value is the
         first arg of the M command which called the macro.
         See the F^X command for a more sophisticated
         way for macros to examine their arguments.
         Note: ^X typed as the first character of a command
         tells TECO to type out the whole error message
         associated with the most recent error.  If the flag
         FS VERBOSE$ is zero (normally true on printing terminals)
         TECO normally types only the 3-letter code.  Use ^X
         to see the whole message if you don't recognize the code.
```

	Note: ^X in search string is a special char which is matched by any character.
^Y	like ^X, only second or only arg of the M command. If ^Y is the first char typed in in a command string, the most recently typed command string longer than 7 characters (not counting the 2 altmodes) is inserted in the buffer. This is a loss recovery procedure.
^Z	normally causes an interrupt to DDT when typed. However, one can be given to TECO by quoting it with ^_ , in which case it is a normal command: with no arg, its value is a pseudo-random number.
<n>^Z	inserts <n> random letters before the pointer.
Altmode	terminates following text argument to certain commands; two successive altmodes terminates command string and begins command execution. Execution of an altmode as a command depends on the setting of FS NOOP ALTMODES$. If the flag is >0 (old-fashioned mode), altmode acts like the ^_ command. If the flag is negative (default mode), altmode is a no-op. If the flag is zero (losing mode), altmode is an error as a command.
^\	exits from the innermost macro invocation, unwinding the q-register pdl to the level it had when the macro was entered, and popping all iterations that started inside that macro. Note that if Q..N is popped this way, it's previous contents (before the pop) will be macroed (after the pop is done). This enables macros to arrange arbitrary actions to be performed whenever the macro is exited, no matter for what reason.
:^\	exits from the innermost macro invocation, without unwinding the q-register pdl. It does pop iterations.

^] string substitution

^] is not really a command. It is a special character that makes it possible to substitute the contents of a q-reg into a TECO command at any point (such as, inside an I or S command). ^] is processed when TECO reads a character from the command buffer (ie. Before anything like insertion or execution is done to the character.). It gobbles the next character and decodes it as follows:

^A	sets the one-character flag (see below) then reads another character and interprets it as if it had been typed after a ^].
^Q	gobbles another character and returns it to TECO superquoted (i.e. It will not act as a text terminator, in a search string, it will have no special effect, etc.)
^R	is the beginning of the name of a q-reg to be substituted.
^S	cause the superquote flag to be turned on (see below) then read another character as in ^A
^T	cause the delimiter flag to be turned off (see below) then read another character as in ^A
^V	followed by a q-register name, causes the char whose ASCII value is in that q-register as a number to be substituted in. That is, after ^^AU0, ^]^V0 will substitute an "A".
^X	reads a string argument to the M command that called the current macro, and substitutes it in. ^]^X pushes the current command buffer onto a special pdl, then causes the normal macro pdl to be popped one level (the macro pdl is pushed onto each time an M command is executed. It is also pushed onto by ^]<q-reg name> (see below)). TECO will then proceed normally, reading from what is essentially a string argument to the current macro, until an altmode is encountered. This altmode will not be passed to TECO, but will instead cause the command buffer to be repushed on the macro pdl and the special pdl to be popped, thus restoring the state of the world. If a real altmode is desired in a string argument, ^]$ (dollar sign) should be used. If TECO had been in any state other than reading commands (i.e. Reading a string to be inserted) then the characters read in the string argument will be protected from being taken as text delimiters. Thus I^]^X$ is guaranteed not to terminate somewhere in the macro argument. If this is for some reason undesirable, a ^T (see above) should be used between the ^] and the ^X (^]^T^X). Characters are

not normally protected from being interpreted specially in searches, etc. If this is desired, use ^S (eg. ER^]^S^X bar$ will cause the file <macro argument> bar to be selected for read, even if the macro argument has spaces, semicolons, etc. in it.).
If the one character flag had been on only one character will be read as an argument instead of an entire string.

^Y	acts like ^]^X, but only one character is taken from the previous command level. Has the same effect as ^]^A^X. Additional ^] calls will be chained through, with the final character comming from the last command level not indirected.
$ (altmode)	pass a superquoted altmode back to TECO (same as ^]^Q$)
^]	pass an actual ^] to TECO
$ (dollarsign)	pass an ordinary altmode back to TECO (see ^X above)
.	is the beginning of a q-reg name. Multi-character q-regs such as Q..A can be substituted with ^] just like single-character q-regs.
0-9	the current command buffer is pushed onto the macro pdl, and the q-register named by the character read becomes the new command buffer (eg. I^]1$ is the same as G1, but G is optimized for that operation.). Protection (superquoting) is the same as in ^X (qv).
@	@ ("indirect") causes the characters substituted in by the ^] to be treated as if they in turn had a ^] in front of them. Thus, after :IA.B$, ^]@A will substitute q-reg .B. After :IA.Bfoo$, ^]@A will substitute the contents of .B, followed by "foo". I may change that if I can see an easy way.
A-Z	like 0-9 (insert q-reg)

^^<char>
(ctl uparrow) has the value of the 7-bit ASCII code for <char>.

^_ (note that in order to type this character to a program, it must be typed twice, due to ITS hackery) ends execution of the command string "successfully"; the TECO will log out if disowned, or return to its superior if a ^C ended the typed-in command string. Otherwise, or after TECO is $P'd, TECO will reset all stacks (if FS *RSET$ is 0), then maybe display the buffer or dircetory (using the user's supplied macros in Q..B and Q..G if any), and go on to read another command string.
It is not wise to use this as a nonlocal exit from a macro; that is what F< is for. The main use is to restart TECO's command reading loop at the current stack levels - useful when a user-defined error handler wants to transfer to a TECO break loop. TECO's command loop puts a ^_ at the end of every command string to make sure that it gets control back when the command string terminates. Otherwise, in a break loop, control would return right back to the suspended program.

Space same as "+", except that space by itself does not constitute a nonnull argument, while "+" does.

!<label>! defines <label> for use by O command (q.v.).
This contruct is also the standard way of putting comments in TECO macros. It is completely transparent if it is between commands.

" starts a conditional. The character after the " gives the condition. It is followed by conditionalized commands, up to a matching '. If an else-clause is desired, the ' should be followed immediately by "#, with perhaps CRLFs, spaces, or comments (see "!") in between, followed by the contents of the else-clause, followed by another '. A conditional may return a value.
The argument to the conditional is normally gobbled up by the conditional, and the first conditionalized command receives no argument; see F" for a variant conditional that passes the argument along instead.
The conditions that now exist are:

Char:	Condition succeeds if numeric arg to " is
B	the ASCII code for a delimiter character
C	the ASCII code for a non-delimiter character.
E	zero.
G	positive.
L	negative.
N	nonzero.

The delimiter characters are those characters which are specified as delimiters by the contents of q-reg ..D. Initially, q-reg ..D is set up to specify that all non-squoze characters are delimiters, but the user can change that by setting q-reg ..D.

Squoze characters are letters, digits, ".", "%" and "$".

Conditionals operate by skipping the text up to the matching ' if they fail, and doing nothing if they succeed. If the ' terminating a failing conditional is followed by "#, they will be skipped as well. If the conditional succeeded, they would be executed - and "# is really a conditional that always fails.

For example, an expression whose value is the signum of the number in q-reg 0 is: `Q0"G 1'"# Q0"L -1'"# 0''` .

#	exclusive or (an arithmetic operator).
$	(dollar sign) the old lower-case edit mode: "-1$" is the same as "-1F$/$" (first dollar, then altmode) "0$" is the same as "0F$$" (first dollar, then altmode). "1$" is the same as "1F$$". For more info, see the "F$" command (that's dollarsign, not altmode).
%<q>	increments the number in q-reg <q> by 1, and returns the result as a numeric value. Meaningless if the q-reg contains text.
&	logical and (an arithmetic operator)
'	terminates a conditional (see "). This character is actually a no-op when executed. It is for the " to search for if the condition fails.
(,)	fill usual role of parentheses in arithmetic calculations. (turns off the colon and uparrow flags;) turns them on iff they were on before the (, but will never turn them off. See also F(and F) for variants of these commands.
*	multiplication (an arithmetic operator). Note that in TECO there is no operator precedence. Evaluation of arithmetic operators is left-to-right.
+	addition (an arithmetic operator).
,	separates arguments for commands taking two numeric arguments. Doesn't affect the colon and uparrow flags.
-	subtraction (an arithmetic operator).
.	equals the number of chars to left of the pointer.

..n Q-registers

..0,..1,..2
"^P" sort puts its 3 arguments into these q-regs.
These q-regs are also used by "F^A".

..A holds the string to be used to represent the cursor in standard buffer display. Initially "/\" on displays, "^A^B" on Imlacs (looks like an I-beam), and "-!-" on printing terminals (of course, TECO's default is not to display the buffer on printing terminals unless FS TTMODE$ is set).
In the cursor, backspaces always really backspace and all other control characters are treated as non-spacing characters.

..B holds the user buffer display macro.
After each command string whose last command was not an E-command, TECO does "normal buffer display", as follows:
if ..B is 0, as it initially is, the default is:
on graphics devices, do "standard buffer display";
on printing terminals, do so only if FS TTMODE$ is set;
otherwise do nothing. For details of standard buffer display, see "^ V".
If q-reg ..B is nonzero, TECO simply macroes it. Normal buffer display in this case consists of whatever that macro happens to do.
Q-reg ..H and flags FS ERRFLG$ and FS ERROR$ will contain information about the command string that just ended.
If either Q..H or FS ERRFLG$ is nonzero, there is text on the screen that should not be immediately covered over.
The buffer display macro should check ..H and not display if it is nonzero. FS ERRFLG$ need not be checked, since if -1, it will automatically cause all typeout on the first line of the screen to be ignored on displays. This is the right thing if the buffer display macro doesn't wish to worry about errors. If it is desirable to write on the first line and overwrite the error message, just zero FS ERRFLG$.

..D holds the delimiter dispatch table, which tells several commands how to treat each of the 128 ASCII characters.
These commands are FW, FL, "B, "C and the special search character ^B. The treatment of the character with ASCII code <n> is determined by the values of the characters in positions 5*<n>+1 and 5*<n>+2 in the delimiter dispatch table.
The first of the dispatch characters says whether the character <n> is a delimiter. The dispatch character should be " " for a delimiter and "A" otherwise.
This dispatch character is used by FW, "B, "C and ^B.
The second dispatch character describes the character's syntax in LISP. The possibilities are "(", ")", "/", "'" "|", " " and "A". Each says that the character <n> should be treated by FL and ^ FW as if it were an open, a close, a slash, etc.
Initially, the first dispatch character is "A" for squoze characters (letters, digits, "$", "%" and "."), and " " for all others. The second dispatch character is set up to reflect the default LISP syntax definitions as closely

as possible.
The delimiter dispatch must be at
least 640 characters long so that every character has
a dispatch entry. ..D should always contain a buffer
or a string; if it holds a number an error will result.

..E holds the output radix for = and \. Initially decimal.
Negative radices work - somewhat. If the radix is 0 or 1,
the next attempt to use it will change it to decimal
and also cause an error "..E".

..F holds the ^R secretary macro. If nonzero,
it will be macroed every (FS ^RMDLY$) characters
while ^R mode is in use. More precisely, the counter
FS ^RMCNT$ is decremented each time through ^R's main
loop, and if it becomes 0, it is reset from FS ^RMDLY$
and ..F is macroed. ..F is also macroed whenever the
outermost level of ^R mode is exited (but not when
inner recursive invocations of ^R are exited).
When ..F is macroed because ^R is being exited, the
FS ^RMODE$ flag will be 0; otherwise it will be nonzero.
Note that the ^ V command in ^R mode counts as one
pass through the ^R main loop and thus may run the
secretary macro.

..G holds the user-specified directory-display macro.
Whenever TECO wants to display the directory in the
usual manner (that is, when ^U or E^U is executed or
at the end of a command string whose last command was
an E-command), if this q-reg isn't zero TECO will simply
macro it (otherwise, TECO has defaults - see "^U").
When that is done, q-reg ..H will contain useful info.

..H is the "suppress display" flag. It is set to zero at the
start of each command string, whenever the screen is
cleared. It is set nonzero when any typeout or display
takes place, except for error message typeout.
TECO's default is not to display the buffer if this q-reg
is nonzero at the end of the command string. User buffer
and dir display macros should also look at this flag.
If ..H is nonzero on entering or returning to ^R, ^R will
wait until a character is typed in (and executed, unless
it is a space) before allowing any redisplay.

..I at the start of each command string,
.'s value is saved in this q-reg.
At the end of each command string, Q..IFS PUSHPT$ is done.
Those actions are what enable the ^V command to work.

..J initially 0, if this q-reg contains a string that string
will appear on the screen just above the echo area, on
the same line that --MORE-- sometines appears on. The
--MORE-- will still appear, following the ..J string,
if it is appropriate. The displayed string is not
updated immediately when ..J is changed, but rather at
the next opportunity for redisplay of the buffer or
the next time typeout reaches the bottom of the screen.
It is possible to put a buffer in ..J but that has the
problem that TECO will not always be able to detect it
when the buffer's contents change, and thus will not be

able to update the screen when it should.

..K each ^K or ^W command in ^R mode puts the deleted text
in this q-reg so it can be reinserted if desired.

..L whenever TECO is $G'd, this q-reg is executed.
Also, after an EJ, the macro loaded into ..L is run.
If you don't like the way TECO initializes certain FS
flags (namely FS ECHOLINES$, FS TRUNCATE$, FS VERBOSE$,
FS WIDTH$, FS ^HPRINT$, FS ^MPRINT$, and FS SAIL$) each
time it is $G'd, put something in ..L to change them.
When a TECO dump file is made with ^ EJ, ..L should
contain a macro to do whatever must be done when the
file is loaded back in. However, since that macro would
be re-executed if TECO were $G'd afterward, it should
replace itself with something innocuous that just
resets the terminal-related flags.

..N this q-reg is special in that whenever it is popped by
automatic unwinding of the q-register pdl, the previous
contents are macroed after the pop.
Thus, it is possible for a macro to set up an
action that will be performed when the macro is exited,
no matter what causes it to be exited, by pushing Q..N
and putting the commands for that action in Q..N. For
example, [0 .U0 [..N :I..N Q0J $ saves . in such a
way that it will always be restored. That string,
unfortunately, has a timing error in that a ^G-quit
after the [..N will find an inconsistent state. The
remedy is to use the FN command which is the same as
"[..N:I..N": [0 .U0 FN Q0J$
Within a macro that has already set Q..N up in this
way, the easiest way to add another action to be
performed is to append to ..N using
:I..N^]^S..N<new-stuff>$.
Note that popping Q..N explicitly with]..N does not
macro it.
If you wish explicitly to pop ..N and macro the old
value, the way to do it is "-FS QPUN$". "M..N]..N"
has the disadvantage that when ..N is executed it is
still on the q-reg pdl; that may make it execute
improperly and also is a timing error.

..O this q-reg is defined to hold the current buffer. That is,
all the commands that use "the buffer" use whatever
buffer happens to be in Q..O at the time. An attempt
to put a string or number in Q..O causes an error.

..P holds the user-defined error-handler macro, if any.
Whenever an error occurs that is not caught by an errset,
this macro will be invoked. If it is 0, TECO will
instead print out the error message and set up for "?"
in the normal manner.
The executing command string will have been pushed on the
macro pdl, so FS BACKTRACE$ can be used to examine it.
Also, the arguments will have been saved with "(" so that
they can be examined with ")F(=".
FS ERROR$ will contain the error code for use with
FE or FG in obtaining the error message.
Note that the error handler is invoked for quits and when

TECO is restarted, as well as after errors; at those other times FS ERROR$ will be zero.
If the error handler prints an error message in the main program area of the screen, it may wish to allow buffer display to occur as usual but prevent the error message from being overwritten by it. Setting Q..H to zero permits buffer display, and setting FS ERRFLG$ to minus the number of lines of error message preserves them.
The FG command takes care of this automatically.

The error handler can return to the erring program with ")^\" or "F)^\" (return whatever you like, but make sure to close the parentheses somehow). However, do not expect the command that signalled the error to be retried.
It may also use F; to return to a catch that was made at a higher level, or use ^W to pop out to TECO's top level loop, flushing the pushed program and all its callers. Otherwise, it can pop to the appropriate place, or, if FS *RSET$ is nonzero, make a break loop.
To make a break loop, just do a ^_ which will transfer to TECO's command string reader. To throw to the appropriate place, you can do FS ^R THROW$ to go to the innermost ^R, or you can do ^_ or ^W to go to TECO's command reader.
FS ^R MODE$ might help you decide which one to do.
If an error happens during the invocation of the error handler, in order to prevent an infinite error loop TECO does an automatic ^W command to pop all stacks.
This condition is a likely result of an error in the error handler itself, since the recursive invocation of the error handler will eventually lead to stack overflow.

..Q holds a q-vector which serves as the symbol table for TECO variables, such as QFoo accesses. The symbol table is in the format that FO likes. Initially, the q-vector in ..Q has only one element, which contains 2, the number of words per symbol table entry. See FO and Q for more details.

..Z initially holds the same thing as ..O (the initial buffer), on the assumption that your main editing will be done in it, so that if you accidentally leave something else in ..O you can do Q..ZU..O to recover and not lose all your work.

```
/        division (an arithmetic operator).

0-9      a string of digits is a command whose value is a number.
         If it is not followed by a ".", the normal input radix
         (the value of FS IBASE$, initially 8+2) is used.
         If the number ends with ".", the radix used is
         the value of FS I.BASE$, initially 8.
         An attempt to type in a number too large for a 36-bit
         word to hold will cause a "#OV" error, unless the radix
         is a power of 2.

:        used before certain commands,
         modifies function of that command
         in a way described separately for each such command.
         Arithmetic operators and comma do not affect the
         colon flag.  Parens save it just like arguments, and
         don't deliver it to the commands inside the parens.
         Commands that don't return values always turn it off;
         commands that do, either ignore it or use it and turn it off.

;        does nothing if arg<0.  Otherwise
         sends command execution to char
         after next > (see < description).
         If no arg, uses value returned by
         last search (see S) as arg.

:;       like ;, but with the opposite condition:
         end iteration if arg is <0, or last search succeeded.

<        begin iteration.  Commands from here
         to matching > are executed arg times
         if there is an arg or indefinitely if
         no arg.  Execution of iteration can
         be terminated by ; command (q.v.).
         It is an error if the iteration remains unterminated
         at the end of the macro level it began on.
         Within iterations, failing searches do not cause
         errors, unless FS S ERROR$ has been changed to
         disable this "feature".

:<       begin errset.  This is like < except that errors
         occurring inside it are caught and will return after the >.
         The value returned after the > will be 0 iff there was
         no error;  otherwise it will be a negative number which
         is the error code, and may fed as argument to the
         FE command to find out what sort of error it was.
         Note that FE can also be used to find the error code
         corresponding to a three-letter error name.
         Note also that :< iterates like <.  Perhaps you want 1:< ?
         Errsets to not prevent failing searches from erroring
         (luckily), and in fact undo the effects of any iterations
         farther out in the stack.

=        is for printing numbers:
<n>=     types out <n> in the current output radix, and a CRLF.
         The output radix is kept in q-reg ..E .  It is initially 8+2.
<m>,<n>=  types both <m> and <n>, with a comma between.
:=       is like = but omits the CRLF.
^=       is like = but types in the echo area.  ^:= also works.
```

```
>       end of iteration, errset or catch (see "<", ":<", "F<").

?       if this is the first char input after
        typeout of an error message from TECO
        several command chars before the one
        causing the error will be typed.
        Otherwise, enter trace mode, or, if in trace mode
        already, leave trace mode.
        When in trace mode all command
        chars are typed out as they are executed.
        Trace typeout never uses the first line so that
        error messages won't wipe it out.
        The flag FS TRACE$ is nonzero when in trace mode.
:?      leaves trace mode whether in it or not.

A       if no arg, append next page of
        input file to current contents of
        buffer, i.e., like "Y" only don't empty buffer first.
        If virtual buffer boundaries are in use, the appended
        text goes just below the upper virtual boundary.
        Does not close the input file.
<n>:A   appends <n> lines of the file (but won't append beyond
        a page boundary).  Uses the same conventions for throwing
        away padding as "Y" does.  Does not close the input file.
^ A     appends all the rest of the file.  A cross between
        "A" and "^ Y".  Closes the input file.

<n>A    value is the 7-bit ASCII value of char arg chars
        to the right of the pointer.  Note that
        "0A" is the character immediately to the left of
        the pointer and "-<n>A" is the character <n>+1
        characters left of the pointer.  If .+<n>-1 is not
        within the bounds (real or virtual) of the buffer, a
        "NIB" error occurs.
<m>,<n>A
        is like <n>A except that when <n>A would cause a
        "NIB" error, <m>,<n>A will return <m>.  Thus,
        13,1A will return 13 iff the pointer is either at the
        end of the buffer or before a carriage return.

B       normally 0.  Actually, the number of the first character
        within the virtual buffer boundaries - but that will be
        the first char in the buffer (char number 0) unless you
        have used FS BOUND$ or FS V B$ to change that.

C       moves in the buffer relative to pointer:
<n>C    move pointer <n> chars to the right (1 char, if no arg).
        If that's out of the buffer, a "NIB" error results.

:C      like C, but returns -1 ordinarily,
        or 0 if C without colon would cause an error.
        :C is to C as :S is to S.

D       delete arg chars to right of pointer.
        If arg<0, delete to left of pointer.
```

"E" commands

```
E       is the prefix for most operations on files.

E^U<dir>$
        displays in the usual manner the directory of the
        device specified in the string argument, or the default
        device.  More precisely, reads the string arg and sets
        defaults, then does "^U".

E?<file>$
        tries to open <file>, and returns 0 if successful.
        Otherwise, the value is the TECO error code for the
        error that would occur if you tried to ER the file.
        The file does not stay open, and the open input file
        if any is not interfered with.

E_<old>$<new>$
        makes a copy of the file <old> and names it <new>.
        I/O is done in ASCII block mode.  The currently open
        input and output files are not affected.

EA      if drive <n> is already the default, does "<n>EA".
<n>EA   does .ASSIGN on dectape drive <n> (1 <= <n> <= 4).
        Dectapes may not be used if they are not assigned.
        The UNAME of the assigner becomes the tape's SNAME,
        and no program may open the tape unless it specifies
        that SNAME.
        The specified tape becomes the default device.

EC      close the input file, if any.  This should always be done
        whenever an input file is no longer needed;  otherwise, one
        of the system's disk channels will be tied up.
        ^ Y, EE and EX automatically do an EC.
        All other input operations always leave the input file open.

ED<file>$
        deletes <file>.

EE<file>$
        like infinity P commands then EF<file>$ and EC.

EF<file>$
        files output accumulated by PW and
        P commands with the name <file>.  <file> may not contain
        a device or SNAME;  they must have been specified when
        the file was opened (with EI or EW) and may not be changed.

EG      inserts in buffer on successive lines
        the current date (as YYMMDD),
        the current time (as HHMMSS),
        TECO's current sname,
        TECO's default filenames for E-commands,
        the real names of the file open for input (or,
         if there is none, the names of the last one there was)
        the date in text form,
        a 3-digit value as follows:
                1st digit = day of week today (0 = Sunday)
                2nd digit = day of week of 1st day of year
                3rd digit should be understood as binary:
                    4-bit = normal year, and after 2/28
                    2-bit = leap year
                    1-bit = daylight savings time in effect.
        and the phase of the moon.

EI      opens a file for writing on the default device.
        The filenames used will be "_TECO_ OUTPUT".
        When the output file is closed, it will normally be
        renamed to whatever names are specified.  However, if
        the TECO is killed, or another output file is opened,
        anything written will be on disk
        under the name "_TECO_ OUTPUT_"
0EI     sets the default to DSK:, then does EI.
<n>EI   sets default to dectape <n>, then does EI.
:EI     like EI, but uses the current filename defaults
        instead of "_TECO_ OUTPUT".  This is useful for opening
        on devics which do not support rename-while-open
        fully, such as the core link.
^ EI    like EI, but opens an old file in rewrite mode
        if there is one, rather than creating a new file
        in all cases.  Together with FS OFACCP$ and
        FS OFLENGTH$ this can be
        used to update an existing file in arbitrary ways.
        However, what you really want to use is:
^:EI    like ^ EI but uses the default filenames
        rather than "_TECO_ OUTPUT".

EJ<file>$
        restores the complete environment (q-reg values,
        buffer contents, flag settings, etc) from the
        specified file, which should be in the format
        produced by ^ EJ.  This restores all q-regs, buffers,
        and flags to what they were when the file was
        dumped.  Exception:  pure (:EJ) space is not changed,
        nor is FS :EJPAGE$.  After loading, TECO restarts itself,
        which implies that if a nonzero value was
        loaded into Q..L, it will be macroed.
        This is intended to be used in init files, for
        loading up complicated macro packages which would
        take a long time to load from source files.
        If the file isn't a dump file, or was dumped
        in a different TECO version, an "AOR" error occurs.

^ EJ<file>$
        dump all variable areas of TECO on the file open
        for writing (it must already be open), and file it
        under the specified filenames.  One should not
        write anything in the file before doing "^ EJ".
        Files written with ^ EJ can be loaded into a TECO with
        the EJ command, or they can be run as programs directly,
        in which case they will bootstrap in all the constant
        parts of TECO from the canonical place:
        .TECO.;TECPUR <TECO version>.  If you ^ EJ a file
        TS TECO on your home directory, then TECO^K will
        always get you that environment.

:EJ<file>$
        inserts the specified file into core, shareable and
        read-only, and returns a string pointer to the beginning
        of it.  :EJ assumes that FS :EJPAGE$ points to the lowest
```

page used by :EJ's, and inserts the file below that
page (updating the flag appropriately). Memory is
used starting from the top of core and
working down to page 340.
See the sections "buffers - internal format"
and "buffer and string pointers - internal format"
for information on what can go in the file.
An ordinary ASCII text file is not suitable for :EJ'ing.
A file to be :EJ'ed must, first of all, be a single
string whose length (including its header) must be a
multiple of 5120 (1K words of characters). Within that
string lives the other strings or whatever that are
the data in the file. Their format is unrestricted
except that the first thing in the file (starting after
the header for the file as a whole) should be a string
which is the file's "loader macro" which must know how
to return the data in the file when asked for it.
The loader macro shoudl expect to be called with the
name of the desired data (as a string) as the first
argument (^X), and a pointer to the whole file (as a string)
as the second argument (^Y). The reason for passing it
the pointer to the file is so that the loader itself can
be pure (independent of the particular file containing it).
The pointer to the file, plus 4, gives a pointer to the
loader itself, if the loader wishes to examine its body.
The loader macro should return as its value the string
which is the value associated with the specified name.
If the name is undefined in the current file, the loader
should pass the request on to the loader in the next file.
The next file can be assumed to start right after the
end of the current one, so that ^Y+FQ(^Y)+4 is a pointer
to it. If there is another file, FQ of that will be
positive; otherwise (this is the last file in memory)
FQ of that will be -1.
If there are no more files, the loader should return 0.
The goal is that several files with different loader macros
should be :EJ'able in any order, and yet allow things to be
loaded out of any of them at any time.

EK:
<n>EK unload dectape number <n>.

EL display in the standard manner the directory of the
default device. This command
does not use the user's buffer display macro; in fact,
the buffer display macro might well use this command.

EM insert in buffer file directory of the default device.

EN<old>$<new>$
renames the file <old> to have the name <new>.
The device and SNAME may not be changed; they should not
be specified in <new>.

EO<file>$
sets the dumped-on-tape bit of <file>, thus preventing
it from being included in the next incremental dump.
To clear the bit, rename the file as itself.

EP<file>$

```
        does ER<file>$, then bigprints file name
        twice on device(s) open for writing.

EQ<from>$<to>$
        creates a link named <from> pointing to the file <to>.
        devices COM:, TPL: and SYS: are understood.
        An attempt to link to a non-disk device is an error.

ER<file>$
        opens <file> for input.  The "Y", "A" and "FY" commands
        in various forms may be used to read from the file.
        As soon as the file is no longer needed (eg, if all
        of it has been read), an "EC" should be done to close
        the input channel.  "^ Y" and "EE" do an automatic "EC".
        FS IF ACCESS$, FS IF LENGTH$, and FS IF CDATE$ make it
        possible to get or set various parameters of the file.
0ER     is similar but defaults device to DSK:

ES:
<n>ES<nam>$
        sets the tape name of dectape number <n> to <nam>.
        <nam> must contain at most 3 characters.  If <n> is
        omitted, the current default device is used.

ET<file>$
        sets the default filenames to <file>.

EU:
<n>EU   deassign dectape number <n> (see "EA").

EW<dir>$
        like EI but device specified by
        following text string rather than by a numeric arg.
:EW<file>$
        like EW, but uses the specified filenames
        instead of "_TECO_ OUTPUT".  This is useful for opening
        on devics which do not support rename-while-open
        fully, such as the core link.
^ EW<dir>$
        like EW, but opens an old file in rewrite mode
        if there is one, rather than creating a new file
        in all cases.  Together with FS OFACCP$, this can be
        used to update an existing file in arbitrary ways.
        However, what you really want to use is:
^:EW<file>$
        like ^ EW but allows filenames to be specified
        rather than using "_TECO_ OUTPUT".

EX<file>$
        if an output file is open, first does "EE<file>$"
        (in this case, <file> may not have a device or sname).
        Then, tells DDT to tell MIDAS to assemble the program,
        making an error output file, and to load
        it into a job named DEBUG.
        When TECO is proceded later the screen will be cleared
        and the execution of the command string will continue.
:EX<file>$
        is similar but will direct MIDAS to cref the
        program and DDT to run CREF, outputting to the TPL.
```

EY<dir>$
        like EL but specified device and SNAME.

EZ<dir>$
        like EM but etc.

E[push the input channel, if any.
        Saves the current input file and position in it, or saves
        the fact that no file is open.
        Useful for reading in a file without clobbering any
        partially read input file.
        Note: for this and the next
        three commands ("E]", "E\", "E^"),
        the file open for input must be
        randomly accessible (=DSK). The one
        open for output need not be.
        FS PAGENUM$ and FS LASTPAGE$ are saved by E[
        and restored by E].

E]   pop the input channel.
        If any input file was open, the rest of it is flushed.
        Further input will come from the file that was popped.
        (see "E[".)

E\   push output channel.
        (see "E[".)

E^   pop output channel.
        If an output file is open, it is closed without being
        renamed, so it is probably filed as "_TECO_ OUTPUT".
        (see "E[".)

"F" commands

F   further decoded by the next character as follows:

F^@:
<m>,<n>F^@
returns 2 values, which are <m> and <n> in numerical order. Thus, "1,2F^@" ans "2,1F^@" both return 1,2. "<m>,<n>F^@T" is the same as "<m>,<n>T" except that the former will never cause a "2<1" error.

<n>F^@ returns, in numerical order, 2 args that delimit a range of the buffer extending <n> lines from the pointer. Thus, "<n>F^@T" is the same as "<n>T".

F^A:
<m>,<n>F^A<q>
this command scans the range of the buffer from <m> to <n> using the dispatch table in q-register <q>. That is, each character found in the buffer during the scan will be looked up in the dispatch table and the specified actions will be performed. The dispatch table should be a string or buffer with at least 128*5 characters in it - 5 for each ASCII character. Each character seen has its ASCII code multiplied by 5 to index into the table, and the 5 chars found there are executed as TECO commands. When that is done, the char that was found in the buffer is in Q..0 as a number, Q..3 holds the dispatch table that was in use (so that the dispatch commands can change it if they wish) and Q..2 holds the end of the range to be scanned (the commands executed may modify Q..2 to cause the scan to end early or to account for insertions or deletions they do). For efficiency, if the first of the 5 chars in the dispatch table is a space, the 5 are not macroed. Instead, the second character, minus 64, is added into Q..1, and the third is specially decoded. " " means no action; this feature makes to easy to skip over most chars, keeping track of horizontal position. Other permissible third characters are "(" and ")". Their use is in counting parens or brackets. "(" means that if the scan is backwards and Q..1 is positive, the scan should terminate. ")" means that if the scan is forward and Q..1 is negative, the scan should terminate. If an open-paren-like character is given the dispatch " A(" and the close is given " _) ", the same dispatch table may be used to find the end of a balanced string going either forward or backward. "F^A" may be given 0 or 1 arg - it turns them into 2 the way "K", "T", etc. do.

<m>,<n>^ F^A<q>
the uparrow modifier causes the scan to go backwards.

F^B:
<ch>F^B<string>$
searches for the character <ch> in <string>. <ch> should be the ASCII code for a character. If that character does not occur in <string>, -1 will be returned. If the char does occur, the value will be the position of its first occurrence (eg., 0 if it is the first char).

F^E<string>$
        replace <string> into the buffer at point. Replacing
        means inserting, and deleting an equal number of
        characters so that the size of the buffer does not
        change. The advantage of this command over
        "I<string>$ FKD" is that the gap need not be moved.
<n>F^E<string>$
        replaces <string> in at <n>. Point does not move.
        Like ".(<n>J F^E<string>$)J".
<n>:F^E<q><string>$
        replaces <string> into the string or buffer in q-reg <q>
        starting at the <n>'th character. This is the only way
        that the actual contents of a string can be altered,
        although other commands copy pointers to strings, or
        create new ones. If this command is done, it may be
        necessary to sweep the jump cache (see "F?") if the
        string being altered might be a macro that might
        contain "O" commands.

F^X   within a macro, this command returns as its values the
        arguments that were given to the macro. As many values
        are returned as args were given. To find out how many
        there were, use F^Y.

F^Y   returns a value saying how many args it was given. For
        example, WF^Y returns 0; W1F^Y, 1; W1,F^Y, 2; W1,2F^Y, 3.

F"<condition>
        F" is a conditional. It works like ", except that
        whereas " throws away its argument after testing it,
        F" returns its argument, whether it succeeds or fails.
        Thus, QA-QBF"LW'+QB implements max(QA,QB).

F$ is used to read or set the status of case conversion
        on input and case flagging on output, for terminals
        that do not have lower case. What those features do
        when activated is described below. F$ controls them thus:
        with no arg, returns the value of FS CASE $
        and inserts in the buffer before the pointer the
        case-shift char, if any, and the case-lock char, if any.
        With an arg, sets FS CASE $ to that arg, and takes a string
        argument whose 1st char becomes the new case-shift,
        and whose 2nd char becomes the new case-lock.
        (if the chars are the same it is only a case-shift)
        (if there are no chars, you get no case-shift or -lock, etc)
        the old case-shift and case-lock, if any, become normal
        characters before the string arg is read. Thus, repeating
        an F$ command will not screw up.

        Case conversion on input:

        When FS CASE$ is nonzero, all letters are normally
        converted to the standard case, which is upper case
        if FS CASE $ is positive; lower if negative.
        The case-shift char causes the next char to be read in
        the alternate case. The case-lock char complements
        the standard case temporarily
        (it is reset for each cmd string).
        The case-shift quotes itself and the case-lock.
        The "upper case special characters" which are "@[\]^_"

are not normally converted, but if one of them is preceded
by a case-shift it will be case shifted into a "lower
case special character" (one of "`{|}~<rubout>").
(note that case conversion happens during command execution
now, so that it makes sense to change modes in the middle
of a command string. However, no case conversion is done
on characters that come from macros)
(note also that it doesn't work well to have FS CASE$
and FS *RSET$ simultaneously nonzero, for complicated reasons).

Case-flagging on output:

If FS CASE $ is odd, chars in the nonstandard case
(and the "lower case special characters") will
be preceded by case-shifts on typeout from the buffer.
If FS CASE$ is even, no flagging is done.

F(is like (except that whereas (returns no values,
F(returns its arguments. F(therefore facilitates
putting the same information in two places without the
use of a q-reg.

<n>F(<m>)
converts <n> feet <m> inches to inches.

F) resembles), but whereas) returns its arguments
combined with the values stored by the matched (,
F) returns precisely its arguments. The data saved
by the corresponding (is discarded.

F* reads and ignores a string argument. Useful in macros
because "F*^]^X$" reads and ignores a string argument
passed to the macro.

F+ clears the screen. Like "^L", but does not separate
pages in files. If only a part of the screen is in
use (FS LINES$ or FS TOP LINE$ is nonzero), only that
part is cleared. To be sure to clear the whole screen,
bind both of those flags to 0 around the F+.

F6<string>$
returns a word of SIXBIT containing the first six
characters of <string>.

<sixbit>F6
interpreting <sixbit> as a word of SIXBIT, converts
it to ASCII which is then inserted in the buffer
before the pointer.

<sixbit>^ F6
returns a string containing the characters of <sixbit>.

F;<tag>$
is a "throw", a la LISP. See "F<" below.

F<!<tag>! ... >
is a catch. If anywhere in the arbitrary code which
may replace the "..." a "F;<tag>$" command is executed,
control will transfer to after the ">" that ends the
catch. If no "F;" is executed, the catch acts like
an iteration, so if the code should be executed only once,
"1F<" should be used. When a "F;" or throw happens,
all macros, iterations and errsets

entered within the catch are exited and the
q-reg pdl is unwound to the level it had at the time
the catch was entered. Example:

```
    F<!FOO! [A FIUA QAI QA-32"E F;FOO$' ]A>
```

reads characters from the terminal and inserts them, up to
but not including the first space, and does not modify
the q-reg pdl (never mind that this macro might be improved).
If a throw ("F;") is done to a tag that does not belong
to any catch containing it, an error "UCT" occrurs,
at which time nothing has been unwound.
The ">" ending a catch will return 0 if the catch was
exited normally; if it was thrown out of, the argument
given to the throw will be returned.
Note that case is not significant in the F; or in the F<.

:F< is an errset and a catch at the same time!
Amazing what happens when your program works
by simply examining a bunch of flags!

F= does an ordered comparison of strings.
If "F=" has numeric args, they specify the range of buffer
to be used as the first comparison string. Otherwise,
the "=" should be followed by the name of a q-reg which
should hold the first comparison string.
The second comparison string should follow the command as
a string argument, as for the "I" command. (the ^ modifier
works just as it does for the "I" command)
the two strings are compared, and if they are equal
0 is returned as the value of the "F=" command.
If the first string is greater, a positive value
is returned; if the second, a negative value.
If the value isn't 0, its absolute value is 1 +
the position in the string of the first difference
(1 if the first characters differ, etc.).
A string is considered to be
greater than any of its initial segments.

F? mbox control; argument is bit-decoded.
No arg, or arg=0, implies arg=30 .

bit 1.1 - close gap.
May be needed for communication with other
programs that don't understand the gap.

bit 1.2 - GC string space.
Useful before dumping out, or if it is suspected
many strings have recently been discarded.

bit 1.3 - sweep the jump cache.
Necessary if a string's contents have been
altered by the F^E command, and might be a macro
that might have contained "O" commands.
Also necessary if :EJ is used after increasing
the value of FS :EJPAGE$ (thus replacing one
file with another in core).

bit 1.4 - flush unoccupied core.
Good to do every so often,
or if it is likely the buffer has just shrunk.

bit 1.5 - close the gap, if it is > 5000 characters long.
It is good to do this every so often,
in case the user deletes large amounts of text;
say, whenever excess core is flushed.

FA performs text justification on a range of the buffer
specified by 1 or 2 args (as for K, T, commands, etc.).
The idea is that whenever you edit a paragraph,
you use FA or a macro that uses FA to re-justify it.
The line size is kept in FS ADLINE $.
A CRLF followed by a CRLF, space or tab causes a break.
So does a CRLF, space or tab as the first character
of the text being justified.
An invisible break can be produced before or after a
line by beginning or ending it with space-backspace.
CRLFs that do not cause breaks are turned into spaces.
Excess spaces (or CRLFs turned into spaces) are not removed;
if a CRLF is being inserted where there are multiple spaces,
it replaces the last space, so that the others stay around
invisibly at the end of the line. Thus, if you once put
two spaces at the end of a sentence, there will always
be two spaces there.
Spaces at the beginning of a line are treated as part of
the first word of the line for justification purposes,
to prevent indentation of paragraphs from changing.
The last part-line of stuff to be justified is only filled.
Tabs prevent alteration of what precedes them on a line.
I suggest using "FA" in the following macro:

```
    [0 Z-^YU0
        ^XJ <.,Z-Q0FB.
^O?
$;  :0L I $>
    !""Make sure .'s and ?'s at end of line have spaces!
        ^X,Q0FA !actually justify!
    ]0
```

If you want indented paragraphs, simply indent them the
right amount when you type them in. "FA" will leave the
indentation alone. "FA" knows about backspace.
Sometimes it is desirable to put a space in a word. To
do that, use space-backspace-space.

^ FA like "FA" but only fills (doesn't justify)

FB bounded search. Takes numeric args like K, T, etc.
Specifying area of buffer to search, and a string
argument like S, N, etc. The colon and uparrow flags
are used as they are by other search commands.
:FB is like :S, not like :L. That is, :FB returns a
value indicating extent of success, and searches the
same range of the buffer as FB with no :.
If two args in decreasing order are given searching is
done in reverse. With one negative arg, the search is
forward, but through a range that ends at the pointer.

FC takes arguments like K and
converts the specified portion of the buffer to
lower-case. Only letters are converted.

^ FC converts a specified portion of the buffer to upper case.

FD a list manipulating command whose main use is in
"<arg>FDL", which moves down <arg> levels of parens.

FD returns a pair of args for the next command.
If <arg> is positive, they specify the range of the buffer from the pointer rightward to the first character that is <arg> levels up;
if negative, leftward to the first character -<arg> levels up.

FE inserts a list of TECO error messages and explanations in the buffer before the pointer, one message per line.

<arg>FE inserts only the line describing the error of which <arg> is the error code.
<arg> might have been returned by an errset, or might be the value of FS ERROR $.
Since error codes are actually strings, <arg>FE is equivalent to G(<arg>) I<crlf>$

^ FE<errname>$
returns the error code associated with the given error name <errname>. Only the first three characters of <errname> are used. This is useful for analysing anticipated possible errors and recovering appropriately. Another way to do that is to compare the first three characters of the error code, which is a string, against the expected ones, with F~.

:FE inserts a list of FS flag names in the buffer before the pointer, one name per line.

FG does error processing. With no argument, it simply rings the terminal's bell. Given the ^ modifier, it also throws away type-ahead. Given a nonzero numeric argument (which should be s string), FG prints its contents as an error message (obeying FS VERBOSE$). Accompanying the numeric argument with the : modifier causes the error message to be typed at the top of the screen (think of : FT). Unlike most commands that do typeout, FG does not change Q..H, so that typing an error message will not inhibit buffer display or ^R redisplay. Instead, FG sets FS ERRFLG$ so that the next buffer display will not overwrite the line(s) occupied by the error message.

FI input one character from the terminal and return its ASCII value. (same as vw without the v)
if the "mode" (in q-reg ..J) has changed, the new value will be put on the screen, unless input is already waiting when the FI is executed.
Note that CR (but not TV control-M) causes a LF to be put in FS REREAD$. Thus, CR typed in is read as CRLF. To flush the LF that may be present after something that reads in as CR, do -1FS REREAD$.

:FI similar to FI, but doesn't flush the character.
It will be re-read by the next fi or by TECO's command string reader.

^ FI is like FI, but returns a character in the 9-bit TV character set, rather than converting it to ASCII as FI does. Note that CR causes a LF to be put in FS REREAD$, and control-CR causes a control-LF to go there.

```
        In the TV character set, the 400 bit means "meta",
        the 200 bit means "control", and the bottom 7 bits
        are a printing character (if < 40, it is one of the
        new TV graphics, or else it is a formatting character).
        Note that there exist controlified lower case letters
        different from their upper case counterpart (for example,
        341 octal is control lower case a).
^:FI    analogous.

FJ      insert the job's command string as read from DDT
        in the buffer.  Will normally end with a CR-LF
        but may be null.

FK      returns minus the value of FS INSLEN$;  that is,
        minus the length of the last string insreted by "I", "G" or
        "\", or found by a search or "FW".  FK is negative except
        after a successful backward search, or backwards "FW".
        Thus "SFOO$FKC" will move to the beginning of the FOO found.
        "-SFOO$FKC" will put it at the end of the FOO found.
        "SFOO$FKDIBAR$" will replace FOO with BAR.  See "^F".
        IBLETCH$FKC" inserts BLETCH and backs over it.

FL      parses lists or S-expressions:
<arg>FL a list maniulating command, that returns 2 values specifying
        a range of the buffer.  If <arg> is >0, the range
        returned is that containing the next <arg> lists
        to the right of the pointer;  if <arg> is <0,
        the range is that containing <arg> lists
        to the left of the pointer.  This command should be
        followed by a command such as K, T, X, FX ...
        which can take 2 args;  the specified number of lists
        will be deleted, typed, put in q-reg, etc.
        To move to the other side of the lists, do "<arg>FLL".
        The syntax parsed by FL is controlled by the delimiter
        dispatch table in Q..D;  the character types known
        are "A", " ", "|", "/", "(", ")" and "'", and any character
        can be redefined to be of any of those types.

<arg> ^ FL
        is like "<arg>FL", but refers to <arg> s-expressions
        rather than <arg> lists.  An s-expression is either
        a list or a LISP atom, whichever is encountered first.
        "^ FW" is used to find LISP atoms when necessary.

FM:
<m>,<n>FM
        attempts to move the pointer so that the cursor will
        appear at hpos <n>, <m> lines below where it started out.
        "FM" without the "^" modifier can move only toward the
        end of the buffer.  It operates by moving the pointer
        downward in the buffer until either 1) the exact desired
        absolute hpos and relative vpos have been reached, in
        which case "FM" simply returns, or 2) the end of the buffer
        is reached, which causes a "NIB" error, or 3) the line
        below the desired one is reached, in which case it is
        known that the desired combination of hpos and vpos does
        not exist, so FM reverses its motion until it is back
        on the desired line, then issues a "NHP" error.
        "FM" tries to avoid leaving the cursor between
        a CR and the following linefeed.
```

"FM" will not currently work if ^R mode has never been
entered, but it need not be in ^R mode.
The ":" modifier causes "FM" to accept any hpos greater
than or equal to the second argument as a condition for
success, rather than demanding exact equality.
The "^" modifier causes "FM" to scan toward the beginning
of the buffer rather than toward the end. The first argument
should not be positive. The algorithm
is otherwise unchanged and ":" has the same meaning
(accept any hpos >= the specified one).

FN is the same as [..N :I..N. It is needed because it
elminates the possibility of a ^G-quit between the
push and the insert. If such a quit happened, the
previously set up undo action would be performed
twice instead of once, and that might have bad results.
To perform the opposite action - pop and macro Q..N -
just do "-FS QPUN$".
The ..N macro has no effect on the value(s)
returned by the macro that set it up.
The uparrow flag allows the user to specify a string
delimiter, as with the I command.

FO<q><name>$

binary-searches tables of fixed-length entries.
It is intended for searching and constructing symbol tables.
<q> should be a q-vector or pure string containing
the table, and <name> the item to search for.
The table's data must be an integral number of words.
The first word of the table must contain the number of words
per table entry; the rest of the table is then divided into
entries of that size. The first word of each entry should be
the entry's name, as a TECO string pointer. This name is what
FO will match against its string argument. The second word
of each entry should be the value; the use of any extra
words is up to the user. The entries' names must always be
kept in increasing order, as F~ would say, or FO's binary
search will lose. Also, they should not contain leading,
trailing, or multiple spaces, or any tabs. Their case is
ignored.

FO, without colon, will return the value from the entry if
the name is found; otherwise, an UVN or AVN error results.

:FO returns the offset (in words) of the entry found;
if the name is not found, :FO returns minus the
offset (in words) of the place the name ought to be inserted
in the table. The offset of the first entry in the table is
1, to skip the word in the front that contains the entry
size.

<arg>FO is like plain FO except that if the name is
undefined <arg> is returned as its "default value".
Ambiguous names still cause errors.

Here is a macro that uses FO to create variables that
can then be used with the Q$<name>$ construction:

```
        [0 :I0≥←$              ! Get variable name in Q0 !
   ! Find it, or where to put it if not found !
        [1 :FO..Q≥0$U1
        Q1"L                   ! If not found, put it in !
           Q..Q[..O            ! Symbol table lives in ..Q !
           -Q1*5J 10,0I 10R    ! Make space at right place !
```

```
    ! Install string containing variable name !
            Q0,.FSWORD$'
```

If this macro is put in QV, then MVFoo$ will create a variable named Foo.

If the table is a pure string, the data must start on a word boundary, which means that the string's header must start in the second character in its word. In addition, the pointers to the entries' names are taken to be relative to the table itself. That is, the "pointer" should be an integer which, when added to the TECO string pointer to the table, should give a TECO string pointer to the name of the entry.

FP:

<obj>FP returns a number describing the data type of the object <obj>. The possible values are:

-4	A number, not even in range to point into pure or impure string space.
-3	A number that is in range for pure string space but does not point at a valid string header.
-2	A number that is in range for impure string space but does not point at a valid string header.
-1	A dead buffer.
0	A living buffer.
1	A Q-vector.
100	A pure string.
101	An impure string.

FQ<q> returns the number of characters in q-reg <q> (-1 is returned if the q-register contains a number).

FR tells TECO to update the displayed mode from q-reg ..J, provided it has changed, and no terminal input is available.

FS reads in a flag name as a text argument. Flag names may be any length, but only the first six characters are significant. Spaces are totally ignored. Only enough of the flag name to make it non-ambiguous is required. However, in programs, abbreviation should be minimized. The result of the FS is the current value of the flag. If an argument is given to the FS and the flag can be set, is then set to that value. If a flag can be set, to make it's value the second operand of an arithmetic operator put the FS command in parens. Otherwise, FS will think it has an arg and set the flag. Flags labeled "read-only" do not require that precaution. Flags currently implemented are:

FS % BOTTOM$ specifies the size of the bottom margin as a percentage of the number of lines being displayed. Initially 10. Rather than let the cursor appear inside the bottom margin, TECO will choose a new buffer window - unless the bottom of the buffer appears on the screen already.

FS % CENTER$ specifies where TECO should prefer to put the cursor when choosing a new window, as a percentage of the screen from the top. Applies even if the end of the buffer appears on the screen - in fact, the purpose of this variable is to make sure that when you go to the end of the buffer some blank space is provided to insert into without total redisplay. Initialy 40.

FS % END$	specifies (as a percentage of total size) the size of the area at the bottom such that TECO should never choose a new window putting the cursor in that area. Initially 30.
FS % TOP$	the size of the top margin. Analogous to FS %BOTTOM$. Initially 10.
FS %TOFCI$	(read only) nonzero if the terminal can generate the full 9-bit character set. This flag reflects the bit with the same name in the terminal's TTYOPT variable, and it is updeat whenever TECO is restarted or FS TTY INIT$ is done.
FS %TOHDX$	(read only) nonzero if the terminal is half-duplex. See FS %TOFCI$.
FS %TOLWR$	(read only) nonzero if the terminal can generate lower case characters. See FS %TOFCI$.
FS %TOMOR$	(read only) nonzero if the user wants --MORE-- processing, in general. See FS %TOFCI$.
FS %TOOVR$	(read only) nonzero if the terminal is capable of overprinting. See FS %TOFCI$.
FS %TOROL$	(read only) nonzero if the user has selected scroll mode. See FS %TOFCI$.
FS %TOSAI$	(read only) nonzero if the terminal can print the SAIL character set. See FS %TOFCI$.
FS *RSET$	initially 0. Nonzero suppresses automatic unwinding of TECO's various pdls each time through the top level loop. In other words, when FS *RSET$ is non-zero, errors not caught by errsets enter break-loops in which q-regs may be examined (unless the user's error-handler macro in q-reg ..P intervenes). The break-loop may be returned from with ")^\", or thrown out of with "^W" or "F;". The suspended program and its callers may be examined with FS BACKTRACE$. For more info on break-loops, see under q-reg ..P.
FS .CLRMOD$	(normally -1) if < 0, TECO clears the screen whenever it gets the terminal back from its superior. If 0, that is not done (used mainly for debugging TECO). If > 0, screen clearing is totally eliminated, even if requested by the program (use this for debugging macros that try to destroy trace information).
FS .KILMOD$	normally -1. If 0, FS BKILL$ doesn't actually kill.
FS :EJ PAGE$	is the number of the lowest page used by :EJ'd shared pure files. Initially 256. If multlipied by 5*1024, and then added to 400000000000 (octal), the result is a suitable string pointer to the last file :EJ'd. :EJ looks at this flag to

figure out where to insert the next file to avoid clobbering the previous ones. By resetting the flag a file may be essentially flushed.

FS ADLINE$ is the line-size used by the FA command.

FS ALTCOUNT$ the number of $$'s that TECO has seen at interrupt level.
That is, an approximation to the number of command strings that the user has typed ahead.
Useful in user-defined buffer display macros (q-reg ..B).

FS BACKTRACE$

used to see what program is running at higher levels of the mcro pdl. The program is inserted in the buffer, and point is put at the place it is executing. 0FS BACKTRACE$ gets the routine one level up from the one that uses it;
<n>FS BACKTRACE$ gets the routine <n>+1 levels up.

**FS B BIND$** is useless, but F[B BIND$ and F] B BIND$ are useful for pushing to a temporary buffer, or popping back from one. F[B BIND$ pushes ..O, then does FS B CREATE$, but with the extra feature that if an error happens instead of just popping back ..O, the temporary buffer will be killed. That is because instead of doing]..O, F] B BIND$ will be done, which is just like FS B KILL$ with no argument.
If after creating a buffer with F[B BIND$ you change your mind and want to keep it, pop the previously selected buffer off the pdl with the] command. The F[B BIND$ will no longer be on the stack to kill the new buffer when you return.

<n>FS B CONS$ returns a newly cons'ed up buffer <n> characters long. The contents are all initially zeros, and the pointer starts out at the beginning of the buffer. If <n> is not specified, 0 is assumed.
When a buffer is newly created it is at the top of memory. The closer a buffer is to the top of memory, the more efficient it is to do large amounts of insertion in it.

**FS BCREATE$** is like FS B CONS$ U..O - the buffer is selected instead of returned.

**FS BKILL$** see q-reg ..O. FS BKILL$ is used for freeing buffers explicitly. With an argument, it frees the argument, which should be the result of applying the q command to a q-reg containing a buffer. Attempting to kill the currently selected buffer is an error. For example, QAFS BKILL$ kills the buffer in qa. After that is done, qa still contains a buffer pointer, but it has been marked "dead". If there were other pointers to the same buffer in other q-regs, TECO will regard them too as "dead" buffer pointers. An attempt to select the buffer using one of those pointers will result in an error.

	FS BKILL$ may be used without an argument, in which case it pops the q-reg pdl into Q..O, and if the new value of Q..O is different from the old, the old value is killed.
FS BOTHCASE$	(initially 0) >0 => searches ignore case of letters. That is, the case used in the search string is irrelevant, and either lower or upper case will be found. <0 => searches ignore case of special characters also ("@[\]^_" = "`{|}~<rubout>").
FS BOUNDARIES$	reads or sets the virtual buffer boundaries (this command returns a pair of values) the virtual boundaries determine the portion of the buffer that most other commands are allowed to notice. Normally the virtual boundaries contain the whole buffer. See the B, Z and H commands.
FS CASE$	like F$ (F-dollar) but neither inserts the case-shift and case-lock if no arg, nor expects a string arg. That is, it gets or sets the numeric quantity which determines the standard case and whether to flag on output.
FS CTL MTA$	if negative suppresses the ^R-mode definitions of all control-meta-letters (and ctl-meta-[, \,], ^ and _) to make it easy to insert control characters. This mode is convenient for editing TECO commands.
FS DATA SWITCHES$	(read only) the contents of the PDP10 console switches.
FS DATE$	(read only) is the current date and time, as a number in file-date format. It can be fed to FS FD CONVERT$ or to FS IF CDATE$.
FS D DEVICE$	is the current default device name, as a numeric sixbit word. See F6.
FS D FILE$	is the current default filename (that ER$ would use) as a string pointer. Do G(FS D FILE$) to insert it in the buffer. This flag is very useful for pushing and popping with F[and F]. The exact format of the string is the device name, a colon, a tab, the SNAME, a semicolon, a tab, the FN1, a tab, and the FN2. To extract specific names from the string, use the tabs. It is not reliable use the colon and semicolon, because there may be other colons and semicolons ^Q'd in the names themselves.
FS D FN1$	is the current default first file name, as a numeric sixbit word. See F6.
FS D FN2$	is the current default second filename, as a numeric sixbit word. See F6.
FS D SNAME$	is the current default SNAME, as a numeric sixbit word. See F6.

FS ECHO DISPLAY$

(write only) like FS ECHO OUT$, but outputs in display mode, so that ITS ^P-codes may be sent. See .INFO.;ITS TTY for details on available options.

FS ECHO LINES$ the number of lines at the screen bottom to be used for command echoing. Default is 1/6 of screen size, except 0 on printing terminals. If this flag is set to -<n>, echoing is turned off, and there are <n>-1 echo lines. Thus,

 -1FS ECHOLINES$ makes no echo and no echo area;
 -5FS ECHOLINES$ makes no echo but a 4 line echo area;
 0FS ECHOLINES$ makes echo but no echo area.

Even if echoing is off, FS ECHOOUT$ may be used.

FS ECHO OUT$ (write only) treats its numeric argument as the ASCII code for a character, and outputs the character in the echo area as it would be echoed. Thus, sending a CR will actually do a CRLF, and sending a ^A will print either downarrow (in :TCTYP SAIL mode) or uparrow-A.

FS ERR$ same as FS ERROR$ if read; if written, causes an error with the error code that is written in it. Thus, to cause a "You Lose" error with 3-letter code LUZ, do :I*LUZ<tab>You Lose$ FS ERR$. Your error message should not contain any CRLFs. Users who wish to generate errors themselves with the same codes that TECO uses should use TECO's standard strings for those errors (that is, do ^ FE IFN$ FS ERR$) so that comparing FS ERROR$ against ^ FE values will still work.

FS ERRFLG$ is used to signal the buffer display routine (whether built-in or user-written) that an error message is on the screen and should not be overwritten. Its value is -<n> if the first <n> lines contain an error message, or nonnegative if there is none. If typeout is done when FS ERRFLG$ is negative, TECO will not actually type the first <n> lines of it. The <n>+1st line of typeout will appear in its normal position, beneath the error message. By that time, FS ERRFLG$ will be zero again.

FS ERROR$ the error code of the most recent error. Errors caught by errsets are included. A TECO "error code" is now just a string containing the text of the error message. Everything up to the first tab is the "brief" part of the error message; if FS VERBOSE$ is 0, that is all that TECO will print out. You can see if an error was (for example) an "IFN" error by doing F=(FS ERROR$)IFN$ and seeing if the result's absolute value is 4. It still works to compare against ^ FEIFN$, which returns the standard string that TECO always uses for internally-generated IFN errors.
 I.T.S. I/O errors now have messages starting with "OPNnnn" where nnn is the I.T.S. open-failure code. Macros which used to

decode I/O errors by numeric comparison must switch to using F=, since the strings for such errors are consed up by TECO as needed.

FS EXIT$	(write only) does a .BREAK 16, using the argument to FS EXIT$ as the address field. AC 2 will contain the address of the 7-word "buffer block" describing the current buffer. See the section "buffer block" at the end.
FS FDCONVERT$	converts numeric file dates to text and vice versa. If there is a numeric arg, it is assumed to be in the format for its file dates, and converted to a text string which is inserted in the buffer. The form of the string is dd/mm/yy hh:mm:ss. In this case, no numeric value is returned. If there is no arg, a text string is read from the buffer starting at ., and . is moved over the string. The string should be in the format inserted by FS FDCONV$ with argument, and will be converted to a numeric file date which will be the value of FS FDCONV$. See FS IFCDATE$ and ofcdate.
FS FILE PAD$	the character used to pad the last word of files written by TECO. Normally 3 (for ^C).
FS FLUSHED$	is nonzero if a --MORE-- has been flushed, and type-out is therefore suppressed. The flag is positive if the flushage was due to a rubout, negative otherwise. You can stop generating type-out when FS FLUSHED$ is nonzero, or you can clear it to make type-out start actually appearing again.
FS FNAM SYNTAX$	controls TECO's filename readed. If 0, when only one filename is present, it is used as the fn2 (this is the default). If positive, a lone filename is used as the fn1. If negative, a lone filename is used as the fn1 and automatically defaults the fn2 to ">". The file COM:.TECO. (INIT) uses this flag to process a DDT command line.
FS GAP LENGTH$	(read only) the length of the gap. This is the value of EXTRAC (see "buffer block").
FS GAP LOCATION$	(read only) the buffer position of the gap. This is GPT-BEG (see "buffer block").
FS HEIGHT$	(read only) number of lines on the screen, on display terminals (including --MORE-- and command lines). On printing terminals, wil be a very large number.
FS H POSITION$	(read-only) returns the number of character positions there would be to the left of the type ball if the contents of the buffer (or at least everything after the previous carret) were printed on a hardcopy terminal with hardware 8 character tabbing and backspace.
FS IBASE$	the input radix for numbers not followed by "."

	(initially 8+2)
FS I.BASE$	the input radix for numbers ended by ".". Initially 8.
FS IF ACCESS$	(write-only) sets the access pointer of the current input file - the argument is the desired character address in the file.
FS IF CDATE$	the creation date of the currently open input file. Arg and value are in numeric file date form - see FS FD CONVERT$.
FS IF LENGTH$	(read-only) the length, in characters, of the currently open input file; or -1, if that length is unknown (because the file is on a device for which the fillen system call is unimplemented). Error if no file is open.
FS IMAGE OUT$	outputs arg as to the terminal as a character in superimage mode. Returns no value. 7FS IMAG$ types a bell, but in future versions of ITS it may not, so use FG instead.
FS INSLEN$	length of last string inserted into the buffer with an "I", "G" or "\", or found with a search command or "FW". FS INSLEN$ will be negative after a backward search or "FW" with negative arg. See "FK" and "^F".
FS JNAME$	(read only) returns the JNAME of the job TECO is running in, as a numeric SIXBIT word, which can be converted into text by the F6 command. Note that the XJNAME is also available, and might be better for your purpose - see FS XJNAME$.
FS LAST PAGE$	(read only) set to -1 when an input file is opened; set to -1 as soon as the last char of the file is read in. Saved by E[- E]. Updated by FS IF ACCESS$. "<AZJ FS LASTPA$; 12I>" gobbles a whole file.
FS LINES$	(initially 0) determines the number of lines displayed by standard buffer display, and, for display terminals, the number of lines to use at all. 0 => a full screen on displays, 2 lines on printing terminals. <n> not zero => <n> lines.
FS LISPT$	normally 0, this flag is set nonzero if TECO is started at 2 + its normal starting address. This is intended to indicate to TECO programs that passing of text between TECO and its superior is desired.
FS LISTEN$	returns nonzero if there is input available to be read by "FI". If given an arg, then if no input is available, the arg is typed out using FS ECHOOUT$.
FS MACHINE$	(read only) returns the name of the machine TECO is running on, as a numeric SIXBIT word, which can be

	turned into text with F6.
FS MSNAME$	is the user's working directory name (set up from the SNAME that TECO was given when it started up), as a numeric SIXBIT word, which can be converted into text by the F6 command.
FS NOOP ALTMODE$	if negative, altmode is always a noop as a command. If 0, altmode is an error as a command. If >0, always ends execution (as ^_ does). Initially -1. The old treatment of altmodes was as if this flag were set to 1.
FS NOQUIT$	gives the user control of TECO's ^G-quit mechanism. See "^G".
FS OF ACCESS$	(write-only) sets the access pointer in the output file. The argument must be a multiple of 5. If the last output done did not end on a word boundary, rather than throwing away the remaining characters, an error occurs.
FS OF CDATE$	the creation date of the currently open output file, in numeric file-date form.
FS OUTPUT$	if nonzero, suppresses output to the EW'd file. Output commands (P, etc) are errors.
FS PAGENUM$	the number of formfeeds read (with non-uparrow Y and A commands) from the input file since it was opened.
FS PROMPT$	the ASCII value of the prompt character (initially 38 for "&"). TECO will prompt on printing terminals only, whenever it is about to read from the terminal, and FS PROMPT$ is not 0.
FS PUSHPT$	(write-only) pushes its argument on the "ring buffer of the pointer", but only if the argument differs from the value already at the top of the ring buffer. See the ^V command. Note that TECO's top level loop does Q..IFS PUSHPT$ each time it is about to read in a command string.
FS QP PTR$	the q-register pdl pointer (0 if nothing has been pushed, 1 if one q-reg has been pushed, etc.) <n> FS QP PTR$ sets the pointer to <n> if <n> > 0; adds <n> to it if <n> is negative. It is illegal to increase the pointer value with this command.
FS QP SLOT$	<n> FS QP SLOT$ reads q-reg-pdl slot <n>. <m>,<n>FS QPSLOT$ sets it to <m>. The first slot is numbered 0. If <n> is negative, it is treated as FS QPPTR$+<n>. Thus, -1FS QP SLOT$ is the last slot pushed.
FS QP UNWIND$	(write only!) like FS QP PTR$ but pops slots back into the q-reg's they were pushed from instead of simply decrementing the pdl poinetr.

This unwinding is automatically done when an error is caught by an errset, and at the end of each command string, and by ^\ and FS ^REXIT$.
If Q..N is popped by this command, it is macroed first (see ..N).
If <n> is negative, <n>FS QP UNWIND$ pops -<n> slots.

FS QUIT$	^G-quit works by setting this flag negative. Whenever quitting is possible (see FS NOQUIT$) and FS QUIT$ is negative, quitting will occur (and FS QUIT$ will be zeroed automatically). When TECO's quitting is inhibited, the user can test this flag explicitly to do his own special quitting.
<n>FS Q VECTOR$	(a pseudoflag) returns a newly cons'ed up q-register vector, <n> characters long. <n> should normally be a multiple of 5. The contents are initialized to zeros, and the pointer is initially at the beginning of the qvector.
FS RANDOM$	reads or sets the ^Z-cmd random # generator's seed.
FS REAL ADDRESS$	returns the value of BEG, the character address of the beginning of the current buffer. Useful for communicating with other programs that need to be given addresses of data in their commands. Also useful for executing the buffer as PDP10 code (do FS REALAD$/5U0 M0; the code should be position-independent, expect its address in accumulator 6, and start with SKIP (skip never) instruction).
FS REFRESH$	if nonzero, is macroed whenever ^R is about to clear the whole screen (because the TTY was taken away from TECO temporarily). It is subject to the same conventions as FS ^R DISPLAY$. If FS REFRESH$ is zero but FS ^R DISPLAY$ is not, the latter will be macroed instead.
FS REREAD$	(usually -1) if nonnegative, FS REREAD$ is the 9-bit TV code for a character to be re-read. Putting 65 into FS REREAD$ will cause the next "FI" command to return 65 (and set FS REREAD$ back to -1).
FS RGETTY$	(read-only) 0 if printing console, otherwise equal to the tctyp word of the terminal. However, it is better to decode the FS %TOFCI$, etc., flags than to decode FS RGETTY$ when trying to determine what kind of display the terminal is, and what functions it can perform.
FS RUB CRLF$	if nonzero causes the initial definitions of ^D, rubout and control-rubout to delete both characters of a CRLF at one blow, as if it were a single character.
FS RUNTIME$	(read-only) TECO's runtime in milliseconds.

FS SAIL$	if nonzero, the terminal is assumed to be able to print non-formatting control chars as 1-space graphics. TECO outputs them as they are instead of outputting an ^ and a non-ctl char. Terminal initialization zeros this flag if the terminal's TOSA1 bit is 0 (this bit is set by :TCTYP SAIL).
FS S ERROR$	if 0, as it is initally, a failing serach within an iteration or a ^P-sort key is not an error - it simply fails to move the pointer. If not 0, such searches cause sfl errors like all other searches.
FS S STRING$	is the default search string (which S$ will use), as a string pointer. Do G(F S STRING$) to insert it in the buffer. This flag is most useful for pushing and popping.
FS STEP$	(normally 0) if nonzero, every CR in the program displays the buffer and waits for input before proceding. See ^M for details.
FS S VALUE$	the value stored by the last search command (0 if search failed; else negative, and minus the number of the search alternative which was actually found).
FS TOP LINE$	is, on display terminals, the number of the first line on the screen that TECO should use. Normally 0, so TECO starts output at the top of the screen.
FS TRACE$	(read only) nonzero iff TECO is in trace mode. See "?".
FS TRUNCATE$	says what to do with long lines of type-out. Negative => truncate them. Positive or zero => continue them to the next line. Entering ^R sets this flag to 0.
FS TTMODE$	(initially 0) non-zero tells TECO that normal buffer display should display on printing terminals. (if there is a user buffer display macro, this flag has no effect unless the macro checks it)
FS TTY INIT$	(no argument or value) causes TECO to reexamine the system's terminal description and reset various flags, and the cursor in ..A, appropriately, this is done automatically when TECO is started. In detail, ..A is set to "-!-" on printing terminals, and to "/\" on displays (but "^A^B" on imlacs). FS RGETTY$ is set up to be 0 on a printing terminal, nonzero otherwise. FS VERBOSE$ is set equal to FS RGETTY$. FS TTYOPT$ is read in from the system. FS ^H PRINT$ is zeroed unless the terminal can backspace and overprint; FS ^M PRINT$ is zeroed unless the terminal can overprint. FS SAIL$ is set nonzero if the %TOSA1 bit is set in TTYOPT (this is the bit ":TCTYP SAIL" sets). FS WIDTH$ and FS HEIGHT$ are read in from the system.

	FS ECHOLINES$ is set to 0 on printing terminals; 1/6 of the screen size on displays.
FS TTYOPT$	(read-only) the TTYOPT word for the terminal. However, if you think you want to use this flag, see FS %TOFCI$, etc., first.
FS TYO HPOS$	(read-only) while typeout is in progress (FS TYPEOUT$ nonnegative), holds the current typeout horizontal position, in which the next typed character will appear.
FS TYPEOUT$	is -1 if typeout has not been happening recently, so typeout starting now would appear at the top of the window. FS TYPEOUT$ is not -1 when typeout was the last thing to happen and any more typeout will appear after the previous typeout. :FT types at the top of the window by putting -1 in FS TYPEOUT$ before typing.
FS UNAME$	(read only) returns the UNAME of the job TECO is running in, as a numeric SIXBIT word, which can be converted into text by the F6 command. See also FS XUNAME$ and FS MSNAME$, one of which might be better for your purpose.
FS UPTIME$	(read only) returns time system has been up, in 30'ths.
FS UREAD$	(read-only) -1 iff an input file is open, else 0. Once an input file is opened, it remains open until "EC", "^ Y", "^ A", "EE" or "EX" is done.
FS UWRITE$	(read-only) -1 if an output file is open, else 0.
FS V B$	is the distance between the real beginning of the buffer and the virtual beginning. See FS BOUNDARIES$, but unlike that flag, FS V B$ can be pushed and popped.
FS VERBOSE$	if not 0, TECO will print the long error message of its own accord when an error occurs. Otherwise it will print only the 3-char code and the long message must be requested by typing ^X. Initially 0 except on displays.
FS VERSION$	(read-only) the current TECO version number
FS V Z$	is the distance between the virtual end of the buffer and the real end - the number of characters past the virtual end. See FS BOUNDARIES$ for more info, but note that FS V Z$ can be pushed and popped.
FS WIDTH$	width of terminal's screen or paper, in characters.
FS WINDOW$	the number of the first character in the current display window, relative to the virtual beginning of the buffer (that is, FS WINDOW$+B is the number of that charcter). FS WINDOW$+BJ (FS HEIGHT$-(FS ECHOLINES$)/2)L will put the pointer in the middle of the window (usually). Setting FS WINDOW$ will make TECO try to use the window specified. However, if the

constraints of FS %TOP$ and FS %BOTTOM$ are not
met, TECO will choose another window rather than
use the specified one.

FS WORD$

gets or sets words in the current buffer. This
flag makes it possible for TECO programs to
edit binary data bases.
<n>FS WORD$ returns the contents of the word
containing character <n>; <val>,<n>FS WORD$
sets that same word.
When handling binary data, it is unwise to
insert or delete characters other than in units
of five, on word boundaries. The way to delete
a word is to delete its five characters; insert
a word with 5,0I (it will contain either 0 or 1).
To read in a file of binary data, FY should be
used, since Y might pay special attention to the
characters in the file. Copying out of another
buffer with G works, provided the transfer starts
and ends on word boundaries (blt is used).
For writing out binary data, use "HP" rather than
"PW" - "PW" may add a "^L". "EF" is OK for closing the
file - it will add no padding if it is done at a
word boundary.

FS XJNAME$

(read only) returns the XJNAME of the job TECO is
running in, as a numeric SIXBIT word, which can be
converted into text by the F6 command. The XJNAME is
"what the JNAME was supposed to be", so if you want
your init file to do different things according
to how TECO was invoked, you should use the XJNAME
rather than the JNAME.

FS X PROMPT$

On printing terminals, all commands that type out
first print and zero FS X PROMPT$ if nonzero (using
FS ECHO OUT$). Do not try to use it on displays.

FS XUNAME$

(read only) returns the XUNAME of the job TECO
is running in, as a numeric SIXBIT word, which
can be converted into text by the F6 command.
The XUNAME is "who the user really is".
For example, it is what TECO and other programs
use to decide whose init file to use.

FS Y DISABLE$

controls treatment of Y command.
0 => Y is legal.
1 => Y is illegal (gives "DCD" error).
-1 => Y is always treated as ^ Y.

FS Z$

(read only) the number of characters in the buffer.
Will differ from value of Z command when virtual
buffer boundaries don't include the whole buffer.
This is Z-BEG (see "buffer block").

note: in the names of the following flags,
"^" represents uparrow, not a control character.
Control characters cannot be part of FS flag names.

FS ^H PRINT$

controls how ^H is typed out.
Negative => actually backspace (and overprint),

otherwise type the ^H as uparrow-H.
FS TTYINIT$ and $G cause this flag to be zeroed
if the terminal cannot handle overprinting.

FS ^I DISABLE$	controls the treatment of the tab character as a TECO command. 0 => ^I is self-inserting. 1 => ^I is illegal (gives "DCD" error). -1 => ^I is a no-op.
FS ^L INSERT$	(initially 0) if 0, formfeeds in files that terminate Y commands' reading, are thrown away, and the P and PW commands output a formfeed after the buffer. If FS ^LINSERT$ is nonzero, formfeeds read from files always go in the buffer, and P and PW never output anything except what is in the buffer. Either way, a Y and a P will write out what it reads in.
FS ^M PRINT$	says when a stray CR or LF should be typed out as one, as opposed to being printed as "^M" or "^J". Possible values and initialization like FS ^H PRINT$
FS ^P CASE$	if nonzero, ^P sort ignores case (lowercase letters sort like the corresponding uppercase).
FS ^R ARG$	is the explicit numeric argument for the next ^R mode command, or 0 (not 1!) if there was none.
FS ^R ARGP$	contains two bits describing this ^R-command's argument: bit 1.1 (1) is set if any argument was specified (either numerically or with ^U). bit 1.2 (2) is set if a numeric argument was specified. If this bit is 0, the contents of FS ^R ARG$ are ignored, and 1 is used instead.
FS ^R CCOL$	the comment column, for ^R's comment mode.
FS ^R CMACRO$	<n>FS ^RCMAC$ gets the ^R-mode definition of the character whose ASCII code is <n>. <m>,<n>FS ^RCMAC$ sets it to <m>. <n> should be an ASCII code; it will be converted to 9-bit TV code which is what is actually used to index the ^R-mode dispatch table. If you wish to supply a 9-bit code yourself, use "^ FS ^RCMAC$" which skips the conversion. Also, these definitions may now be referred to as q-regs in all the q-reg commands - see "Q". The definition is either a built-in command or a user macro. In the former case, the definition is a positive number of internal significance only. However, built-in definitions may be copied from one character to another using FS ^RCMAC$. For a character to be a user macro, its definition must be one of the funny negative numbers which are really string pointers. They can be obtained from strings by applying the "Q" command to a q-reg that contains text. For example, to make the

definition of the character " " be the string which is at the moment in q-reg A, do "QA,^^ FS ^RCMAC$W". To copy the definition of "A" into the definition of rubout, do "^^AFS ^RCMAC$,127FS ^RCMAC$W". This will make rubout self inserting (unless "A" had been redefined previously).

FS ^R DISPLAY$ if nonzero is macroed every time ^R is about to do nontrivial redisplay (anything except just moving the cursor).
If the evaluation alters the needed redisplay (either by returning 0 or 2 values to ^R, or by doing some of the redisplay with ^V) then ^R will take note. If a FS ^R DISPLAY$ returns no values, it will force a full redisplay, thus effectively disabling ^R's short-cuts, so beware.
If FS REFRESH$ is nonzero, then it will be used instead of FS ^R DISPLAY$, at those times when the whole screen is being cleared because the TTY was taken away and returned.

FS ^R EXIT$ (write-only) exits from the innermost ^R invocation. Pops q-regs pushed within that ^R level, and ends iterations started within it.

FS ^R ECHO$ 1 => characters read in by ^R should not be echoed.
0 (the default) => they should be echoed only on printing terminals.
-1 => they should be echoed on all terminals.
Note that this "echoing" is explicit typeout by TECO. System echoing is always off in ^R mode, currently. Also, rubout is not echoed on printing terminals. However, FS ^R RUBOUT$ on a printing terminal when FS ^R ECHO$ is <= 0, types out the char being deleted.

FS ^R ENTER$ is macroed (if nonzero) whenever ^R is entered at any level of recursion.

FS ^R EXPT$ is the ^U-count for the next ^R-mode command.

FS ^R HPOS$ the current horizontal position of the cursor in ^R mode. Not updated when the pointer moves, unless ^R gets control back or ^ V is done.

**FS ^R INIT$** <ch>FS ^R INIT$ returns the initial definition of the character whose ASCII code is <ch>; in other words, <ch>FS ^R INIT$ always returns what <ch>FS ^R CMACRO$ initially returns.
The uparrow modifier works for FS ^R INIT$ just as it deos for FS ^R CMACRO$; it says that the arg is a 9-bit code rather than ASCII.

FS ^R INSERT$ the internal ^R-mode insert routine's user interface. It takes one argument - the ASCII code for the character to be inserted. This command itself take care of notifying ^R of the change that is made, so when returning to ^R this change should not be mentioned in the returned values (so if this is the

	only change made, return 1 value). This command is very sensitive; if the buffer or even "." has changed since the last time ^R was in progress or an ^ V was done, it may not work. Its intended use is in macros which, after thinking, decide that they wish only to insert 1 or 2 characters (such as a space-macro which might continue the line but usually inserts a space).
FS ^R LAST$	holds the most recent character read by any ^R invocation (quite likely the one being processed right now). Commands that wish to set up arguments for following commands should zero FS ^R LAST$, which tells ^R not to flush the argument when the command is finished.
FS ^R LEAVE$	is macroed (if not zero) whenever ^R returns normally (including FS ^R EXIT$ but not throws that go out past the ^R).
FS ^R MARK$	holds the mark set by ^T in ^R mode, or -1 if there is no mark.
FS ^R MCNT$	the counter used by ^R to decide when to call the secretary macro. It starts at FS ^R MDLY$ and counts down.
FS ^R MDLY$	sets the number of characters that should be read by ^R mode before it invokes the secretary macro kept in q-register ..F . Characters read by user macros called from ^R are not counted.
FS ^R MODE$	(read-only) non-zero while in ^R-mode.
FS ^R MORE$	if nonzero, the --MORE-- is disabled in ^R mode.
FS ^R NORMAL$	all "self-inserting" characters in ^R mode are really initially defined to go indirect through this word, if it is nonzero. If it is zero, as it is initially, the default definition is used for such characters.
FS ^R PREVIOUS$	holds the previous (second most recent) command read by ^R's command loop, not counting argument setting commands.
FS ^R REPLACE$	if nonzero puts ^R in "replace mode", in which normal characters replace a character instead of simply inserting themselves. Thus, the character A would do DIA$ instead of just IA$. There are exceptions, though; a ^H, ^J, ^L or ^M will not be deleted, and a tab will be deleted only if it is taking up just one space. Also, characters with the meta bit set will still insert. Replace mode actually affects only the default definition of "normal" characters. Characters which have been redefined are not affected, and if FS ^R NORMAL$ is nonzero no characters are affected (unless the user's definitions check this flag). Making FS ^R REPLACE$ positive has the additional

	effect of forcing all meta-non-control characters to be come normal, suppressing their definitions.
FS ^R RUBOUT$	the internal ^R rubout routine's user interface. Takes 1 arg - the number of characters to rub out. This command is very sensitive and may fail to work if the buffer or "." has been changed since the last time ^R was in control, or an ^ V, FM, FS ^R RUB$ or FS ~R INSERT$ was done. Its intended use is for macros which, after thinking, decide to do nothing but rub out one character and return; it gives extra efficiency but only when rubbing out at the end of the line.
FS ^R SCAN$	if nonzero causes ^R commands, when using a printing terminal, to try to imitate a printing terminal line editor by printing the characters they insert/delete/move over. FS ^R ECHO$ should be 1, to avoid double-echo.
FS ^R SUPPRESS$	(initially -1) nonnegative => builtin ^R-mode commands are suppressed, except for rubout, and user-defined commands are suppressed unless their definitions begin with "W". Suppressed command characters become self-inserting. The char whose 9-bit value is in FS ^R SUPPRESS$ is the unquoting char. It reenables suppressed commands temporarily by setting FS ^R UNSUPPR$ to -1. If FS ^RSUPPRESS$ is > 511, there is no unquote char.
FS ^R THROW$	returns control to the innermost invocation of ^R. This is different from FS ^R EXIT$, which returns control FROM that invocation.
FS ^R UNSUPP$	(initially 0) actually, builtin commands are suppressed only if this flag and FS ^RSUPRESS$ are nonnegative. However, this flag is zeroed after each command except ^U and ^V. Thus, setting this flag to -1 allows one builtin comand.
FS ^R VPOS$	the ^R-mode cursor's vertical position.
FS _ DISABLE$	controls treatment of the "_" command. If 0 (the default), "_" is "search-and-yank" as it originally was. If 1, "_" is illegal (gives "disabled command" error). If -1, "_" is treated like "-" (good on memowrecks).

FT	types its string argument.
:FT	similar, but always goes to top of screen first (actually, to the line specified by FS TOP LINE$).
^ FT	similar to FT, but types its argument in the echo area rather than the display area. Characters are typed normally, in ITS ASCII mode, rather than as they would echo, so to do a CRLF you need a CR and a LF.
^:FT	like ^ FT, but types the argument only if no input is available (FS LISTEN$ would return 0). If input is available, the argument is skipped over and ignored.

FU	a list manipulating command whose main use
	is in <arg>FUL, which moves up <arg> levels of parentheses.
	<arg>FU where <arg> is positive returns a pair of args for
	the next command, specifying the range of the buffer from .
	Moving rightward to the first place <arg> levels up.
	If <arg> is negative, it moves left -<arg> levels up.

FV	displays its string argument.
:FV	types its string argument, then clears whatever is
	left of the screen.

FW	similar to FL but hacks words instead.
	A word is defined as a sequence of non-delimiters.
	Initially, the non-delimiters are just the squoze
	characters but the user can change that - see q-reg ..D.
	This command returns a pair of args for the next one.
	Also, FW sets FS INSLEN$ equal to the length of the
	last word moved over.
	Main uses: FWL moves right one word,
	-FWL moves left one, FWK deletes one word to the right,
	FWFXA deletes and puts in q-reg A,
	FWFC converts one word to lower case.

:FW	similar to FW but stops before crossing the word instead
	of after. Thus, :1FWL moves up to before the next
	non-delimiter. :2FWL is the same as 2FWL-FWL.
	:FW sets FS INSLEN$ to the length of the last inter-word
	gap crossed.

^ FW	like FW, but finds LISP atoms rather than words.
	Understands slashes and vertical bars but not comments.

FX	like X and K combined. "3FXA" = "3XA 3K".
	The uparrow flag causes appending to the q-reg, as for X.

FY	insert all that remains of the current open input file
	before point. Error if no file is open. The input data are
	unaltered; no attempt is made to remove padding or ^L's.
	If the transfer is on a word boundary in the file and in
	the buffer, word operations will be used, so this command
	is suitable for use with binary data.
	The input file is not closed - use EC for that.

<n>FY	like FY, but inserts at most <n> characters, or until EOF,
	whichever comes first. Note that <n> characters of
	space are always needed, even if the file is not really that
	long; thus, 1000000FY to read in the whole file will
	not work. The input file is not closed.

F[<flag>$
	pushes the value of FS<flag>$ on the q-reg pdl,
	so that it will be restored on unwinding.
<arg>F[<flag>$
	pushes the flag and sets it to <arg>
<ch>F[^R CMACRO$
	pushes the definition of character number <ch>.
<arg>,<ch>F[^R CMACRO$
	pushes the definition of character number <ch> and sets it.

F_	this command has the same meaning that _ normally has;

namely, search for a string arg and keep yanking till end of file. However, this command works regardless of the setting of FS _DISABLE$

F]<flag>$
pops from the q-reg pdl into FS <flag>$.

<ch>F] ^R CMACRO$
pops from the q-reg pdl into the definition of character number <ch>, and returns the old definition.

F~ compares strings, ignoring case difference. It is just like F= except that both strings are converted to upper case as they are compared.

```
G<q>     insert in buffer to left of pointer the text in q-reg <q>.
         If q-reg specified contains # rather than text, decimal
         representation thereof will be inserted.  If the q-reg
         contains a buffer the gap in the buffer may have to be
         moved before the G can be done.
         FS INSLEN$ is set to the length of the inserted text.
<m>,<n>G<q>
         insert only a part of the text in the q-reg;  specifically,
         the range from <m> to <n>-1 inclusive.  This feature works
         only for q-regs containing text;  if a q-reg holds a number
         the whole q-reg will be inserted despite the args.

H        equvalent to B,Z;  i.e., specifies whole buffer
         (or all within the virtual boundaries if they're in use)
         to commands taking two args for character positions such as
         K, T, or V.

I        if no arg, insert following chars
         up to altmode in buffer to right of
         pointer.  If preceded by an uparrow ("^")
         following char is delimiter to end
         text string instead of altmode,
         e.g., ^I/text/.
         The length of the inserted string is kept in FS INSLEN$
         (see the "FK" and "^F" commands).
:I<q>    takes a q-register name immediately after
         the I and inserts into that q-reg,
         replacing the previous contents.
         Uparrow works as with the
         normal I command, with the delimiter
         following the q-reg name.
         FS INSLEN$ is not set by :I.
         Self-inserting chars will not take
         the colon modifier.
<n>I     inserts the character with ASCII code <n>.
<n>:I<q>
         puts the character in a string in q-reg <q>.
<m>,<n>I
         inserts <m> copies of the character with ASCII code <n>.
<m>,<n>:I<q>
         puts <m> copies of the character in a string in q-reg <q>.
J        position pointer after argth char
         in buffer.  If no arg, arg=B (usually 0).

:J       is to J as :S is to S.

K        if no arg or one arg, kill chars
         from pointer to argth line feed
         following.  (no arg, arg=1;  negative
         arg, back up over 1-arg line feeds,
         space over last line feed found, and
         kill from there to pointer.  A colon
         after the arg will move back over
         carriage return+linefeed before deleting.
         If no carriage return exists,
         TECO will only move back one character.
         There is an implicit line-feed at the end of
         the buffer.  (note: this is the action of colon
         for all commands which take
```

```
        one or two args like K.)
<m>,<n>K
        kills characters <m> through <n>-1.
        The pointer is moved to <m>.

L:
<arg>L  move to beginning of <arg>th line after
        pointer (0L is beginning of current
        line.).  Colon acts as in the K command.
        Note that :L moves to end of current line
        0:L moves to end of previous line
        and -:L moves to end of line before
        previous line
        <m>,<n>L is the same as <m>+<n>-.J

M<q>    calls the function in q-reg <q>.  If <q> contains
        a string or buffer, its contents are "macroed" -
        that is, treated as TECO commands.
        If <q> contains a number, that number should be the
        initial definition of some ^R-mode character;
        that "built-in" function will be called.
        Built-in finctions take 1 arg;  user macros,
        0, 1 or 2, which it may access using "^X" and "^Y"
        (or, more winningly, with the F^X and F^Y
        commands).  The macro may read string arguments using
        the ^]^X construction;  such arguments should be
        supplied after the M command.
        Note that if you macro a buffer, you may screw yourself
        if while that buffer is executing you either
        modify its contents or throw away all pointers to it.
        If the q-register specified in the M command is a
        ^R-mode character definition (as in M.^RX), the code
        for that character is put in Q..0 in case the definition
        looks at it there.

^ M     is like M except that if no second argument is
        supplied the default value is 1, not 0.
        As with plain M, F^X in the called macro will
        not return the value;  the default is available
        only through ^Y.  ^ M is useful because its
        convention is the same as ^R's.

:M      does the same thing as M, except that
        it is a JRST instead of a PUSHJ.  Ie.,
        when a :M-called macro terminates, it
        acts as if the macro calling it
        had terminated.  Doesn't work if q-reg holds
        a built-in function (that is, a number).

N       does search (see S) but if end of
        buffer is reached does P and
        continues search.

O<tag>$ sends command execution to char after
        the occurrence of <tag> as a label ("!<tag>!")
        ( OX$ goes to !X!).  Case is not significant in tags,
        so OFOO$ and Ofoo$ will both find both !FOO! and !foo!.
        Label must be on same iteration level as O command,
        i.e., no unmatched < or > between O and !.
        The label must also be within the same macro as
```

the O; in other words, non-local gotos are not
implemented.
If the tag is not found, an "UGT" error occurs at
the end of the O command. For convenience's sake,
:O is just like O but simply returns if the tag
is not found.
The ^ modifier allows the tag to be abbreviated.
OFOO$ will not find !FOOBAR!, but ^ OFOO$ will find it.
This is for the sake of those using O to do command
dispatching.
TECO has a cache containing the locations of
several recent O commands and where they jumped to.
If an O command is in that table, searching is
unnecessary. This increases efficiency. However, if
there is a ^] call in the arg to O, it might be
intended to jump to different places each time, so
TECO refuses to cache such jumps to force itself to
search each time. Also, jumps in buffers and in
top level command strings cnnot be cached, since the
data in the buffer (including the argument of the O
command) might change at any time; if TECO then did
not read the argument and search, it might jump to the
wrong place.

P output contents of buffer to device
open for writing, followed by form feed (^L)
if FS ^LINSERT$ is 0; clear buffer and read
into buffer from file open for
reading until next form feed or end
of file. Takes one arg, meaning
do it arg times, or two args, meaning
output specified portion of buffer
(args as in K command) without
following form feed, and without
clearing buffer or doing input.
Note: if next command char
is W, this is not P command but

PW which outputs like P but does not
clear buffer or do input. Takes
arg, meaning do it arg times.

Q<q> returns the value in q-reg <q>, as a number. If <q>
is holding a number, that number is the value.
If <q> "holds text", then it really contains a pointer
to a string or buffer, and Q<q> wil return the pointer,
which if put in another q-reg (using "U") that q-reg
will "hold the same text" as <q>.

A q-reg name is either an alphanumeric char preceded
by 0, 1 or 2 periods,
a "variable name" of the form $<name>$,
a subscripting expression such as :Q(<idx>),
a * (for certain commands),
an expression in parentheses (for certain commands),
or up to 3 periods followed by a "^R" or "^^" and any
ASCII character.

Periods plus alphanumeric character q-reg names refer
to TECO's q-registers, which are what serve as variables

for TECO programs. Each distinct such name names a
distinct variable. Names with two periods are
reserved for special system meanings; those that
are now assigned are documented starting at "..A".

While names like "A" or ".8" are fine for local variables
in programs, for global parameters mnemonic names are
necessary. Variables with long names are accessible through
the $<name>$ construct. Variable names may be abbreviated,
and extra spaces and tabs may go at the beginning, the end,
or next to any space or tab. Also, case is not
significant inside variable names. Thus, a variable named
"Foo" could be accessed with QFOO, Q$ foo $ or (if there
is no FO or FOX, etc.) with Q$ Fo$. Because of this
latitude, variables are not created if they are referenced
and do not exist; instead, they must be entered explicitly
in the symbol table by the user. This is easy to do,
because the entire symbol table data structure is
user-accessible. See the FO command for a sample macro
for creating variables.

The elements of a q-vector may be accessed as q-registers
in their own right. If q-reg A contains a q-vector, then
the "q-register" :A(0) is the first element of it, and
:A(1) is the second, etc. Indexing starts at zero for the
first element of the q-vector, but only those elements
within the virtual boundaries of the q-vector may be accessed.

A star ("*") may be used only with commands like] and X
that wish only to store in a q-register; it causes such
commands to return their data as a value instead. Thus,
:I*FOO$ returns a string containing FOO.

Expressions in parentheses can be used only with
commands that wish only to examine the contents of a
q-register; the value of the expression is used as
the contents to be examined. Commands that allow
this option include F^A, F^E, F=, FQ, F~, G and M.
Thus, G(Q0) is equivalent to plain G0.

Q-reg names containing ^R or ^^ refer to the definitions
of ^R-mode command characters. When "^R" is used,
the ^R-mode definition of the specified ASCII
character is referred to; when "^^" is used, the
^R mode definition of the specified charcter xor'ed
with 100 (octal) is meant. The periods specify the
control and meta bits since ^R-mode definitions
belong to 9-bit characters but only 7-bit characters
can be inside TECO command strings; one period sets the
control bit; 2, the meta bit; 3, both control and meta.
If the char is obtained from a ^]^V, then all 9 bits
may be obtained from that source; the periods xor into
the number in the q-reg.
For example, "Q^RA" refers to the definition of "A",
and "Q.^RA" refers to that of control-A, as does
"Q^R^]^VX" when QX holds 301 (octal).
"Q^R^A" refers to the definition of downarrow,
one of the new TV printing characters, as does "Q^^A".
"Q^^J" refers to the definition of linefeed, whereas
"Q.^RJ" refers to the definition of control-J, which

```
         can be typed in only on a TV (and which is usually
         defined to execute the definition of linefeed).

R        move pointer left arg chars (no arg, same as arg=1).

:R       is to R as :S is to S; as :C is to C.
<m>,<n>r
         does "<m>+<n>-.J".  This is for FLR to work.

S        search.  Takes following text string
         and looks for it in the buffer,
         starting from the pointer.
         (if the string arg is null, the last nonnull arg
         to any search command is used)
         if it finds it, it positions the
         pointer after the string.  If it
         does not find it, it does not
         move the pointer but generates an
         error message unless the search is
         inside an iteration (see <.  See also FS S ERROR$
         which may be used to disable this "feature").
         If the search is inside an iteration,
         the value as if produced by :s (read on)
         will be saved whether or not the
         colon is used, for use by the ;
         command.  The effect of iterations on searches is
         cancelled by errsets, so what matters is whether
         the search is more closely contained in an iteration
         or in an errset.
         A positive arg to the search means do it arg times,
         i.e., find the argth appearance of the string;
         a negative arg means search the buffer
         backwards from the pointer and
         position the pointer to the left of
         the string if successful.  If the S is
         preceded by "^", the char after
         the S is used to delimit the text
         string instead of altmode.  In this case, a null
         arg causes a search for the null string, instead of
         a search for the last string searched for.
         (this for the sake of macro-writers using ^])
         if the s is preceded by :, val=-1 if the
         search is successful and val=0 if
         not--there is no error condition.
         Note also N and _ commands.
         There are some special characters
         used inside search strings which
         do not have their usual meanings:
         ^B matches any delimiter char (normally this means it
         matches any non-squoze char, but see q-reg ..D).
         ^N matches any char other than
         the char following it in the search
         string (i.e., "not").  ^N^B matches non-delimiters, and
         ^N^X matches nothing.  ^N^Q^B matches all but ^B, etc.
         ^O divides the string into substrings
         searched for simultaneously.  Any one
         of these strings, if found, satisfies
         the search.  If :s is used,
         finding the nth substring sets val=-n.
         ^Q quotes the following char, i.e.,
        deprives it of special significance.
        The delimiter and rubout cannot
        be quoted.
        ^X matches any char.
        Note that SFOO^O$ will move the pointer
        over the next three characters
        if and only if they are FOO.  It will always
        succeed.  However, -2-(:sfoo^O$) does the
        right things.

T       type:  takes one or two args like K
        and types out the selected chars.
^ T     types in the echo area.

U:
<n>U<q>
        puts the number <n> in q register <q>.
        Returns no value.
<m>,<n>U<q>
        puts <n> in <q>, and returns <m>.
        Thus "<m>,<n> U<q1> U<q2>" does "<n>U<q1> <m>U<q2>".

V       takes arg like K and displays chars, representing
        the cursor by "/\" (or whatever is in ..A).
        When, after being proceded from a --MORE--,
        a new screenfull is started, the place it began
        is remembered in FS WINDOW$ so that an attempt
        to display the buffer will try to start at the same place.
        This may make redisplay unnecessary if you search for
        something that appears on the screen.
        Nothing is typed on printing terminals.

^ V     (uparrow-v) performs standard buffer display.
        That is, "^ V" always does what automatic buffer
        display does as a default (when ..B holds 0).
        When in ^R mode, ^ V does a ^R-style display.
        Note that ^ V will display on any type of terminal,
        although TECO does not normally display automatically
        on printing terminals.
        In ^R mode, ^ V treats its arguments as ^R does
        (as hints on how to redisplay).  When not inside a ^R,
        ^ V ignores its arguments.

        If V is followed by W it becomes
VW      which does V, then waits for terminal
        input of one char whose 7-bit ASCII
        value is returned as val.

W       flushes current value except when
        part of VW or PW.

X       takes one or two args like K and
        enters selected chars as text into
        q-register named by next char in
        command string.  Can be retrieved
        by G command and ^] substitution, q.v.
        ^ X acts like X but
        appends text to q-reg rather than
        replacing q-reg contents.  If q-reg
        does not already contain text this
        works like ordinary X.
        See also :I.

Y       kills the buffer, then inserts one page from the
        current input file (until first formfeed or eof).
        Point is left at the beginning of the buffer.
        If reading is terminated by a ^L, the ^L will go in
        the buffer iff FS ^LINSERT$ is nonzero.
        (FS ^LINSERT$ is initially 0)
        The input file is not closed, even if eof is reached.
        To close the input file, use EC.  However, EE does
        close the input file.  Closing the input file is not
        necessary but will lighten the drain on system resources.
        Trailing ^C's or ^@'s just before eof are considered
        padding and are flushed.  To do input without having
        any padding characters removed, use FY.
        The virtual buffer boundaries are understood.
        If no file is open, the buffer is left empty.
        Because Y is an easy command to be screwed by, and
        isn't really necessary since the A command exists,
        there is a way to disable it.  See FS YDISABLE$.

^ Y     (uparrow Y) yanks in all the rest of the file.
        ^L's within the file go in the buffer.
        A ^L at the end of the file will go in the buffer
        iff FS ^LINSERT$ is nonzero.  Trailing ^C's and ^@'s
        are conidered padding, and flushed.
        The input file is closed automatically.

Z       val=number of chars in buffer (more generally, the
        character number of the virtual end of the buffer,
        if virtual buffer boundaries are in use).

[<q>    push text or number in q-reg <q> onto the q-register pdl.
        There is only one q-reg pdl, not one per q-reg.
        At various times (for example, the ^\ and F; commands,
        and after errors) TECO unwinds the q-reg pdl to a
        previous level by popping everything back to where it was
        pushed from.
        The [ command does not allow subscript expressions (such
        as "[:A(5)") because automatic unwinding would have no way
        to know how to pop the pushed value back where it came from.
        If you wish to push the value and don't mind that errors,
        etc. won't pop it back, do "[(Q:A(5))" or something similar.

<new>[<q>
        is equivalent to [<q> <new>U<q>.

\       Converts digits in the buffer to a number, or vice versa.
        If no arg, value is the number
        represented as optionally signed
        decimal or octal-with-point digits
        to right of pointer in buffer.
        (actually, the inpit radix comes from FS IBASE$
        or FS I.BASE$, as with numbers in commands).
        Moves pointer to right of number.
        If one arg, inserts printed representation of arg in
        buffer to right of pointer;  usually the number is
        "printed" in decimal, but the radix is controlled
        by q-reg ..E.
```

If two args, first specifies field size such that if
2nd is shorter than that many chars
leading blanks will be added.
"\" with 1 or 2 args sets FS INSLEN$ to the number of
characters inserted. See "FK" and "^F".

]	pop from q-register pdl into q-reg named by next char.
^	used with I, N, S, and _ commands (and others) to specify text delimiter other than altmode. Used with T, FT and = commands to specify typeout in the echo area. A few other commands also use the flag this command sets. The uparrow flag is handled generally like the colon flag. See the : command.
_	if FS _ DISABLE$ is 0, then _ is like S, except at end of buffer do Y command and continue search until end of file on input or until text string found. If FS _ DISABLE$ is 1, "_" is illegal. If FS _DISABLE$ is -1, "_" is the same as "-". Use "F_" in a macro to be sure of doing the search.
Rubout	deletes last char typed in, and types deleted char. Done during type-in, not during command execution. If executed (rather hard to do), same as _. Rubouts are typed out by TECO as ^? (rubout is ctl-?)

Lower case letters are interpreted like
upper case letters when they are commands.
Inside insert and search strings they are
treated as themselves.

Various special topics of interest are treated below

When TECO is started for the first time,
it initializes various data areas, prints its version number, and initializes several flags associated with the terminal (by executing FS TTY INIT$).
If TECO was started at 2 + the normal starting address, FS LISPT$ is set nonzero. Otherwise, it is set to 0.
In either case, TECO looks for a "TECO INIT" file (see below), executing it as a program if it is found.

When TECO is restarted,
it does not clobber the buffer, q-regs or open files.
It does, however, execute FS TTY INIT$ which resets some flags whose preferred setting depend on the type of terminal.
Then, it quits to top level and executes whatever is in q-reg ..L (unless it is 0).

Init files:
whenever TECO is started for the first time, it checks for a file .TECO. (INIT) on the directory of its initial XUNAME, or, if there is no such directory, for a file <xuname> .TECO. on (INIT);.
If either one is found, it is read in and executed.
The user is said to "have an init file" in that case.
If neither file exists, (INIT);* .TECO. is used instead. Its only function is to interpret DDT command lines as follows:

```
":TECO FOO BAR <cr>"      typed at DDT causes
        "ET FOO BAR $ EI ER$ Y$$" to be done by TECO --
        that is, TECO starts editing FOO BAR.
":TECO FOO <cr>" edits FOO >  .
        Because COM:.TECO. (INIT) sets FS FNAMSY$ temporarily.
":TECO <filename>$<TECO commands> <cr>"
        typed at DDT executes " ET <file> $ <commands> ".
":TECO FOO;<cr>"
        reads and executes FOO's init file (an error if
        he has none).
```

A user's own init file should interpreted the JCL by reading it in with the "FJ" command. It may have any command format it wishes except that it should always respond to "<foo>;" in the JCL by flushing the JCL (do "^ ^K^W:JCL<cr>$P^V") and loading and executing <foo>'s init file. See the default init file for how to do those things.

TECO's Data Structures (Strings, Buffers and Qvectors)

TECO has two different data structures for storing sequences of characters: strings, and buffers. They differ in what operations are allowed on them, and how efficient they are.

Strings have less overhead than buffers, but as a penalty they are not easily altered. Once a string has been created, its contents usually do not change; instead one might eventually discard the string and create a new one with updated contents. The sole exception is F^E, which makes it possible to alter characters in a string (but not to insert or delete). Commands which "put text in a q-register" all do so by consing up a string and putting a pointer to it in the q-register.

Buffers are designed to be convenient for insertion and deletion. Each buffer has its own pointer, and its own virtual buffer boundaries, which are always associated with it. The contents of a buffer can be accessed just like the contents of a string (in which case only the part between the virtual boundaries is visible), but it can also be "selected" and then accessed or altered in many other ways: insertion, deletion, searching, etc.
Each buffer has about 42 characters of overhead,
and the number of buffers is limited (about 40).
Initially, there is only one buffer in a TECO (pointers to which are initially found in q-registers ..O and ..Z), and new ones are only made when explicitly asked for with F[B BIND$, FS B CONS$ or FS B CREATE$.

Strings and buffers are normally represented in TECO by pointers. When a q-register "contains" a string, it actually contains a pointer to the string (see the sections on internal format for details). If q-register A contains a string, QA returns the pointer, which can be stored into q-register B; then QB and QA both point to the same string. A buffer is selected by putting a copy of a pointer to it into q-register ..O. TECO has a garbage collector, so that if all pointers to a buffer or string are eliminated, the storage it occupies will eventually be reclaimed. Most of the space occupied by a buffer can be reclaimed explicitly with the FS B KILL$ command; the buffer is becomes "dead", and even though pointers to it may still exist, any attempt to use them to select the buffer or examine its contents will be an error.

Vectors of objects can also be represented in TECO, with either buffers or qvectors. Buffers can be used to as vectors of numbers, while qvectors are used as vectors of arbitrary objects (numbers, or pointers to strings, buffers or qvectors). The difference is due to the fact that the garbage collector knows that the objects in a qvector might be pointers and therefore must be marked, while the objects in a buffer cannot be pointers and are ignored. The words in a buffer or q-vector can be accessed easily with subscripted q-register names; if QA contains a q-vector, then Q:A(0) is its first element. To access the elements in hairier ways, you can select the buffer or q-vector and the insert or delete, etc.

The buffer block, and what buffers contain (and the gap):

The current buffer is described by the 7-word "buffer block" which contains these variables:

```
        BEG     char addr of start of buffer,
        BEGv    char addr of lower buffer boundary,
        PT      char addr of pointer,
        GPT     char addr of start of gap,
        ZV      char addr of upper buffer boundary,
        Z       char addr of top of buffer,
        EXTRAC # chars in gap.
```

Note that all character addresses normally used in TECO have BEG subtracted from them; "B" returns BEGV-BEG; "Z", ZV-BEG; "FS Z$", Z-BEG; ".", PT-BEG;

"FS GAP LOCATION$", GPT-BEG. "FS GAP LENGTH$" gives EXTRAC. The actual value of BEG is available as "FS REAL ADDRESS$". GPT and EXTRAC describe the "gap", a block of unused space in the middle of the buffer. The real amount of space used by the buffer is Z-BEG+EXTRAC. BEGV, PT, Z and ZV are "virtual" addresses in that they take no account of the gap. To convert a virtual address to a real one, add EXTRAC to it if it is greater than or equal to GPT. Real address 0 refers to the first character in word 0; real address 5 refers to the first character in word 1, etc. It is OK for the superior to alter those variables or the contents of the buffer, if TECO is between commands or has returned because of ^K, FS EXIT$ or ^C; except that BEG should not be changed and the sum of Z and EXTRAC (the real address of the end of the buffer) should not be changed, unless appropriate relocation of other buffers and TECO variables is undertaken.

Strings - internal format:

A string containing <n> characters takes up <n>+4 consecutive characters in TECO. It need not start on a word boundary. The first four characters are the string header; the rest, the text of the string. The header starts with a rubout. The second character is <n>&177; the third, (<n>/200)&177; the fourth, <n>/40000 (numbers in octal).

Buffers - internal format:

A buffer consists of a buffer-string, which points to a buffer frame, which points to the buffer's text. The buffer-string is similar to a string, and exists in a string storage space, but begins with a "~" (ASCII 176) instead of a rubout. It has only three more characers; the second is <addr>&177; the third, (<addr>/200)&177; the fourth, <addr>/40000; <addr> being the address of the buffer frame. The buffer frame is a seven-word block whose purpose is to save the buffer block for buffers which are not selected. While a buffer is selected, the buffer frame contents may not be up to date. The first word of the frame contains a few flag bits in the left half. The sign bit will be set to indicate that the block is in use as a buffer frame. The 200000 bit is the GC mark bit and should be ignored. The 100000 bit, if set, indicates that the buffer is really a qvector. These bits are only in the buffer frame, not the buffer block (BEG).

Buffer and string pointers - internal format:

When a q-reg is said to hold a buffer or a string, it really contains a pointer to the buffer or string. The pointer is in fact a number, distinguished from other numbers by its value only! A range of the smallest negative numbers are considered to be pointers (this is why QAUB copies a string pointer from QA to QB without any special hair). They

are decoded by subtracting 400000000000 octal (the smallest negative integer) to get a character address. This may either be the exact address of a character in pure space (what :EJ loads into), or the relative address of a character in impure string space (what "X" allocates within. The char address of the start of impure string space is held in location QRBUF).
In either case, that character should be the rubout beginning a string or the "~" starting a buffer-string.
For example, 400000000000.+(FS :EJPAGE$*5*2000.)
is a string pointer to a string whose first character is at the very beginning of the last :EJ'd file. If the file has the proper format (see "strings" above), that number may be put in a q-reg and the string then executed with "M" or gotten by "G", etc. The file might contain a buffer-string except that causing it to point to a legitimate buffer frame would be difficult. Making it point to a counterfeit buffer frame inside the file would lose, since TECO tries to write in buffer frames.

How superiors can put text (and other things) into TECO:

A standard protocol for communication from a superior to TECO is hereby defined, which allows the superior to request space in the buffer for inserting text, or request that a file be loaded and a certain function be found. Macro packages may supply functions to handle the requests instead of TECO's default handler.

A superior can make a request whenever TECO has deliberately returned control (by a ^K valret, FS EXIT$ or a ^C) by restarting TECO at a special place: 7 plus the address of the "buffer block", which address can be found in accumulator 2 at such times.
Save the old PC before setting it, since you must restore the PC after the request is handled. The word after the starting location (8 plus the buffer block address) is used for an argument.

There are two types of requests. If you wish to supply text for TECO to edit, the argument should be the number of characters of text you need space for (it may be 0). In that case, TECO will return (with an FS EXIT$) after making at least that much space in the buffer, after which you should restore the PC at which TECO had stopped before you made the request. You can then insert the text in the buffer and restart TECO.

If you want TECO to read in a file, supply a negative argument and be prepared to supply TECO with JCL, when it asks with the standard .BREAK 12, describing what it should do. When TECO does a .BREAK 16, (FS EXIT$) you can assume it has done its work, and should restore the old PC. The formats for the JCL string are <filename><cr>, <filename>,<decimal number><cr>, and <filename>,<function name><cr>. A decimal number should be the address within TECO of the place to start editing. A function name can be anything that isn't a number, and its interpretation is not specified.

TECO macro packages can supply a handler for requests from the superior by putting it in FS SUPERIOR$. It will receive the argument describing the type of request as its numeric argument (^Y), and can read the JCL with FJ and do an FS EXIT$ when finished.
If FS SUPERIOR$ is zero, TECO's default actions will be taken
Note that TECO's default handling of a request to load a file is

to do nothing.

TECO's character sets:
(numbers in this section are in octal)

The most important TECO character sets are ASCII (7-bit) and the 9-bit TV set. The contents of all files, strings, and buffers, and thus all TECO commands, are in ASCII; 9-bit is used only for terminal input. Here is how TECO converts between character sets:

14-bit to 9-bit conversion when characters are read in:

When a character is actually read from the terminal, it is in a 14-bit character set which contains a basic 7-bit code, and the control, meta and top bits (also shift and shift-lock, which are ignored since they are already merged into the basic 7-bit character). TECO converts it to 9-bit as follows: if top is 0, and the 7-bit character is less than 40 and not bs, tab, LF, CR or altmode, then add control+100; then clear out top, shift and shift-lock.
Thus, TV uparrow comes in as top+013 and turns into 013;
TV control-K comes in as control+113 and stays control+113;
TV "VT" comes in as 013 and turns into control+113;
TV control-VT comes in as control+013 and becomes control+113;
non-TV control-K comes in as 013 and becomes control+113;
TV control-I comes in as control+111 and stays control+111;
TV "tab" comes in as 011 and stays 011;
TV control-tab comes in as control+011 and stays control+011;
non-TV "tab" or control-I comes in as 011 and stays 011.

9-bit to ASCII, when TECO wants to read an ASCII code:

input read in using "^ FI", or read by the ^R-mode comand dispatch, is used as 9-bit. However, when input is read by "FI", or by the "^T" command reader, or by TECO top level, it must be converted to ASCII as follows: meta is thrown away; if control is 0 then nothing changes; otherwise, control is cleared and the following actions performed on the 7-bit character that is left: rubout stays the same; characters less than 40 stay the same; characters more than 137 have 140 subtracted; other characters (40 through 137) have 100 complemented.
Thus, control+111 (TV control-I) becomes 011;
control+011 (TV control-tab) becomes 011;
and 011 (TV tab, or non-TV control-I) stays 011.
Similarly, TV uparrow, TV "VT", TV control-K and non-TV control-K all become 013.

ASCII to 9-bit in FS ^RCMACRO$ and FS ^R INIT$:

The ^R command dispatch table is indexed by 9-bit characters. For compatibility with the time that it was not, the commands FS ^R CMACRO$ and FS ^R INIT$, when not given the uparrow modifier, accept an ASCII argument, and try to have the effect of referring to the definition of that ASCII character - in fact, they convert the ASCII character to 9-bit and then index their tables. The conversion is as follows:

if the character is less than 40, and is not bs, tab, LF, CR or altmode, then add control+100.
Thus, 013 (^K) becomes control+113 (TV "VT" or control-K, not TV "uparrow"), which is just right. Tab, etc. have a harder time doing the right thing, since both 011 control+111 are plausible ways that the user could type what corresponds to ASCII 011. The solution chosen is to leave 011 ASCII the same in 9-bit, since the ^R-mode definition of control-111 is to use 011's definition.

The initial ^R-mode definitions of all 9-bit characters:

All characters whose bottom 7 bits form a lower case letter are defined to indirect through the corresponding upper case character. Their definitions are all 40,,RRINDR, where RRINDR is the indirect-definition routine, and 40 specifies the character 40 less.
Control-BS and Control-H indirect through BS, and similarly for Tab and LF. Control-CR and Control-Altmode (but not Control-M and Control-[) indirect through CR and Altmode.
An isomorphic indirection-pattern exists for meta characters.
All meta characters are self-inserting. A few (mentioned above) are self inserting because they go indirect through other meta characters.
All non-control non-meta characters, except for CR, altmode and rubout, are self-inserting. CR inserts CRLF; altmode leaves ^R-mode; rubout deletes backwards.
Of the rest, ^H, ^I and ^J are defined to insert themeselves straight away, while the rest are defined to be "normal" and do whatever FS ^R NORMAL$ and FS ^R REPLACE$ say.
Control-rubout has its own special routine, which deletes treating spaces as if they were tabs.
Control-digits update the numeric arg for the next command.
All other control characters not in the range control+101 through control+135 are errors.
Control-M inserts just a CR. Control-[is an error.
The remaining control characters from control-101 to control-135 do what the ^R command table says, or else are errors.

TECO program writing standards

1) Each line that doesn't begin inside a string argument should be indented with at least one space, preferably, with the number of spaces indicating how deep in conditionals and iterations it is.
2) The semantic content of one line of TECO program should be no greater than that of one line of any other language, if the program is to be understandable. in other words, break lines frequently - and put a comment on each line. There should be spaces between logical groups of commands, every few characters, as in "3K J IFOO$", which also shows how long a line should be.
3) The standard way to write a comment is to make it look like a tag: !<comment>!.
4) Follow value-returning commands with "W"'s when the value is not used for anything.
5) An example of a well-commented TECO program is RMAIL.

see .TECO.;RMAILX > and RMAILZ >.

TECO program debugging aids:

1) Trace mode causes all TECO commands to be typed out as they are executed. See the "?" command. a good technique is " ? FN:?$ MA " for running q-reg A in trace mode.
2) FS STEP$ causes TECO to pause at each line, displaying the buffer and waiting for input before continuing execution. This works best when lines are short, as they ought to be anyway.
3) Break-loops on errors are available, by setting FS *RSET$ to nonzero.
4) It is easy to edit a "FTHere I am$" or "Q0=" into the program and re-execute it.
5) If the standard top level is in use, "?" typed in after an error will cause a printout of a short section of command string up to the point of the error.
6) Setting FS .CLRMODE$ to 1 disables the ^L and F+ commands, which normally clear the screen. This may be useful for debugging programs that wipe out their trace output.