17.2 Data Structures

This section briefly describes how BASIC keeps some of its data structures. If the information contained in these is added to or changed incorrectly, then BASIC may enter an unstable state.

17.2.1 Memory Map

BASICs memory map is not straight forward because of paging considerations. The top segment (page 3) always contains the code that is currently executing, except for the case of extensions running out of page 0. This will usually be BASICs main (cartridge) ROM, but may also be the second half of the internal ROM (which contains about 8K of BASIC) or an extension ROM.

Page 0 will always contain the same RAM segment once BASIC has started up. This contains BASIC's variables, the keyword table and the system part of the symbol table. Above these is the program, symbol table and stack of program 0, which may grow into a segment in page 1 and 2. Other program numbers start at the the bottom of a segment in page 1 and may grow up to the top of a segment in page 2.

The following is a more detailed description of the various areas in page 0:

address	function
0-7	Not used.
8	Restart to call an address in a segment. Before using this restart, HL should contain a page 3 address within the segment to call, and A should contain the segment number.
10H	Restart to do BASIC system calls. The restart instruction to do BASIC system calls should be followed by two or more bytes which indicate which system routines are to be called. These are described in detail in section 6 (System Calls). The calling process preserves all register parameters passed to the routine being called, and the returning process preserves all register results passed back from the routine.
18H	EXOS error check. The EXOS error check restart can be called immediately after calling EXOS. If the return code in A is zero, then it will return immediately; otherwise it will convert the EXOS error code into a BASIC error number and cause an exception.
20H	Cause a BASIC exception. The BASIC error restart is called with the desired exception number in HL. The exception is then caused. This restart will not return.
28-2FH	Not used.
30H-5BH	Used by EXOS.
5CH-1FFH	Used internally by BASIC.
200H	Start of BASICs variables.
E54H	Start of BASICs segment allocation table (variable size).
variable	default keyword table. This area is pointed to by the variable KEYPTRS.
variable	default part of symbol table. Entries in this table are pointed to by entries in the symbol table hash table.
variable	space used by extensions.

variable	space ALLOCATEd by the user. The start of this area is pointed to by EXTOP.
variable	start of program 0. The start of program 0 is pointed to by MCTOP, and also TEXT when program 0 is the current program.

17.2.2 Program Storage and Tokenisation

The program is stored in RAM in a tokenised form. The start of the current program is pointed to by the variable TEXT. This will usually be at the start of a segment in page 1, but program 0 is a special case and may start somewhere in page 0.

The format of each line is:

1 byte	- Offset to next line in program (ie. length+1).
2 bytes	- Line number in binary.
1 byte	- Indentation level.
n bytes	- Tokenised program text.

A zero in the length byte indicates the end of the program. The indentation byte takes a value from 0 to 127, and is the number of nested blocks deep that the program line is in. This count is used both in the running of the program and in listing it. The top bit of the indentation byte is used internally by BASIC as a flag.

The tokenised program text is made up of a series of items which correspond to the various syntactic items in the original source.

Each item consists of at least one byte. This byte describes the item, and the following bytes (up to the start of the next item) are data. Thus a program line can be viewed as:

1 byte	2 bytes	1 byte	1 byte	1 byte	etc.
offset	line number	indentation	item	item	...

In general, the item decriptor byte is split into two parts; the top three bits describe the type of the item and the bottom 5 bits contain the number of bytes in the item (with one exception).

The following is a list of valid items and their formats:

floating-point constants

Floating point numbers in the source are not stored in ASCII, but in their internal BCD representation (see section 2.6 (numbers) for more details), as in the symbol table (see section 2.3 (Symbol Table)).

The descriptor byte of a floating point number has the value C6H, and is followed by the six BCD bytes of the number.

integer constants

Integer numbers in the range 0 to 9999 are not stored in the floating-point format for speed, but are stored as a 16-bit binary number in the normal Z80 format.

The descriptor byte of an integer number has the value C2H, and is followed by the two binary bytes of the number.

line numbers

Line numbers are stored differently from integer constants, so that RENUMBER knows what to renumber.

The descriptor byte of a line number has the value A2H, and is followed by the two binary bytes of the number, as with integers.

string constants

The actual characters of strings are stored in ASCII as in the original source text, but are preceded by the length byte of the string. Opening and closing quotes are not stored as part of the string, and double quotes within the string ("") are stored as a single quote character (").

The descriptor byte of a string has the value 80H, and is followed by the length byte of the string (0 to 254) and the actual ASCII characters of the string.

keywords

Keywords are stored as a byte containing the number of the keyword within the keyword table (see section 3 (Extension Statements) for more details). Every statement and program line must begin with the keyword descriptor byte.

The keyword descriptor byte has a value of 60H and is followed by the byte containing the keyword number.

identifiers

Identifier items also include text used within statements such as TO in a FOR statement, and VIDEO MODE in the SET statement. Identifiers are stored as upper case ASCII characters, with the length in the lower 5 bits of the descriptor byte.

The identifier descriptor byte has a value of 20H (numeric) or 40H (string) + number of characters in the identifier (1 to 31).

signs

The punctuation symbols that appear in programs (such as <> and ;) are called 'signs'. These are converted to a single number 0 to 31 and encoded into the lower 5 bits of the descriptor byte.

The sign descriptor byte has a value of 0 + token for the sign. The tokens used are:

end of line	!	1
channel	#	3
concatenation	&	6
left parenthesis	(	8
right parenthesis	)	9
splat	*	10H
plus	+	11H
comma	,	12H
subtract	-	13H
divide	/	15H
colon	:	16H
semi	;	17H
less than	<	18H
eq	=	19H
gt	>	20H
ne	<> ><	21H
le	<= =>	22H
ge	>= =>	23H
exp	^	27H

17.3 Symbol Table

The symbol table contains all the user's variables and function names in ASCII, as well as the built-in functions. The table consists of a 32-entry hash table of pointers which point to 'chains' of entries. Each entry contains a pointer to the next entry with the same hash value, a pointer of 0 indicating the end of a chain.

When an identifier is added to the symbol table, the entry is put into RAM, and the appropriate hash table pointer adjusted to point to the new entry. The old hash table pointer is then put into the new entry, thus ensuring that new entries are found before old entries with the same name. Any entry and subsequent entries can be removed again simply by adjusting the hash table pointer, thus providing the ability for functions to have local variables.

In RAM, entries are added in sequential order. Because of the nested nature of function calls, an entry never has to be removed without also removing all entries subsequently defined.

The variable VARPTR points to the next free byte in the symbol table, and the variable VARBASE points to the first variable defined within a function. When a function is called, VARPTR is copied into VARBASE, and when the function exits VARBASE and VARPTR are restored (together with the hash table), thus unlinking any local variables used within the function.

The hash table starts at HASHTAB and contains 32 16-bit pointers. Initially these point within page 0 to the built-in functions, but as user symbols are added they may point outside page 0. Thus the symbol table must always be cleared of the user's symbols before paging to another program number.

Each symbol table entry follows the following format:

2 bytes - pointer to previous entry, 0 if none.
1 byte - flags byte
n bytes - name of symbol, length byte first.
m bytes - data.

Within the flags byte, the bits have the following meanings:

bit 0 - Set if string
bit 1 - Set if an array (reset for extension)
bit 2 - Set if a user function or handler (reset for extension)
bit 3 - Set if a machine-code function (set for extension)
bit 4 - Set if identifier is a function parameter passed by reference (reset for an extension)

The name of the symbol is upper case ASCII preceded by a byte containing the number of characters in the name. String identifiers end in '$'.

The data and the space it occupies is determined by the type of identifier as described by the flags byte.

If the identifier is a reference parameter to a function, then the data is a 16-byte pointer to the start of the entry of the identifier which it references.

If the identifier is a machine code function, then the first two bytes of the data form a page 3 (or page 0 for an extension) pointer to the code of the function. The next byte contains the segment number of the code, or 0 if the code is in page 0 (since the segment numoer cannot be pre-determined).

If the identifer is a simple (ie. non-array) numeric or string variable, then the following byte contain the data of the number or string. See section 2.5 (Strings) and 2.6 (Numbers) for more details.

If the identifier is a numeric or string user-defined function, then the following bytes contain the data for a number or string (as for a simple variable described above) followed by a 16-bit pointer to the start of the line on which the DEF or HANDLER corresponding to the function occurs.

If the identifier is a numeric or string array, then the first data consists of information about the the dimensions and the upper- and lower- bounds of the array.

The first byte of data contains the number of dimensions in the array (1 or 2), followed by 6 bytes containing the total number of elements in the array in floating point format (as described above)

Following this are the lower bound of the array and the number of elements in the dimension, both in the normal 6-byte floating point format. These are repeated for the second dimension if there is one

Finally, there are a number of numeric or string entries, as described above, each corresponding to an element. Elements in the second dimension are the most rapidly changing in memory.

17.4 Stack

BASIC keeps the Z80 stack in a fixed place in page 0 to avoid paging problems, but has another stack on which is kept numbers and strings during expression evaluation and information about currently active FOR..NEXT and DO..LOOPs, GOSUBs, WHEN blocks and function calls.

The top of this stack is pointed to by the variable STKPTR, and the base is pointed to by STKTOP.

Whenever a channel open call to EXOS is attempted (including GRAPHICS and TEXT statements etc.), the stack is moved down in memory to just above the symbol table, and any segments no longer used are given back to EXOS.

Before any data is put onto the stack, BASIC ensures that there is enough room between it and the symbol table top, and if there is insufficient room then the stack is moved to the top of the segment currently being used. If there is still not enough room, then BASIC allocates another segment, pages it in to the next highest page not being used, and moves the stack to the top of that segment (or to the EXOS boundary if the segment is shared). An out of memory error is given when either no more segments are available or another segment is required when the top segment being used is already page 2.

This is a simplified view of the way BASIC can move its stack, but the above should be borne in mind when an extension statement or function calls the expression evaluator, or indeed most other parts of BASIC, since any expression can invoke a user-defined function which may cause the stack to move. In practice this means never calling BASIC whilst saving a pointer to or into the stack. If this becomes necessary, then a negative offset from the contents of STKTOP should be saved, and the actual stack address re-calculated after calling BASIC.

Each item on stack is preceded by a byte describing that item. The remainder of the bytes in the stack item are data.

The stack items are:

Type 6

Stack type 6 is the GOSUB...RETURN stack item, and has 2 bytes of data following it. These two bytes form a pointer to the start of the line following the line containing the GOSUB statement.

Type 7

Stack type 7 is the DEF stack item, and has 5 bytes of data following it. These 5 bytes are as follows:

2 bytes - value of VARPTR when function invoked.
1 byte - bit 0 set if a result is required on stack.
2 byte - value of VARBASE when function invoked.

This stack item is also used when a HANDLER is called as a result of an exception occuring whilst an exception handler is active.

Type 8

Stack type 8 is the WHEN stack item, and in version 2.0 is followed by 2 bytes of data which form a pointer to the next outer exception handler (0 if none). In 2.1 there are 8 bytes of data as follows:

2 bytes - pointer to END WHEN line.
4 bytes - used internally.
2 bytes - WHEN nest count.

Numbers and

2.5 Strings

Strings are stored in the symbol table and on the stack in the same format. The first byte is the number of characters in the string (from 0 to 254) and the remainder of the string contains the actual characters of the string, which may be any byte. A length byte of 255 in a symbol table entry indicates that the identifier is uninitialised.

In the symbol table, a fixed amount of string space is set aside for each string, and if the length of the string is less than the number of bytes allocated to it, then the remaining bytes at the end of the string will contain garbage. The byte before the length byte in the symbol table contains the maximum length of the string ie. the number of bytes allocated+1.

On the stack, the number of bytes occupied is length of string+2 (ie. one extra byte for the length byte and one for the stack type byte - see section 2.4 (Stack) above).

2.6 Numbers

In the symbol table, numbers are stored as a 6-byte BCD number. The digit pair at the lowest memory address are the least significant digits. The top byte of the number is the binary exponent and sign, and the top-but-one byte contains the most significant digit pair.

The decimal point is always (for a normalised number) considered to be before the most significant digit. The exponent has a bias of 63, with the top bit of the exponent byte set if the number is negative

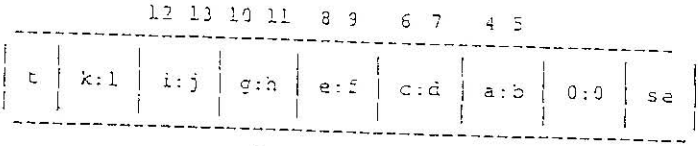
If the exponent byte has a value of 7F hex., then the first and second bytes of the number contain a signed 16-bit integer in normal Z80 format, with the remaining bytes normally unused. The number zero is represented in this format.

An exponent byte of 7F hex. and most significant digits byte of FF hex. (invalid BCD) indicates that the numeric identifier is uninitialised.

On the stack, the format is similar except that there is an extra byte between the most significant digits byte and the exponent. This byte is used during the arithmetic operations for greater accuracy.

The stack format also has an extra byte before the least significant digits byte. This is used for increased accuracy during the calculation of the transcendental functions.

The format of numbers on the stack can thus be viewed as:



where a is the most significant digit, b is the next significant digit etc.

t is the type byte (see section 2.4 (Stack) above).

se is the exponent, with the top bit indicating the sign of the number.

BASIC sometimes refers to digits individually, and the numbers above the diagram above show how BASIC regards the digits of the number as being numbered. The system calls PUTDIG and GETDIG use this scheme (see section 6 (System Calls) for more details).

Some of the functions use another number format for extra accuracy. This is a sort of cross between the symbol table format and the stack format. The digits k and l above are present in this format, but the two digits marked 0:0 are not. Thus this format takes up 7 bytes. There are system calls (see section 6 (System Calls) for details) and variables (see section 5 (Variables) for details) which deal with this format.