ATARI PILOT EXTERNAL SPECIFICATION

Preliminary release	(23-AUG-79)
Revision A	(06-SEP-79)
Revision B	(18-SEP-79)
Revision C	(22-OCT-79)
Revision D	(25-FEB-80)
Revision E	(27-OCT-80)

Prepared by: Harry B. Stewart

ATARI PILOT EXTERNAL SPECIFICATION

1.0 Introduction

This document provides an external specification of the Atari PILOT language and operating environment. The operating modes of the PILOT interpreter, the syntax for the language elements and commands, and the behavior of the system in response to commands will all be discussed. The specification was prepared as an implementation aid for the implementor and as a design review document for Atari. Accordingly, the presentation is somewhat formal and brief, rather than tutorial, in nature.

2.0 Design Goals

The primary design goals associated with this implementation of Atari PILOT are:

Make the behavior of the system as consistent and easy to learn as possible.

Maintain compatibility with the core PILOT definition ('Guide to 8080 PILOT, Version 1.1') as written by John A. Starkweather.

Allow a reasonable amount of access to the Atari system capabilities, but not at the expense of having a small, clean, and easy to learn language.

Provide as much information to the user as possible, in a form that is easy to use. For example: error messages instead of error numbers, many debug features, etc.

Maintain compatability with Atari BASIC where it does not directly conflict with the PILOT language.

3.0 PILOT Operating Modes

PILOT is always in one of three operating modes, either immediate mode, run mode or auto-number input mode. In immediate mode, the interpreter reads and executes command lines from the Screen Editor; while in run mode, the interpreter executes a program from the deferred program storage area; and while in auto-number input mode, the interpreter reads PILOT statements from the Screen Editor and saves them for deferred execution.

3.1 Immediate Mode

The system initializes to immediate mode and signals that it is ready to accept a PILOT input line by issuing a prompt message 'READY' to the screen and displaying a visible cursor (white box) at the beginning of the following line. The operator then has three choices: either enter a PILOT statement for immediate execution, enter a PILOT statement for deferred execution, or delete a deferred PILOT statement. The mechanism for making the choice is an optional line number at the beginning of the input line, as shown below:

```
T:HELLO                 Immediate execution (no line number).
100 T:HELLO             Deferred execution (line number).
100                     Delete line 100 (no statement).
```

The system remains in immediate mode until one of the commands shown below is entered for immediate execution, at which time the system changes mode.

Command	New mode
Run	Run
Jump	Run
End	Run
Use	Run
Auto-number	Auto-number
Dos	Leaves PILOT environment

3.2 Run Mode

In run mode the PILOT interpreter is executing the program residing in the program storage area; the program will execute until one of the following conditions occurs:

The operator presses the BREAK key (preferred to RESET because BREAK produces an orderly stop).

The operator presses the RESET key (potentially dangerous because PILOT could be in the process of altering the string variable list; the interruption would leave the list in an unusable state).

The program runs out of statements to execute.

A PILOT End command is executed in the program with the Use stack empty (the normal termination technique).

A run-time error is encountered, such as a jump to a non-existent label.

When the deferred program ceases execution due to one of the above stated events, the system will return to immediate mode. In most cases, a message will be generated to indicate why. See Appendix E for a table showing the differences between these ways of terminating run mode.

3.3 Auto-number Input Mode

In auto-number input mode, the PILOT interpreter is reading PILOT statements from the Screen Editor, syntax checking the statements, and if correct then appending an internally generated line number and storing the statement in the program storage area. When a totally blank line is input by the operator, or if an invalid line number is generated, PILOT returns to immediate mode.

4.0 PILOT General Statement Syntax

The syntax specifications contained in the body of this document are similar to BNF, except that square bracket pairs "[...]" are used to delimit optional fields.

```
<PILOT input line> ::= [<line #>][<PILOT statement>]<EOL>
<line #> ::= <numeric constant>
<PILOT statement> ::= [<label>][<command>][<comment>]
<label> ::= *<label name>
<label name> ::= <alphanumeric character>[<label name>]
<command> ::= [<command name>][<condition>]:[<command
                                                parameters>]
<command name> ::= A|T|M|C|J|U|E|R|Y|N|RUN|LIST|DUMP|NEW|VNEW|
                   READ|WRITE|CLOSE|MS|JM|GR|SO|PA|POS|TAPE|
                   LOAD|SAVE|DOS|TSYNC|AUTO|REN|TRACE
<condition> ::= [Y|N][(<nexp>)]
<command parameters> ::= as defined for each command in Section 6.
<comment> ::= [<any character other than EOL>
<EOL> ::= <end-of-line character (9B hex) supplied by the
           Screen Editor on every line.>
```

Some examples of PILOT input lines with valid syntax follow:

```
100 T:HELLO, HOW ARE YOU TODAY?
20 A:
30
8000 *START
T:HELLO THERE.
160 :
999 *LOOP C:#A=#A+1 [ INCREMENT THE LOOP COUNT.
84 TY:VERY GOOD!
60 J(#A>1):*MORE
```

The elements of a PILOT input line are described in the following paragraphs:

4.1 Line Number

The optional line number must be within the range of 0 to 9999, when entered.

4.2 Statement Label

The statement label may be of any length (as governed by the longest input line allowed) and all characters in the label are significant and are retained. There is no test made for duplicate labels; if there are two statements with identical labels, the one with the lower line number will be the target of Jump and Use commands. The label is delimited by the first non-alphanumeric character found.

4.3 PILOT Command

The PILOT command must be spelled precisely as there are no short forms or long forms supported. For example: "TYPEN:" would be interpreted as TY<junk>: rather than T<ignored>N:. The Atari PILOT commands are specified in Section 6.

4.4 Condition

The optional condition field allows for the selective execution of any and all PILOT statements. There are two criteria for the selection: the Match result flag tests, Y and N, and/or the arithmetic test (<numeric expression>). When both tests are part of the condition, both must be true for the statement to execute.

If Y is the condition, the statement is executed only if the most recent Match command found a match; conversely, if N is the condition, the statement is executed if the most recent Match command did not find a match.

For the arithmetic test, if the expression within parens has a value greater than zero, the statement is executed; if the value is less than or equal to zero, the statement is not executed.

4.5 Field Delimiting

All fields are delimited by the first occurrence of a character which is not valid for that field. For example, the line number field is delimited by the first non-numeric character, which may be (but is not required to be) a blank. Blanks are ignored to the left of the ':', except to delimit fields, so they may be used liberally to format a PILOT input line; note, however, that in deferred execution statements, the blanks do use up storage space (one byte per blank). Some examples of valid input lines (without and without blanks) are shown below:

```
100T:THIS IS VALID.
100*LOOP A:
100J( #I - 3):*LOOP
100  J(#I-3):*LOOP
```

The comma is a general purpose operand delimiter (to the right of the ':') and commas and blanks may be used interchangeably in that context. Note that any consecutive string of commas and/or blanks is treated as a single separator, except within the Match commands.

```
<sep> ::= <space>|,|<sep><sep>
```

For example the statements shown below are all equivalent:

```
LIST:100 200                    Single space as separator.
LIST:100,200                    Single comma as separator.
LIST:100       200              Multiple spaces as separator.
LIST:100,,, , ,,, ,,200         Multiple commas and spaces.
```

4.6 Comment

The optional statement comment is delimited at the beginning by the square left bracket character ('[') and at the end by the EOL character; any character may appear inside a comment.

4.7 Command Continuation

There is one special case not yet described, and that is command "continuation", in which the command name and condition result are used from the most recently seen statement. This option is specified by having the command name and condition field both absent from the statement, as shown in the examples below:

```
100 T(#A>40):
105 :THE RESULT IS TOO LARGE.
110 :PLEASE TRY AGAIN.
```

which is equivalent to:

```
100 T(#A>40):
105 T(#A>40):THE RESULT IS TOO LARGE.
110 T(#A>40):PLEASE TRY AGAIN.
```

Continuation is allowed only after the Type commands (T, Y & N) and the Remark command when in run mode; a continuation after any other command will result in a run-time error, which will stop the execution of the offending program. Continuation is allowed after any command when in immediate mode.

The default continuation command at power-up is Type; that same default is also established after the following conditions: RESET, an error of any kind and transition from run mode to immediate mode.

5.0 PILOT Data Types

PILOT allows the user to manipulate two distinct types of data: numeric data and text data.

5.1 Numeric Data

Numeric data is maintained internally as 16-bit signed integers with a range of -32768 to +32767. All operations upon numeric data will produce integer results, truncated to 16 bits, with numbers in the range of 32768 to 65535 being treated as negative numbers in the range -32768 to -1 (standard two's complement arithmetic). A result of this convention is that the positive number domain wraps into the negative number domain, e.g. 32767 + 1 = -32768. Note that this is not considered an error by PILOT.

The specifications for language elements that represent numeric data of various types are given below:

5.1.1 Numeric Constants -- Numeric constants consist of one or more numeric digits, terminated by any non-numeric character. Numbers may have any (non-zero) number of digits, with the resultant value being truncated to a 16-bit signed number. In most, but not all, contexts a unary minus sign is allowed to precede the first digit.

```
<numeric constant> ::= [-][<numeric constant>]<digit>
```

Examples:

```
1
32767
00000000000023
345698765243621864
-475
```

5.1.2 Numeric Variables -- Numeric variables are specified by the prefix delimiter '#' followed by one of the letters 'A' through 'Z'. There are 26 numeric variables that can be accessed by the language, #A through #Z.

```
<numeric variable> ::= #<alphabetic character>
```

Examples:

```
#L
#J
```

5.1.3 Random Number -- A hardware generated random number may be used anywhere a numeric expression is allowed; the '?' character is used to specify that a random value is to be selected.

Examples:

```
J(?):*SOMETIMES          Toss of the coin Jump.
C:#A=?                   #A takes on a random value.
GR:TURN ?                Rotate theta by random amount.
```

5.1.4 Atari Controller Sense -- The Atari paddle controllers, joysticks, lightpen and their associated trigger inputs may be sensed as numeric data using the following constructs:

```
<joystick sense> ::= %J<select>
<paddle sense> ::= %P<select>
<trigger sense> ::= %T<select>
<lightpen sense> ::= %H|%V|%L
<select> ::= <number constant>|<numeric variable>|<pointer>
```

The sense constructs are recognized anywhere a numeric constant is allowed and in text expressions. See Appendix F for more information regarding the controllers and the sense values.

Examples:

```
J(%T0):*TRIGGER          Jump if trigger set.
C:#A=%P1                 Read Paddle #1 to numeric variable.
C:#J=%J#I                Read Joystick specified by #I.
C:#H=%H                  Read lightpen horizontal value.
```

5.1.5 Special Variables -- In addition to the controller sense variables, there are several other special "read only" variables which may be sensed (as if numeric constants) using the following constructs:

```
<free memory> ::= %F
<graphics x-coordinate> ::= %X      See section 6.1.12
<graphics y-coordinate> ::= %Y
<graphics theta angle> ::= %A
<graphics data value> ::= %Z
<Match result> ::= %M               See section 6.1.4
```

The special variables are recognized anywhere a numeric constant is allowed and in text expressions.

Examples:

```
T:FREE MEMORY = %F.      Displays the amount of unused memory.
J(%M=5):*CASE5           Jump on Match result.
GR:GOTO #X,%Y            Goto computed x, without changing y.
J(%X>=80):*OUTX          Test for graphics cursor out window.
```

5.1.6 Pointer -- A pointer is the specification of the memory address of a numeric quantity, the data at the specified address being one or two bytes in length. A pointer may be used anywhere a numeric variable is allowed.

The syntax for the pointer specification is shown below:

```
<word pointer> ::= @<address>
<byte pointer> ::= @B<address>
<address> ::= <numeric constant>|<numeric variable>|<word pointer>
```

The 'B' is an optional modifier which specifies that the pointer points to a memory byte. If the 'B' is not present the pointer is assumed to point to the l.s.b. of a two-byte word, where the next higher memory location contains the m.s.b. of that word. The 6502 has no inherent word organization, so words may start on either even or odd addresses.

In a multi-level indirect pointer reference the 'B', if present, may only be placed at the outermost level, as shown below:

```
C:#A = @B@4096       is valid.
C:#A = @@B4096       is not valid.
```

Examples:

```
Content of location 4096 (& 4097) = 5000
Content of location 5000 (& 5001) = 4103
Content of location 4103 (& 4104) = 5160

C:#A = 4096          #A = 4096.
C:#A = @4096         #A = 5000.
C:#A = @@4096        #A = 5160.

C:#A = @B4096        #A = 136 (lsb of value 5000).
C:#A = @B4097        #A = 19  (msb of value 5000).
                     Note that 5000 = (19 * 256) + 136.
```

The PILOT equivalents of the BASIC PEEK function and POKE command are shown below:

```
PEEK (<addr>)          @B<addr>
POKE <addr>,<data>     C:@B<addr> = <data>
```

Since the internal storage for all numeric quantities within PILOT is as word quantities, the following transformations will occur when byte pointers are used:

Byte value as target -- the l.s.b. of the integer source value is stored and the m.s.b. is not used. The next higher memory byte above the byte target is neither accessed nor altered.

Byte value as source -- the l.s.b. of the integer result is set to the value of the referenced byte and the m.s.b. of the integer result is set to zero. The next higher memory byte above the byte source is not accessed.

When a pointer is used as a target for a numeric assignment, the byte or word is always read before it is written to; this may cause problems when writing to some types of hardware devices, such as a PIA, where the hidden read may clear a status bit before it is read by the program.

5.1.7 Numeric Expression -- A numeric expression (<nexp>) is a list of one or more numeric elements separated by numeric or logical operators, as in an algebraic formula. Numeric expressions are evaluated from left to right, with no operator precedence rules; parentheses are allowed (encouraged?) to either clarify the formulae or to alter the left to right evaluation scheme. Any number of redundant parens are allowed (those that don't alter the evaluation order) and up to 2 levels of nested non-redundant parens are allowed.

```
<nexp> ::= <operand>[<operator><nexp>]|(<nexp>)|-<nexp>
<operand> ::= <numeric constant>|<numeric variable>|?|
              <pointer>|<controller sense>|<special variable>
<operator> ::= +|-|*|/|\|>|<|=|<=|>=|<>
```

The operators have the following meanings:

```
+  is the numeric addition operator.
-  is the numeric subtraction operator.
*  is the numeric multiplication operator.
/  is the numeric division operator (truncated result, as
   opposed to the floor value or rounded result).
\  is the modulus operator (result is always positive).
```

The relational operators all have the characteristic that they compare the term on their left with the term on their right and generate a boolean result based upon the results of the comparison. A value of 1 is generated when the comparison is true, 0 when false.

```
>  is the "greater than" operator (test for left greater than
   right).
<  is the "less than" operator (test for left less than right).
=  is the "equal to" operator (test for left equal to right).
<= is the "less than or equal to" operator (left not greater
   than right).
>= is the "greater than or equal to" operator (left not less
   than right).
<> is the "unequal to" operator (test for left not equal to
   right).
```

Examples:

```
56
#A+4
((((7))))                    Extra (redundant) parens don't matter.
1+(142/(#J/6))
1+2+3+4+5+6+7+8+9
(1+2+3+4+5+6+7+8+9)
#J/-3
#V<3
(#A<3) * (#B=#C)             Logical "and" of two relations.
(#D<>5) + (#Z/2=1)           Logical "or" of two relations.
@#P>%F
```

5.2 Text Data

Text data comes in two flavors, text literals and named strings. Unlike most languages, PILOT does not require quotes or any other kinds of delimiters for text literals; within the command context, anything that cannot be identified as being otherwise is assumed to be a text literal.

5.2.1 Text Literal -- All textual data that is not explicitly identified as being otherwise, is assumed to be a text literal. Text literals may contain any ATASCII character except for the following:

```
  $ -- this is the string variable name prefix delimiter. *
  # -- this is the numeric variable name prefix delimiter. *
  % -- this is the controller sense/special variable prefix
       delimiter. *
  @ -- this is the pointer specifier. *
  [ -- this is the comment prefix delimiter.
  <EOL> -- this is the end of line delimiter.
```

* Note: these characters may be used as literals if the character that follows does not start a valid data variable specification; a space is always suitable for this purpose. Some other possibilities are shown below:

```
  T:YOUR WEIGHT IS 30#.          the '.' makes the '#' literal.
  T:#X%, ARE YOU SURE?           the ',' makes the '%' literal.
  T:THE COST IS $#V.             the '#' makes the '$' literal.
```

See Appendix G, item 11 for a further discussion of using and generating key words in text expressions.

Examples:

```
  HERE IS SOME TEXT
  THIS & THAT.  AND SOME MORE.
  %'''""""!;+@?????
  BLANKS       ARE           SIGNIFICANT TOO!
```

5.2.2 Named Strings -- A named string is a dynamic data element that consists of a name portion and a data portion. Just as a numeric variable has both a name (#A) and a value (-45), a string variable has a name ($BIRTHPLACE) and a value (SAN JOSE); however, for a string variable, both the name and value are text. A string variable name consists of the prefix delimiter '$' followed by at least one alphanumeric character (but up to 254 total). All characters in the string name have significance and are retained. String variable data consists of a single text literal of from zero (null string) characters up to 254 ATASCII characters. The data portion never includes an <EOL> character.

```
  <string variable name> ::= S<alphanumeric character>
                             [<alphanumeric characters>]
```

Examples:

```
$NAME
$RESPONSE
$BIRTHPLACE
$S1
$6
$A
```

5.2.3 Text Expression -- A text expression is a list of one or more textual elements which are concatenated to form a textual result. A text expression is scanned from left to right, with all data assumed to be literal text unless it is first delimited by '#', '$', '%' or '@' in which case it is then assumed to be a numeric variable (if '#'), a string variable (if '$'), a controller sense or special variable (if '%') or a numeric pointer (if '@'). If a string variable is undefined, its name is substituted for the missing value. At the end of a variable name, scanning resumes assuming literal text once again. An optional underscore ('_'), if present as the last character of the text expression, will be converted to a blank as part of the evaluation of the text expression; this features allows trailing blank(s) to be part of text expressions, which the Atari Screen Editor would otherwise not allow.

```
<texp> ::= <operand>[<texp>][_]
<operand> ::= <text literal>|<string variable>|<numeric
              variable>|<controller sense>|<special variable>|
              <pointer>
```

Examples:

```
THIS IS TEXT
HELLO $NAME
YOUR AGE IS #A.
HERE IS A TRAILING BLANK_
FREE MEMORY IS NOW %F BYTES.
THE VALUE IS @5000.
```

Text expression are evaluated into an internal buffer before being used as data for a PILOT command; the buffer size is 254 characters and if a text expression exceeds that size it is truncated without a warning message or error being generated.

5.2.4 String Indirection -- String indirection is a special syntactic form in which the data portion of one string is made to specify the name of another string. Just as the value of text literal ABC is 'ABC', and the value of $ABC is the data portion of the string named ABC, so the value of $$ABC is the data portion of the string whose name is the data portion of the string named ABC. As many '$'s as desired may precede the name portion, and this form may be used wherever a string variable is allowed. If the indirection cannot be carried to the number of specified levels, because of undefined string names, the result will be the same as that obtained by specifying a simple undefined string name. The examples below may clarify string indirection:

```
$LADDER=JANE              existing named strings.
$JANE=ATARI
$ATARI=LUNCH

T:$LADDER                 will produce 'JANE'.
T:$$LADDER                will produce 'ATARI'.
T:$$$LADDER               will produce 'LUNCH'.
T:$$$$LADDER              will produce '$LUNCH'.
```

Note that '$'s are not expected to be part of the string data in order to do indirection; in fact, if they are present, they will cause the indirection to fail, as they are not valid characters in a string name. In other words, all indirection is specified at the outer (command) level; no additional indirection is obtained via '$'s in the strings being scanned.

String indirection is allowed anywhere a simple string variable is allowed. See Appendix G, item 3 for one possible use for string indirection.

5.3 Explicit Delimiting

Note that there may be up to four separate PILOT entities based upon a single alphabetic letter, let us use the letter 'F' for an example:

```
F -- the text literal
#F -- the numeric variable
$F -- the string variable
*F -- the statement label
```

The PILOT system will not be confused, as all but the text literal are explicitly delimited.

6.0 PILOT Commands and Syntax

This section specifies the syntax of each of the commands that are implemented in Atari PILOT. First the commands that are executed in either run mode or immediate mode are defined, followed by those that are executed only in immediate mode.

6.1 PILOT run/immediate mode commands

The commands described in this section may be executed from the console while in immediate mode or may be entered to the program storage area to form a program. The same syntax checking and semantic processing is applied in either mode, so no special rules have to be remembered. Immediate mode command execution accesses the same data base as run mode command execution, so the user may interact with his program's data as an aid in debugging.

6.1.1 TYPE Commands (T, Y or N)

These commands output information to the text screen. The data to the right of the ':' is evaluated as a text expression (as described in section 5.2.3) and then output to the screen. Logic within the Type command processor assures that no word will be broken at the right margin of the text screen, unless that word's length exceeds the defined screen width. Every physical line output is terminated by an <EOL>, so that logical and physical lines are synonymous, except when the last character to be output is a '\', in which case the '\' and the <EOL> are both suppressed.

There are two alternate names provided for the Type command: 'Y' is an abbreviation for 'TY' and 'N' is an abbreviation for 'TN'; note that the command forms 'YY', 'YN', 'NN' and 'NY' are syntactically proper under this implementation, but are either redundant (as in 'YY') or result in statements that will never execute (as in 'YN').

```
<type operand> ::= <texp>[\]
```

Examples:

```
T:$NAME IS #A YEARS OLD
will produce JACK IS 10 YEARS OLD if $NAME=JACK & #A=10,
or $NAME IS 0 YEARS OLD if $NAME is undefined & #A=0.

T:
will produce an empty line (<EOL> only).
```

6.1.2 ACCEPT Command (A)

The Accept command allows the user to enter data from the text screen to an internal storage area known as the accept buffer. The Accept command allows one or two optional operands to the right of the ':'. If a string variable name is the first operand, then the accepted data is stored in that variable as well as in the accept buffer. If the first operand is a numeric variable, then if the accepted data

contains a numeric constant anywhere in the text, the value is stored in the variable or if the accepted data is totally non-numeric then the value will be zero (no error message will be generated). The input of an empty line (<EOL> only) will result in a numeric variable being set to zero, and a string variable being set to the null (empty) value; note that null strings are not the same as undefined strings.

If an '=' operator is present, then the text expression to the right of the '=' will be assigned to the accept buffer, rather than having PILOT go to the console to get accept data.

All accepted data goes through the following transformation as it goes to the accept buffer (but not to the optional string variable):

A space is inserted at the beginning of the accepted data.

A space is inserted at the end of the accepted data.

All lower case alpha characters are converted to upper case.

All multiple spaces are converted to a single space.

The accept buffer is 254 characters in length and if the accepted data exceeds that length the tail end of the data will be truncated.

```
<accept operand> ::= [<numeric variable>|<string variable>]
                     [=<texp>]
```

Examples:

```
A:                  Accepts data to the accept buffer only.
A:#A                Sets #A to the numeric value entered.
A:$NAME             Assigns to $NAME the text literal entered.
A:=$LEFT            Assigns the value of $LEFT to accept buffer.
A:$WHAT=HI $NAME    Complex assignment to buffer and variable.
```

If the user program requires that numeric data be input when asked for, the code sequence shown below will suffice:

```
*NUMIN A:#N          [ ACCEPT A NUMBER.
M:0,1,2,3,4,5,6,7,8,9
TN:Please enter a number.
JN:*NUMIN
```

6.1.3 MATCH Command (M) -- The Match command scans the current content of the accept buffer, trying to find an exact match with one of the Match command fields. The operand is a text expression which will evaluate to one or more match fields separated by commas or, optionally, vertical bars. As always, blanks have significance in text expressions. If a match is found, the internal match flag is set true (= 1 to n, where n is the ordinal number of the match field producing the match); if no match is found, the match flag is set false (= 0). This flag is the one that is tested by the statement conditional operators 'Y' and 'N', is used in evaluating the Jump match command, and

also generates the value for the special variable '$M'.

```
<match operands> ::= [<vertical bar>][<skip>]<match list>
                     [<field sep>]
<skip> ::= <right arrow>|<skip><right arrow>
<match list> ::= <match field>|
                 <match list><field sep><match field>
<field sep> ::= <comma>|<vertical bar>
<match field> ::= [<ATASCII characters, not including the
                   field separator or EOL character>]
```

If the first character of the operand is a vertical bar ('|'), then that character will be the field separator for the match list, instead of a comma. If the first character of the operand is anything other than a vertical bar, then the field separator will be a comma. There is no situation in which both the vertical bar and comma may be field separators at the same time.

Cursor right characters (ESC CTRL-*) appearing at the beginning of the operand (they may start after the optional vertical bar) will cause pattern matching to start at the n+1th character of the accept buffer, where n is the number of right arrows specified. A single right arrow is used to allow pattern matching to start after the omnipresent leading blank character in the accept buffer.

Examples:

M: YES , YEAH , SURE_	Tries to match words (note blanks).
M: YES , YEAH , SURE_,	Functionally identical to above.
M:YE,SURE	A less precise matching of character substrings within words.
M:$VERBLIST	Uses a string variable containing the match fields.
M:|.|;|,|:	Matching for punctuation marks, including a comma.

The scan algorithm used in the Match command scans the accept buffer to find a possible occurrence of the first specified match field, if none is found then the accept buffer is scanned to find an occurrence of the second specified match field, etc. until a match is found or there are no more match fields. Several special cases are shown below:

M:	Null operand is not allowed.
M:,	Will match anything (null match).
M:THIS,THAT,,OTHER	Will match anything (null match).

6.1.4 MATCH (Producing) STRINGS Command (MS) -- The Match Strings command behaves exactly the same as the Match command, and also produces three named strings as a result of any successful match. $LEFT will have a value equal to everything to the left of the match, $MATCH will have a value equal to the match data, and $RIGHT will have a value equal to everything to the right of the match. Any of these strings may have null values, depending upon where the match occurred within the accept buffer data. If the attempted match is unsuccessful, the three strings will retain the values they had prior to the

command execution.

When <right arrow> operands are specified, and a successful match is made, the value of $LEFT does not contain the Accept buffer characters skipped over by the <right arrow>s.

```
  <match operands> ::= <same as for the Match command>
```

Example:

```
  A:=THIS IS A TEST.
  MS: IS , WAS , WILL BE ,
  $LEFT = ' THIS'
  $MATCH = ' IS '
  $RIGHT = 'A TEST. '


  MS:→_                    Match first imbedded blank.


  A:=WHAT WILL HAPPEN?
  MS:→→→_
  $LEFT = 'AT'
  $MATCH = ' '
  $RIGHT = 'WILL HAPPEN? '
```

6.1.5 COMPUTE Command (C) -- The Compute command assigns the value of a numeric expression to a numeric variable or assigns the value of a text expression to a string variable.

```
  <compute operand> ::= <numeric variable>=<nexp>|
                        <string variable>=<texp>
```

Examples:

```
  C:#L=#L-1                              Decrements variable #L.
  C:#C = 1                               Assigns the value 1 to #C.
  C:#A=314*#R*#R/100                     #A = 3.14 * #R**2.
  C:#J = #J-(#A/2)                       Assigns new value to #J.
  C:$ADDRESS=#N $STREET GILROY           Assigns to $ADDRESS the
              concatenation of the value of #N converted to
              ATASCII, followed by the value of the string
              $STREET followed by the text literal GILROY.
  C:$STEMP = $RIGHT                      String indirection O.K.
```

6.1.6 REMARK Command (R) -- The Remark command allows for the insertion of remarks into the body of a PILOT program. The condition field result has no effect upon the action of this command, except for slight variations in the time to execute. That is to say, 'R', 'RY' and 'RN' are all equivalent.

```
  <remark operand> ::= <anything>
```

Examples:

```
  R:****************************************
```

```
R:*                                        *
R:*     PILOT PROGRAM "TEACHER"            *
R:*                                        *
R:*              21-AUG-79                 *
R:*                                        *
R:******************************************
```

6.1.7 JUMP Command (J) -- A run mode Jump command allows the running program to alter its normal linear flow by continuing execution at the statement with the specified label. An immediate mode Jump will cause PILOT to enter run mode and start program execution at the specified label; contrary to Run command execution, no initialization of variables, Use stack, accept buffer or the screen will take place. If two or more statements have the specified label, the one with the lowest line number will be the target of the Jump.

<jump operand> ::= <label>

Examples:

```
J:*LOOP1
J:*BEGINNING
```

6.1.8 JUMP (on) MATCH (JM) -- The Jump on Match command allows the running program to jump to one of several labeled statements, depending upon the result of the most recently executed Match command. Each of the labels specified as an operand, corresponds to a Match field; if the match was successful with the nth match field, the nth operand label will be used for the jump. If the prior Match was unsuccessful, or if there is no nth operand label, no jump will be executed. Immediate mode execution of a Jump on Match command is as described for the Jump command.

<jump match operand> ::= <label>[<sep><jump match operand>]

Examples:

```
JM:*L1 *L2 *L3
JM:*HERE,*THERE,*EVERYWHERE
```

6.1.9 USE Command (U) -- The Use command corresponds to the BASIC GOSUB command. It allows the program to alter the linear flow to invoke a subroutine, and when the subroutine is done, control is returned to the statement following the Use command. Up to eight (8) Uses may be nested before the system responds with an error message. Immediate mode execution of a Use command is as described for the Jump command, with the exception that the Use stack is cleared.

<use operand> ::= <label>

Example:

```
U:*GETDATA
U:*SQRT
```

6.1.10 END Command (E) -- The End command tells the interpreter to return to the statement following the most recently executed Use command, or to stop the execution of the program if there is no Use return stacked. This command, as all others, may be conditional, and there may be any number of them in a program and/or subroutine. Immediate mode execution of an End command is as described for the Jump command, with the exception that execution starts at the statement at the top of the Use stack. If the Use stack is empty, PILOT immediately reverts to immediate mode.

```
<end operand> ::= <null>
```

Examples:

```
EY:
EN:
E:
```

6.1.11 NEW VARIABLES Command (VNEW) -- The New variables command allows the selective clearing of the numeric variables and/or the string variables. If the operand is null then both variable types are cleared, and if the operand is '$' or '#', then the string or numeric variables (respectively) are cleared. Note that when the string variables are cleared, any currently active READs or WRITEs are closed.

```
<vnew operand> ::= [$|#]
```

Examples:

```
VNEW:$                  Clears the string variables.
VNEW:#                  Clears the numeric variables.
VNEW:                   Clears both the numeric & string vars.
```

6.1.12 GRAPHICS Command (GR) -- The Graphics command allows the user to move a cursor around the graphics screen and to draw line segments and plot points of various colors. There is one PILOT command provided to handle all of the graphics capabilities, as the operand field actually contains graphic sub-commands that do the graphic manipulations. The graphics sub-commands constitute a language within the PILOT language and differ syntactically from core PILOT; for example: multiple sub-commands may appear in one line, there is an iteration construct, etc.

```
<graphic operand> ::= <graphic sub-command>[;<graphic operand>]|
                      <iterate count>(<graphic operand>)|
                      <iterate count><graphic sub-command>
```

<iterate count> ::= <numeric constant>|<numeric variable>

For example: GR:GOTO 0,0;TURNTO 0;4(DRAW 10;TURN 90) draws a square with the lower left corner at the screen center.

Note that the iterate count must be specified as a positive integer that will be decremented until zero is reached; thus iterate counts may range from 0 to 65535.

PILOT supports two drawing systems, a cartesian system of X,Y points and a polar system of R,THETA vectors (turtle graphics). Both systems are available at all times, and there is no problem in mixing sub-commands that deal with both systems. The coordinate directions and reference points are listed below:

X=0, Y=0 is at the center of the graphics screen.
Increasing X goes to the right.
Increasing Y goes upward.
THETA=0 is straight up.
Increasing THETA is clockwise.
Increasing R is in the direction of THETA.

The inside left edge of the graphics screen is X = -79.
The inside right edge of the graphics screen is X = 79.

The inside top edge of the graphics screen is Y = 47.
The inside bottom edge of the graphics screen (top of text window) is Y = -31.
The inside bottom edge of the graphics screen (bottom of text window) is Y = -47. Although the graphics data in the region -47<Y<-31 is not visible to the user, it may be sensed by the program (using the %Z special variable).

If, when a Graphics command is executed, the screen is not in the graphics mode, the screen is put into the graphics mode and cleared, the cursor is put at home position, THETA is set to zero and the pen color is set to YELLOW. This same initialization of the graphics parameters will occur also upon the following conditions:

Power-up.
RESET or return from DOS.
Run command.

For all commands that draw lines or set the graphics cursor, if the cursor or line leaves the bounds of the visible graphics screen, the position will be maintained within a +/- 32767 segment address space; whenever the cursor and/or a line re-enters the visible graphics screen, drawing will resume. The graphics sub-commands will be described in the paragraphs that follow:

PEN -- Pen color select.

Sets the drawing pen to one of the five options provided.

<sub-command> ::= PEN <color>
<color> ::= RED|YELLOW|BLUE|ERASE|UP

Example: PEN RED

QUIT -- Quit graphics mode.

Returns the screen to the text screen.

<sub-command> ::= QUIT

Example: QUIT

GOTO -- Move cursor to X,Y without drawing line.

Moves the cursor to the specified X,Y coordinate and plots a point at that coordinate in the current pen color. If the pen is UP, then no point will be plotted.

<sub-command> ::= GOTO <x-coordinate><sep><y-coordinate>
<x-coordinate> ::= <nexp>
<y-coordinate> ::= <nexp>

Examples: GOTO #J+3,20
GOTO 40,-10 Note that the ',' is required as '40 -10' would be processed as a single expression yielding a value of 30.

DRAWTO -- Move cursor to X,Y while drawing a line.

Moves the cursor to the specified X,Y coordinate while drawing a line of the current pen color. If the pen is UP, then no line will be drawn. THETA will remain unchanged.

<sub-command> ::= DRAWTO <x-coordinate><sep><y-coordinate>
<x-coordinate> ::= <nexp>
<y-coordinate> ::= <nexp>

Example: DRAWTO #I #J/2

FILLTO -- Move cursor to X,Y while drawing a line and filling.

Moves the cursor to the specified X,Y coordinate while drawing a line of the current pen color; in addition, blank regions to the right of the line being drawn are filled with the current pen color also. If the PEN is UP, then no line or fill will be drawn.

<sub-command> ::= FILLTO <x-coordinate><sep><y-coordinate>
<x-coordinate> ::= <nexp>
<y-coordinate> ::= <nexp>

Example: FILLTO 40,-10

TURNTO -- Set polar angle to value in degrees.

Sets the polar angle THETA to the value of the expression, modulo 360.

```
<sub-command> ::= TURNTO <angle>
<angle> ::= <nexp>
```

Example: TURNTO 90

GO -- Move the cursor forward by n units.

Moves the cursor forward (along the direction specified by THETA) the number of units specified and then plots a point in the current pen color at the end point. If the pen is UP, then no point will be plotted. If the number of units specified is negative, the cursor will move backward instead of forward along THETA.

```
<sub-command> ::= GO <units>
<units> ::= <nexp>
```

Example: GO 20

DRAW -- Moves the cursor forward (along the direction specified by THETA) the number of units specified, while drawing a line of the current pen color. If the pen is UP, then no line will be drawn. If the number of units specified is negative, the cursor will move backward instead of forward along THETA.

```
<sub-command> ::= DRAW <units>
<units> ::= <nexp>
```

Example: DRAW #L

FILL -- Moves the cursor forward (along the direction specified by THETA) the number of units specified, while drawing a line of the current pen color; in addition, blank regions to the right of the line being drawn are filled with the current pen color also. If the PEN is UP, then no line of fill will be drawn. If the number of units specified is negative, the cursor will move backward instead of forward along THETA.

```
<sub-command> ::= FILL <units>
<units> ::= <nexp>
```

Example: FILL 30

TURN -- Increments the polar angle by the number of degrees specified: THETA = (THETA + increment) MOD 360.

```
<sub-command> ::= TURN <angle>
```

<angle> ::= <nexp>

Example: TURN 30

CLEAR -- Clear the graphics and text screens.

Clears both the graphics screen and the text screen window.

<sub-command> ::= CLEAR

Example: CLEAR

There are four special "read only" variables (as mentioned in section 5.1.5) associated with the Graphics command; the attributes of these variables are discussed in the paragraphs that follow:

%X returns the current value of the x-coordinate of the graphics cursor, rounded to an integer value.

%Y returns the current value of the y-coordinate of the graphics cursor, rounded to an integer value.

%Z returns the current numeric equivalent of the screen color at the current graphics cursor location. The numeric equivalents are shown below:

ERASE (background) = 0
RED = 1
YELLOW = 2
BLUE = 3

If the cursor is outside the bounds of the graphics screen, or if the screen is not in graphics mode, a value of 0 is returned.

%A returns the current value of the graphics THETA angle.

6.1.13 SOUND Command (SO) -- The Sound command enables or disables the sound generating facilities within PILOT. The operands, of which there may be up to four, specify the numeric variable(s), pointer(s) and/or numeric constants which are to be selected to produce sounds. The sound generation is accomplished by assigning a specific note on a chromatic scale to each integer value between 1 and 31, and silence to the value 0, where:

```
1 = C below middle C
2 = C# below middle C
3 = D below middle C
.
13 = middle C
.
25 = C above middle C
.
31 = F# above C above middle C
```

When the value exceeds 31, the value modulo 32 is used.

Since the hardware sound generation circuitry will support four voices in parallel, up to four voices may be specified. The hardware sound registers are updated after the excution of every PILOT statement. At that time the current value of the variables and pointers are obtained and converted to tones, with the tones changing as the variables (and pointed to data) change; the numeric constants select constant tones, as would be expected. Note that due to the nature of the implementation, pointers to addresses above 32767 will produce static tones.

```
<sound operands> ::= [<sound variable>[<sep><sound variable>
                      [<sep><sound variable>
                      [<sep><sound variable>]]]]
<sound variable> ::= <numeric variable>|<pointer>|
                      <numeric constant>
```

If no variables are present, the sound generation will be disabled.

Examples:

```
SO:#A                          Generate sounds using variable #A.
SO:#D #G #K                    Generate sounds using 3 variables.
SO:                            Disable sounds.
SO:@4096,@B764,@#P             Generate sounds using pointers.
SO:1,5,8,13                    Generete constant tones.
```

6.1.14 PAUSE Command (PA) -- The Pause command delays the indicated number of 1/60s of a second before executing the next command. This command is used to synchronize music and graphical presentations to some fixed time base as the delay is accomplished by examining the hardware generated timer rather than by a software delay loop.

```
<pause operand> ::= <nexp>
```

Examples:

```
PA:60                          Delays one second.
PA:0                           No delay.
PA:1                           Delays to the next clock tick.
PA:2                           Delays to the next clock tick + 1.
PA:#D                          Delay specified by variable #D.
```

6.1.15 CASSETTE TAPE CONTROL Command (TAPE) -- The Tape command turns the Cassette peripheral motor on or off as indicated by the supplied operand, which must be either 'ON' or 'OFF'.

```
<tape operand> ::= ON|OFF
```

Examples:

```
TAPE:ON                        Turns the Cassette peripheral on.
TAPE:OFF                       Turns the Cassette peripheral off.
```

6.1.16 CASSETTE TAPE SYNC Command (TSYNC) -- The Tape Sync command allows a PILOT program to synchronize itself to a specially prepared cassette tape which has an audio track plus synchronization information on the digital track, such as the 'Invitation to Programming' tape. The Tsync command checks the cassette motor drive status and if the motor is not on, then the command is done. However, if the motor drive is on (probably as a result of a prior 'TAPE:ON') then the Tsync command will read the cassette digital track and wait for a MARK (1) to SPACE (0) transition.

```
<tsync operand> ::= <null>
```

Example:

```
TSYNC:
```

6.1.17 I/O Commands -- The I/O commands (READ, WRITE & CLOSE) provide the ability to read data from and write data to any of the peripheral devices. The syntax and behavior of each of these commands will be described in the paragraphs to follow, but first a discussion of their common points would be in order. All three commands require a device specification as the first command parameter; this specification may be in the form of a text literal (e.g. P or C or D:ELIZA) or may be a string name (e.g. SDEVICE or $F1 or $PRINTER) where the value of the string is a valid device/filename. There is no explicit OPEN type of command provided; the first use of the name in an READ or WRITE command will attempt to OPEN the device (subject to available internal resources and the legality of the device/filename). The number of devices which may be accessed in parallel is 4 and there are no restrictions as to the mix of input and output types.

```
<device spec> ::= <text literal>|<string variable>
```

The evaluation of <device spec> must produce a valid Atari

400/800 <device/filename>.

READ Command (READ) -- The Read command allows data to be read from one of the attached peripheral devices to the accept buffer, with the data transformation rules being applied as described for the Accept command (section 6.1.2). Optionally, data may be read to either a numeric variable or a string variable as well.

```
<read operand> ::= <device spec>[<sep><input variable>]
<input variable> ::= <numeric variable>|<string variable>
```

Examples:

```
READ:C,$DATA
READ:$FILE1,#N
```

If a device end-of-file status is read, the Read command will return null data.

WRITE Command (WRITE) -- The Write command allows data to be written to one of the attached peripheral devices. The data to be written will be the evaluation of a text expression.

```
<write operand> ::= <device spec><sep><texp>
```

Examples:

```
WRITE:C,JOE IS #A YEARS OLD.
WRITE:$DEVICE #N
WRITE:P,$J32 #C ABCD.
```

CLOSE Command (CLOSE) -- The Close command is the equivalent of CLOSE in most languages. Internally, the IOCB associated with the named file is freed for use by another named file, and for some peripherals (such as the disk) special termination actions are initiated.

```
<close operand> ::= <device spec>
```

Examples:

```
CLOSE:C
CLOSE:D:STAR
CLOSE:$DEVICE4
```

Advanced I/O discussion -- The next few paragraphs discuss special aspects of the Atari PILOT I/O subsystem.

I/O Errors -- Any I/O error will normally cause termination of a running program and will also cause the file in error to be closed. An error message will be generated which will inform

the user of the type of error, as shown in Appendix C.

However, all of the above activities may be inhibited (in run mode) by setting location $0500 (1280 decimal) to any non-zero value. The status of the prior READ, WRITE or CLOSE command may then be checked by examining location $00E4 (228 decimal); a value of 0 or 1 indicates that the operation was normal, and values 128-255 indicate I/O errors as shown in Appendix C. On READ errors, an EOL character is returned; WRITE and CLOSE errors are simply ignored. The inhibit flag is ignored in immediate mode and all immediate mode I/O errors will produce an error message.

The inhibit flag is set to zero by PILOT upon the following conditions:

Power-up.
RESET.
Any reported error.
Return to immediate mode from run mode.
Immediate mode BREAK.

End-of-file -- End-of-file status is not considered to be an error, and will result in null data being read. Since the I/O status is not easily accessible via a PILOT construct, the user should write some extra data at the end of any file to be read by PILOT, to indicate the EOF to the program doing the reading of the file. See the program example below:

```
100 C:$FILE=C                    Set filename to Cassette.
110 C:#L=1                       Initialize loop count.
120 *LOOP1
130 WRITE:$FILE,#L               Output numeric data from 1 ...
140 C:#L=#L+1
150 J(#L<=100):*LOOP1            ... to 100.
160 WRITE:$FILE,*** EOF ***      Write user EOF data.
170 CLOSE:$FILE                  Close the file.

200 C:$FILE=C                    Set filename to Cassette.
210 *LOOP2
220 READ:$FILE,#N                Read numeric data.
230 M:*** EOF ***                Check for EOF.
240 TN:NUMBER = #N.              Print data if not EOF.
250 JN:*LOOP2                    Loop back if not EOF.
260 CLOSE:$FILE                  Close the file.
270 E:                           End of program.
```

Bi-directional I/O -- Atari PILOT I/O is inherently unidirectional; there is no OPEN command and the direction is determined by the initiating I/O command (READ or WRITE). However, some devices are bi-directional, such as the screen editor (E); and the user may want to perform concurrent reads and writes when using this class of device. This is done by specifying synonyms for the device/filename, e.g. 'E', 'E:', 'E1', and 'E1:' are all valid ways of specifying the Screen Editor. Note that each synonym is treated as a separate device by the PILOT I/O subsystem.

List of open files -- PILOT maintains a list of currently open files; this list is comprised of string variables, where the string name is the device specification appended to '@' and the string value is a single character which indicates the internal IOCB assignment. These strings are of no interest to the PILOT user except that they appear in the string variable list produced by the Dump command, and clearing the string variables using the Vnew command has the effect of closing all files. An example of these special strings is given below:

```
$@C='@'                    Cassette is assigned to IOCB 4.
$@D:ELIZA='P'              Disk file 'ELIZA' on IOCB 5.
```

Note that '@' is IOCB 4, 'P' is IOCB 5, '◆ is IOCB 6 & 'p' is IOCB 7.

Graphics conflict error -- There is a check that prevents I/O operations that would destroy the graphics screen. If the user desires to READ from or WRITE to the Screen Editor ('E') or the Display handler ('S') when the screen is in graphics mode, the first READ/WRITE (with its implied OPEN and screen clear) must occur before establishing the graphics screen, then subsequent READ/WRITE operations will be allowed.

IOCB AUX1 & AUX2 control -- Normally PILOT establishes the values for AUX1 and AUX2 of each IOCB at OPEN time, as shown below:

Command	AUX1	AUX2
READ	4	Ø
WRITE	8	Ø
LOAD	4	Ø
SAVE	8	Ø

The advanced user may force other values for these variables by writing a byte into location 1373 (decimal) for AUX1 and into location 1374 (decimal) for AUX2. PILOT will then "inclusive-or" the user supplied byte with the constant shown in the table above, and use the result in the IOCB at OPEN time. The two user controllable bytes are reset to zero as shown below:

Power-up.
RESET.
After every usage (OPEN).

6.1.18 POSITION CURSOR Command (POS) -- The Position command allows the user to control the position of the cursor while in the text screen, just as the GOTO sub-command controls the cursor for the graphics screen. The two Position operands specify a column number ranging from Ø to 39 and a row number ranging from Ø to 23; the upper left corner of the text screen being Ø,Ø.

When the screen is in graphics mode, the specified column number applies to the text window and the row number is ignored.

```
<position operand> ::= <column><sep><row>
<column> ::= <nexp>
<row> ::= <nexp>
```

Examples:

```
POS:40,2
POS:#C,#R
```

6.1.19 TRACE Command (TRACE) -- The Trace command allows the user to monitor the execution of the program while in run mode. When turned on, the trace will print to the screen each statement scanned, prior to its execution. Conditional statements will be printed whether or not the condition is true. Every trace line is preceded by the four characters '-->' in order to make the trace lines stand out from Type, Accept, Read and Write data on the screen.

```
<trace operand> ::= ON|OFF
```

Examples:

```
TRACE:ON
TRACE:OFF
```

A typical trace output to the text screen is shown below:

```
--> 10 T:HELLO, WHAT IS YOUR NAME?
HELLO, WHAT IS YOUR NAME?
--> 20 A:$NAME
JACK
--> 30 T:GOODBYE $NAME.
GOODBYE JACK.
--> 40 E:

READY
```

6.1.20 DUMP Command (DUMP) -- The Dump command is used to list to the text screen the contents of the string variable list. Shown below is an example of Dump command output:

```
$NAME='JOE'
$RESPONSE='NO '
$STRING1='I LIKE TO PLAY BALL'
$STRING2='GREEN'
$NULL=''
```

6.1.21 LOAD Command (LOAD) -- The Load command allows the user to read a previously saved PILOT program to the stored program area (see section 6.2.3 for information on how to save a program). The required operand is a text literal (or string variable) that specifies the device/filename.

```
<load operand> ::= <device/filename>
```

Examples:

```
LOAD:C                  Loads a program from the cassette.
LOAD:D:SQUARE           Loads the file 'SQUARE' from Disk #1.
LOAD:$NAME
```

If the Load command is executed in immediate mode, the stored program area is not cleared prior to the loading; if a program is in the storage area prior to a load, the two programs will be "merged".

If the Load command is executed in run mode, the stored program area is cleared prior to the loading of the specified program (and the Use stack is cleared). If the load process encounters no errors (file I/O or statement syntax) the newly loaded program will be executed without any inititialization of the program environment, except that the Use stack is cleared.

Load reports any syntax errors encountered during the load process and continues loading until either an I/O error or end-of-file is encountered. Any statement with a syntax error is not stored in the deferred program storage area.

6.1.22 CALL Command (CALL) -- The Call command allows the PILOT user to execute 6502 machine language code by having the PILOT interpreter execute a JSR to a user specified address. The 6502 code starting at that address need only execute an RTS to return to the PILOT environment. The 6502 A, X, Y and P registers are available to the called routine and need not be saved and restored. The PILOT interpreter will enable IRQ interrupts and clear decimal mode immediately upon return.

```
<call operand> ::= <nexp>
```

Examples:

```
CALL:4096               JSR to location 4096 decimal.
CALL:#A                 JSR to location specified by value of #A.
CALL:@4096              JSR indirectly through location 4096.
```

6.2 PILOT immediate mode only commands

This section describes those commands that may only be executed while the system is in immediate mode; that is, they are restricted to use by the operator.

For the immediate mode only commands, the condition field delimiter (':') may be omitted if desired. Thus, for example, either 'RUN' or 'RUN:' will be accepted as a legal form of the Run command. In addition, the following run/immediate mode commands have this same feature: TRACE, VNEW, DUMP and LOAD.

6.2.1 LIST Command (LIST) -- The List command is used to list to the text screen the current contents of the program storage area. Screen control characters will be displayed rather than being interpreted. The List command will display either the entire program area or a selected portion thereof (if beginning and ending line numbers are provided).

```
<operands> ::= [<line #>[<sep><line #>]]
<line #> ::= <numeric constant>
```

Examples:

```
LIST                        Lists all of the program area.
LIST 100 200                Lists lines 100 through 200.
LIST 500                    Lists line 500.
```

6.2.2 RUN Command (RUN) -- The Run command changes the operating mode from immediate to run; execution starts at the lowest numbered line in the program storage area. Before the stored program is started, the Use return address stack is cleared, the accept buffer is cleared, all variables are cleared, the screen is cleared and the Match result flag is set to false.

```
<run operand> ::= <null>
```

Example:

```
RUN
```

6.2.3 SAVE Command (SAVE) -- The Save command allows the user to save all, or portions of, the current PILOT deferred program to a specified external device. The required first operand is a text literal that specifies the device/filename; the optional operands specify line numbers which have the same function as for the List command (see Section 6.2.1).

```
<save operands> ::= <device/filename>[<line #>[<sep><line #>]]
<line #> ::= <numeric constant>
```

Examples:

```
SAVE P                  Prints the program to the Printer.
```

```
SAVE C              Saves the program to the Cassette.
SAVE P,200,300      Prints program lines 200 through 300.
```

6.2.4 DOS Command (DOS) -- The Dos command allows the user to leave the PILOT environment and enter the environment of the Disk Operating System Utility. If the DOS is not resident, control will be passed to the Blackboard program. PILOT may be resumed by use of the DOS 'B' command or by pressing the RESET key. The return to PILOT finds PILOT in the same state as after the RESET key has been pressed, even if the 'B' command is used to return.

```
<dos operand> ::= <null>
```

Example:

```
DOS
```

6.2.5 NEW Command (NEW) -- The New command allows the user to delete the program stored for deferred execution, to remove all string variables from storage and to zero the numeric variables; the Use stack is also cleared in the process. The memory that was used to store the program and strings is then available for any use.

```
<new operand> ::= <null>
```

Example:

```
NEW
```

6.2.6 AUTO-NUMBER INPUT Command (AUTO) -- The Auto-number Input command allows the user to enter the auto-number input mode. In that mode PILOT statements are entered from the screen and those that are error free are appended to an internally generated line number and stored to the deferred program area. The line numbers start with the number specified by the optional first operand of the command and are thereafter incremented by the value of the optional second operand. The operands, if not specified, default to the value 10.

The text screen changes from white characters on a blue background to black characters on a dark gold background when auto-number input mode is active, and reverts to the normal colors when the mode is inactive. The entry of an empty line, or the generation of an invalid line number, terminates the mode and causes a return to immediate mode.

```
<auto operands> ::= [<line #>[<sep><increment>]]
<line #> ::= <numeric constant>
<increment> ::= <numeric constant>
```

Examples:

```
AUTO                    Enter auto-number input mode.
AUTO 10,10              Same as above.
AUTO 100                Start with line 100, increment by 10.
AUTO 100,20             Start with line 100, increment by 20.
```

6.2.7 RENUMBER Command (REN) -- The Renumber command allows the user to renumber the PILOT program statements in the program storage area. The new line numbers start with the value of the optional first operand and are thereafter incremented by the value of the optional second operand. The operands, if not specified, default to a value of 10.

If during the course of the renumber process an invalid line number is generated (outside of range 0-9999), an error message is generated and the renumber process stops. The stored program is never reorganized by the Renumber command, so this condition can be easily corrected by renumbering again with different operand values. Note that if a partially renumbered program is SAVEd and then LOADed, the program will be reorganized at load time and a simple recovery will be impossible.

```
<renumber operands> ::= [<line #>[<sep><increment>]]
<line #> ::= <numeric constant>
<increment> ::= <numeric constant>
```

Examples:

```
REN                     Renumbers the program.
REN 10,10               Same as above.
REN 100                 Renumbers to: 100, 110, 120, 130, ...
REN 200,100             Renumbers to: 200, 300, 400, 500, ...
```

7.0 PILOT Message Responses

Atari PILOT produces two types of messages, informative (non-error) messages and error messages. The display format for both types is similar. Atari PILOT responds to errors by displaying the statement in error, color inverting the character at (or just beyond) the source of the error, and generating a message explaining the error. There are two classes of errors: syntax errors, which are detected by scanning the PILOT statement when it is entered, and run-time errors, which are only detected as the statement is executed. Informative messages do not invert a character in the statement, because there is no error to point out. The rest of this section itemizes the Atari PILOT messages and elaborates on them somewhat.

7.1 Syntax Errors

WHAT'S THAT -- Indicates one of the error conditions specified below.

The condition field is improperly specified or the ':' is missing from the statement.

After a specific command has been completely scanned, there are additional characters present in the statement.

The statement command name (or Graphics sub-command field) does not specify an Atari PILOT command.

The indicated character or word has no meaning to Atari PILOT within the context of the statement being examined:

'$' not followed by alphanumeric character (string var?).
'#' not followed by alphabetic character (numeric var?).
'*' not followed by alphanumeric character (label?).
'@' not followed by numeric data (pointer?).
'%' not followed by recognized character (special var?).
An unrecognized special character outside of a text literal.

The command operand flagged has one of the following problems:

Out of range value.
Incorrect data type.
Missing operand where one is required.

The numeric expression has been incorrectly specified and has one of the following errors:

Too many levels of nested parentheses.
Unmatched left paren.
Unmatched right paren.
Non-numeric operand.
Missing or incorrect numeric operator.

The Graphics sub-command operands have been scanned and one of the following conditions has occurred:

Missing ';'.
Unmatched left paren.

Too many or too few operands for a sub-command.

LINE #? -- The statement line number specified is negative or is greater than 9999. The error may occur during any of the following conditions:

Immediate mode entry of a statement for deferred execution.

Auto-number input mode entry of a statment.

Renumbering a program.

IMMEDIATE ONLY -- The command specified is allowed only in immediate mode and cannot be entered for deferred execution.

7.2 Run-time Errors

WHAT'S THAT? -- Indicates an error in a command operand that is syntactically correct but has a semantic error, such as an out of range numeric variable, assignment of data to an invalid string indirection, etc.

I/O ERROR xxx -- In the course of performing an I/O operation the I/O subsystem detected an error and returned the status indicated. See Appendix C for a list of I/O error codes.

NO ROOM -- Indicates that the requested operation could not be performed because there was not enough free memory. The creation of string variables, entry of deferred execution statements, and I/O initiation may all generate this message.

WHERE? -- Indicates that the target label for a Use or Jump command does not exist in the program storage area.

U: TOO DEEP -- The program has exceeded 8 levels of nested Use commands.

TOO MANY I/OS -- Indicates that there was an attempt to perform more than 4 concurrent READ/WRITE operations.

DIVIDE BY 0 -- A Compute command has just attempted to perform a division by zero.

OOPS -- An I/O operation involving the 'E' or 'S' device has been aborted because it would destroy the graphics screen and greatly confuse PILOT. See section 6.1.17 for a way to avoid this error.

7.3 Informative (non-error) Messages

READY -- Indicates one of the following conditions:

An End command was executed with no corresponding Use return address in the Use stack (normal program termination).

The last line in the program storage area was executed and it was not an End or a Jump command, therefore there was no next program statement to execute.

The operator has pressed the BREAK key; when this occurs while a PILOT program is running, the message is surrounded by '***'. The statement printed when a running program is stopped is the last to be executed or partially executed.

One of the immediate mode only commands has just finished execution (or VNEW or LOAD while in immediate mode).

The operator pressed the RESET key.

A PILOT program has just generated a run-time error; the error message precedes the READY message.

Appendix A -- PILOT DATA SYNTAX SUMMARY

This appendix lists the Atari PILOT data types and operators, giving an abbreviated version of the expected syntax for each.

Sub-group	Name	Syntax	Section
Statement	Line #	<Ø-9999>	4.1
	Label	*<alphanumerics>	4.2
	Comment	[<text literal><EOL>	4.6
Numeric data	Variable	#<alpha>	5.1.2
	Constant	[-]<digits>	5.1.1
	Random num	?	5.1.3
	Pointer	@[B]<constant|variable|pointer>	5.1.6
	Controller	%<alpha>[<number>]	5.1.4
	joystick	%J<Ø-3>	App F
	paddle	%P<Ø-7>	App F
	trigger	%T<Ø-11>	App F
	lightpen	%H, %V, %L	App F
	Special	%<alpha>	5.1.5
	free mem	%F	
	match	%M	6.1.3
	graphics	%X, %Y, %Z, %A	6.1.12
	Expression		5.1.7
	nexp	<numeric entity>[<operator><nexp>]	
	unary -	-<nexp>	
	eval	(<nexp>)	
Text data	Variable	$<alphanumerics>	5.2.2
	Literal	<any ATASCII character>	5.2.1
	Expression		5.2.3
	Indirection	$<text variable>	5.2.4
Operator	Arithmetic	<numeric><operator><numeric>	5.1.7
	add	+	
	subtract	-	
	multiply	*	
	divide	/	
	modulus	\	
	Logical	<numeric><operator><numeric>	5.1.7
	equal to	=	
	not equal	<>	
	gtr than	>	
	gtr/eq	>=	
	less than	<	
	less/eq	<=	

Appendix B -- PILOT COMMAND SUMMARY

This appendix lists the Atari PILOT commands, giving an abbreviated form of the operand syntax. The complete command descriptions are to be found in Section 6. The type indicates whether the command is executed in immediate mode only (I) or both run and immediate modes (R/I). The core PILOT commands are listed first, followed by the Atari extensions; note that only the core PILOT commands have one character names.

Command	Function	Operand syntax	Note	Type	Section
T	Type	<text expression>	1	R/I	6.1.1
Y	Type if match	<text expression>	1	R/I	6.1.1
N	Type if no match	<text expression>	1	R/I	6.1.1
A	Accept	[<var>][=<texp>]		R/I	6.1.2
M	Match	<match list>	1	R/I	6.1.3
C	Compute	<nvar>=<nexp> <svar>=(texp>		R/I	6.1.5
R	Remark	<comment>		R/I	6.1.6
J	Jump	<label>		R/I	6.1.7
U	Use	<label>		R/I	6.1.9
E	End	<null>		R/I	6.1.10
JM	Jump on match	<label list>		R/I	6.1.8
MS	Match Strings	<match list>	1	R/I	6.1.4
VNEW	New variables	[#|$]		R/I	6.1.11
GR	Graphics	<sub-commands>		R/I	6.1.12
SO	Sound	<vars>|<pntrs>|<consts>		R/I	6.1.13
PA	Pause	<nexp>		R/I	6.1.14
TAPE	Cassette control	ON|OFF		R/I	6.1.15
TSYNC	Cassette synch.	<null>		R/I	6.1.16
READ	I/O input	<device>[<var>]	3	R/I	6.1.17
WRITE	I/O output	<device><texp>	3	R/I	6.1.17
CLOSE	I/O complete	<device>	3	R/I	6.1.17
POS	Position cursor	<column><row>		R/I	6.1.18
TRACE	Trace execution	ON|OFF		R/I	6.1.19
DUMP	Dump string vars	<null>		R/I	6.1.20
LOAD	Load stored prog	<device>	3	R/I	6.1.21
CALL	Call assy program	<nexp>		R/I	6.1.22
LIST	List stored prog	<line#><line#>	2	I	6.2.1
RUN	Run stored prog	<null>		I	6.2.2
SAVE	Save stored prog	<device><#><#>	2	I	6.2.3
DOS	Go disk utility	<null>		I	6.2.4
NEW	Clear stored prog	<null>		I	6.2.5
AUTO	Auto-number input	<line#><line#>	2	I	6.2.6
REN	Program renumber	<line#><line#>	2	I	6.2.7

Notes:

1. The entire operand is evaluated as a text expression before command scanning of the parameters commences.

2. Line numbers must be numeric constants only.

3. <device> may be a text literal or a string variable.

The Graphics command sub-commands and operand syntax are shown below

PEN	Pen color select	RED|YELLOW|BLUE|ERASE|UP
QUIT	Quit graphics mode	
GOTO	Move cursor	<x-coord><y-coord>
DRAWTO	Draw line	<x-coord><y-coord>
FILLTO	Draw line & fill	<x-coord><y-coord>
TURNTO	Rotate	<angle>
GO	Move cursor relative	<units>
DRAW	Draw line relative	<units>
FILL	Fill line relative	<units>
TURN	Rotate relative	<angle>
CLEAR	Clear screen	

Appendix C -- PILOT I/O ERROR CODES

This appendix lists the Atari 400/800 system I/O error codes within the context in which they will be seen in Atari PILOT. Not all of the the system codes are presented here, because some of them cannot occur within the PILOT environment.

130 A non-existent device was specified.

131 A READ command followed a WRITE command with the same device specified.

135 A WRITE command followed a READ command with the same device specified.

136 End of file condition.

138 Device timeout; device doesn't respond. (Note 1)

139 Device NAK. (Note 1)

140 Serial bus framing error. (Note 1)

141 Screen cursor out of range (READ from or WRITE to 'S').

142 Serial bus data frame overrun. (Note 1)

143 Serial bus data frame checksum error. (Note 1)

144 Device DONE error. (Note 1)

145 Disk read after write compare error. (Note 1)

146 Function not implemented for device (e.g. OUT:K).

147 Insufficient RAM for operating the graphics screen.

160 Disk drive # error.

161 Too many concurrent disk files being accessed.

162 Disk is full (no free sectors).

163 Fatal system data I/O error.

164 File # mismatch. (Note 1)

165 Disk file naming error.

167 Disk file locked.

169 Disk directory full (64 files).

170 Disk file not found in directory.

Note 1 -- These errors indicate problems over which the user has no direct control; they are due to hardware problems and should seldom be seen.

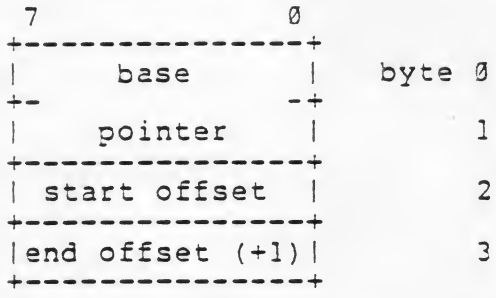
Appendix D -- SIGNIFICANT MEMORY ADDRESSES

This appendix provides the addresses of many of the Atari PILOT interpreter's internal variables, buffers, pointers and stacks. The knowledgeable user may be able to play some interesting tricks using these elements. For more information regarding the implementation of the Atari PILOT interpreter see the ATARI PILOT INTERNAL SPECIFICATION.

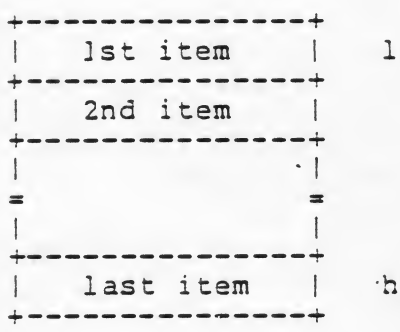
Address decimal	Address hex	Length (bytes)	Content
144	0090	1	Use stack index (1 byte pointer).
1291	050B	16	Use stack.
1307	051B	52	Numeric variables (#A to #Z).
182	00B6	2	Variable/pointer address.
184	00B8	2	Numeric item value.
147	0093	2	Expression value.
174	00AE	2	Pointer to start of program area.
176	00B0	2	Pointer to end of program area.
132	0084	4*	Pointer to next statement to execute (run mode).
178	00B2	2	Pointer to start of string list.
180	00B4	2	Pointer to end of string list.
190	00BE	4*	String NAME pointer.
194	00C2	4*	String VALUE pointer.
140	008C	4*	Pointer to text expression buffer.
1399	0577	255	Text expression evaluation buffer.
128	0080	4*	Pointer to command input buffer.
1654	0676	123	Command line input buffer.
136	0088	4*	Pointer to accept buffer.
1280	0500	1	I/O error disable flag.
228	00E4	1	I/O status byte.
1373	055D	1	User AUX1 byte.
1374	055E	1	User AUX2 byte.
1363	0553	1	Graphics pen color.
1777	06F1	1	Graphics screen mode.
1778	06F2	14	Spare bytes for user.

The 4-byte pointers flagged with '*' above have an internal format as shown on the following page:

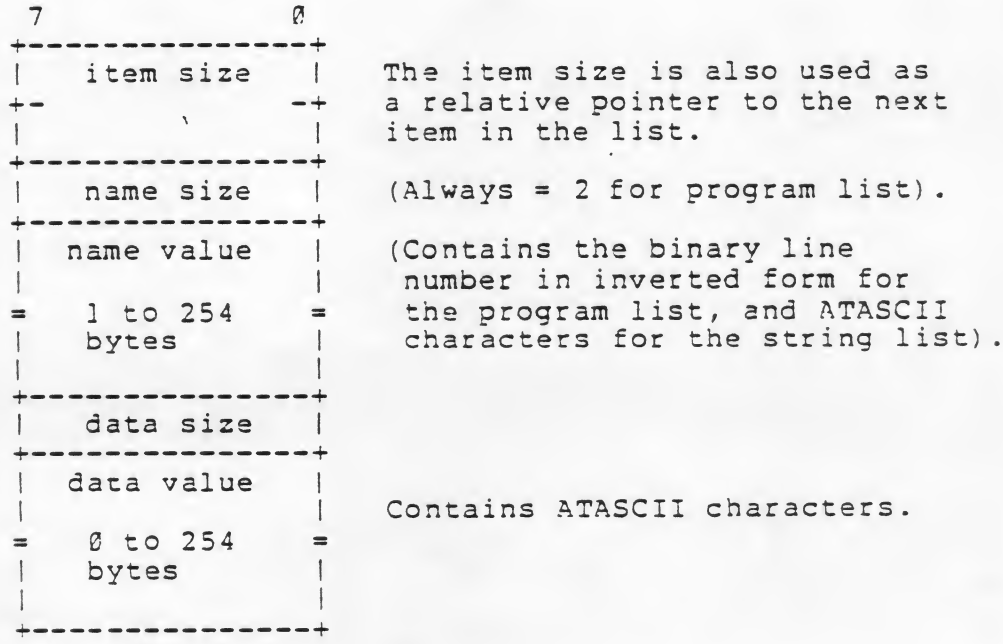
Atari PILOT interpreter 4-byte pointer format:



Atari PILOT interpreter string list & program list format:



Where each item has the format shown below:



Appendix E -- MODE CHANGE BEHAVIORS

This appendix describes the changes in the PILOT environment that may occur as the result of various events of consequence.

Event	Quit Graphics mode?	Close RD/WRT files?	Clear Vars?	Clear Program?	Clear screen?
POWER-UP	YES	Note 1	YES	YES	YES
RESET & WARMSTART from DOS	YES	Note 1	NO	NO	YES
BREAK	NO	Note 2	NO	NO	NO
I/O ERROR	NO	Note 3	NO	NO	NO
RUN:	NO	YES	YES	NO	YES
U: (immed)	NO	NO	NO	NO	NO
J: (immed)	NO	NO	NO	NO	NO
E: end	NO	YES	NO	NO	NO
E: (immed)	NO	Note 4	NO	NO	NO
Program off end	NO	NO	NO	NO	NO
Run-time err	NO	NO	NO	NO	NO
GR:QUIT	YES	NO	NO	NO	YES
NEW:	NO	YES	YES	YES	NO
VNEW:$	NO	YES	$ only	NO	NO
VNEW:#	NO	NO	# only	NO	NO
VNEW:	NO	YES	YES	NO	NO
LOAD (immed)	NO	NO	NO	NO	NO
LOAD (run)	NO	NO	NO	YES	NO
Line insert/ delete	NO	NO	NO	NO	NO

Note 1 -- All READ/WRITE files are terminated, but not formally closed; in some cases, information (or complete files) may be lost.

Note 2 -- If the BREAK causes an I/O error, then the affected file will be closed; if the BREAK does not cause an I/O error, then the file will not be closed.

Note 3 -- Only the file causing the error will be closed.

Note 4 -- YES if Use stack is empty, else NO.

Event	Clear accept buffer?	Clear Match flag?	Clear Use stack?	Stop Cassette motor?	Clear sounds?
POWER-UP	YES	YES	YES	YES	YES
RESET & WARMSTART from DOS	YES	YES	NO	YES	YES
BREAK	NO	NO	NO	YES	YES
I/O ERROR	NO	NO	NO	YES	YES
RUN:	YES	YES	YES	NO	Note 1
U: (immed)	NO	NO	YES	NO	NO
J: (immed)	NO	NO	NO	NO	NO
E: end	NO	NO	NO	YES	YES
E: (immed)	NO	NO	NO	Note 2	Note 2
Program off end	NO	NO	NO	YES	YES
Run-time err	NO	NO	NO	YES	YES
GR:QUIT	NO	NO	NO	NO	NO
NEW:	NO	NO	YES	NO	Note 1
VNEW:$	NO	NO	NO	NO	NO
VNEW:#	NO	NO	NO	NO	Note 1
VNEW:	NO	NO	NO	NO	Note 1
LOAD (immed)	NO	NO	NO	NO	NO
LOAD (run)	NO	NO	YES	NO	NO
Line insert/ delete	NO	NO	YES	NO	NO

Note 1 -- Does not clear sound selects, but does zero the numeric variables, which may make sounds silent.

Note 2 -- YES if the Use stack is empty, else NO.

The Trace flag is cleared only by TRACE OFF, power-up and RESET.

Appendix F -- ATARI CONTROLLER CHARACTERISTICS

This appendix describes the manner in which the Atari controllers are sensed in Atari PILOT.

Paddles -- There are up to eight paddles that can be sensed (%P0 through %P7); each paddle yields a numeric value from 0 (for full counterclockwise rotation) to 227 (for full clockwise rotation).

Joysticks -- There are up to four joysticks that can be sensed (%J0 through %J3); each joystick yields a numeric result, depending upon the joystick position, as shown below:

Triggers -- There are twelve triggers that can be sensed, eight paddle triggers (%T0 through %T7) and four joystick triggers (%T8 through %T11); each trigger yields a value of 0 if not pressed or 1 if pressed.

Lightpen -- A single lightpen may be sensed using the three special variables shown below:

%H = lightpen horizontal position.
%V = lightpen vertical position.
%L = lightpen trigger (0 if not pressed, 1 if pressed).

Any executed reference to one of the lightpen special variables will change the background (ERASE) color from black to light gray so as to allow the lightpen to operate at all regions of the screen.

See Appendix G, item 7 for more information regarding the Atari lightpen.

Appendix G -- APPLICATION NOTES

This document contains a collection of techniques and tricks that enable the user to perform functions that might not otherwise appear possible in Atari PILOT. This is by no means comprehensive, but merely represents the solutions to some of the problems that have been encountered.

1. DISPLAYING THE CONTENT OF THE ACCEPT BUFFER

To display the content of the Accept buffer while in immediate mode, enter the following commands:

```
MS:,
DUMP
```

The string named '$RIGHT' will have as a value the then current content of the Accept buffer and the Accept buffer will remain unaltered.

2. SAVING AND RESTORING THE ACCEPT BUFFER

The Accept buffer may be saved and restored using the techniques shown below:

```
*SAVEACCEPT
MS:,
C:$ASAVE=$RIGHT
E:

*RESTOREACCEPT
A:=$ASAVE
E:
```

3. USING STRING ARRAYS

While string arrays are not a separate syntactic element within Atari PILOT, they may be simulated by using string indirection. Strings may be created which are the concatenation of text and numbers and then if those strings are used as the names of other strings, the function of string arrays will be provided. One advantage for many applications is that the amount of space allocated corresponds to the total of the sizes of each individual string stored to the array, rather than to the subscript range and/or to the maximum string length; a definite advantage for sparse array and variable length string applications.

```
C:#S=5
C:$NAME=STRING#S
C:$$NAME=DATA PORTION
```

The result of this sequence will be to produce a string named 'STRING5' which will have the value 'DATA PORTION'.

4. CONVERSION OF TEXT/NUMERIC DATA

The Compute statement allows the data type assignments shown

below:

```
C:<numeric variable>=<numeric expression>
C:<string variable>=<text expression>
```

The Accept statement allows the data type assignments shown below:

```
A:<numeric variable>=<text expression>
A:<string variable>=<text expression>
```

Thus, between the two statements, the four combinations of data assignments shown below are possible:

NUMERIC TO NUMERIC

```
C:#A=#B+1       source may be a numeric expression.
A:#A=#B         source may be numeric variable or constant.
```

TEXT TO TEXT

```
C:$ABC=YOUR NAME IS $NAME.
A:$ABC=YOUR NAME IS $NAME.
```

NUMERIC TO TEXT

```
C:$VALUE=#X     source may be numeric variable or constant.
A:$VALUE=#X     source may be numeric variable or constant.
```

TEXT TO NUMERIC

```
A:#V=$VALUE     source contains a number as part of text.
```

5. NUMERIC/TEXT STACK SIMULATION

A numeric stack may be simulated by concatenating numeric data to a string; depending upon whether the new data is concatenated to the beginning or the end of the string, either a LIFO stack or FIFO buffer may be simulated.

First we shall examine a LIFO stack simulation. Shown below are three routines which perform initialization, push and pop operations, using the string named 'STACK' as the simulated stack, and the numeric variable #D for the stack data. A period ('.') will be used to terminate each stack entry, an exclamation point ('!') will terminate the stack for overflow detection purposes and the Match Flag will be false when an attempt is made to pop data from an empty stack or when the stack overflows.

```
*INITSTACK
C:$STACK=!
E:

*PUSH
A:$STACK=#D.$STACK
M:!
TN:STACK OVERFLOW.
E:
```

```
*POP
A:=$STACK
MS:.
TN:STACK UNDERFLOW.
EN:
A:#D=$LEFT
C:$STACK=$RIGHT
E:
```

Next we shall examine a FIFO buffer simulation. The very same techniques are used here as for the LIFO example, although the subroutine and string names have been changed. The only logic differences are to be found between the *PUSH and the *WRITEFIFO routines.

```
*FIFOINIT
C:$FIFO=!
E:

*WRITEFIFO
A:=$FIFO
MS:!
CY:$FIFO=$LEFT#D.!
MY:!
TN:FIFO OVERFLOW.
E:

*READFIFO
A=$FIFO
MS:.
TN:FIFO EMPTY.
EN:
A:#D=$LEFT
C:$FIFO=$RIGHT
E:
```

Buffer/stacks of non-numeric data may be handled in a similar manner using string names or string data in place of the numeric variable #D. Stacking the string names is preferable, where the string value is static once assigned, assuming that the name is shorter than the value, because the stack may then contain more entries before overflow occurs than when the string values are stacked.

6. TOKENIZING A TEXT STRING

A text string may be broken up into words (tokens) by a left to right scanning technique as shown below; the sample program will accept any text, and then print out the individual words within the text (in single quotation marks).

```
T:Please enter a line of text.
A:
*LOOP
MS:→_                [ skip over 1st blank and match on 2nd.
EN:                  [ nothing left -- all done.
T:'$LEFT'            [ this is the next word.
A:=$RIGHT            [ put the remainder to the acc. buff ...
J:*LOOP              [ ... and continue scanning.
```

A text string may be broken up into single characters (excluding blanks) by a scanning technique shown below:

```
T:Please enter a line of text.
A:
MS:,                  [ get accept buffer to $RIGHT, ...
A:=$RIGHT!            [ ... append ! to end & store back.
*LOOP
MS:→→,                [ skip over blank and char of interest.
EN:                   [ nothing left -- all done.
MS:$RIGHT             [ $LEFT will contain blank and single char.
C:$SAVE=$MATCH        [ save remainder as we will clobber acc. buff.
A:=$LEFT              [ accept buffer contains blank, char, blank.
MS:→                  [ skip 1st blank & match on 2nd (last) blank.
T:'$LEFT'             [ aha!  here is our character.
A:=$SAVE              [ restore the accept buffer ...
J:*LOOP               [ ... and continue scanning.
```

7. READING THE LIGHTPEN

A subroutine to read the lightpen and convert the position to PILOT graphics coordinates is shown below:

```
*GETPT J(%L=0):*GETPT
C:#H=%H
C:#X=#H-152
C(#H<6):#X=#X+227
C:#Y=64-%V
E:
```

8. SIN/COS VALUES

The SINE and COSINE functions can be derived from the graphics screen capabilites, as shown below:

```
T:Please enter an angle (in degrees).
A:#A
GR:GOTO 0,0; TURNTO #A; DRAW 10000
T:Sine(%A) = %X E-4.
T:Cosine(%A) = %Y E-4.
E:
```

9. PEEK/POKE MAGIC

The following examples show interesting things that can be performed using PILOT pointers.

pen color

The pen color can be altered without using the GR:PEN xxx command by storing a byte value to location 1363 (decimal).

```
C:@B1363=0            is equivalent to GR:PEN ERASE.
C:@B1363=1            "       "       "  GR:PEN RED.
C:@B1363=2            "       "       "  GR:PEN YELLOW.
C:@B1363=3            "       "       "  GR:PEN BLUE.
C:@B1363=4            "       "       "  GR:PEN UP.
C:@B1363=?\4          is a random pen color selection.
C:@B1363=@B1363+1\4   selects the next color.
```

reassigning color registers

The actual colors assigned to each pen color name may be altered by poking a byte value to one of four color registers.

```
C:@B712=xx        changes the color assigned to ERASE.
C:@B708=xx           "     "    "       "      "  RED.
C:@B709=xx           "     "    "       "      "  YELLOW.
C:@B710=xx           "     "    "       "      "  BLUE.
```

The form of the color register byte is shown below:

```
 7 6 5 4 3 2 1 0
+-+-+-+-+-+-+-+-+
| color | lum |0|
+-+-+-+-+-+-+-+-+
```

Where: color

```
 0 = gray
 1 = light orange
 2 = orange
 3 = red orange
 4 = pink
 5 = purple
 6 = purple-blue
 7 = blue
 8 = blue
 9 = light blue
10 = turquoise
11 = green-blue
12 = green
13 = yellow-green
14 = orange-green
15 = light orange
```

lum

```
0 = minimum luminance
1 =       |
2 =       |
3 =   (increasing
4 =    luminance)
5 =       |
6 =       |
7 = maximum luminance
```

A color register value can be calculated as shown below:

```
C:@B708 = (#C * 16) + (#L * 2)    [ #C = color, #L = lum.
```

finding free space for pointer work

The unused RAM memory within the Atari PILOT environment is defined by two addresses: 1) the lowest free memory address is contained in the word at location 176 (decimal) and 2) the highest free memory address is contained in the word at location 178 (decimal). Note that Atari PILOT adds stored program statements to the low memory region (causing the address at location 176 to increase) and adds named strings to the high memory region (causing the address at 178 to decrease). Since a PILOT program can add to the string storage requirements, but can't alter the program size (except through the use of LOAD), the low memory region is the safer area from which to start allocating.

```
C:#L=@176          [ free memory low address to #L.
C:#H=@178          [ free memory high address to #H.
```

stop execution from within a Use routine

If it is desired to immediately stop execution from within a Use routine, the following sequence may be employed:

```
C:@B144=0
E:
```

Storing a zero to byte 144 (decimal) has the effect of clearing the Use stack, thus making the E: that follows appear to be an outer level Exit.

changing the screen margins

The left and right text screen margins may be altered as shown below:

```
C:@B82=0          left margin = 0.
C:@B83=39         right margin = 39.
```

The default values for the left and right margins are 2 and 39, respectively, but other values may be set at the user's discretion.

10. READING THE KEYBOARD

The console keyboard may be read directly using the program shown below:

```
*WAIT J(@B764=255):*WAIT  [ wait for a keystrike.
C:#C = @B764              [ save the keycode.
C:@B764 = 255             [ clear the code just read.
```

The keycode read by this program is not ATASCII and cannot be readily converted to ATASCII by an algorithmic process. If conversion is required, a lookup technique (using the Match command or a pointer) would do the trick. The keycodes may be obtained empirically or by referencing the ATARI PERSONAL COMPUTER SYSTEM O.S. USER'S MANUAL or the ATARI PERSONAL COMPUTER SYSTEM HARDWARE MANUAL.

11. GENERATING KEYWORDS IN TEXT EXPRESSIONS

Some difficulty may be experienced in generating certain text literals within an operand that is evaluated as a text expression, for example in a Type command. There are a couple of tricks that will allow the user to generate just about anything, based upon the fact that all "keywords" in PILOT are two or more characters in length (except for '?' which is not evaluated in a text expression anyway).

Technique #1 -- The first technique is to insert into the keyword an innocuous set of screen control characters, such as <cursor up> followed by <cursor down>.

```
T:#↑↓A                Will put '#A' to the screen.
T:$↑↓NAME             Will put '$NAME' to the screen.
```

Technique #2 -- The second technique is to create one or more named strings that contain portions of the keywords to be generated. These strings may then be concatenated with text literals or other named strings to generate the keywords.

```
C:$A=A
T:#$A                    Will put '#A' to the screen.
```

Technique #3 -- String variables names will be generated automatically just by being undefined.

```
T:$NAME                  Will put '$NAME' to the screen, if the
                         variable has never been defined.
```

12. WRITING LARGE CHARACTERS TO THE SCREEN

Using facilities explained in section 6.1.17, and in the PERSONAL COMPUTER SYSTEM USER'S MANUAL, large characters may be written to the screen as shown below:

```
C:@B1373 = 16                [ split screen select to AUX1.
C:@B1374 = 2                 [ screen mode 2 to AUX2.
WRITE:S,LARGE LETTERS.
WRITE:S,large letters.
T:Press return to continue.
A:
E:
```

13. READING A DISK DIRECTORY

Using facilities explained in section 6.1.17, and in the PERSONAL COMPUTER SYSTEM USER'S MANUAL, a disk directory may be read as shown below:

```
C:$DIR=D:*.*                 [ wildcards to read all filenames.
C:@B1373 = 2                 [ read director select to AUX1.
*LOOP
READ:$DIR $FNAME             [ read directory information.
J(@B228=136):*DONE           [ exit loop on EOF status.
T:$FNAME                     [ type directory information.
J:*LOOP
*DONE CLOSE:$DIR
E:
```

14. USING ATARI CONTROLLERS WITH PILOT GRAPHICS

The Atari contollers can be used to interactively control graphics presentations as shown by the progrm segment shown below. The example assumes that a pair of Paddle Controllers is inserted into the first controller port.

```
*LOOP
GR(%T0):CLEAR                [ clear screen on paddle 0 trigger
C:@B708=%P1                  [ reassign color of PEN RED.

 <graphics program body>

PA:%P0/100                   [ variable delay.
J:*LOOP
```

ATARI PILOT PROGRAM OPTIMIZATIONS -- 15-April-81

This memorandum describes two techniques which a PILOT programmer may wish to consider in order to increase the execution speed of an Atari PILOT program.

1. String operation optimization.

The assignment of a value to a named string causes a prior occurrence of that named string to be deleted and the new occurrence to be inserted. Since strings are stored in memory in name collation order, the time to perform this operation is a linear function of the amount of string data which precedes the new named strings. The user may optimize string handling by using naming conventions, utilizing the following rules:

a. If large numbers of strings are to be created, and the names are to be generated algorithmically, create the strings such that the names are generated in reverse collation order (e.g. $S99, $S98, $S97 .... $S10).

b. If Match String (MS:) is to be used extensively, and other named strings are to be present, make the names of these other strings higher in the collation sequence than $RIGHT (the higher of $LEFT, $MATCH and $RIGHT).

2. Jump/Use optimization.

The PILOT interpreter always searches for a label definition starting with the lowest numbered PILOT statement and working upward. The execution speed of a program which has many references to a subroutine label or loop label may be increased by positioning that label (and associated code) nearer to the beginning of the program. The use of this optimization should be balanced against the possible loss of readability of the resultant program (however, Pascal programmers may find this technique perfectly reasonable).