**Tokenisation**

The following tokens are stored in low/high order. Each is 16 bits wide except for that representing an ASCII string.

**End Token : 0000 0000 0000 0000 $0000**

This token is used to mark the end of a sequence of tokens (e.g. a line of tokenised code)

**ASCII String : 0000 0000 LLLL LLLL $0001-$00FF**

This represents a single ASCII string. The token identifies it as a string and gives the overall length in bytes of the string.

This is not the same as the length of the string. It also includes the header word (2 bytes/1 word) and a trailing zero byte. The latter is mandatory. Hence there cannot be a string token where the length is zero.

The string is stored in such a way that the address of the following token is itself the address of the token added to the value of the token. All tokens are 16 bit, so if the length of the string, and the trailing zero byte is odd, an extra zero byte spacer must be added.

**Constant Shift : 0001 CCCC CCCC CCCC $1000-$1FFF**

This allows the extension of constants from 15 bits to 27 bits, e.g. enough to encompass the address range of a 65816 CPU.

When there is a constant shift, it is shifted left 15 times and added to the 15 bit “standard” constant. Processing the standard constant resets the constant shift to zero, e.g. it does not persist to the next constant.

**Keywords : 001T TTTK KKKK KKKK $2000-$3FFF**

These represent tokenised keywords. The numbers are not continuous though the K KKKK KKKK value is – this is the actual keyword token. TTTT identifies it’s type as follows.

0000-0111 Binary operators

0000 and or xor

0001 > = < >= <= <>

0010 + -

0011 \* / mod

0100 on *not used.*

1000 Unary function

1001 Syntactic only keywords ( , ; : TO etc)

1010-1100 *not used*

1101 Keyword that decrements the structure level (until, wend, …)

1110 Keyword that increments the structure level (repeat, while, …)

1111 Keyword that does not affect the structure level.

**Constant : ??CC CCCC CCCC CCCC $4000-$BFFF**

This value represents the constants 0-32767 in order, e.g. 0 = $4000 32767 = $BFFF. This can be extended from 0-2^15 to 0-2^27 using the constant shift. After a constant has been extracted the constant shift is cleared.

**Identifier : 11ET ACCC CCCC CCCC $C000-$FFFF**

The identifier token has 2 ASCII values packed into a 16 bit word.

E is a continuation bit, if zero it marks the last token in the identifier, if 1 the token continues after this token.

TA are type bits. T is 1 if the type is a string, A is 1 if the type is an array. The post identifier characters $ and ( are included as part of the tokenisation. (So a2$( only occupies one token space)

CCC CCCC CCCC is 2 numbers from 0-44 multiplied together as 2nd Character \* 45 + 1st Character. The values are :

0 filler if there is an odd number of characters / space

1-26 A-Z

27-36 0-9

37-44 Unused

**Memory Blocks**

Memory is divided into blocks. Blocks can be up to 64k in size. Each block must all be in the same 64k address space. They do not have to start at the start of a block, so all these address are offsets from the base address.

|  |  |
| --- | --- |
| **Offset** | **Contents** |
| $0000 | 4 byte type ID. A Basic program has the ASCII characters for “BASC” here. |
|  |  |
| $0008 | Low Memory pointer. This is a 16 bit address (not offset) which is the next free byte available in low memory. |
| $000A | High Memory pointer. This is a 16 bit address(not offset) which is the byte after the topmost free byte. |
| $000C | Empty string. This contains $0000, which is an empty string (the length byte is zero). Newly created string arrays, and strings that result in an empty string are stored as this. This reduces G/C as they are not allocated high memory. |
| $000E | If non-zero, execute the code in the tokenised buffer first. |
| $0010 | 26 Fast Variables A-Z, 4 bytes each ; so @ is at $0010 and Z is at $0074 |
| $0080 | 4 x 16 word addresses which are linked lists of variables, these are in line with the TA bits, so are integer, integer array, string, string array. |
| $0100 | Buffer for tokenised command line |
| $0200 | The program |
| … | $0000 indicating end of the program |
| … | By convention $EEEE $EEEE is stored in the 4 bytes following the end of the program, to place a clear marker between the BASIC code and the variables. |
| … | First variable. |
| … | More variables/ |
| … | Free unused Space. The current value of the low memory pointer is the first byte of this space. |
| … | Data Space. This is used for strings. Some space above this may be reserved so the program creating blocks can create identifiers. |
| … | End of Block. The byte after the last in the block. The High Memory Pointer is initially set to this value normally, though in testing test data may be stored above here both arrays and strings as well as tokenised names. |

**Program Storage**

Programs are a collection of records in line number order.

The header is 2 words:

Firstly an ***offset*** to the next line. If this is zero this indicates the program end.

Secondly the tokenised line number, in unencoded format.

The tokens making up the program line follow this, ending in the $0000 token.

**Variable Storage**

Variable storage immediately follows the end of program storage.

Each record occupies 6 information bytes and 4 bytes for each item of data.

+0..1 The *address* of the next variable in the list

+2..3 The *address* of the identifier as a sequence of tokens (stored in program code normally).

+4..5 Highest array index for an array, 0 if non-array.

+6.. First data element.

All addresses are absolute addresses not offsets. Variables are all erased when a program is RUN or the code is edited (except A-Z)

Integer variables are stored as 32 bit integers, String variables as 16 bit addresses in the current data page.

Variables are stored in linked lists, ending with a next variable address (the link) of $0000. There are up to 16 hash entries for variables, and four sets of linked lists for each of the four available types.

**String Storage**

Strings are stored in the top of memory. While evaluating a string memory is filled downwards from the top with intermediate results. When these are required to be persistent – most obviously when assigned to something, they are “concreted”.

A 256 byte gap is left between the top of memory and the temporary string area (this is reset at the start of each instruction) ; this is sufficient for any string length required. When a string is concreted it is copied into this space and the high memory pointer reduced to allow for it.

Garbage collection is reduced as intermediate values are always thrown, so complex expressions will only produce a single permanent entry in the heap, any intermediate values are wasted.

This should be borne in mind if such values are passed to any function which needs to retain a string value ; say one was going to name a music object “fred” using a command like MusicName “Fred” ; Fred will not be concreted and will disappear if string operations are performed.

Do not use the standard concreting routine for this. It will work, but because the name is not in the variable system the garbage collector will overwrite it. It should be stored in the Music object as a **copy**.