```
#       *Sfield(^D)$V(ESC)$(%)      The user finds a "field",
#       a great battlefield(%)      checks, and sees that
#       (%)                         it is the wrong one.
#       *Sfield(^D)$V(ESC)$(%)      The user finds the next
#                                   "field",
#       field as a(%)               and makes sure it is
#       (%)                         the right one.
#       *I,(^D)$V(ESC)$(%)          He inserts a comma,
#       field, as a(%)              and checks his work.
#       (%)
#
#       *Rbefoer(^D)$before(^D)$V(ESC)$(%)
#       remaining before us,(%)     The user corrects
#       (%)                         a typing error.
#
#       *JSbefoer(ESC)$(%)          The user checks for
#                                   another
#       ?SEARCH?35(%)               instance of this error,
#       JSbefoer$(%)                but there is none.
#
# The last Search Command in the dialog produced an error message;
# nevertheless, it illustrates a useful and legitimate application
# of the Search Command. The user wanted to know if there was an
# instance of "befoer" in the buffer, and the error message
# supplied the answer "no". A Search Command which fails does not
# move the pointer.
#
# MOVING TEXT
#
# It is often necessary to transport a string of text from one
# place in the editing buffer to another. This operation is
# required when multiple copies of a given string must be made or
# when a string must be moved which is too long to be conveniently
# deleted and retyped. The Q-Registers and their associated
# commands are used for this operation.
#
#     There are 37 Q-Registers, and each of them can hold a
# character string of virtually unlimited length. Each Q-Register
# has one of the 26 letters or the 10 digits or @ as its name, and
# all the Q-Register Commands (except one) end with the name of a
# Q-Register. The upper and lower case forms of a letter are
# equivalent as a Q-Register name, just as they are when used in a
# command name.
#
# Setting a Q-Register
#
# There are two commands for moving a substring of the buffer into
# the Q-Register. The Extract-String Command (m,nXq) extracts from
# the buffer everything from just after the m-th character to just
# after the n-th character. The Extract-Lines Command extracts
# everything between the pointer and the beginning of the n-th line
```

after the current line. Each command removes the selected string
from the buffer and places it in the Q-Register q. The pointer
is left where the extracted string was, and the previous contents
of the Q-Register are lost.
#
HXF Extracts the contents of the entire buffer (leaving the buffer empty) and puts it in Q-Register F.
#
Ø,23XA Extracts the first 23 characters and places the sequence in Q-Register A.
#
5XX Extracts a substring and puts it in Q-Register X. The substring extends from the pointer to the beginning of the 5-th line after the current line.
#
-4X@ Extract the preceding four lines and put them into Q-Register @.
#
:Xl Extract the remainder of the current line except the end-of-line.
#
The two Extract Commands move exactly that substring of the buffer which the corresponding Type-String or Type-Lines Command types out. Thus the user can use one of these Type-Out Commands to simulate an extraction.
#
Using a Q-Register
#
The Get-QR Command (Gq) inserts a copy of the character string in Q-Register q into the buffer just before the pointer. The contents of the Q-Register is not changed. When an Extract Command which refers to Q-Register q is immediately followed by "Gq", the total effect is to copy a substring of the buffer into a Q-Register without deleting it from the buffer.
#
The ;Type-QR Command (Qq;T) types out the complete character string contained in Q-Register q. It can be used to check on the successful execution of an Extract Command. This is the one Q-Register Command which does not have the Q-Register name, q, at the end of the command.
#
GA Inserts a copy of the contents of Q-Register A into the buffer just before the pointer.
#
5,62X3G3 Extracts characters 6 through 62, puts them in Q-Register 3, and copies Q-Register 3 into the buffer. Leaves the buffer unchanged.

```
#      QZ;T      Types-out the character string in
#                Q-Register Z.
#
# Merging Files
#
# The commands just described can be used for very general and
# extensive manipulations of files -- merging, interleaving, and so
# on.  For example, to merge parts of File A into File B proceed as
# follows:
#
#   -- On listings of Files A and B, mark the parts of
#      File A with the names of Q-Registers, and use these
#      Q-Register names to indicate where the parts are to
#      go in File B.
#
#   -- Read File A into the buffer, extract the parts into
#      the appropriate Q-Registers, and delete the
#      remainder of the file.
#
#   -- Read File B into the buffer and, for each
#      insertion, position the pointer in the buffer and
#      insert the contents of the appropriate Q-Register.
#
# This is a good procedure for making large-scale patches to any
# program or document.
#
# An Application
#
# The following dialog places at the beginning of the buffer an
# introductory sentence which cites the opening words of the text
# which is already in the buffer:
#
#      *JSago(ESC)$(%)          Finds end of the opening.
#      (%)
#      *Ø,.XAGA(ESC)$(%)        Extracts and restores
#      (%)                      the opening.
#      *QA;T(ESC)$(%)           Types out the contents
#      Fourscore and seven(%)   of Q-Register A
#      years ago(%)
#
#      *J(ESC)$(%)              Moves pointer to the
#      (%)                      top of the buffer.
#
#      *IThe Address begins(%)  Makes up the sentence.
#      "(^D)$GAI"(%)
#      The entire text is:(%)
#      (ESC)$(%)
#      (%)
#
```

```
#       *HT(ESC)$(%)                Types the result.
#       The Address begins(%)
#       "Fourscore and seven(%)
#       years ago"(%)
#       The entire Text is:(%)
#       Fourscore an(DEL)(BEL)(%)
#       (%)
#       (%)
#       *
#
```

STORING INTEGERS

A Q-Register can also be used to hold an integer value. This value is almost always used to save the position of the pointer in the buffer, but it is not restricted to that purpose. If the user wants to do some integer calculations, he can use Q-Registers as his variables.

Only two commands are required and they are very simple. The Update-QR Command (nUq) loads the integer value n into Q-Register q, destroying the previous contents. The Q-Value Command (Qq) is actually a function. Its value is the contents of Q-Register q, and it can be used wherever an integer value is accepted.

23UA	Puts the integer 23 into Q-Register A.
.U5	Puts the current pointer position (the number of characters before the pointer) in Q-Register 5.
Q5J	Puts the pointer at the position specified by the integer in Q-Register 5.
QA+2UA	Increases Q-Register A by 2.
QA=	Types out the integer in Q-Register A.

The Q-Register Commands for character strings and for integers are not mutually compatible. For example, if a Q-Register is loaded with the character string "398" by an Extract-String Command and an attempt is then made to get its value with the Q-value Function, TECO will object and type an error message. All of the Q-Registers initially contain the integer value Ø (as if a ØUq had been executed).

An Application

In the preceding dialog based on the Gettysburg Address, the first sentence of the Address was extracted. That was relatively easy because the beginning of the sentence had a known position,

Ø, in the buffer. In the dialog which follows, a sentence is extracted from the middle of the Address and both ends must be located. Q-Register Ø is used to save the position of the beginning of the sentence while the end is found.
#
*Sdid here.(ESC)$(%) Finds previous period,
*V(ESC)$(%) types end of previous
they did here.(%) sentence.
*LT(ESC)$(%) Gets to beginning
It is for(%) of desired sentence and
*.UØ(ESC)$(%) saves pointer.
*S.(^D)$T(ESC)$(%) Finds end of sentence
It(%) and checks.
*QØ,.XC(ESC)$(%) Picks up sentence.
*QC;T(ESC)$(%) Types out sentence.
It is for(%)
us the living,(%)
rather, to be(%)
...
#
#
#
#
#
#
#
LOOPS
#
Although TECO has general facilities for both conditional and unconditional transfer of control, it is the simple and specialized Iteration Command which is most useful.
#
In order to repeat any command string n times, (1) place the command string in angle brackets, "<" and ">", and (2) put n in front of the bracketted string. If n is omitted, an approximation to infinity (2^35) is assumed, and the loop will continue until something in the command string stops it.
#
3<R;(^D),(^D); V> Starts at the current position
of the pointer and replaces the
next 3 semicolons with commas.
Types each modified line.
#
5<L18<I.(^D)>> Puts 18 periods at the
beginning of each of the next
five lines.
#
J<Ryclepped(^D)yclept(^D); V>
Corrects all misspellings of
"yclept" in the buffer, however
many there are.

```
#      J<S(^S)command(^S)(^D); V>
#                          Types out every line in which
#                          the word "command" appears.
#
```

S and R commands inside iteration brackets cause the iteration to terminate if the search fails. Thus, if the buffer contains exactly one occurrence of the string "abc" the command J<Sabc(^D)V> will print the line containing the "abc" twice -- once when the search succeeds and again when it fails. This is because the V command is seen before the > which will cause the iteration to stop.

A Q-Register can be used to count the number of times a loop is executed. A special function, %q, is provided which increases the integer in Q-Register q by one and then assumes the resulting integer value. This function can be used as a free-standing command if the user types Control-D after it to "absorb" its integer value.

```
#      96UA26<%AI>          Inserts a complete lower-case
#                           alphabet into the buffer.
#
#      ØUAJ<S;(^D)%A(^D)>   Counts the semicolons which
#                           appear in the buffer and leaves
#                           the result in Q-Register A.
#
```

A Final Example

The following loop searches the buffer for the occurrences of the word "command". For each occurrence of the word, the loop types the line in which the word appears, stops to let the user type in a single character, and then capitalizes the words if the user typed "Y" (for "yes").

```
#      J<Scommand(^D); V^T-^^Y"E-7CRc(^D)C(^D)'>
#
```

This command string uses some commands which have not been described yet and, in any case, needs some explanation. An annotated listing follows:

```
#       J               Puts pointer at beginning.
#       <               Starts loop.
#       Scommand(^D)    Searches for "command".
#       ; V             Skips to end of loop if search
#                       fails.  Types out the line
#                       containing "command".
#       ^T-^^Y          "^T" is a function which assumes the
#                       value of the code for the next
#                       character the user types (TECO waits
#                       for the user).  "^^Y" is the code
#                       for "Y".
#       "E              If the preceding expressions is
#                       Equal to zero, the following
#                       commands are executed; otherwise,
#                       they are skipped up to the
#                       apostrophe.
#       -7C             Back up to just before "command".
#       Rc(^D)C(^D)     Capitalize "c".
#       '               End conditional expression skip.
#       >               End loop.
#
# This example is important.  It represents TECO at its best,
# namely in a short, interactive loop in which the user makes the
# difficult judgements and TECO carries out the editing details.
#
# 3.  SPECIAL EDITING
#
# All the TECO commands which have not been described in the
# previous sections are mentioned in this section.  Each division
# of this section describes a group of commands designed for a
# particular purpose.  The descriptions are brief because the
# commands are not of universal interest.  The purpose of this
# section is to inform the reader of the existence of specialized
# commands of TECO.  A complete description of each command appears
# in the TENEX TECO Handbook and can be found by looking up the
# command name in the command index.
#
# Automatic Indentation
#
# An effective technique for organizing information on a page is to
# use the "outline" form; that is, to indent lines by varying
# amounts to indicate the grouping and relative importance of the
# information.  This formatting technique is often applied to
# improve the readability of programs in block-structured
# languages, such as LISP, Algol, and PL/I.
#
#       The simplest means of indenting a line is to type in the
# appropriate number of leading spaces.  For a long program with
# many different indentations, this process is tedious and error
# prone.  TECO has a set of four special commands which can be used
# to supply these leading blanks automatically.  The commands are:
```

(^W) Records the number of spaces in a new indentation.

(^U) Inserts spaces required for the indentation which is currently in use.

(^B) Reverts to the indentation which was recorded just before the current one and inserts the required spaces.

(^Y) Reverts to the indentation which was recorded just after the current one and inserts the appropriate spaces.

The first time a particular indentation is required, the user types in the necessary spaces himself and uses the Control-W command to record the indentation. Indentations are recorded in a list which is a special part of TECO storage.

Logical Operators

An integer may be represented as a sequence of binary digits (as the reader may know). TECO has operators which convert an integer into a bit string, apply a logical operation to the strings, and convert the result back to an integer. The operators are:

m#n the i-th bit of the result is the logical "or" of the i-th bits of m and n

m&n the i-th bit of the result is the logical "and" of the i-th bits of m and n.

Conversion of Integers

In some cases it is useful to convert a sequence of digits which occurs in the buffer into an integer argument for a command. Conversely, it may be useful to convert an integer argument into a character-string representation of the integer. The necessary commands are:

n;N This function interprets the digit string which follows the pointer and assumes that integer value. The argument n specifies the base to be used in interpreting the digits (for example, n=8 for octal digits).

n\ This command expresses its argument, n, as a (possibly signed) sequence of digits of base 10 and inserts the sequence into the buffer just before the pointer.

Flow of Control

TECO has a rather complete set of commands for flow of control. These commands are supplied because even a short command string occasionally needs a conditional transfer or a custom-made loop to make it go. The following commands provide conditional execution of an arbitrary command string:

n"Ec' Executes the command string c if n Equals 0; otherwise, skips over c.

n"Nc' Executes c if n Not-equals 0.

n"Lc' Executes c if n Less-than 0

n"Gc' Executes c if n Greater-than 0

n"Cc' Executes c if n is the character code of a letter, digit, ".", "$", or "%".

The following commands provide an unconditional transfer of control:

!s! This is a label.

Os(^D) This is a transfer to the label "!s!".

Sometimes it is not appropriate to have a Search Command produce an error message when the search fails. This is the case, for example, when a search is part of a stored program. The following alternative is provided:

:Ss(^D) The ":" before a Search Command causes the whole command to assume an integer value of -1 or 0 according as the command succeeds or fails. Thus the command can be used wherever an integer argument is accepted. Note: Search commands and Replace commands inside iteration brackets (< ... >) act as if they have the : modifier on.

n;(SP) The ";(SP)" causes a skip out of the enclosing loop for non-negative values of n and otherwise is ignored. (The semicolon must be followed by a Space character.) If a ":"-Search Command is used as the argument to

this command, the command will skip out of the enclosing loop when the search fails.

Stored Programs

TECO has important commands which make it possible to create a TECO program, store it, and later execute it. Specifically, the command string is typed into the buffer like any other passage of text, is placed in a Q-Register by an extract Command, and is then executed by the Macro Command, as follows:

Mq — This command causes TECO to execute as a command string the contents of Q-Register q.

Since the character string in the Q-Register q may itself contain a Macro Command, this command provides for subroutines which can be called through one another as well as directly by the user.

A minor difficulty arises in preparing programs. An ordinary Insert Command cannot be used to enter into the buffer an Insert or Search Command because the Control-D which is a part of the stored command will prematurely terminate the overall insertion. The following commands solve this problem:

@Itst — This command inserts the character string s, which may include (^D). Any character which does not appear in s can be used as the character t which delimits the character string.

@Stst — This command searches for the character string s, which may include (^D).

@Rtslts2t — Replace string s1 by s2, both of which may contain the terminator t.

TECO has a few additional commands whose principal use is in stored programs. They are:

^T — This is a function whose value is the integer code for the next character typed in by the user. (TECO waits until the character is typed.)

;Ts(^D) — Types out s and is used to output a message from a running program.

[q Pushes down into a special pushdown stack the current value of Q-Register q.

]q Pops up the pushdown stack into Q-Register q.

? Causes TECO to "trace" its execution; that is, to type out commands as they are executed. The second "?" turns the trace off, the third turns it on again, and so on.

The basic input/output commands of TECO, as described in Section 1, obtain a file designator not from the command string but from a special dialog with the user. This operation is not appropriate for use in a stored program, and the following command sequences can be used instead:

;Rf(^D);Y
Reads in all of file f and adds it to whatever is in the buffer.

;Wf(^D);U
Writes out the entire buffer onto file f and clears the buffer.

These commands are further described in the discussion of paged input/output which follows this section.

The stored program commands and the rather complete set of flow of control commands combine to make possible some relatively complicated symbol manipulation. However, the proper use for the programming facility is to bridge the gap between simple editing and full scale symbol manipulation. TECO should not be used in competition with complete programming systems like LISP or SNOBOL. TECO does not have the debugging facilities, the data structures, or the efficiency to support such an activity.

TECO Paging

The computer on which TECO was originally implemented did not have virtual memory and its actual memory was relatively small. It was therefore necessary to devise a means by which a file could be edited piece by piece. Toward this end, the Form Feed character (Control-L) was selected as an "End-of-Page" character and each substring of a file which ended with a Form Feed was called a TECO page. (There is no relation between this "TECO page", which is purely a software notion, and the "page" which is the unit of the TENEX virtual memory.)

With the introduction of virtual storage and the consequent very great increase in the capacity of the buffer, the need for paging declined. For TENEX TECO, the possibility of the division

# of a file into TECO pages arises under the following conditions:
#
-- When a file exceeds the (approximately) one-million-character capacity of the buffer, it must be broken into parts.
#
-- When a file is to be merged with another file or rearranged in some way, it must be broken into parts.
#
-- When a file exceeds 65,000 characters (a very rough estimate) and will be subjected to extensive editing (that is, many insertions and deletions), efficiency considerations suggest that it should be broken into parts.
#
However, a decision to break a file into parts does not necessarily lead to paging the file. It will often be more appropriate to maintain each part of the file as a file in itself. Thus the cases in which paging commands are essential is rare.
#
For LISTing or TYPEing it is useful to have ^Ls on at logical places to make page boundaries simply for human use. Also can be handy in moving through file using n;BJ.
#
The commands for handling paged files will now be described briefly. They fall into several subgroups according to the steps of the input/output process.
#
The first step is to open files for input and/or output, as follows:
#
;Rf(^D) Selects the file whose designator is f and opens it for input.
#
;Wf(^D) Selects the file whose designator is f and opens it for output.
#
The following commands read a page from the file which is open for input.
#
Y Deletes the contents of the buffer and reads the next page of the input file into the buffer.
#
A Adds the next page of the input file to the end of the current contents of the buffer.
#
The following commands output a portion of the buffer which is specified by its arguments. Two, one, or no arguments can be

used, and their significance is described in the Handbook section
of the TECO Manual.
#
PW Does output without modifying the buffer.
#
W Does output and deletes from the buffer the portion which is output.
#
The following commands perform a combination of input and output.
The ;Y and ;U Commands which appear below were discussed in
Section 1; but here they have a different interpretation because
they are used when an input or output file is open.
#
P Outputs the current contents of the buffer and then reads in the next page of the input file. In effect, the command "turns a page" of the file being processed.
#
;Y Reads in the remainder (however many pages) of the input file and adds it to the end of the buffer.
#
;U Repeatedly executes P (Page-Turn) Commands until the remainder of the input file has been copied into the output file and then closes the output file.
#
The following command concludes the output process:
#
;C Closes the output file. (For certain technical reasons, the output file is not permanently saved until it is closed.)
#
Two commands are provided to search an entire paged file in a
single operation. Each command starts at the position of the
pointer (as usual) but does not stop at the end of the buffer;
instead, subsequent pages are read from the input file and
scanned until a match is found or the end of the file is reached.
The commands are:
#
nFs(^D)
Searches page after page. After a page has been searched unsuccessfully, it is written on the output file so that nothing is lost. This command is used when a file is being modified.

n;Fs(^D)
: Also searches page after page. However, after a page is searched unsuccessfully, it is discarded. This command is used when the file is being examined but not modified.

In contrast to the Search commands just mentioned, the indices described here are designed for use with a multiple-page file which is entirely in the buffer; no automatic input/output is involved.

;B
: Supplies the number of characters in the buffer before the current page. The current page is the page which contains the character just before the pointer.

;Z
: Supplies the number of characters in the buffer through the end of the current page.

n;B
: Supplies the number of characters in the buffer before the beginning of the n-th page in the buffer.

n;BJ
: Jumps to start of n-th page.

Quoting Control Characters

Certain control characters can be used literally in the character string argument of an Insert or Search Command when they are properly quoted. The commands for this purpose are:

(^V)
: In most cases, when this character is used immediately before a command control character it will cause that character to be entered into the command register literally.

(^V)(^Q)
: When this character pair is used before one of the three match control characters (as used in a search command), the match control character is interpreted literally.

Complete instructions for quoting characters on input, recognizing them on output, and specifying them in a search are given in the Character Set Appendix.

Abbreviations

There are a number of commands in TECO which can be described as abbreviations for commonly used command strings. Since TENEX TECO is very concise in any case, abbreviations do not play an important role and most of them are of limited interest. They

are:
#
EDIT f This is an Executive command which is an
alternative to the "TECO" command. It enters
TECO and then causes the file f to be read
into the buffer. For a subsequent editing of
the file in the same TENEX session, the user
can just type "EDIT".
#
EG   This is the exit from TECO which is used when
TECO has been called automatically by some
other subsystem of TENEX.
#
(LF)   This Command (the Line Feed character) moves
the pointer to the beginning of the next line
and types that line. The command must be the
first character in a command string. (The
command was imported from DDT). Note: (LF)
and (^H) are done immediately (i.e. (ALT) is
not needed).
#
(^H)   This command moves the pointer to the
beginning of the previous line and types that
line. The command must be the first
character in a command string. (The command
was imported from DDT).
#
^^c   This is a function whose value is the integer
code of the character c.
#
;P   This is a function whose value is the integer
code of the current character and which moves
the pointer one character to the right. It
thus combines a "1A" Function with a "1C"
Command.
#
B   This is an index which is always Ø. It is
useful because "B" is slightly easier to type
than "Ø".
#
(SP)   This operator (the Space Character) is
equivalent to the "+" operator. It is easier
to type than "+".
#
(HT)s(^D)
This command can be used to insert a string
which begins with a Tab (HT) character. In
effect, TECO supplies an "I" at the beginning
and makes this into an Insert-String Command.
#
This completes the list of useful TECO commands. The Command

```
# Index contains some other commands; they are either obsolete
# names which have modern synonyms or are characters such as
# End-of-Line or "$" which have no effect when they appear as TECO
# commands.  See TENEX TECO Manual for further information.
```

TTYTRB

TTYTRB is a teletype trouble report form for use within BBN Cambridge.
#
TTYTRB asks you for two things: the location of your terminal and a description of the problem.
#
When it prompts for "LOCATION OF TERMINAL:" type up to one line of information, terminated by carriage return. For details on typing this information, see "Subject" under SNDMSG in this manual.
#
When it prompts for "DESCRIBE TROUBLE:" type as much as you need to, terminated by control-Z. For details on typing this information, see "Message" under SNDMSG in this manual.

TTYTST

A teletype-testing program developed at BBN.

The program cycles through a series of tests which are described below. Any test may be prematurely aborted by hitting a single rubout. This will cause the next test to begin.

TEST 1 OUTPUT TEST

A series of lines of output are generated. All of the characters of the alphabet and all of the numerals are typed as well as all of the special characters. If your teletype is consistently misprinting on output, it will probably show up with this test.

TEST 2 CARRIAGE RETURN TEST

Line of somewhat random length repeating the same string of characters are typed. If these characters double up at or near the left margin, your teletype probably needs a dashpot adjustment.

TEST 3 ORDINARY TEST

Some lines of ordinary text are typed out. If your teletype is only occasionally misprinting on output, observe the output of Test 3 carefully, the problem may show up in it.

TEST 4 INPUT TEST

At this point you may type anything you like into a buffer in the program. The program will repeatedly type out the contents of this buffer after you type control D. If you make a number of mistakes use a number of control A's to erase them. If you hit a rubout during this last test, the program will give you a chance to type in more input.

TYPBIN

TYPBIN is a subsystem which does an octal dump of a packed file. A packed file is a file where every bit is considered to be an information bit of a continuous bit string. A packed file of 8-bit bytes, for example, will contain exactly nine bytes per each two words. In contrast, a standard format file would only contain eight of these bytes per two words, left-justified in PDP-10 ILDB/IDPB format.

When TYPBIN is started, it asks:

```
BINARY FILE INTERPRETER.
INPUT FILE=
```

The correct response is any readable file name. The program asks:

```
BYTE LENGTH (DECIMAL BITS) =
```

Any number from 1 to 36 is acceptable. The program then asks:

```
OUTPUT FILE =
```

Any writeable ASCII file is acceptable. The program then reads each successive "BYTE SIZE" of bits from the input file, writing out the octal value in ASCII to the output file. This output is columnated into easily readable format for 72-column wide paper.

On completion, the program reports the total number of bytes converted and exits. Example:

```
@TYPBIN
BINARY FILE INTERPRETER.
INPUT FILE = ABC [CONFIRM]
BYTE LENGTH (DECIMAL BITS) = 18
OUTPUT FILE = ABC.DUMP [NEW FILE]
DONE.
TOTAL BYTES (DEC) = 18
EXIT.
^C
@
```

At this point, file ABC.DUMP contains:

```
777772 000137 200740 000313 402000 000362 402000 000363 561040 000314
104000 000076 205040 120003 254000 000140
```

In the past, TYPBIN has proved invaluable in deciphering the unknown formats of data read from imported magtapes using the MTACPY subsystem.

TYPREL

Typeout of .REL Files

TYPREL analyzes the contents of .REL files created by MACRO, FAIL, and FORTRAN (F40).

TYPREL first asks for an output file where analysis results are to be written. The question is "OUTPUT FILE =", and an appropriate response is any file capable of receiving ASCII output. (TTY:, LPT:, FOO, etc.).

Next, the program requests the name of the .REL file of interest: "REL FILE=".

The results written on the output file are a tabulation of each block in the REL file, giving block type, the number of data words, (excluding relocation words), and the total number of words in that block. (See DECsystem10 Assembly Language Handbook, for explanation of block types.) For blocks declaring a program name, that name is also typed out. For blocks defining entry points, the names of the first 4 entry points are also typed out.

If a block is encountered which is entirely zeroes, it is noted as a block of type "ERR", and its length is counted but it is considered to contain no data words.

On completion of the file the total (decimal words) length of the file is typed out.

TYPREL provides a good doublecheck of FUDGE2 manipulations.

Example:

```
@TYPREL

OUTPUT FILE= TTY:

REL FILE= ANOM.REL;2

TYPE    DATA    TOTAL

4       Ø       2       ENTRY
6       1       3       NAME    ANOMAL
1       22      24      PROG
1       22      24      PROG
1       14      16      PROG
1       17      21      PROG
1       22      24      PROG
1       7       11      PROG
2       22      24      SYM
2       22      24      SYM
2       2Ø      22      SYM
7       1       3       START   'Ø
5       2       4       LAST    '135
Ø       Ø       126     ERR

TOTAL FILE LENGTH (DECIMAL WORDS) = 256

DONE.
@
```

WATCH

WATCH is a program that makes continuous on-line measurements of system activity.

WATCH requests an output file where it will put its measurements.

At a specifiable interval, it measures and outputs the following information in the form of readable ASCII text:

1. The percentage of the last minute spent

    a. In user computations
    b. Idle (no requests for service)
    c. In an I-O wait for a page to be read from the disc or drum.
    d. Running the core manager (a,b,c, and d should always sum to 100%)
    e. In page faults (this time is charged to users)
    f. In each of JOBS 0 to 15

2. The number of jobs in the balance set (runnable and loaded into core) averaged over the last minute.

3. The number of the following events in the last minute

    a. Drum pages read
    b. Drum pages written
    c. Disc pages read
    d. Disc pages written
    e. Terminal wakeups (break characters typed)
    f. Terminal interrupts (^C)

WATCH is terminated by typing ^C.

EXAMPLE:

```
@WATCH
OUTPUT TO: TTY: [OK]
INTERVAL IN SECONDS: 30       (any reasonable # in seconds)

second group values? y            (either answer Y or N)
JOB BREAKDOWN? Y                  (either answer Y or N)

USED IDLE IOWT CORE NCOR PURG NREM TRAP NRUN NBAL BSWT DSKW DRMW
DMRD DMWR DKRD DKWR TTIN TTOU TTBK TT^C

JOB0 JOB1 JOB2 JOB3 JOB4 JOB5 JOB6 JOB7 JOB8 JOB9
JB10 JB11 ...
 ..
 ..

62.2  0.0 33.8  3.8   8;  0.3   16  8.5  7.9  3.1  1.9  2.3 97.7
1688  620   16   50  391 1523  353    2
30.2 SEC, 21 JOBS.
11.1  0.7  0.0  0.0  8.6  2.2  0.0  0.0  4.0  0.0
 1.3  1.3  0.2  0.0  0.7  0.0 12.3  2.0  0.0  0.0
```

(on the next example, we just gave a yes on job breakdown)

```
OUTPUT TO: TTY: [OK]
INTERVAL IN SECONDS: 30

SECOND GROUP VALUES? N
JOB BREAKDOWN? Y

USED IDLE IOWT CORE NCOR PURG NREM TRAP NRUN NBAL BSWT DSKW DRMW

JOB0 JOB1 JOB2 JOB3 JOB4 JOB5 JOB6 JOB7 JOB8 JOB9
JB10 JB11 ...
 ..
 ..

60.0  0.0 33.0  3.8  109  0.3   23 10.610.9  4.5  2.4  3.2 95.7
30.3 SEC, 22 JOBS
12.0  3.0  5.4  0.0  0.2  2.6  0.0  0.0  0.0  3.0
 0.8  0.0 23.4  0.2  0.0  2.5  0.0  0.3  1.7  4.3
 0.8  0.7
^C
```

```
@
```

(on the next example, we just gave a yes on second group values)

```
OUTPUT TO: TTY: [OK]
INTERVAL IN SECONDS: 30

SECOND GROUP VALUES? Y
JOB BREAKDOWN? N

USED IDLE IOWT CORE NCOR PURG NREM TRAP NRUN NBAL BSWT DSKW DRMW
DMRD DMWR DKRD DKWR TTIN TTOU TTBK TT^C

74.4  0.0 21.3  3.9   95  0.7   53  8.2 12.6  6.9  2.4  2.6 96.7
1844  661   23   57  531 1693  274   17
30.5 SEC, 22 JOBS.
```

The following game programs in the directory are normally available to in house BBN people only; contact Tony Calleva with requests for exceptions. (e.g. ARPANET demos). See example below for using them.

```
@RUN <HACKS>CHESS

@RUN <HACKS>DOCTOR

@RUN <HACKS>JOTTO

@RUN <HACKS>LIFE

@RUN <HACKS>MAXIM
```

CHESS

CHESS is the chess-playing program developed by Richard Greenblatt, Donald Eastlake, and Stephen Crocker at M.I.T. It was described in "The Greenblatt Chess Program", (authors above), P801 - 810 of 1967 Fall Joint Computer Conference. The program is an honorary member of the United States Chess Federation and the Massachusetts Chess Association, under the name Mac Hack Six. In the April 1967 amateur tournament the program won the class D trophy; it wins about 80% of its games against non-tournament players.

During play, the program understands moves typed in using standard chess notation, some examples of which are given below.

P-KN3	Pawn to king's knight 3
B*P	Bishop captures pawn
O-O	Castle kingside
QR-Q1	Queen's rook to queen 1
R/K2-Q2	Rook on king 2 to queen 2
P-R8	Promote pawn (to queen assumed)
Q*P/Q6	Queen captures pawn on queen 6
O-O-O	Castle queenside

Other commands are available for control and information:

BD	Type out board
PW	Play white
PB	Play black
PN	Play neither
PS	Play self (both sides)
M	Make next move
U	Undo last move
DRAW	Request machine to acknowledge draw
PG	Print game (history)
LIST	List commands
RESET	Overturn board (for bad sports)

EXAMPLE:

```
@CHESS
 PB

P-KB3
 B P/K2-K4
BD
WR WN WB WK WQ WB WN WR
WP WP ** WP WP WP WP WP
-- ** WP ** -- ** -- **
** -- ** -- ** -- ** --
-- ** -- BP -- ** -- **
** -- ** -- ** -- ** --
BP BP BP ** BP BP BP BP
BR BN BB BK BQ BB BN BR

P-KN4
 B Q/Q1-KR5 CHECKMATE
PG

1  P/KB2-KB3      P/K2-K4
2  P/KN2-KN4      Q/Q1-KR5
```

DOCTOR

Simulated Psychiatrist

DOCTOR is a LISP program written by Joseph Weizenbaum and described in 'ELIZA - A Computer Program For the Study of Natural Language Communication Between Man And Machine', Comm. ACM, Vol. 9, #1. Jan. 1966.

DOCTOR simulates a psychiatric interview with a Rogerian psychotherapist. The 'subject' types in some statement in natural language using normal punctuation and sentence structure. Two editing characters are provided: Control A deletes the previous character typed in (repeatable back to the previous space or comma), and # deletes the entire line typed in.

Each input statement is terminated by a period or question mark. The session is terminated by typing GOODBYE.

EXAMPLE:

```
@DOCTOR

BBN LISP-10  05-27-72 ...

GOOD AFTERNOON.

TELL ME YOUR PROBLEMS. PLEASE TERMINATE INPUT WITH A PERIOD OR A
QUESTION MARK.

*I HAVE A BIG PROBLEM.

I AM NOT SURE I UNDERSTAND YOU FULLY.

*MY GRANDMOTHER HAS A GRAND GZORENPLATZ.

IS IT IMPORTANT TO YOU THAT YOUR GRANDMOTHER HAS A GRAND
 GZORENPLATZ?

*IT WAS THIS MORNING?

PLEASE GO ON.

*HAVEN'T GOT THE TIME.

LETS DISCUSS FURTHER WHY YOUR GRANDMOTHER HAS A GRAND GZORENPLATZ.

*GOODBYE.

IT'S BEEN MY PLEASURE, THAT'S $9.72 PLEASE.
NIL
```

LIFE

LIFE is the Mathematical Game described in that section of Scientific American, Vol. 223, #4, October 1970. It was originated by the mathematician John Conway at Cambridge.

LIFE simulates a colony of organisms living on a 72x72 rectangular grid. Each point except for those on the edges, has 8 neighboring points, 4 horizontally and vertically and 4 diagonally. The rules of LIFE are:

1. Birth
   A new organism is created on an empty grid point if exactly 3 neighbors are adjacent to the grid point.

2. Death
   An organism dies of overcrowding if it has 4 or more neighbors.

   An organism dies of isolation if it has fewer than 2 neighbors.

Deaths and births happen simultaneously.

The program requests an initial colony pattern from the user. This is input by typing for instance,

```
 * * *
*     *
 * * *
```

using asterisks also spaces and carriage returns. Control-A will delete the previous character, Control-X deletes the line, and Control-R retypes the line. The pattern is terminated with an altmode.

Each successive generation will be typed out until one of three things happens:

1. The colony dies
2. A stable pattern is established
3. Any teletype key is pressed

At that point, the program requests another initial pattern.

FTP

Introduction

FTP (File Transfer Protocol) provides facilities for file transfer between HOSTs on the ARPA Computer Network (ARPANET). The primary function of FTP is to transfer files efficiently and reliably among HOSTs and to allow the convenient use of remote file storage capabilities. The objectives of FTP are 1) to promote sharing of files (computer programs and/or data), 2) to encourage indirect or implicit (via programs) use of remote computers, 3) to shield a user from variations in file storage systems among HOSTs, and 4) to transfer data reliably and efficiently.

FTP Command Interpreter

Instructions to the FTP program are given via the FTP Command Interpreter. Characters typed on the user's terminal are read by the FTP Command Interpreter and decoded as commands to perform various actions by FTP.

Typing a "?" to the FTP Command Interpreter will yield a message to use the "HELP" command to type a summary of the FTP commands.

The FTP Command Interpreter provides command completion whenever a terminator is typed (full-duplex terminals only) and an exact match is achieved with some command or a unique initial substring is typed. Terminators are space, comma, alt-mode, and carriage return. Terminators are often not distinguished and are thus equivalent. Where necessary, comma is used to separate list items, space terminates a command or option and signals the desire to specify more options, carriage return ends a command unless more information is necessary. Altmode is the same as space except that it will cause command completion in those modes where it is normally suppressed.

Escaping Back to EXEC Mode

At any time, typing a Control-C (^C) will cause FTP to stop whatever it is doing and return to the EXEC mode.

Data Transfer Functions

Data and files are transferred only via the data connection. The transfer of data is governed by FTP data transfer commands received on the FTP connections. The data transfer functions include establishing the data connection to the specified HOST (using the specified byte size) transmitting and receiving data in the specified representation type and transfer mode, handling EOR and EOF conditions, and error recovery (where applicable).

Making a Connection

There are two ways of making a connection. Typing "CONNECT host-name" or "CONNECT octal-number" will cause a connection attempt to be made. If successful, the connection will be said to be complete. If unsuccessful, the connection will be said to be incomplete with a reason given.

Disconnecting

There are three disconnect commands. "DISCONNECT" disconnects the user from the remote host without returning to the EXEC, "BYE" is the same as "DISCONNECT", and "QUIT" returns the user to EXEC without closing the connection. Thus, to close the connection and return to the EXEC, the user should type the "DISCONNECT" command followed by the "QUIT" command.

In the event that the network connections are severed by a network failure, the user will receive a message that the network has been severed and/or that the data transfer is incomplete.

FTP Command Summary

Access Control Commands

CONNECT host-name or octal-number

Performs ICP to connect to the indicated host. This must be the first command issued.

LOGIN username optional-password optional-account

User identification that is required by the server for access to its file system.

username
The argument field is an ASCII string identifying the user. Special characters may be quoted by ^V.

optional-password
The argument field is an ASCII string identifying the user's password and may be optional. This field must be immediately preceded by the username field. The typeout of this field will either be "masked" or suppressed. Special characters may be quoted by ^V.

optional-account
This argument field is optional and is a number or ASCII string identifying the user's account.

CWD anothername

CWD is used to change the working directory to anothername. Using it requires the "QUOTE" command at present. (Example: *QUOTE CWD anothername)

ACCOUNT number or string

The argument field is a number or ASCII string identifying the user's account.

DISCONNECT

Disconnects user from remote host without returning to EXEC.

BYE

Same as DISCONNECT.

QUIT

Returns user to EXEC without closing connection.

Transfer Parameter Commands

BYTE size-of-data-connection

The argument is an ASCII-represented decimal integer specifying the byte size for the data connection. The size must be 36, 32 or 8 bit bytes.

TYPE data-type

The argument is a single ASCII character code specifying the representation types.

The following codes are assigned for type:

A - ASCII