ATARI PILOT INTERNAL SPECIFICATION

Initial release (11-NOV-80)

Prepared by: Harry B. Stewart
NECTERIC
15816 San Benito Way
Los Gatos, CA 95030
(408) 395-6478

ATARI PILOT INTERNAL SPECIFICATION

TABLE OF CONTENTS

1. Introduction

This manual describes the implementation details of the Atari PILOT language interpreter for the Atari Personal Computer System. The Atari PILOT language is described in a separate manual entitled 'ATARI PILOT EXTERNAL SPECIFICATION'.

The PILOT interpreter resides in an 8K byte cartridge and uses all of the available RAM memory for its own static and dynamic memory requirements. Static memory requirements include:

Interpreter working variables.
Mode control and option flags.
Numeric variable storage.
Numeric expression stack.
Use stack.
Accept, text expression and command line buffers.

Dynamic memory requirements include:

Program statement storage.
String variable storage.
Graphics/text screen area.

The interpreter utilizes the Screen Editor as a source for command lines, which may be immediate commands, deferred PILOT statements or line deletions; thus, full line editing is provided before the interpreter sees the input line.

The interpreter utilizes PILOT source statements for all operations; the statements are never tokenized, compiled or transformed in any way, except that line numbers are converted to integer form for internal program storage.

2. Interpreter operation

The interpreter is at all times reading PILOT statements, but depending upon the operating mode different actions are taken. The primary operating modes and their behaviors are:

Immediate mode -- PILOT statements are read from the Screen Editor and result in either immediate execution, a storage to the program storage area or a statement deletion.

Run mode -- PILOT statements are read from the program storage area and executed.

Auto-number input mode -- PILOT statements are read from the Screen Editor, a line number is appended and the resultant statement is stored to the program storage area.

Load mode -- PILOT statements are read from a user specified device, otherwise this mode is identical to immediate mode. Normally PILOT loads from a file containing only numbered statements which are then stored to the program storage area; however, un-numbered PILOT statements contained within a loading file will be executed as encountered.

The modes will be discussed in more detail in the paragraphs that follow.

2.1 Command/statement scanning

PILOT statements are scanned on a strict left to right basis using a context independent recursive lexical analyzer which identifies and evaluates all data elements and operators. PILOT statements undergo a double scan process: first a syntax scan which rejects all syntactically incorrect statements and then an execute scan. Even immediate mode commands are double scanned to minimize the occurrence of unwanted side effects from partially executed invalid commands. PILOT statements in the program storage area are syntax checked at the time of entry (immediate mode, load mode or Auto-number input mode) and then are not syntax checked again during run mode, in order to maximize the execution speed.

The same routines are used for both the syntax scan and the execute scan with a flag indicating which type of scan is to be performed.

2.2 Immediate mode

Immediate mode is the default operating mode; when in this mode the interpreter reads PILOT commands/statements from the Screen Editor and operates upon them. Input lines consists of an optional line number and an optional PILOT statement with the following rules governing the interpreters actions:

line #	PILOT statement	resultant action
YES	YES	Store line in program storage area

YES	NO	Delete line from program storage.
NO	YES	Execute statement immediately.
NO	NO	No-operation (ignore).

2.3 Run mode

While in run mode, the interpreter executes previously stored PILOT statements from the program storage area. Run mode is entered by the immediate mode execution of a Run, Jump, Use or End command. Run mode is terminated by a run-time error, the end of program execution, an operator BREAK or system RESET.

2.4 Auto-number input mode

While in auto-number input mode, the interpreter accepts unnumbered PILOT statements from the Screen Editor, appends an internally generated line number to each statement and stores the resultant statement to the program storage area. Syntax errors are reported and the offending statement is not stored.

Auto-number input mode is entered by the immediate mode execution of an Auto command and is terminated by the operator entry of an empty line.

2.5 Load mode

Load mode is identical to immediate mode, with the exception that statements are read from a specified device/file rather than from the Screen Editor. Load mode is entered by the execution of a Load command and is terminated by an I/O error or an end-of-file condition from the device/file being read.

2.6 Graphics mode

Graphics mode is a screen mode, rather than an operating mode as discussed in the preceding paragraphs. The screen is usually in one of two modes, text mode or graphics mode. The default mode is text mode in which the screen is organized as 24 lines of 40 characters each. The user may select graphics mode by the execution of the Graphics command, in which case the screen is reorganized to a graphics screen with a 4 line text window at the bottom.

The current mode and the transitions between modes are carefully controlled and monitored by the interpreter because:

The highest available RAM location changes when going into and out of graphics mode, thus requiring a movement of the string variable list.

Different O.S. interfaces and database variables are utilized for screen I/O in the two modes.

3. Special considerations

The paragraphs that follow discuss some of the special considerations that were made during the design and implementation of the Atari PILOT interpreter.

3.1 Error reporting

In order to aid the user in the correction of syntax errors, the interpreter highlights a character, in the offending statement, that is the source of the error or is immediately to the right of a field that is incorrectly specified. The fact that the highlighting is usually to the right of the error is a consequence of the fact that the lexical analyzer never backtracks, that data type errors are detected after lexical scanning and that syntactical ambiguities may lead to the postponement of error detection.

3.2 BREAK key monitoring

The BREAK key is supposed to provide a graceful termination of any on-going process; accordingly the pilot interpreter monitors the BREAK key at the following points:

Between the scanning of each statement (all modes).
Periodically during the execution of the Pause command.
Periodically during the execution of the Tsync command.
Between the execution of each Graphics sub-command.

When an operator BREAK is detected, the interpreter stops the then current activity and returns to immediate mode execution.

3.2 Trace mode

Trace mode is a special mode which may be utilized in conjunction with run mode to aid in the debugging of PILOT programs. While in trace mode, the interpreter will write to the text screen (or window) the PILOT statement about to be executed; this is done regardless of whether or not the statement condition field evaluates to true or false.

3.3 Run-time error minimization

Some steps have been taken to minimize the possibilites for producing run-time errors in PILOT programs; among the items accounted for are:

All statements are syntax checked before being stored to the program storage area.

Accept buffer and text expression buffer results are truncated on overflow, with no error reported.

The accept buffer and command line input buffer are larger than the largest line which may be entered from the Screen

Editor.

The graphics screen cursor control includes in-bounds/out-bounds tests and line clipping to eliminate "cursor out of bounds" errors from the Display Handler.

No explicit OPEN command is provided, the device/filename and the data direction being provided by the READ and WRITE commands themselves.

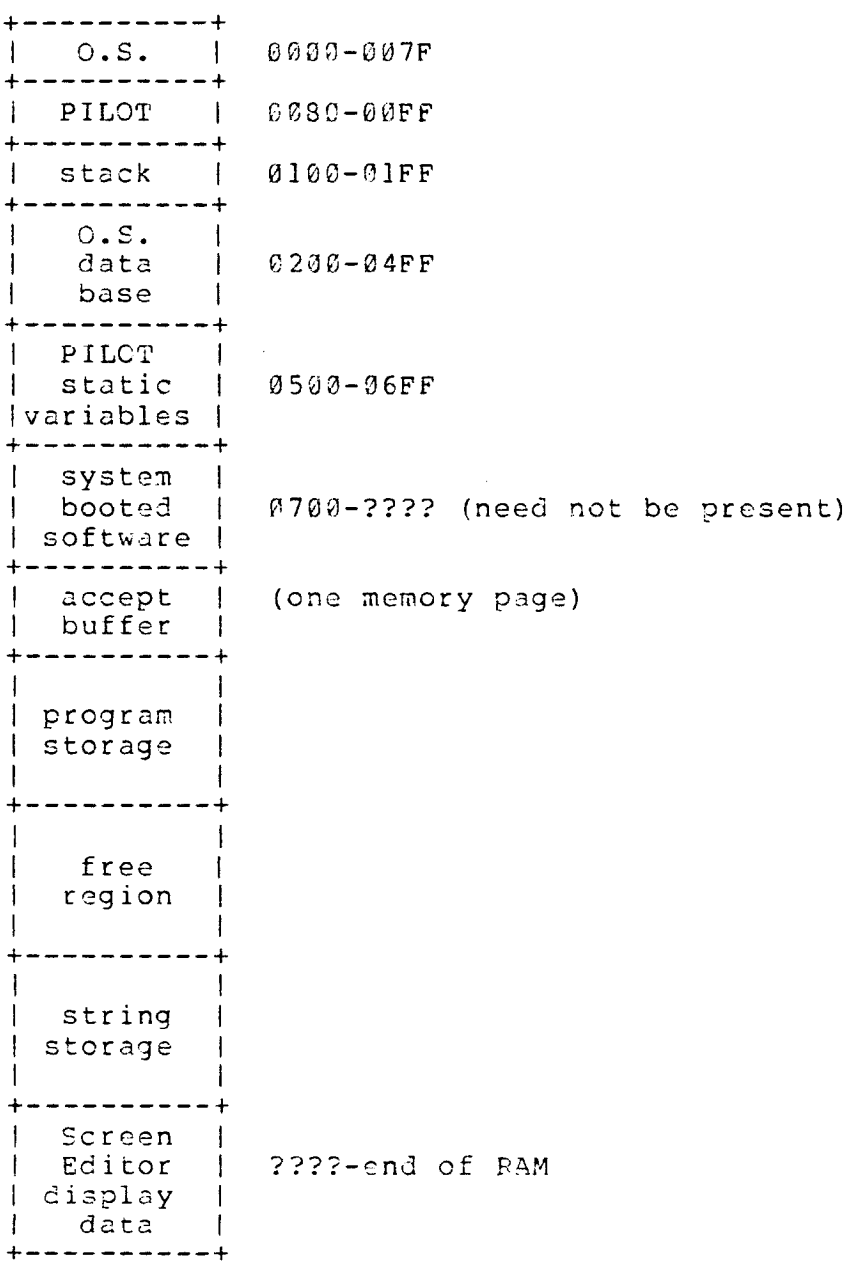
The string variable list is moved when entering graphics mode so that "insufficient memory" error will not occur except when there is really not enough memory.

The Use stack is cleared on immediate mode execution of Run, New, program statement insertion, program statement deletion and run mode execution of Load, so that the Use stack is guaranteed always to be empty or to contain valid return addresses.

String variable operations are allowed on null strings and undefined strings and no explicit declaration of string length is required (or allowed for that matter).

4. Internal data structures

This section details the internal data structures used by the PILOT interpreter. A memory map showing the gross use of RAM is provided below:



PILOT utilizes the entire second half of memory page zero (0080-00FF) for variables, pointers and tables, and in addition, utilizes pages 5 and 6 (0500-06FF) for the same. Memory page 1 contains the 6502 hardware stack and is completely reserved for that purpose.

PILOT starts the program storage area at the bottom of the free memory region at power-up time. If no disk or cassette software was booted, that address will be 0700, otherwise the address

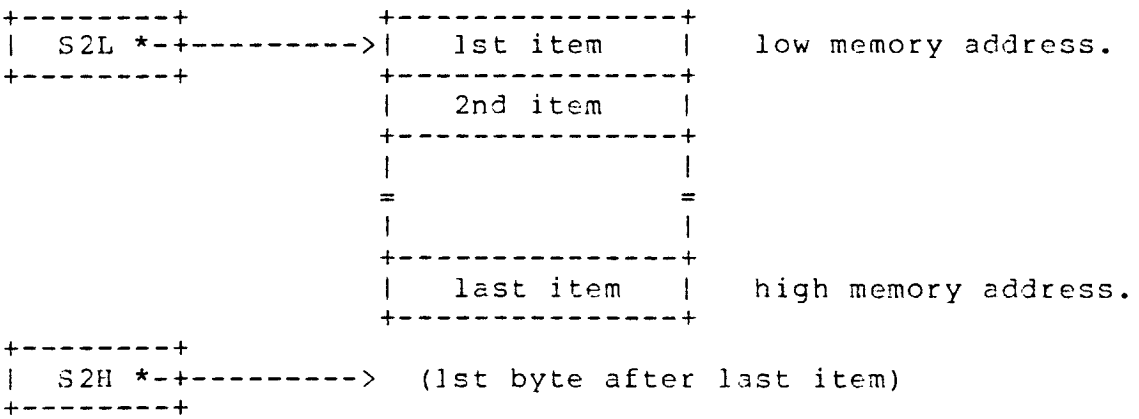
will be at the end of the booted software. PILOT starts the string storage area at the top of memory just below the display list/data region reserved by the Screen Editor.

4.1 Strings

String variables are dynamically assigned from the high address region of the free memory space downward. The variables are stored in a structure called the string list which is ordered by the collation sequence of the variable names contained therein. String names and their values are stored using standard ATASCII encoding, one character per 8-bit byte.

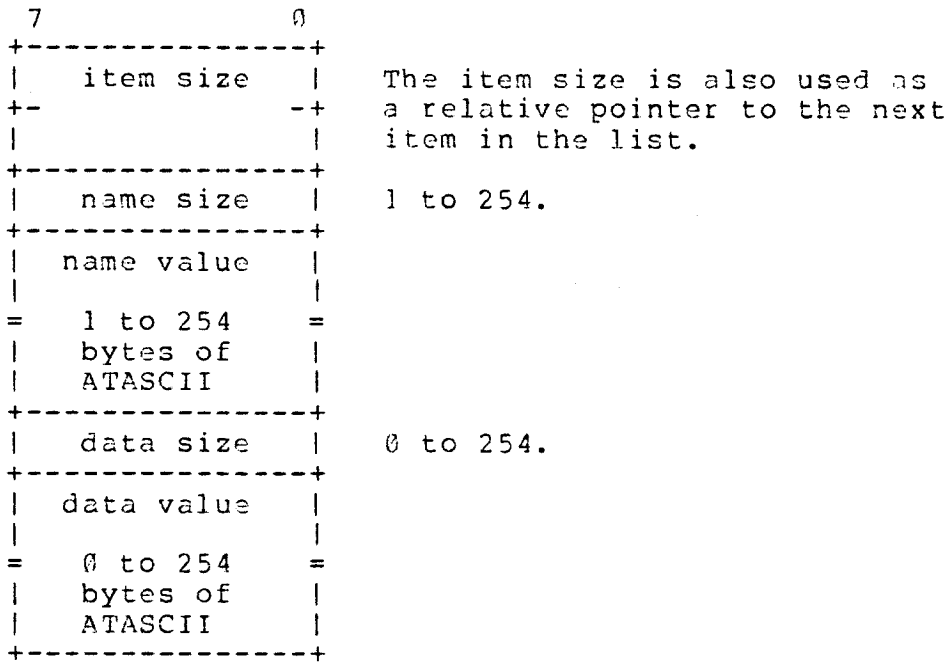
4.1.1 String storage

String variables (strings) are stored in memory using sequential storage, with two pointers demarking the beginning and end of the storage area. Pointer S2H [00B4] marks the end of the list and does not change (except when entering or exiting graphics mode); pointer S2L [00B2] marks the start of the list and changes every time an item is inserted or deleted.



When the string list is empty, S2L and S2H both point to the first unusable byte at the end of memory. The value of S2H is established from O.S. variable MEMTOP [02E5] at power-up time and is readjusted whenever there is a change in the screen mode due to a Graphics command execution.

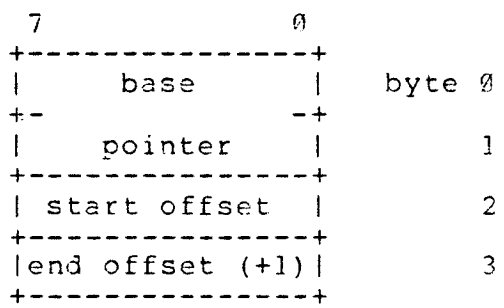
Each item in the string list has the format shown below:



The first item in the list is the variable with the name lowest in the collation sequence, with the rest of the list being ordered accordingly.

4.1.2 String pointers

When scanning and manipulating strings the interpreter utilizes 4-byte pointers which contain a 16-bit base address plus 8-bit unsigned offsets to the beginning and end of the substring being dealt with.



The pointer to a null substring will have the start offset equal to the end offset.

4.1.3 String variable name and value pointers

The lexical analyzer returns, as part of its calling sequence, a pointer to the name and a pointer to the value of each named

string variable it encounters. The name pointer is returned in 4-byte pointer variable NP [00BE] and the value pointer is returned in 4-byte pointer variable DP [00C2]; see section 4.1.2 for the format of the 4-byte pointer variables.

4.1.4 String name temporary

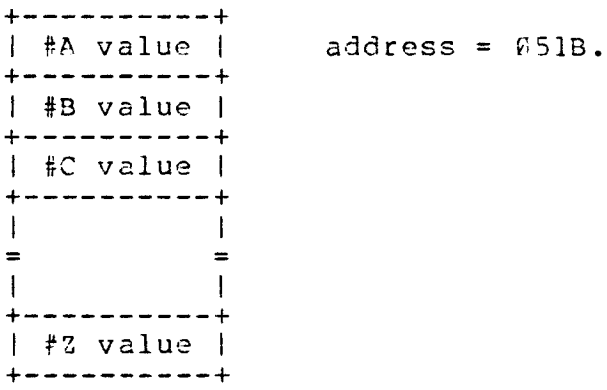
In order to solve a problem arising from the use of string indirection in the target for a Compute or Accept command (e.g. 'C:$$ABC=$DEF'), all target string names are moved to a dynamically allocated memory region prior to evaluating the text expression to the right of the '='. This region is 257 bytes in extent and is allocated upward from the top of the program list, and deallocated at the end of the command execution. If unsufficient memory is available for the allocation, the interpreter will produce a run-time error.

4.2 Numbers

All PILOT language numeric data is stored internally in 16-bit, two's complement integer form, with the least significant byte (lsb) being at the lower address. Numeric overflow may occur as a result of some numeric operations and is not considered an error by the interpreter.

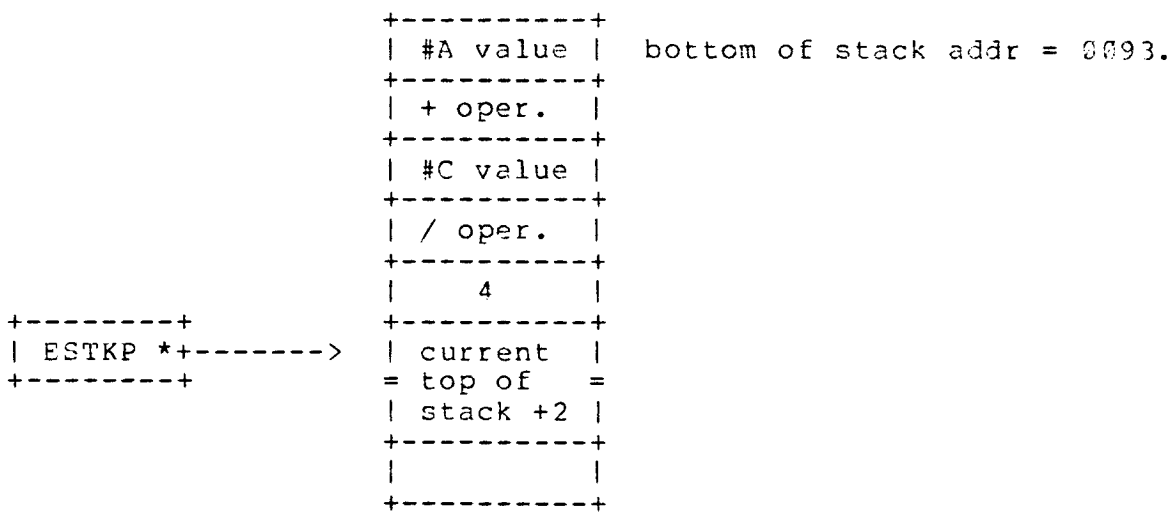
4.2.1 Numeric variables

The PILOT numeric variables '#A' through '#Z' are stored sequentially in a statically assigned table which starts at location 051B.



4.2.2 Numeric expression stack

Numeric expressions are evaluated using an expression stack which is statically assigned and has room for the partial results to support two levels of nested (non-redundant) parentheses. The stack is shown below prior to final evaluation of the expression #A + (#C / 4); notice that the stack contains the expression operands and operators in the same infix form as seen in the statement of the expression. The stack entry for each operator is the ROM address of the arithmetic routine which is to perform the diadic operation; when the operation has been completed, the three words at the then current top of stack will have been replaced with the numeric result of the operation.



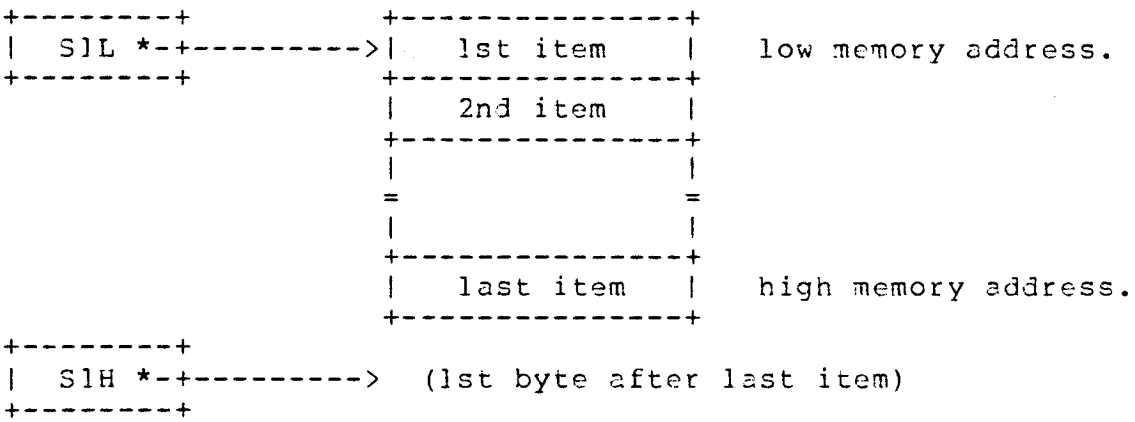
The expression stack pointer ESTKP [0091] is a single byte quantity which contains an index value to the word beyond the current top of stack (next available free word). For the case of an empty stack ESTKP contains 0 and is then incremented by 2 for every item added to the stack and decremented by 2 for every item deleted from the stack.

4.2.3 Numeric result value and address.

The lexical analyzer returns, as part of its calling sequence, the integer value and the address of each numeric variable or pointer variable it encounters. The integer value is returned in variable NUMBER [00B8] and the address is returned in variable POINT [00B6]. For special variables (%x) and numeric constants the integer value is returned in NUMBER, and POINT is not altered.

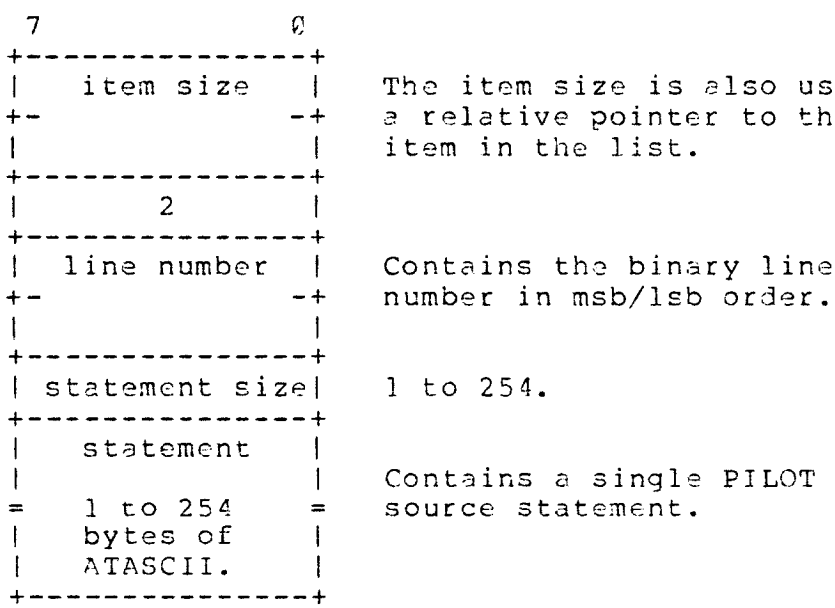
4.3 Program storage

Deferred program statements are stored in memory using sequential storage, with two pointers demarking the beginning and end of the storage area. Pointer SlL [ØØAE] marks the start of the list and does not change; pointer SlH [ØØBØ] marks the end of the list and changes every time an item is inserted or deleted.



When the program list is empty, SlL and SlH both point to the first usable byte at the beginning of the free memory region. The value of SlL is established from O.S. variable MEMLO [Ø2E7] at power-up time and is not altered thereafter.

Each item in the program list has the format shown below:



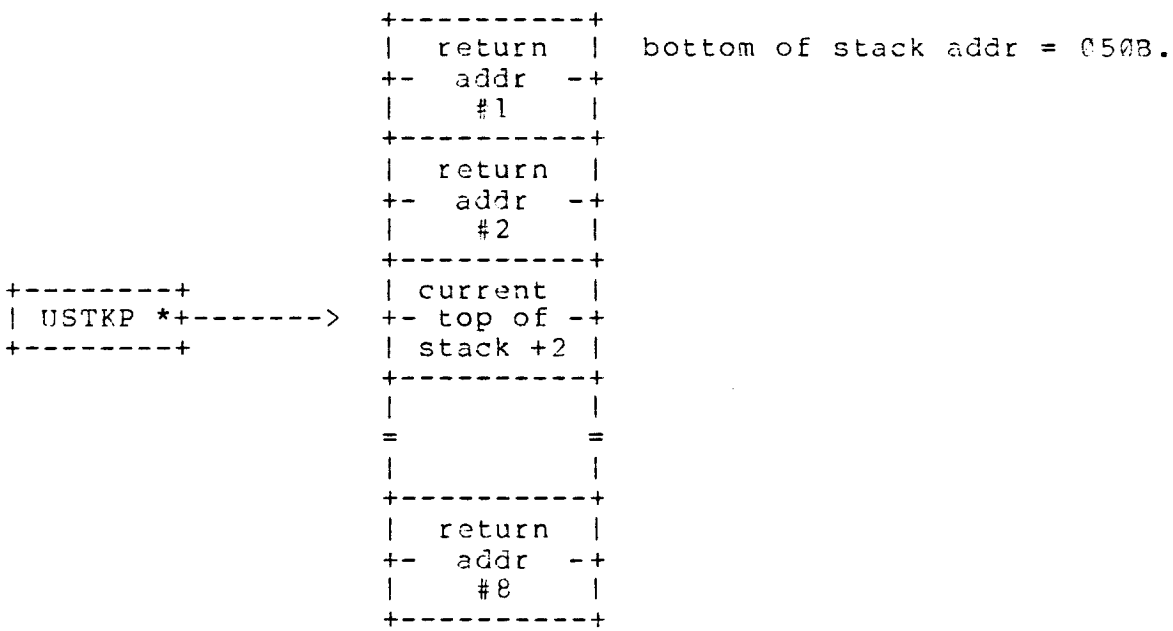
The first item in the list is the statement with the lowest line number, with the rest of the list being ordered accordingly.

The fact that a program list entry is a special case of a string

list entry is purely intentional.

4.4 Use stack

The return addresses for Use commands are retained in a stack which is statically assigned and has room for eight entries. Each stack entry is the memory address of the statement after the one containing the executed Use command.



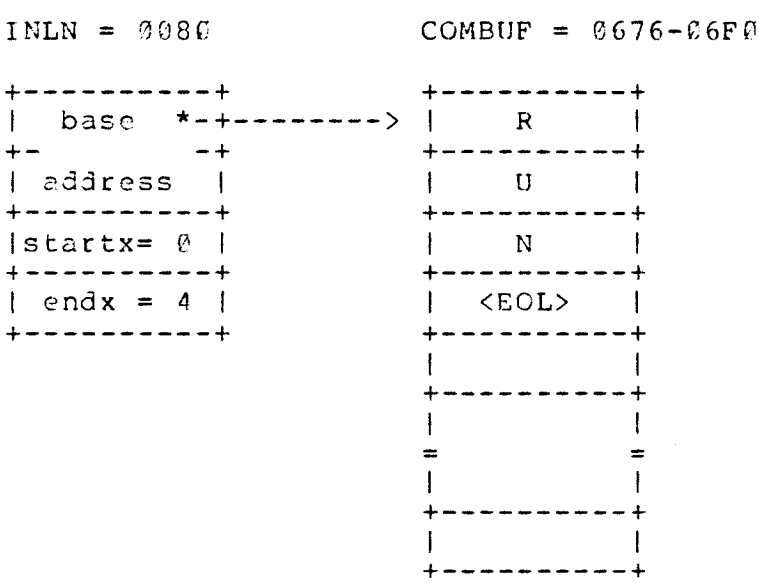
The Use stack pointer USTKP [0090] is a single byte quantity which contains an index value to the word beyond the current top of stack (next available free word). For the case of an empty stack USTKP contains 0 and is then incremented by 2 for every item added to the stack (Use command) and decremented by 2 for every item deleted from the stack (End command).

4.5 Buffers

The PILOT interpreter contains several text buffers, which are each defined and delimited by a correspoinding 4-byte pointer. Each of the buffers is described in the paragraphs that follow.

4.5.1 Command line input pointer/buffer

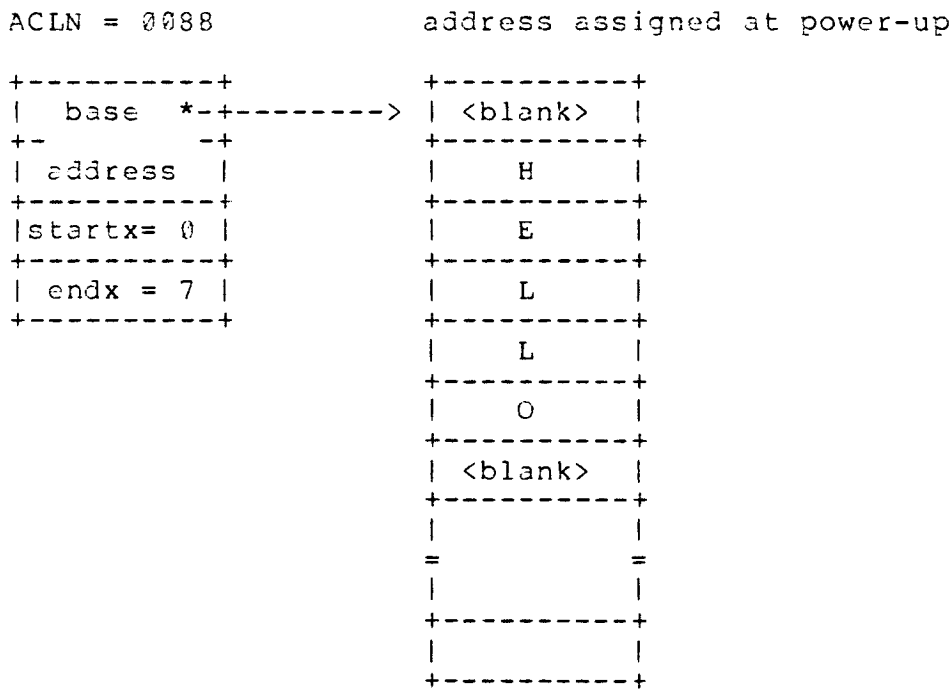
The command line input pointer INLN [0080] is a 4-byte pointer, as described in section 4.1.2, which points to and delimits the PILOT command/statement to be executed. When in immediate mode, INLN points to the buffer COMBUF [0676] which is the target for a logical line of text from the Screen Editor. The example below shows the state of the command line input pointer/buffer immediately after the input of 'RUN' from the screen.



When PILOT is in run mode, the INLN pointer points to program statements in the program storage area (instead of COMBUF). In that case the base address points to the beginning of the statement allocation and the start index contains a value of 6 which offsets the overhead bytes. See section 4.3 for the format of the program statement storage.

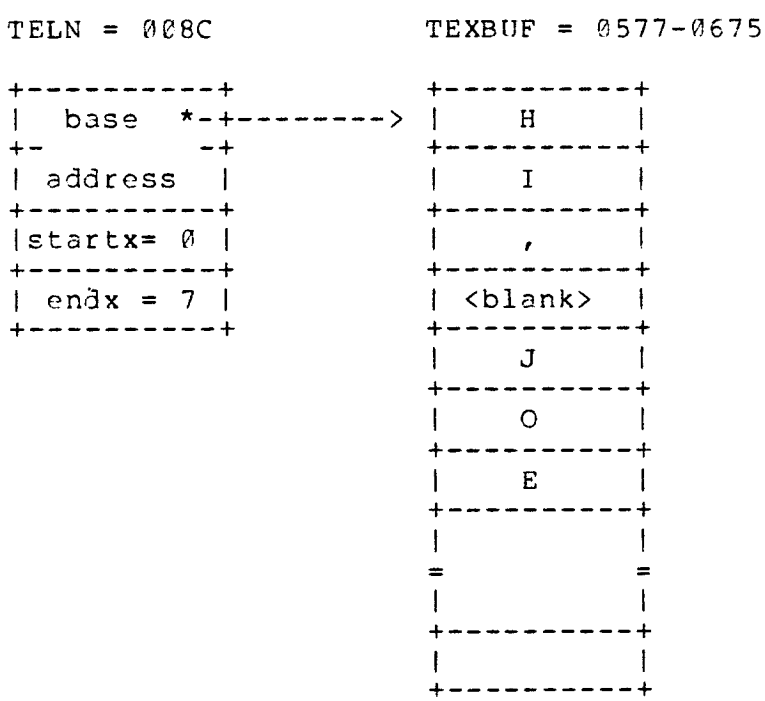
4.5.2 Accept pointer/buffer

The accept buffer pointer ACLN [0088] is a 4-byte pointer, as described in section 4.1.2, which points to and delimits the current accept buffer contents. ACLN always points to the dynamically assigned accept buffer which resides in the 256 byte page at the beginning of the free memory region. The buffer is allocated at power-up time and starts at the then current address contained in O.S. variable MEMLO [02E7]. The example below shows the state of the accept buffer pointer/buffer immediately after the execution of the PILOT statement 'A:=HELLO'



4.5.3 Text expression evaluation pointer/buffer

The text expression buffer pointer TELN [008C] is a 4-byte pointer, as described in section 4.1.2, which points to and delimits the result of the evaluation of a PILOT text expression. TELN always points to the buffer TEXBUF [0577]. The example below shows the state of the text expression evaluation pointer/buffer immediately after the execution of the PILOT statements 'C:$NAME=JOE' and 'T:HI, $NAME\'.



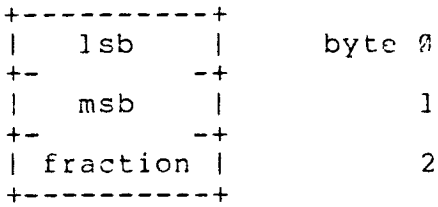
Although TEXBUF could in theory be dynamically assigned, there is code within the interpreter which accesses the buffer by name rather than using the pointer TELN, thus making dynamic assignment impossible (without coding changes to the interpreter).

4.6 Graphics parameters

The internal graphics routines use several different types of variables to perform the screen graphics computations, most of which relate to either line clipping or direction calculations.

4.6.1 X & Y coordinates

To minimize the effects of accumulating errors when performing relative cursor movement (DRAW, GC or FILL), where the cursor positions are not general integral, the graphics coordinates are maintained in memory in a 3-byte format as shown below:



Each coordinate contains two bytes of integer and one byte of fraction, all in two's complement representation. When data is either plotted to the screen, read from the screen or the coordinate values accessed using '%X' and '%Y', the most significant bit of each fraction is used to round the integer values of the coordinates (working values only, not the reference value).

The coordinate reference values are contained in variables GX [00EC] and GY [00EF]; other working variables also utilize this format as shown below:

Name	Function
GX	Current cursor x-coordinate.
GY	Current cursor y-coordinate.
GXNEW	Cursor target position x-coordinate.
GYNEW	Cursor target position y-coordinate.
GX1	Line clipping line end point 1, x-coord.
GY1	Line clipping line end point 1, y-coord.
GX2	Line clipping line end point 2, x-coord.
GY2	Line clipping line end point 2, y-coord.

4.6.2 Theta angle

The graphics theta angle is maintained internally in variable THETA [00F2] as an integer value that ranges from 0 to 359. The graphics variable '%A' returns this value directly.

4.6.3 Pen color

The currently selected pen color is retained in single byte variable PEN [0553] which may have the following values:

0 = ERASE.
1 = RED.
2 = YELLOW.
3 = BLUE.
4 = UP.

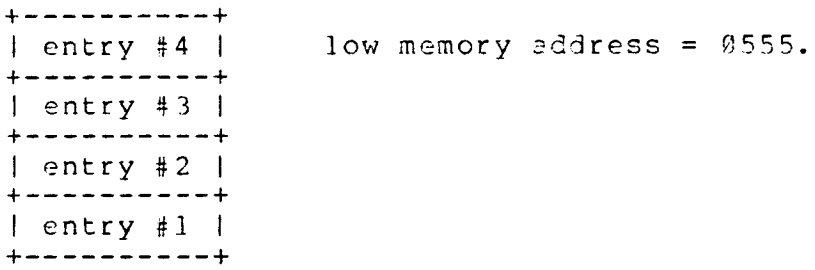
4.6.4 Computations variables

The following variables are used during the computation of line end points by the clipping algorithm:

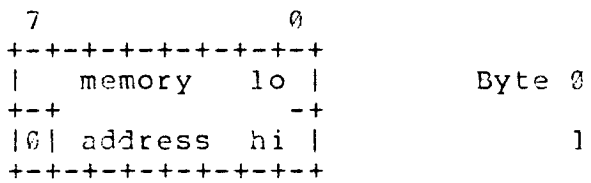
DELX	[00CA]	Integer delta x (line slope).
DELY	[00CC]	Integer delta y (line slope).
GACC	[00CE]	4 byte integer accumulator.
GTEMP	[00D2]	4 byte integer temporary.
GTEMP2	[00D6]	4 byte integer temporary.

4.7 Sounds

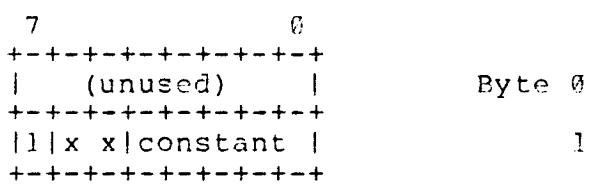
The sound parameters for the Sound command are retained in a statically allocated table as shown below:



Each table entry is a 2-byte quantity with one of the two formats shown below:



or



An entry of all zeros is used to flag the current end of the table unless the table is full. The table is scanned from the high memory address to the low memory address until the logical or physical end of table is reached.

The data value at the address specified by the table entry, or the stored constant, is truncated to a 5-bit quantity which is then used as a table index to obtain one of the 32 values shown in the table below. The value obtained from the table is sent to one of the hardware audio registers to generate the desired tone.

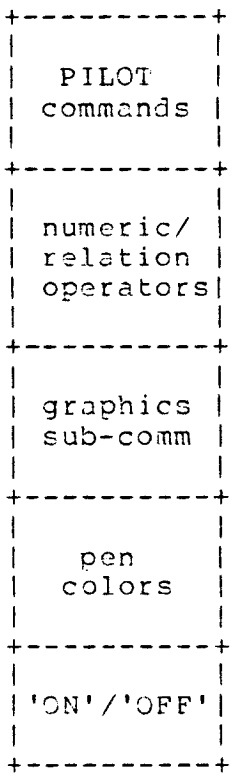
Numeric value	Audio register	Note	Freq. (desired)	Freq. (actual)	Error (%)
0	0	rest	0.0	31960.50	--
1	243	C	130.81	130.99	+0.14
2	230	C#	138.59	138.36	-0.17
3	217	D	146.83	146.61	-0.15
4	204	D#	155.56	155.90	+0.22
5	193	E	164.81	164.74	-0.04
6	182	F	174.61	174.65	+0.02
7	172	F#	185.00	184.74	-0.14
8	162	G	196.00	196.08	+0.04
9	153	G#	207.65	207.54	-0.05
10	144	A	220.00	220.42	+0.19
11	136	A#	233.08	233.29	+0.09
12	128	B	246.94	247.76	+0.33
13	121	C	261.63	261.97	+0.13
14	114	C#	277.18	277.92	+0.27
15	108	D	293.66	293.22	-0.15
16	102	D#	311.13	310.30	-0.27
17	96	E	329.63	329.49	-0.04
18	91	F	349.23	347.40	-0.52
19	85	F#	369.99	371.63	+0.44
20	81	G	392.00	389.76	-0.57
21	76	G#	415.30	415.07	-0.06
22	72	A	440.00	437.82	-0.50
23	68	A#	466.16	463.20	-0.63
24	64	B	493.88	491.70	-0.44
25	60	C	523.25	523.94	+0.13
26	57	C#	554.37	551.04	-0.60
27	53	D	587.33	591.86	+0.77
28	50	D#	622.25	626.68	+0.71
29	47	E	659.26	665.84	+1.00
30	45	F	698.46	694.79	-0.53
31	42	F#	739.99	743.27	+0.44

The formula for converting audio register values to frequency is:

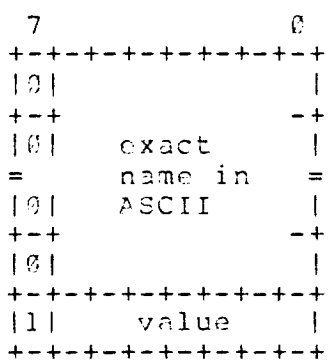
```
frequency = 63921 / (2 * (audio register value + 1 ))
```

4.8 Command/reserved word tables

All command name, numeric and relation operator, and reserved word recognition is performed using a single segmented table contained in the cartridge ROM (CTAB). The format for that table is shown below:



Each segment in the table contains one or more name entries plus a segment terminator byte of zero. A single name entry is shown below:



In some cases the 7-bit value associated with a name is sufficient, for example the graphics pen color definitions. In other cases the 7-bit value is used as an index to a table of 16-bit values (CDTAB).

5. Procedure modules

This section provides module descriptions for the major subsystems that were used to implement the Atari PILOT interpreter.

5.1 Memory management package

This package contains the routines listed below, all having functions relating to the management of memory resources:

MALLOC -- Memory allocate.
MDEALL -- Memory deallocate.
MOVIA -- Move memory block using increasing addresses.
MOVDA -- Move memory block using decreasing addresses.

This memory management scheme utilizes two separate allocation regions; one region starting in low address memory and working upward, and the other region starting in high address memory and working downward. In PILOT, one region is used for program storage and the other region for string variable storage. The regions are initially defined by four pointers:

S1L = Pointer to start (bottom) of region 1 (lower region).
S1H = Pointer to end (top) of region 1.
S2L = Pointer to end (bottom) of region 2 (upper region).
S2H = Pointer to start (top) of region 2.

S1L and S2H define the extent of managed memory and are not altered by the management routines. S1H and S2L are initially equal to S1L and S2H (respectively) indicating null allocations, and thereafter always point to the next available unused memory location. Allocation of memory is accomplished by creating a "hole" in the desired region (by moving memory if necessary); and deallocation of memory is accomplished by eliminating a "hole" in the region. An allocated block of memory always contains its own size as the first (lowest address) two bytes of the block; this is the only memory overhead associated with management. Note, however, that the block size also forms a relative pointer to the next allocation block which can be useful to the application if the blocks within a region form a sequential list.

The user always specifies the address at which allocation is to start; this address may be inside the region ("between" already allocated blocks) or may be the next available address just outside the region. The user specifies the address of the block to deallocate also, the size is assumed to still be in the first two bytes of the block.

The section that follows summarizes the calling sequences of the current routines:

MALLOC -- Memory allocate routine

Calling sequence:

'MEMA' contains the address at which the allocation is to occur.

This address is either the next available address outside the region (content of S1H or S2L) or the address of an already allocated block within the region.

'MEMB' contains the number of bytes to allocate, including the two overhead bytes.

```
JSR     MALLOC
BNE     not enough memory to satisfy allocation
```

'MEMA' contains the lowest address in the allocated block.

The first (lowest) two bytes of the allocated block contain the number of bytes in the block.

MDEALL -- Memory deallocate routine

Calling sequence:

'MEMA' contains the address of the block to deallocate. The first (lowest) two bytes of the block must contain the block size.

```
JSR     MDEALL
```

'MEMA' contains the address of the block following (higher address) the deallocated block after the deallocate.

MOVDA and MOVIA are used by MALLOC and MDEALL; they are general purpose block move utilities and their calling sequences may be obtained from the source file.

5.2 String handling package

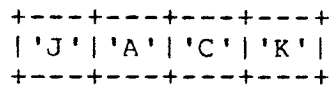
This package contains the routines listed below, all having functions relating to the handling of named strings of variable length.

SFIND -- Find named string in list.
SDELET -- Delete named string from list (if it exists).
SINSRT -- Insert named string into list (by name order).
SMATCH -- Find substring in string, if present.
SCOMP -- Compare two strings for collation.

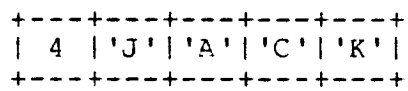
This package utilizes the memory management package described in section 5.1 and deals with two separate lists of named strings, where one list is the program list and the other is the string variable list.

In order to understand the calling sequences, a few definitions must be given first:

Text data -- any contiguous grouping of one or more bytes which are to be treated as a unit. The word 'JACK', when stored in memory as shown below, is an example of text data.



String -- text data to which a string length byte has been appended, as shown below.



Text pointer -- a four byte element consisting of a base memory address and two indices (a starting and an ending index). The text pointer explicitly delimits a (possibly null) group of bytes starting with the byte at address BASE ADDRESS + START INDEX and ending with the byte at address BASE ADDRESS + END INDEX - 1. By convention, if the start index equals the end index, the text data is considered to be null. See also section 4.1.2.

Named string -- a string which can be referenced by a symbolic name consisting of text data; a named string is often known as a string variable. The name and data portions may each be up to 254 bytes in length. See section 4.1.1 for the storage format for named strings.

Parameter area -- a portion of page zero memory which contains text pointers which have assigned meanings for each of the string operations provided by this package. These parameters are initially setup by the user and provide the parameter passing mechanism for all operations.

NP (name pointer) -- when used, this delimits a string name.

DP (data pointer) -- this delimits string data; often associated with the named string specified by NP.

MP (pattern match pointer) -- when used, this delimits comparison or pattern matching data to be used in conjunction with DP.

LP (list pointer) -- points to the string list to be accessed.

The section that follows summarizes the calling sequences of the routines:

SFIND -- Find named string in list.

Function: SFIND scans a list of named strings, attempting to find the name delimited by the NP text pointer. If the named string is found, DP will point to the string data.

Calling sequence:

```
LP = address of start of list of named strings.
NP = text pointer delimiting a string name.

JSR     SFIND
BNE     name null or named string not found

DP = text pointer to string data portion, if found.  Base
     address points to string byte count (n), start index
     = 1 and end index = n+1 (assuming non-null data, else
     both indices = 1).
```

SDELET -- Delete named string, if found.

Function: Finds the named string, if it exists, removes the named string from the list and deallocates the memory utilized by that string.

Calling sequence:

```
LP = address of start of string list to access.
NP = text pointer delimiting a string name.

JSR     SDELET
BNE     name null or named string not found
```

SINSRT -- Insert named string to list.

Function: Deletes a prior occurrence of the named string, if found, and then inserts the new named string into the list. The string is placed in the list so that the names are in standard collation order.

Calling sequence:

```
LP = address of start of string list to access.
NP = text pointer to string name.
DP = text pointer to string data.

JSR     SINSRT
BNE     no room in memory for insertion
```

SMATCH -- Find substring in string if present.

Function: examine string for first occurrence of substring.

Calling sequence:

```
DP = text pointer to source text data.
MP = text pointer to match text data.

JSR     SMATCH
BNE     match text not in source text

SP = text pointer to first occurrence of match data in source.
```

SCOMP -- Compare two string for collation order.

Function: compares two strings to determine their collation order.

Calling sequence:

```
DP = text pointer to text data.
MP = text pointer to text data.

JSR     SCOMP
BEQ     DP text data = MP text data
BCS     DP text data >= MP text data
BCC     DP text data < MP text data
```

Note : The comparison is based upon the numerical encoding of the characters involved; moreover, when one text data is a subset of the other text-data, the shorter one is considered to be lower (<) in the collation sequence.

Routines IFIND, ICOMP, IMATCH, SEND, ILENG, PSETUP, PMOVE, SMOVI and SNXTI are lower level routines used to implement the ones described above. Their calling sequences may be found in the source listing.

5.3 Integer arithmetic package

Double Precision Integer (Signed) Arithmetic Package (DXXXIY)

This package contains the routines listed below, all having functions relating to two-byte signed integer arithmetic (except as noted):

Function	Routine(s)
Addition	DADDI, DADDS, DADDA
Subtraction	DSUBI, DSUBA
Multiplication	DMULI
Division/modulus	DDIVI, DMODI
Comparison	DSCMI, DCMPI, DCWCI, DCMPA
Negation	DNEGI
Relational tests	DEQTI, DNETI, DGTTI, DGETI, DLTTI, DLETI
Number to ASCII	DECASC
ASCII to number	ASCDEC
Move number	DMOVI, DLOADA, DSTORA

These routines have a common calling convention and data base. All data is assumed to be two bytes in length (low byte followed by high byte), and all data is referenced by its position relative to the symbol 'DTAB'. The sample program that follows will hopefully make this clear.

```
;
; DATA REGION
;
*=$B0                           ; ORIGIN OF DATA REGION.
;
DTAB=*                          ; START OF DOUBLE PRECISION DATA.
;
VA      *=*+2                   ; DECLARE 'VA'.
VB      *=*+2                   ; DECLARE 'VB'.
VC      *=*+2                   ; DECLARE 'VC'.
;
; PROGRAM REGION
;
*=$B000                         ; ORIGIN OF PROGRAM REGION.
;
; COMPUTE VC = (VC + VA) * (VB ** 2) - 2.
;
        LDX     #VC-DTAB        ; VC = (VC + VA) ...
        LDY     #VA-DTAB
        JSR     DADDI

        LDY     #VB-DTAB        ; ... * (VB ** 2) ...
        JSR     DMULI
        JSR     DMULI

        LDA     #-2             ; ... - 2.
        JSR     DADDS
```

The double precision subroutines do not, in general, alter the X and Y registers; and since the X register always specifies the destination of the result, great savings in code can be achieved

when computing expressions with several terms by not including the redundant LDXs and LDYs.

The section that follows summarizes the calling sequences of the resident routines.

Name	Function
DADDI	DTAB(X) = DTAB(X) + DTAB(Y)
DADDS	DTAB(X) = DTAB(X) +/- A
DADDA	'ACC' = 'ACC' + DTAB(Y)
DSUBI	DTAB(X) = DTAB(X) - DTAB(Y)
DSUBA	'ACC' = 'ACC' - DTAB(Y)
DMULI	DTAB(X) = DTAB(X) * DTAB(Y)
DDIVI	DTAB(X) = DTAB(X) / DTAB(Y)
DMODI	DTAB(X) = ABS(DTAB(X)) MOD ABS(DTAB(Y))
DNEGI	DTAB(X) = - DTAB(X)
DMOVI	DTAB(X) = DTAB(Y)
DLOADA	'ACC' = DTAB(Y)
DSTORA	DTAB(X) = 'ACC'
DSCMI	cc = DTAB(X) : DTAB(Y) (SIGNED)
DCMPI	cc = DTAB(X) : DTAB(Y) (UNSIGNED)
DCWCI	cc = DTAB(X) : A,Y (UNSIGNED)
DCMPA	cc = 'ACC' : DTAB(Y) (UNSIGNED)
DEQTI	DTAB(X) = 1 IF DTAB(X) = DTAB(Y), ELSE 0.
DNETI	DTAB(X) = 1 IF DTAB(X) <> DTAB(Y), ELSE 0.
DGTTI	DTAB(X) = 1 IF DTAB(X) > DTAB(Y), ELSE 0.
DGETI	DTAB(X) = 1 IF DTAB(X) >= DTAB(Y), ELSE 0.
DLTTI	DTAB(X) = 1 IF DTAB(X) < DTAB(Y), ELSE 0.
DLETI	DTAB(X) = 1 IF DTAB(X) <= DTAB(Y), ELSE 0.

Where: 'ACC' is a 'DTAB' variable.
'cc' refers to 6502 status register bits Z & C.
':' is the comparison operator.

5.4 Graphics package

This package contains the routines listed below, all having functions related to generating memory map graphics using the system Display handler (S:).

```
GMOVE  -- Plot point or draw/fill line (with screen clipping).
INTEST -- Test x,y point for being inside the screen limits.
MOD360 -- THETA = THETA modulo 360.
SETCUR -- Convert coordinate systems and set system cursor.
SINVAL -- Calculate SINE(THETA).
TMULT  -- Triple precision signed multiply.
TADDI  -- Triple precision signed addition.
QMULT  -- Quadruple precision signed multiply.
QDIV   -- Quadruple precision signed divide.
QNEGA  -- Quadruple precision negate.
```

Other routines exist in this package, but they are oriented directly to the syntax of PILOT graphics sub-commands; specifically, there are routines to process the following sub-commands:

```
DRAWTO x,y
DRAW n
GOTO x,y
GO n
FILLTO x,y
FILL n
TURNTO Ø
TURN Ø
CLEAR
PEN c
QUIT
```

The most interesting of the routines is GMOVE which contains a screen clipping algorithm which allows lines to be drawn anywhere within a 65535 by 65535 graphics address space, displaying the line segments which pass through the 160 by 96 visible screen. The algorithm implemented is described in section 5-1 of PRINCIPLES OF INTERACTIVE COMPUTER GRAPHICS, Second Edition.

5.5 Statement scanning utilities

The statement scanning utilites aid in the scanning of PILOT statements. In general, most of these routines expect a single character in the A register or expect the Y register to index into the line pointed to by INLN. The table below lists the primary utilities and summarizes their calling sequences.

Name	Function
CNUMBR	Check A = numeric character. Clear carry if A = '0' - '9'.
CLETTR	Check A = alphabetic character. Clear carry if A = 'A' - 'Z'.
CKEOA	Check A = end of atom (non-alphanumeric character). Set cc non-zero if A = '0'-'9' or 'A' - 'Z'.
CHKEQS	Check A = equal sign. Set cc zero if A = '='.
CHKSEP	Check A = comma or space. Set cc zero if A = ',' or ' '.
CHKTRM	Check A = statement terminator. Set cc zero if A = <EOL> or '['.
SCNEOA	Scan to end of atom (non-alphanumeric character). Y = index for pointer INLN.
SCNLBL	Scan to end of label, if present. Y = index for pointer INLN.
SLB	Skip over leading blanks. Y = index for pointer INLN.
SKPSEP	Skip over separators. Y = index for pointer INLN.
SCNEOL	Scan to end of line. Y = index for pointer INLN.