3 Memory use

Fig 3.1 shows the memory map as seen by BASIC. The memory that BASIC uses can be split up into 3 major areas: workspace, program storage, and *dynamic storage* (the HEAP and STACK).

The workspace includes most of the general memory used by statements and functions. This is described in more detail in section 3.3.

Program storage has already been described in section 2.4.

Dynamic storage is allocated while a program is actually running; whereas workspace and the program occupy fixed areas while this is going on. Dynamic storage includes the storage of variables on the HEAP, and the use of the STACK for storing temporary results, and saving things during FN or PROC calls. The HEAP and STACK are described in more detail in the next sections.

3.1 Variables and the HEAP

3.1.1 The resident integer variables

The resident integer variables, @% and A % to Z%, are not stored on the HEAP where the rest of the variables are: they occupy the lower half of page 4. Because each one occupies a fixed location, they are very fast to access. They are stored in the following format:

&400 to &403	@%
&404 to &407	A% etc.
&468 to &46B	Z%

They are stored in standard 4-byte integer format (i.e. LSB first, MSB last). Here is a short program to list the resident integer variables, and their values (in HEX).

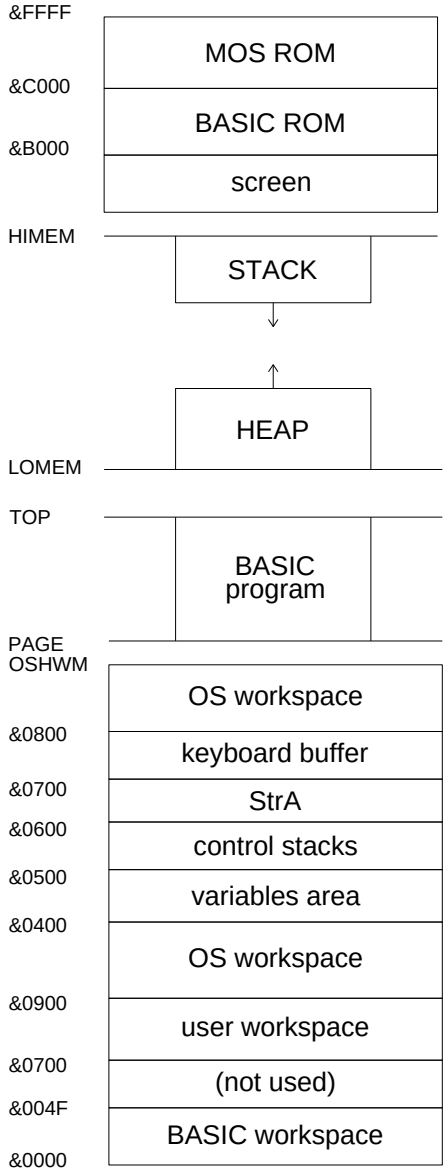

Figure 3.1 — The BASIC memory map.

```
  5 REM Prints out the resident integer variables
 10
 90 vbase = &400
100 FOR char = ASC"@" TO ASC"Z"
110   offset = (char AND &1F)*4
120   value% = vbase!offset
130   PRINT CHR$(char);"% = &";~value%
140   NEXT char
```

3.1.2 Dynamic variables

The rest of the variables used by BASIC are *dynamic* variables, because it allocates space for them when it needs it (i.e. when they are first set). These are stored on the HEAP, which works upwards in memory from LOMEM. To get at the variables once it has put them on the HEAP, BASIC uses a series of *linked* lists.

A linked list starts with a base pointer, which points to the first item in the list. The first item in the list has a pointer which points to the second item in the list, and so on. The end of the list is usually marked by the pointer to the next item being 0. So, if the linked list doesn't contain any items, the base pointer is 0 (a null pointer). Fig 3.2 shows a linked list of three items.

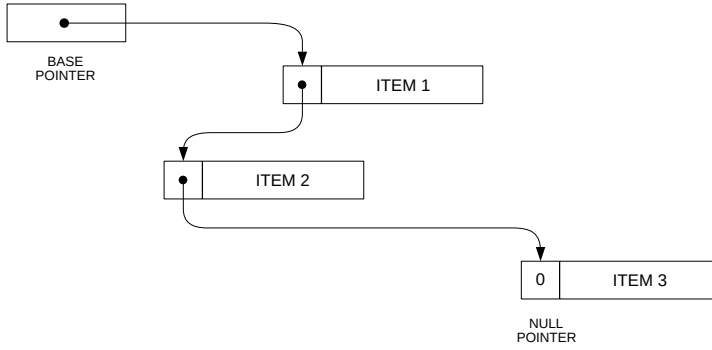

Figure 3.2 − A linked list.

One of the advantages of a linked list is that the items don't need to be in any set pattern in memory, as long as the pointers still point to the next item in the list. This can be very useful for variable storage, as different types of variables occupy a different number of bytes (especially arrays).

In fact, BASIC uses a separate linked list for each possible first letter of a variable name. Although these linked lists are separate, they all use the HEAP in the same way, and the lists link round each other. Using these separate linked lists means that searching for variables is much faster (unless your variable names all start with the same letter!).

The base pointers, which point to the first variable in each particular list, are stored in the upper half of page 4 in the following format:

&482 ,&483	base pointer for the 'A' list
etc.	
&4B4 ,&4B5	base pointer for the 'Z' list
etc.	
&4F4 ,&4F5	base pointer for the 'z' list

A similar linked list is used to store the locations of PROCs and FNs, once they have been called, so that BASIC doesn't have to search through the whole program to find them again. The base pointers for these are:

&4F6,&4F7	base pointer for the PROC list
&4F8,&4F9	base pointer for the FN list

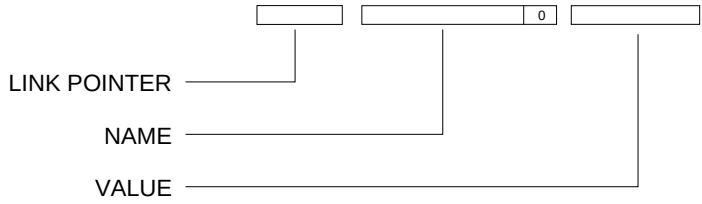

Figure 3.3 − A variable information block.

Each variable (or PROC/FN) on the HEAP is stored as a *Variable Information Block* (fig 3.3). This Variable Information Block is composed of 3 *fields*:

The pointer field (2 bytes).

> This is the pointer which points to the next item in the list (with the same first letter). If this item is at the end of the list, then the MSB of this pointer must be zero (the next item can't be in page zero, so only checking that the MSB is zero saves time).

The name field.

> This holds the name of the variable, with a zero byte to mark the end of the name. For a variable, this name field does not include the first character of the name, because that was used to choose which base pointer to use. It does contain the '$', '%' or '(' characters on the end of the name (if there are any), as this gives the type of the variable.
>
> For a PROC or FN, the first character of the name is included, as there is only one list for all PROCs, and one for all FNs.

The value field.

> This starts with the first byte after the zero byte at the end of the name field. For a variable, the format of this field depends on the type: these are detailed in section 3.1.3.
>
> For a PROC or FN, this field contains a 2-byte pointer to the PROC or FN where it is defined. It points to the first character after the name of the PROC or FN (i.e. to the '(' character if it uses any parameters).

As an illustration of the way variables are stored on the HEAP, the program below will go through the current active variables, printing their names and values. It can be used to print out Variables other than those used by the program itself, by setting them up first, and using 'GOTO 90' to start the program (if 'RUN' is used, all variables are cleared first).

The program follows the linked list for each initial letter of

variable names, using the variable ‘addr’ to hold the current pointer.

PROCvar prints out the name and value of the variable whose *Variable Information Block* (VIB) is at ‘addr’. The last character of the variable gives its type, and this is used to prevent the program from printing out arrays. To print out the value of the variable, it ‘cheats’ by giving the name of the variable to EVAL rather than extracting it directly. Section 7.4 gives a machine code version of this routine.

```
   5 REM ****** VRPRINT ******
  10 REM Prints out variables used by the program.
  15 REM If any others are to be printed, use
  20 REM "GOTO 90" so they won't be cLeared.
  90 @%=0
 100 PRINT'"Variable"TAB(15)"Value"'
 110 FOR char = ASC("A") TO ASC("z")
 120 addr = &400+2*char      :REM Get pointer address
 130 addr = !addr AND &FFFF
 131                         :REM Get ptr to 1st VIB
 140 IF (addr DIV &100)=0 THEN GOTO190
 141                         :REM Exit if null pointer
 150 REPEAT
 160   PROCvar                    :REM Print variable
 170   addr = !addr AND &FFFF     :REM Get ptr to next VIB
 180   UNTIL (addr DIV &100)=0    :REM Exit if null pointer
 190 NEXTchar
 200 END
 999 REM *** Print variable name and vatue ***
1000 DEFPROCvar
1010 name$ = CHR$(char)      :REM First character of name
1020 nptr = 2                :REM Ptr to name in VIB
1030 IF addr?nptr=0 THEN GOTO1100
1031                         :REM End of name?
1040 REPEAT
1050   name$ = name$+CHR$(addr?nptr)
1051                         :REM Add next char to name
1060   nptr = nptr+1
1070   UNTIL addr?nptr=0     :REM Exit if end of name
1100 PRINT name$,TAB(15);
1105 typ$ = RIGHT$(name$,1)  :REM Get type of variable
1110 IF typ$="(" THENPRINT"<array>" ELSEPRINT EVAL(name$)
1111                         :REM Print value if not array
1130 ENDPROC
```

3.1.3 Variable value formats

When writing programs in BASIC, variables can be one of 3 types: 4-byte integers, floating point numbers, or strings (these are called *dynamic* strings, as BASIC allocates memory for them as it is required). However, the indireciton operators ('?', '!' and '$') can be used to manipulate 8-bit bytes, 4-byte integers, and *static* strings (i.e. strings at a fixed address in memory).

Once BASIC has found the location of the variable, these bytes and static strings are treated like just like two more variable types (4-byte indirected integers are stored the same as named 4-byte integer variables). To pass variables between routines, a *Variable Descriptor Block* (not to be confused with the Variable Information Block) is used, which is usually left in IntA (the integer accumulator). The format of this is:

&2A,&2B	pointer to the location of the variable value
&2C	type of the variable

This *Variable Descriptor Block* is used, for example, in the *Parameter Block* passed by the BASIC 'CALL' statement (when any parameters are passed to it). This means that a user routine can read or set any of the variables passed as parameters to the CALL statement.

The format of the different variable types are:

Type number &00: 8-bit byte

Format:

00	8-bit byte	1 byte

This is just a single byte at the specified location. This type of variable can only be accessed by using the '?' operator; either as '?M' to mean 'the byte pointed to by M',or as 'M?3' to mean 'the byte at location M+3'.

Type number &04: 32-bit integer

Format:

00	32-bit integer	4 bytes

This is a 4-byte integer at the specified location. It is stored LSB first, MSB last. This type of variable can be accessed as a named integer variable, like ‘A %’ or ‘integer%’, or by using the ‘I.’ operator.

If a named variable is used, the location of the value has to be found first, either by looking it up in the table of resident integer variables, or by searching through one of the linked lists for it. The name field of the Variable Information Block in the linked list has the ‘%’ on the end of it, so that it is identifiable as an integer.

If the ‘!’ operator is used, the location of the variable is taken as the number following the ‘!’ (for the unary version); or the sum of the variable before the ‘!’, and the number after it (for the binary version).

Type number &05: 40-bit floating point number

Format:

00	exponent (offset &80)	1 byte
01	mantissa	4 bytes
(bit 7 of byte 01 holds the sign bit)		

This is a floating point number at the specified location. The mantissa is stored MSB first, LSB last (the opposite order to 4-byte integers). The top bit of the mantissa is used to hold the sign bit, as this would always be a ‘1’ (see section 2.2.2 for a description of floating point numbers).

This type of variable can only be accessed as a named variable stored on the HEAP; there is no floating point indirection operator. The location of the variable is found by searching through one of the linked lists for it. There is no symbol on the end of the name field of a floating point variable.

Type number &80: static string

Format:

00	ASCII characters of string	nn bytes
nn	&0D terminating character	1 byte

This is a static string at the specified location. It can only be accessed by using the '$' string indirection operator: the location of the string is taken to be the number after the '$'. The carriage return (&0D) terminating character is not counted as one of the characters of the string: it is only used to mark the end.

Space can be allocated for a string of this type, by using the 'reserve space' form of the DIM statement: 'DIM A 20' will allocate space for a string at A of maximum size 20 characters, plus 1 for the terminator.

Type number &81: dynamic string

Format:

00	pointer to string on HEAP	2 bytes
02	space allocated	1 byte
03	current length	1 byte

This is the *String Information Block* of the dynamic string: these 4 bytes will occupy the value field of the Variable Information Block of a string variable. This type of variable can only be accessed as a named variable. The *name field* of the Variable Information Block has the '$' symbol on the end, so it is identifiable as a string.

When a dynamic string is first assigned, the Variable Information Block is created and linked into one of the lists, to hold the name and String Information Block of the string. Then space is allocated on the HEAP for the characters of the string itself, and the String Information Block is set up to point to first character of that string. The string itself does not need a carriage return to mark the end, as the String Information Block holds the length of it.

If the string is empty, no space needs to be allocated for it at all. If the string is a 'small' string (less than 8 characters), just the correct number of bytes is allocated on the HEAP for it. If it is a 'large' string, an extra 8 bytes are reserved for it, to allow some room for expansion (if this would take the allocated space over 255 characters, 255 bytes are reserved).

Whenever a dynamic string exceeds the space which has been allocated, a new area is reserved for it on the HEAP (using the same rules as above). The 'gap' left in the HEAP where the string used to be cannot be recovered (BBC BASIC has no 'garbage collector'): so if memory is not to be wasted, it is usually a good idea to set strings, at the start of a program, to the largest size that they are likely to become.

The amount of memory wasted in this manner is not usually a great deal, but certain operations tend to use quite a lot (for example, a loop which adds one character on the end of a string each time round). In BASIC2 this has been improved by checking to see if the string is on top of the HEAP: if it is, it can be extended without having to throw away the old area.

3.1.4 Array storage

Arrays are stored in the same kind of Variable Information Block as ordinary variables, but the *value field* of an array is usually much bigger than that of an ordinary variable. The *value field* of an array has to hold the number of dimensions, and the size of each dimension, as well as the the value of each cell in the array.

The Variable Information Block for an array is linked into the list when it is dimensioned: any attempt to read from or write to a array which does not exist will result in the 'Array' error (error number 14) being generated.

The *name field* in the Variable Information Block for an array has the '(' symbol on the end, so that it is identifiable as an array. It also has the '%' or '$' symbol before that, if it is an integer array or a string array.

The format of the *value field* of an array with D dimensions is:

00	offset of start of cells (nn)	1 byte
01	size of dimension 1	2 bytes
03	size of dimension 2	2 bytes
05	etc.	
nn − 2	size of dimension D	2 bytes
nn	start of cells	

The first byte of the *value field* gives the offset of the start of the cells from the start of the *value field*, rather than the number of dimensions of the array. If the number of dimensions is D, this offset will be ? D+1 bytes (2 for the size of each dimension, and 1 for the offset byte itself). This will be 3 for single-dimension arrays.

The size of each dimension is stored as the maximum allowed subscript.

Each cell is in the same format as the equivalent variable: if it is an integer array, each cell will contain a 32-bit integer (type number &04); if it is a floating point array, each cell will contain a 40-bit floating point number (type number &05); and if it is a string array, each cell will contain a 4-byte *String Information Block* (type number &81). The actual strings for a string array are stored separately on the HEAP (as for dynamic string variables), as soon as they are first set.

The order of the cells is probably best explained by an example. For the array A(1,1,1) the order of the cells will be:

cell 0	A(0,0,0)
cell 1	A(0,0,1)
cell 2	A(0,1,0)
cell 3	A(0,1,1)
cell 4	A(1,0,0)
cell 5	A(1,0,1)
cell 6	A(1,1,0)
cell 7	A(1,1,1)

The following algorithm can be used to find the required element of an array:

```
C = 0
start at first dimension
REPEAT
        C = (C * size) + subscript
        move on to next dimension
UNTIL no more dimensions left
```

where ‘size’ is one more than the maximum subscript for the dimension of interest (allowing for the subscript O); and ‘subscript’ is the required subscript of the dimension of interest.

At the end of that algorithm, C will give the cell number of the required element.

Taking the example of the array A(1,1,1) again, if the element required was A(1,1,0), the successive values of C after each iteration of the loop in the algorithm would be:

after 1 pass:	C = 1
after 2 passes:	C = 3
after 3 passes:	C = 6

This means that the element A(1,1,0) is cell number 6 of the array A(1,1,1). This agrees with the list given above.

To get the location of the cell, the cell number must be multiplied by the size of each cell: 4 bytes for an integer or a string, or 5 bytes for a floating point number. This gives the offset (in bytes) of the required cell from the start of the cells.

Once the location of the element has been found, this can be put in the *Variable Descriptor Block*, together with the type of the element (integer, floating point, or string). The array element can now be handled inside BASIC as if it was just another variable in memory .

3.2 The BASIC STACK

The BASIC STACK works downwards from HIMEM. The STACK pointer is held in page zero, at &4,&5. It is used to save temporary results in the middle of calculations, and to save the 6502 stack and parameters when a FN or PROC is called (see section 5 .3).

For example, to evaluate the expression:

```
2 + 5 * 3
```

the '2' must be saved while the '5 * 3' is being calculated. The 6502 stack *could* be used for this, but it is very small, and would not allow very complex expressions without overflowing (especially when there are FNs to be dealt with).

Before anything is pushed on the STACK, a check is made to ensure that there is enough room for the new item: otherwise there may be a clash with the HEAP which is growing in the opposite direction, upwards from LOMEM (see fig 3.1). If there is not enough room, the 'No room' error is generated.

There are routines to push any of BASIC's accumulators IntA, FPA, and StrA (and pull them again); these are used quite a lot in the expression evaluator. Chapter 4 describes the expression evaluator in more detail.

The other main use of the BASIC STACK is by PROCs and FNs. When one of these is entered, the 6502 stack is transferred onto the BASIC STACK. If this was not done, the small 6502 stack would soon overflow with return addresses for JSRs if the *recursion* of the PROCs or FNs went very deep (i.e. the PROC or FN called itself).

PROCs and FNs also need to make sure that LOCAL variables and parameters used in the PROC or FN are returned to their original values when the call is finished. When the call is started, the values of the parameters in the PROC or FN definition are pushed on the STACK, together with the *Variable Descriptor Block* for the parameter. That gives the location and type of the variable, so it can be restored after the call. Section 5.3 gives more detail on the action of PROCs and FNs.

3.3 Workspace

This section lists the workspace used by BASIC. In many cases, the use of particular locations may be described in more detail elsewhere.

Page Zero

&00 − &01	LOMEM
&02 − &03	HEAP pointer (section 3.1)
&04 − &05	STACK pointer (section 3.2)
&06 − &07	HIMEM
&08 − &09	ERL
&0A	PTRA offset
&0B − &0C	PTRA base (section 2.2.5)
&0D − &11	psuedo-random number for RND
&12 − &13	TOP
&14	PRINT field width
&15	PRINT hex flag (HEX if bit 7 set)
&16 − &17	ON ERROR pointer (section 5.8, chapter 11)
&18	MSB of PAGE (LSB is always zero)
&19 − &1A	PTRB base
&1B	PTRB offset (section 2.2.5)
&1C − &1D	DATA pointer (points before next DATA item)
&1E	COUNT (no of characters printed on line)
&1F	LISTO mask bit 0: space after line no. bit 1: indent FORs bit 2: indent REPEATs
&20	TRACE flag (&00 = OFF, &FF = ON)
&21 − &22	TRACE maximum line number

&23	WIDTH (or &FF if WIDTH 0 used)
&24	REPEAT stack pointer (section 5.5)
&25	GOSUB stack pointer (section 5.2)
&26	FOR stack pointer (section 5.6)
&27	Temp for expression evaluator
&28	OPT mask: bit 0:produce listing bit 1:give errors bit 2:relocate (BASIC2)
&29	opcode slot for assembler
&2A − &2D	IntA (section 2.2.1)
&2E − &35	FPA (section 2.2.2)
&36	StrA length (characters from &600 on)

Page Zero multi-purpose workspace

&37 − &4E	Main uses are:
&37 − &38	general pointer
&39	name length/variable type
&39 − &40	integer for division and multiplication
&3B − &42	FPB for floating point routines
&43 − &46	floating point multiplyldivide workspace
&3F − &47	PRINT hex digit buiFd area
&48	no. of constants for series evaluator
&49	flag for stringlnumber conversion
&4A	exponent for stringlnumber conversion
&4B − &4C	floating point memory pointer
&4D − &4E	pointer for series evaluator

&4F − &8F (not used)

OS workspace

&90 − &3FF	OS workspace

Page 4 workspace

&400 − &46B resident integer variables (section 3.1.1)

&46C − &470 floating point temp 1
&471 − &475 floating point temp 2
&476 − &47A floating point temp 3
&47B − &47F floating point temp 4

&480 − &4F5 variable list base pointers (section 3.1.2)

&4F6 − &4F7 PROC list base pointer (section 3.1.2)
&4F8 − &4F9 FN list base pointer (section 3.1.2)

&4FA − &4FF (not used)

Page 5 workspace

&500 − &595 FOR stack (section 5.6)
&596 − &5A3 (not used)
&5A4 − &5CB REPEAT stack (section 5.5)
&5CC − &3FF GOSUB stack (section 5.2)

Page 6 workspace

&600 − &6FF characters of StrA (section 2.2.3)

Page 7 workspace

&700 − &7FF keyboard input buffer X