2 The BASIC System

The BBC microcomputer system has been designed to allow many different languages (like LISP or FORTH) to be used with it. However, the language that all BBC micros and Electrons start with is BBC BASIC.

2.1 An overview of BASIC

When BASIC is initialised, it takes control of the computer. It prints ‘BASIC’ on the screen, and prompts for a line to be input. You then type in programs, RUN them, edit and RUN them again until they work, and continue until the power is switched off.

Beneath all of this is 16K of 6502 machine code, in a paged ROM sitting between &8000 and &BFFF, beavering away trying to work out what to do with the line that you just typed in. It is really a whole system all by itself, editing programs, interpreting program statements, evaluating expressions, handling variables; in fact it does everything except actually input and output to the hardware (it leaves that to the Machine Operating System).

Fig 2.1 shows a general overview of BASIC, with its main component parts. The first major section of the BASIC system is the command handler and the statement interpreter. When a line is input at the keyboard, the command handler tokenises it, and decides whether to insert it into the program (if it starts with a line number), or to send it to the statement interpreter. The statement interpreter is also used to handle program statements. The action of the command handler and statement interpreter is decribed in sections 2.3 and 2.4.

The other major section of the BASIC system shown in fig 2.1 is the expression evaluator. This is called by most of the statement handlers (or function handlers) when they want a number or a string to operate on. For example, the MODE statement handler calls the expression evaluator to get the number of the MODE that is to be used. The expression evaluator is described in more detail in chapter 4.

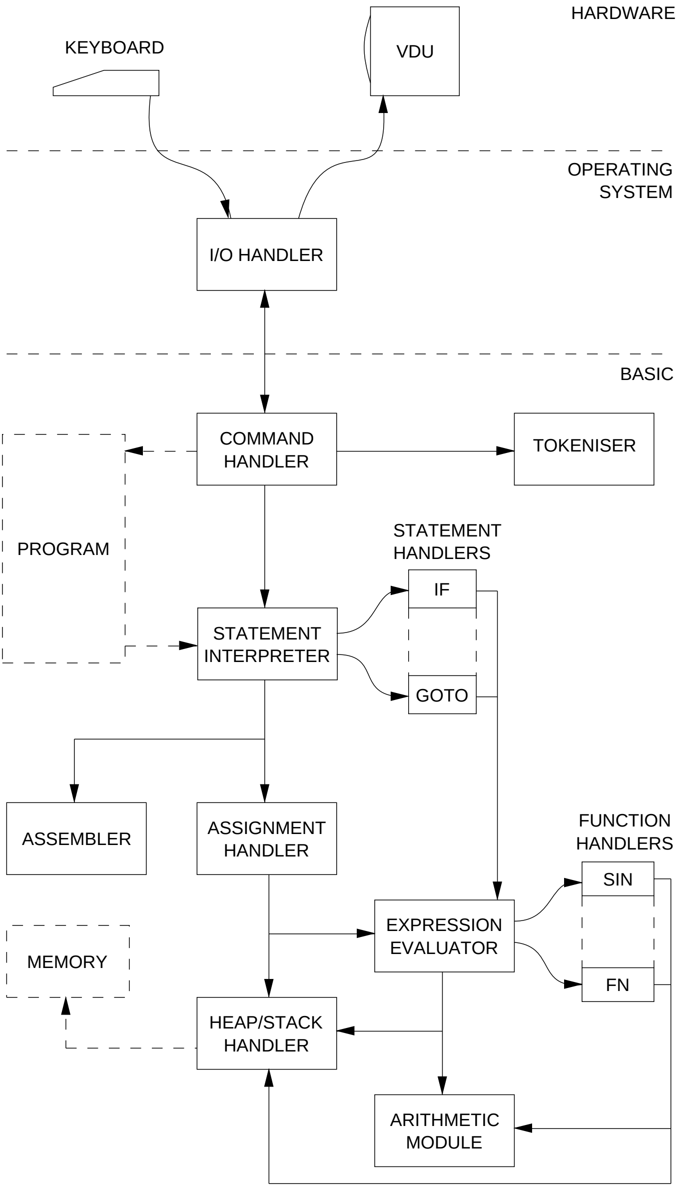

Figure 2.1 − The BASIC system.

The arithmetic module is a collection of routines which is used to perform the calculations required by the expression evaluator (and by the statement and function handlers). Most of these have to be floating point routines, as real numbers are more difficult for the computer to handle than integers or strings. These routines are detailed in chapter 10.

The HEAP/STACK handler is another collection of routines, but these deal with variables and other use of memory by BASIC while the program is running (dynamic memory use). Variables, and BASIC's memory use are described in chapter 3.

2.2 The BASIC 'CPU'

The 6502 CPU is a versatile machine, but on its own it is a bit limited. Its 8-bit accumulator, A, can only handle single byte integers; it can't deal with real numbers or strings; it can't allocate space for BASIC variables, and its stack is only 255 bytes deep. To get round this, BASIC has a softwvare layer on top the 6502, to provide a more versatile service.

This new 'layer' has a collection of page 0 locations as 'registers' , which are manipulated by the 6502. These registers (together with the routines to handle them) make up the 'Central Processing Unit' of the BASIC system. Fig 2.2 compares the 6502 registers with BASIC's registers.

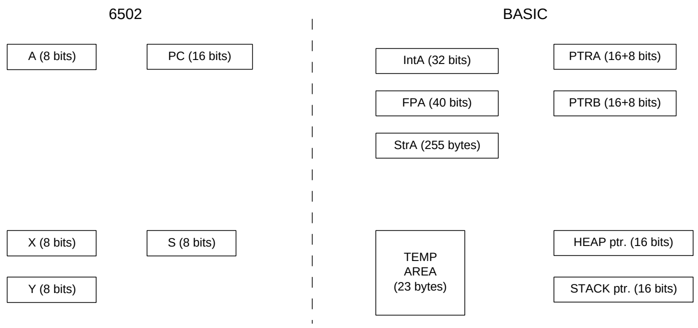

Figure 2.2 − 6502/BASIC registers.

2.2.1 BASIC Integers

Where the 6502 only allows 8-bit integers to be used, most of BASIC's integer work is done with 32-bit (4-byte) integers. For this it has a 4-byte integer accumulator, IntA, stored in page zero at &2A to &2D. The format of the 4-byte integers stored in this accumulator is shown in fig 2.3.

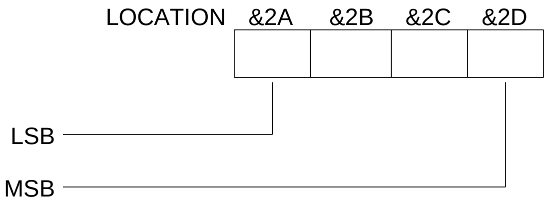

Figure 2.3 − Integer format

Note that the least significant byte (LSB) is stored first, at &2A, with the most significant byte (MSB) at &2D. This means that a single-byte (positive) value at &2A can be converted into a 4-byte integer starting at &2A, by setting the 3 most significant bytes (in &2B, &2C and &2D) to zero.

2.2.2 Real numbers

One of the major advantages of the BASIC 'CPU' over the 6502 equivalent is its ability to deal with real numbers, rather than just integers. For this, it has 2 floating point accumulators, FPA and FPB. For those not familiar with binary floating point representation, here is a brief description.

Decimal integers can be written in binary form, like

9 (decimal)   can be written as: 1001 (binary).

Fractions can be written in decimal by using a decimal point, like '9.6', and binary numbers can be written in a similar form. Thus '0.1' (binary) represents 1/2 (0.5 decimal), '0.01' (binary) represents 1/4 (0.25 decimal), and so on. As an example,

3.625 (decimal)   can be written as: 11.101 (binary)

Using this would give a way to represent numbers on a computer; by holding the integer part as one number, and the fractional part as another. In practice, though, for many applications this is just too limited.

In decimal, for talking about a much wider range of numbers, *scientific form* or *standard form* can be used. For this, the number to be expressed is written down as a number between 1 and 10 (this is the *mantissa*), multiplied by ‘10 to the power of’ another number (this is the *exponent*). Thus 273 can be written as 2.73x10^2 (or 2.73E2).

For the binary representation of real numbers, BASIC uses a similar form to the decimal one: the number to be expressed is written as a number between 1/2 and 1 (not equal to 1), multiplied by ‘2 to the power of’ another number. Thus 11.101 (binary) can be written as 0.11101x2^2 (the exponent is in decimal for clarity). This is often called *floating point* representation, as the actual position of the *binary point* in the number is not fixed to a particular position (in integers, for example, the binary point is always just beneath the least significant bit).

When floating point numbers are stored in variables, they occupy 5 bytes, and are stored as shown in fig 2.4.

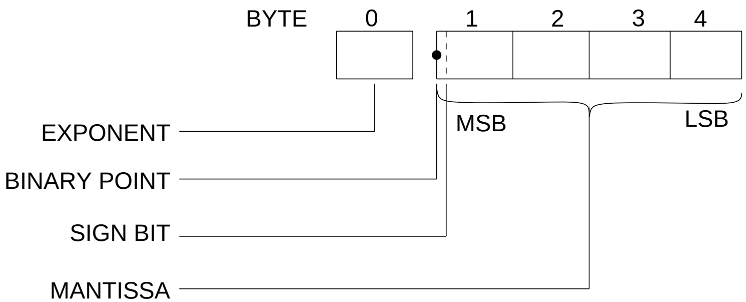

Figure 2.4 − Floating point packed format.

The exponent is stored offset by &80 − i.e. &80 represents 2^0,

&81 represents 2^1 and so on. This allows the number zero to be represented by a floating point number with all its bytes set to 0. Note that zero doesn't fit in to this floating point representation: it is smaller than 2^{-127}, yet it is larger than -2^{-127} . It has to be represented as a special case.

The position of the binary point in the mantissa is just above the most significant bit.

The mantissa is always a number between 1/2 (0.1 binary) and 1 (but not equal to 1), so the top bit of the mantissa is always a '1 '. This means that this bit position is not needed for the mantissa (it can always be retrieved by ORing the MSB of the mantissa with &80), so this bit is used to store the sign bit of the number (the top bit of the mantissa will not be a '1' if the number being represented is zero)

The mantissa occupies 4 bytes. This means that 4-byte integers can be converted to floating point format, and back again, without loss of accuracy. The bytes are stored MSB first, LSB last; the opposite order to integers. The mantissa is stored as a positive number, and not in 2's complement format (so the representation for '6' is just the same as the representation for '−6', except the sign bit will be changed).

When a 'packed' floating point number is loaded into one of the floating point accumulators, FPA or FPB, it is unpacked into 8 bytes. The format of these accumulators is shown in fig 2.5.

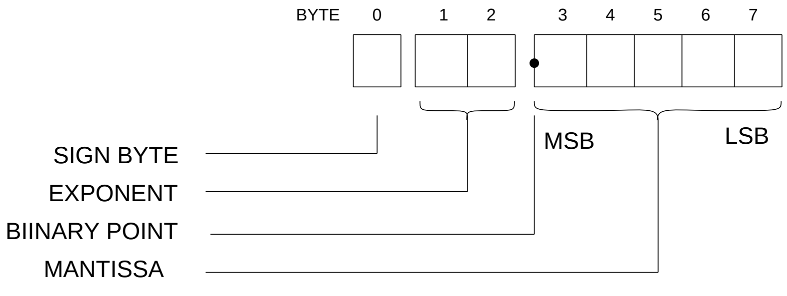

Figure 2.5 − Floating point accumulator format

The exponent has been expanded into 2 bytes; the high-order byte of the exponent is set to zero when the number is loaded in. This allows results of calculations to temporarily overflow (i.e. the exponent becomes too large for the 5-byte representation to handle), providing that they end up in the correct range before being written out to memory again in the 5-byte packed format. The exponent is still offset by &80.

The mantissa has been expanded to 5 bytes instead of 4. This allows for extra accuracy in the middle of calculations. Before the number is written back out to memory, this extra byte is used to round the rest of the mantissa.

The sign bit has been removed to a whole byte by itself, and the top bit of the mantissa has been restored to ‘1’. For calculations, this ‘1’ is needed in the top bit where it is supposed to be.

Often during a calculation, the top bit does not stay set (perhaps due to a number almost equal to it being subtracted from it). If this is the case, the value of the number is still given correctly (as the mantissa multiplied by ‘2 to the power of’ the exponent), but the mantissa is now much less than 1/2. Before the number can be written out into memory, the number must be ‘normalised’ by repeatedly multiplying the mantissa by 2 (i.e. shifting it up by 1 bit), and decrementing the exponent (dividing that part of the representation by 2) to compensate, until the top bit of the mantissa becomes set again.

If this happens, some of the accuracy of the number may have been lost, as some of the bits of the number may have ‘fallen off the bottom’ before the number was shifted back up again.

Floating point numbers do have certain limitations:

(a) The largest number which can be represented (in the 5-byte 38 format) is just less than $1.0x2^{127}$ ($1.7x10^{38}$).

(b) The smallest number (in magnitude) which can be represented (apart from zero) is $1.0x2^{-128}$ ($2.9x10^{-39}$).

(c) Because just 32 bits are used to hold the mantissa of the number, the representation is only accurate to 1 part in 23^2. (1 part in $4x10^9$). This means that if any number stored in this format is printed out in decimal, it will only be accurate to the first 9 decimal digits.

(d) Calculations involving floating point numbers take longer than those involving integers.

The actual format of the floating point accumulators is003A

FPA	FPB	USE
&2E	&3B	Sign byte
&2F	&3C	exponent overflow byte
&30	&3D	binary exponent (offset &80)
&31	&3E	mantissa (MSB)
&32	&3F	mantissa
&33	&40	mantissa
&34	&41	mantissa
&35	&42	mantissa (LSB)

2.2.3 Strings

For string handling, BASIC has a string ‘accumulator’, StrA. All of page 6 is allocated to the string accumulator; the characters of StrA are stored from &600 onwards, with location &36 in page zero used to hold the length of the string.

This makes string handling relatively simple, although it does take up a lot of memory.

2.2.4 General workspace

In addition to these accumulators, BASIC has a general Workspace area, between &37 and &4E, which it uses for general pointers (instead of the 6502 X and Y registers) and for other different purposes, depending on which part of the system is in Operation at the time. FPB is actually in this area, and several routines which do not need to do any floating point calculations may use the same memory that it occupies.

2.2.5 Program pointers

Instead of the Program Counter (PC) of the 6502, BASIC has two pointers, PTRA and PTRB, which it uses to scan through a BASIC program (or a line typed in at the keyboard). Both of these pointers are composed of a 2-byte base pointer, and a single-byte offset from that base. PTRA is mainly used to read the first part of a statement until the statement token is recognised, and PTRB is mainly used for scanning expressions.

The format of these pointers is:

&B,&C	PTRA base
&A	PTRA offset
&19,&lA	PTRB base
&1B	PTRB offset

2.2.6 Dynamic memory pointers

The 6502 only has one way of dynamically allocating space during a program: its stack. This works downwards in page 1 with a maximum size of 256 bytes (i.e. from &1FF down to &100).

Rather than using this, BASIC has a STACK which works downwards in memory from HIMEM. It uses this to hold temporary results from calculations, or when a FN or PROC is called. BASIC also has a HEAP which works upwards in memory from LOMEM (usually the TOP of the program), which is where it puts any variables (apart from resident integers). Together, the BASIC STACK and the HEAP can use up all of the memory between the TOP of the program and the bottom of the screen. Chapter 3 describes how variables are stored, and the use of the HEAP and the STACK.

2.3 Tokenising

When a line is typed in at the keyboard, it is inserted into BASIC's keyboard buffer in page 7 (from &700 onwards). From here, the command handler sends the line to the tokeniser, so that the keywords can be *tokenised*. This involves looking through the line and replacing occurrences of keywords (and their abbreviations) in the line by a single byte *token*, with a value between &80 and &FF. This saves memory when the line is put into a program (as, for example, PRINT takes up only 1 byte instead of 5), and it makes it a lot easier (and faster) to recognise the keyword when it is to be *interpreted*.

2.3.1 Keyword tokenising

The keyword table is stored at &806D (BASIC1) or &8071 (BASIC2), in roughly alphabetical order. The format of each entry is:

Keyword
Single-byte token
Flag byte

Table 2.1 gives a list of the keyword tokens, and the address where they JMP to when recognised, in token value order. From this it can be seen that the tokens are divided up into several groups:

&80 to &84	operators
&85 to &8C	auxiliary tokens
&8D	line number token (see section 2.3.2)
&8E	'OPENIN' for BASIC2
&8F to &93	pseudo-variable functions
&94 to &BC	numeric-valued functions
&BD to &C4	string-valued functions
&C5	'EOF'
&C6 to &CD	commands
&CE	(not used)
&CF to &D3	pseudo-variable statements
&D4 to &FF	statements

The tokeniser does not simply tokenise the line: it obeys certain rules, and can be in several states. The flag byte is used to give instructions to the tokeniser about how to continue tokenising the rest of the line, or how to tokenise this keyword. The flags are used as follows:

Bit 0 **C**onditional flag. If this is set, this tells the tokeniser not to tokenise this keyword if it is followed by an alphanumeric character. This means, for example, that ‘TIMER’ can be used as a variable name, as the ‘TIME’ part of it will not be tokenised.

Bit 1 **M**iddle flag. If this is set, this tells the tokeniser to go to ‘middle of statement’ mode after this token.

Bit 2 **S**tart flag. If this flag is set, this tells the tokeniser to go to ‘start of statement’ mode. The tokeniser must know if it is at the start of a statement or not, because a ‘*’ at the start of a statement will cause tokenising to be abandoned so that the rest of the line can be sent to OSCLI untokenised. If a ‘*’ is found in the middle of a statement, it will be in the middle of an expression, so the rest of the line should be tokenised. It also needs to know if a pseudo-variable found is a statement or a function.

Bit 3 **F**N/PROC flag. If this flag is set (as it is for FN or PROC), this tells the tokeniser not to tokenise the name immediately following the token. This means, for example, that the ‘ERROR’ part of ‘PROCERROR’ will not be tokenised.

Bit 4 **L**ine number flag. If this flag is set, it tells the tokeniser to start tokenising line numbers after this token. This flag is set for keywords like ‘GOTO’ or ‘RENUMBER’. Line number tokenising is usually turned off after any other symbol apart from a ‘,’ , a HEX number, or a string.

Bit 5 **R**EM flag. If this is set, it tells the tokeniser to stop tokenising the rest of the line. This flag is used by the ‘DATA’ and ‘REM’ tokens.

Bit 6 **P**seudo-variable flag. If this is set, it tells the tokeniser to add &40 to this token if it is found at the start of a statement. This is how the tokeniser decides whether a pseudo-variable is a statement or a function. Note that the pseudo-variable *statement* entry in the token table is not used by the tokeniser; it uses the function entry and converts it to the statement token if it is at the start of a statement. The statement entry is used by 'LIST' when the tokens are being printed out.

Bit 7 (not used)

Other symbols

Special symbols found in the input line which affect tokenising are:

&	scans the following hex number
"	scans the following string constant
:	goes to 'start of statement' state
*****	prevents tokenising if at the start of a statement

2.3.2 Line number tokenising

Line numbers can also be tokenised, as well as keywords. However, they will be left alone unless they are found at the start of a line, or after a token with the 'tokenise line numbers' flag set.

Note that the tokenised line number at the start of the line is not inserted into the program (see section 2.4 for program storage).

Tokenising line numbers speeds up the use of GOTOs or GOSUBs in a program, because the numbers are simpler to decode than an ASCII string of digits; but it does not really save very much memory, as each tokenised line number takes up 4 bytes. Fig 2.6 shows how line numbers are tokenised, once the ASCII digits have been read in and converted to a 16-bit integer (it is actually a 15-bit integer, as line numbers greater than 32767 are not allowed).

The bytes after the &8D line number token *must* be less than &80, or they may look like another token. If this was not the case, one of them may look like an 'ELSE' token, and it may be latched on to by the 'IF' statement as something to do if it got a FALSE result (see section 5.4).

Also, the bytes after the line number token must not be allowed to be a control character (i.e. less than &20). If this was not the case, the byte may look like a &0D (carriage return), which marks the end of a line in a program.

The simplest way to ensure that both of these conditions are met, is to fix the top 2 bits of each byte to ‘01’ so that it is in the range &40 to &7F.

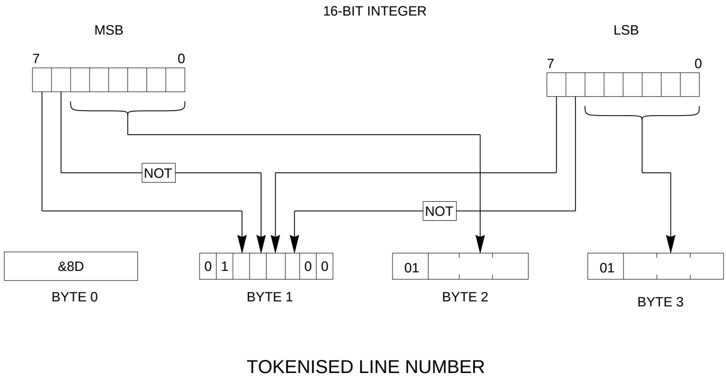

Figure 2.6 − Line number tokenising.

So to convert a 16-bit integer to the tokenised line number format:

1 Set byte 0 to the &8D line number token.

2 Transfer bits 7 and 6 of the MSB of the integer into bits 3 and 2 of byte 1 of the tokenised line number, inverting bit 6 before it is inserted into bit 2.

3 Transfer the bottom 6 bits of the LSB of the integer into byte 2 of the tokenised line number, setting bits 7 and 6 to ‘01’

4 Transfer the bottom 6 bits of the MSB of the integer into byte 3 of the tokenised line number, setting bits 7 and 6 to ‘01’

5 Set byte 1 of the tokenised line number to ‘01000000’ (binary).

6 Transfer bits 7 and 6 of the LSB of the integer into bits 5 and 4 of byte 1 of the tokenised line number, inverting bit 6 before it is inserted into bit 4.

The line number is now tokenised. It is a bit easier to get the line number out of the tokenised form:

1 Shift byte 1 of the tokenised line number up 2 bits, load it into A, and mask off the bottom 6 bits.

2 EOR this with byte 2 of the tokenised line number. A now contains the LSB of the number.

3 Shift byte 1 of the tokenised line number up by a further 2 bits, and load it into A (the bottom 6 bits are all 0)

4 EOR this with byte 3 of the tokenised line number. A now contains the MSB of the number.

Table 2.1. − Keyword Tokens

Token		BASIC 1			BASIC 2	
	Keyword	Flags	Addr	Keyword	Flags	Addr
80	AND	--------	----	AND	--------	----
81	DIV	--------	----	DIV	--------	----
82	EOR	--------	----	EOR	--------	----
83	MOD	--------	----	MOD	--------	----
84	OR	--------	----	OR	--------	----
85	ERROR	-----S--	----	ERROR	-----S--	----
86	LINE	--------	----	LINE	--------	----
87	OFF	--------	----	OFF	--------	----
88	STEP	--------	----	STEP	--------	----
89	SPC	--------	----	SPC	--------	----
8A	TAB(	--------	----	TAB(	--------	----
8B	ELSE	---L-S--	----	ELSE	---L-S--	----
8C	THEN	---L-S--	----	THEN	---L-S--	----
8D	line no.	--------	----	line no.	--------	----
8E	---	--------	----	---	--------	BF78
8F	PTR	-P----MC	BF50	PTR	-P----MC	BF47
90	PAGE	-P----MC	AEEF	PAGE	-P----MC	AEC0
91	TIME	-P----MC	AEE3	TIME	-P----MC	AEB4
92	LOMEM	-P----MC	AF2B	LOMEM	-P----MC	AEFC
93	HIMEM	-P----MC	AF32	HIMEM	-P----MC	AF03
94	ABS	--------	AD8D	AB5	--------	AD6A
95	ACS	--------	A8C6	ACS	--------	A8D4
96	ADVAL	--------	AB56	ADVAL	--------	AB33
97	ASC	--------	ACC4	ASC	--------	AC9E

98	ASN	--------	A8CC	ASN	--------	A8DA
99	ATN	--------	A907	ATN	--------	A907
9A	BGET	-------C	BF78	BGET	-------C	BF6F
9B	COS	--------	A989	COS	--------	A98D
9C	COUNT	-------C	AF26	COUNT	-------C	AEE7
9D	DEG	--------	ABE7	DEG	--------	ABC2
9E	ERL	-------C	AFCE	ERL	-------C	AF9F
9F	ERR	-------C	AFD5	ERR	-------C	AFA6
A0	EVAL	--------	AC12	EVAL	--------	ABE9
A1	EXP	--------	AAB4	EXP	--------	AA91
A2	EXT	-------C	BF4F	EXT	-------C	BF46
A5	FALSE	-------C	AEE9	FALSE	-------C	AECA
A6	FN	----F---	B1C4	FN	----F---	B195
A5	GET	--------	AFE8	GET	--------	AFB9
A6	INKEY	--------	ACD3	INKEY	--------	ACAD
A7	INSTR(	--------	AD08	INSTR(	--------	ACE2
A8	INT	--------	AC9E	INT	--------	AC78
A9	LEN	--------	AF00	LEN	--------	AED1
AA	LN	--------	A804	LN	--------	A7FE
AB	LOG	--------	ABCD	LOG	--------	ABA8
AC	NOT	--------	ACE7	NOT	--------	ACD1
AD	OPENIN	--------	BF85	OPENUP	--------	BF80
AE	OPENOUT	--------	BE81	OPENOUT	--------	BF7C
AF	PI	-------C	ABF0	PI	-------C	ABCB
B0	POINT(	--------	AB64	POINT(	--------	AB41
B1	POS	-------C	AB92	POS	-------C	AB6D
B2	RAD	--------	ABD6	RAD	--------	ABB1
B3	RND	-------C	AF78	RND	-------C	AF49
B4	SGN	--------	ABAD	SGN	--------	AB88
B5	SIN	--------	A994	SIN	--------	A998
B6	SQR	--------	A7B4	SQR	--------	A7B4
B7	TAN	--------	A6C9	TAN	--------	A6BE
B8	TO	--------	AFOB	TO	--------	AEDC
B9	TRUE	-------C	ACEA	TRUE	-------C	ACC4
BA	USR	--------	ABFB	USR	--------	ABD2
BB	VAL	--------	AC55	VAL	--------	AC2F
BC	VPOS	-------C	AB9B	VPOS	-------C	AB76
BD	CHR$	--------	B3EE	CHR$	--------	B3BD
BE	GET$	--------	AFEE	GET$	--------	AFBF
BF	INKEY$	--------	B055	INKEY$	--------	B026
C0	LEFT$(	--------	AFFB	LEFT$(	--------	AFCC
C1	MID$(	--------	8068	MID$(	--------	B039
C2	RIGHT$(	--------	B01D	RIGHT$(	--------	AFEE
C3	STR$	--------	B0C3	STR$	--------	B094
C4	STRING$(	--------	B0F1	STRING$(	--------	BOC2
C5	EOF	-------C	ACDE	EOF	-------C	ACB8
C6	AUTO	---L----	905F	AUTO	---L----	90AC
C7	DELETE	---L----	8ECE	DELETE	---L----	8F31
C8	LOAD	------M-	BF2D	LOAD	------M-	BF24
C9	LIST	---L----	B5B5	LIST	---L----	B59C
CA	NEW	-------C	8A7D	NEW	-------C	8ADA
CB	OLD	-------C	8A3D	OLD	-------C	8AB6

CC	RENUMBER	---L----	8E37	RENUMBER	---L----	8FA3
CD	SAVE	--------	BEFA	SAVE	--------	BEE3
CE	---	--------	9839	---	--------	982A
CF	PTR	--------	BF39	PTR	--------	BF30
D0	PAGE	--------	9239	PAGE	--------	9283
D1	TIME	--------	927B	TIME	--------	92C9
D2	LOMEM	--------	9224	LOMEM	--------	926F
D3	HIMEM	--------	9212	HIMEM	--------	925D
D4	SOUND	------M-	B461	SOUND	------M-	B44C
D5	BPUT	------MC	BF61	BPUT	------MC	BF58
D6	CALL	------M-	8E6C	CALL	------M-	8ED2
D7	CHAIN	------M-	BF33	CHAIN	------M-	BF2A
D8	CLEAR	-------C	9326	CLEAR	-------C	928D
D9	CLOSE	------MC	BF9E	CLOSE	------MC	BF99
DA	CLG	-------C	8E57	CLG	-------C	8EBD
DB	CLS	-------C	8E5E	CLS	-------C	8EC4
DC	DATA	--R-----	8AED	DATA	--R-----	8B7D
DD	DEF	------M-	8AED	DEF	------M-	8B7D
DE	DIM	------M-	90DD	DIM	------M-	912F
DF	DRAW	-------C	93A5	DRAW	-------C	93E8
E0	END	-------C	8A50	END	-------C	8AC8
E1	ENDPROC	------M-	9310	ENDPROC	------M-	9356
E2	ENVELOPE	------M-	B49C	ENVELOPE	------M-	B472
E3	FOR	------M-	B7DF	FOR	------M-	B7C4
E4	GOSUB	---L--M-	B8B4	GOSUB	---L--M-	B888
E5	GOTO	---L--M-	B8EB	GOTO	---L--M-	B8CC
E6	GCOL	------M-	932F	GCOL	------M-	937A
E7	IF	------M-	9893	IF	------M-	98C2
E8	INPUT	------M-	BA62	INPUT	------M-	BA44
E9	LET	-----S--	8B57	LET	-----S--	8BE4
EA	LOCAL	------M-	92D5	LOCAL	------M-	9323
EB	MODE	------M-	935A	MODE	------M-	939A
EC	MOVE	------M-	93A1	MOVE	------M-	93E4
ED	NEXT	------M-	B6AE	NEXT	------M-	B695
EE	ON	------M-	B934	ON	------M-	B915
EF	VDU	------M-	93EF	VDU	------M-	942F
F0	PLOT	------M-	93AE	PLOT	------M-	93F1
F1	PRINT	------M-	8D33	PRINT	------M-	8D9A
F2	PROC	----F-M-	92B6	PROC	----F-M-	9304
E3	READ	------M-	BB39	READ	------M-	BB1F
F4	REM	--R---M-	8AED	REM	--R---M-	8B7D
F5	REPEAT	------M-	8BFF	REPEAT	------M-	BBE4
F6	REPORT	-------C	BFE6	REPORT	-------C	BFE4
F7	REST0RE	---L--M-	BB00	REST0RE	---L--M-	BAE6
F8	RETURN	-------C	B8D5	RETURN	-------C	B8B6
F9	RUN	-------C	BD29	RUN	-------C	BD11
FA	STOP	-------C	8A59	STOP	-------C	8ADO
FB	COLOUR	------M-	9346	COLOUR	------M-	938E
FC	TRACE	------M-	9243	TRACE	------M-	9295
FD	UNTIL	---L--M-	BBCC	UNTIL	---L--M-	BBB1
FE	WIDTH	------M-	B4CC	WIDTH	------M-	B4A0
FF	---	------M-	9839	---	------M-	BEC2

2.4 Program storage

Once the line has been tokenised, the command handler checks to see if it starts with a line number. If it is, it is inserted into the program (and the old line with the same number, if there is one, is deleted). The format of each line is as follows:

00	MSB of line number
01	LSB of line number
02	length byte (= ‘XX’)
03	first character of line text 04etc.
XX−1	&0D (carriage return) line terminator.
XX	start of next line

The length byte is used so that searching for a line number (for a ‘GOTO’ or ‘GOSUB’ statement) is much faster. If this length byte is not set up correctly, BASIC will give a ‘Bad program’ error (see section 9 .2 for a salvage routine).

The first character in memory at PAGE is a carriage return character: this gives something to ‘latch on to’ when BASIC checks for a ‘Bad program’. The routine that checks this also sets TOP to point to the next free location after the end of the program.

The end of the program is marked by a byte with the top bit set (i.e. &80 or greater) in the position which would be the MSB of the lirfe number of the next line. This is why line numbers greater than 32767 are not allowed: if one got in, the MSB of its line number would just mark the end of the program.

For example, the program ‘10PRINT A’ would be stored as (if PAGE = &1900).

&1900	&0D	carriage return at start of program
&1901	&00	MSB of line number
&1902	&0A	LSB of line number (10)
&1903	&07	length byte
&1904	&F1	‘PRINT’ token
&1905	&20	space character

&1906	&41	'A'
&1907	&0D	carriage return end of line marker
&1908	&FF	end of program marker

2.5 Executing statements

If the line input to the command handler did not start with a line number, it passes it on to the statement interpreter to decide what to do with it.

The statement interpreter is also used to RUN programs, as well as just interpreting statements and commands typed in command mode. The command handler has a special entry point to the statement interpreter, so that commands (like 'OLD') can only be executed in command mode, and not in the middle of a program.

The action of the statement interpreter is as follows:

1 It looks at the first character of the statement (skipping any spaces). If it is the token of a BASIC statement keyword (or a command keyword if we came from the command handler), then go to the corresponding statement handler (there is one of these for each statement or command) where the rest of that particular statement will be interpreted.

The *action address* of a particular token (the address to which the statement interpreter jumps when a token is found) is stored in the following format:

BASIC1	**BASIC2**	
&82CB+T	&82DF+T	LSB of action address
&833C+T	&8351+T	MSB of action address

where T is the number of the token (see table 2.1).

2 If the first character of the statement was not a statement keyword token, the statement interpreter checks to see if it is a variable name. If it is, it jumps to the assignment handler. This tries to assign the variable to the expression found after the '=' sign. If there wasn't an '=' after the variable name, it generates a 'Mistake' error (error number 4).

3 If the first character of the statement wasn't a variable name either, the statement interpreter checks to see if it is one of the other special symbols which can be at the start of a line. If it is a '*', it passes the rest of the line to the Operating System Command Line Interpreter (OSCLI) to be acted on. If it is a '[', it jumps into the assembler. If it is an '=', it jumps to the FN return statement handler (as this is the FN return statement).

4 If it wasn't any of those, it checks to see if the first character of the statement actually marks the end of the statement − in other words we have an empty statement. If it was, it goes back to stage I to interpret the next statement (or go to command mode if we have run out of statements to interpret). Most of the statement handlers jump to here when they have finished, to check that the text pointer is set up to point to the next statement.

5 Finally, if the character wasn't a *statement delimiter* either (a character marking the end of the statement), the statement interpreter gives up, and generates a 'Syntax error' (error number 16).