

# Appendix A - IE64 BASIC Keyword Abbreviations and Token Map

---

When you type a BASIC line and press RETURN, IE64 BASIC takes the text apart and writes it back to memory in a shorter form. Each reserved word becomes a single byte called a TOKEN. The body of the program in memory is a stream of these token bytes mixed with the characters of variable names, numeric literals, and quoted strings.

This appendix lists every token byte IE64 BASIC recognises, the keyword it stands for, and what kind of word it is. It also lists the short forms (abbreviations and aliases) the tokeniser accepts.

When you LIST a program, IE64 BASIC walks back through the token bytes and prints the full keyword, so a line typed as ? "HI" is shown as PRINT "HI".

## A.1 What a token is

---

A token is a value in the range \$80 through \$FF. Bytes below \$80 are literal characters - variable letters, digits, punctuation, the contents of quoted strings, and DATA fields. Bytes from \$80 upward stand in for a reserved word.

## A.2 Abbreviations and aliases

---

IE64 BASIC accepts the following short forms. Each one tokenises to the same byte as the keyword it stands for. The "Listed back as" column shows what LIST prints when that byte is detokenised.

You type	Token byte	Listed back as
?	\$9E (PRINT)	PRINT
WEND	\$AF (UNTIL)	UNTIL
TRON	\$92	TRON
TROFF	\$97	TROFF
DEF	\$97	TROFF
BLOAD	\$A3	BLOAD

DEF and TROFF share the byte \$97. When you type DEF and then LIST the program, the line comes back showing TROFF. The statement still behaves as DEF when it is followed by FN, and as TROFF (trace off) when it stands alone. See Chapter 2 for the full syntax of each.

The following reserved words are *not* tokenised at all. IE64 BASIC recognises them as literal characters at run time:

- DIR is a direct-mode command. It is recognised only at the BASIC prompt and has no token byte. It cannot appear inside a program line.
- TYPE is a direct-mode command. It is recognised only at the BASIC prompt and has no token byte. It cannot appear inside a program line.
- COMPILE is a direct-mode command. It is recognised only at the BASIC prompt and has no token byte.
- TRANSPILE is a direct-mode command. It is recognised only at the BASIC prompt and has no token byte.
- ASSEMBLE is a direct-mode command. It is recognised only at the BASIC prompt and has no token byte.

- RUN AOT is a direct-mode form. RUN itself has token byte \$8A, but the prompt recognises the following AOT word before ordinary RUN handling. There is no separate RUN AOT token.
- HOST is recognised as a raw statement when the line runs. It has no token byte. See Chapter 36 for the form and use of the HOST command.
- COSTART, COSTOP, and COWAIT are recognised as raw statements when the line runs. They have no token bytes. See Chapter 32.
- TRICOLOR is left as literal characters so the Voodoo dispatcher can recognise it at run time. See Chapter 9.

Every other reserved word tokenises to exactly one byte.

## A.3 Operator tokens

Hex	Symbol	Name
\$B1	+	PLUS
\$B2	-	MINUS
\$B3	*	MULT
\$B4	/	DIV
\$B5	^	POWER
\$B6	AND	AND
\$B7	EOR	EOR
\$B8	OR	OR
\$B9	>>	RSHIFT
\$BA	<<	LSHIFT
\$BB	>	GT
\$BC	=	EQUAL
\$BD	<	LT

The composite comparison operators <=, >=, and <> are stored as the base byte (\$BD for <, \$BB for >) followed by the raw second character (= or >). IE64 BASIC inspects the pair at run time. This keeps the token alphabet small.

## A.4 Statement tokens (\$80-\$E1)

These tokens share their byte values with the 1980s 68K EhBASIC. Where a word is marked **Function**, it returns a value and may appear only in an expression. Where it is marked **Statement**, it begins a command. **Keyword** denotes a fragment that appears inside another statement (TO, STEP, THEN, ELSE, etc.).

Hex	Name	Keyword	Kind
\$80	TK_END	END	Statement
\$81	TK_FOR	FOR	Statement
\$82	TK_NEXT	NEXT	Statement
\$83	TK_DATA	DATA	Statement

Hex	Name	Keyword	Kind
\$84	TK_INPUT	INPUT	Statement
\$85	TK_DIM	DIM	Statement
\$86	TK_READ	READ	Statement
\$87	TK_LET	LET	Statement
\$88	TK_DEC	DEC	Statement
\$89	TK_GOTO	GOTO	Statement
\$8A	TK_RUN	RUN	Statement
\$8B	TK_IF	IF	Statement
\$8C	TK_RESTORE	RESTORE	Statement
\$8D	TK_GOSUB	GOSUB	Statement
\$8E	TK_RETURN	RETURN	Statement
\$8F	TK_REM	REM	Statement
\$90	TK_STOP	STOP	Statement
\$91	TK_ON	ON	Statement
\$92	TK_EXT	Extended	Prefix
\$93	TK_INC	INC	Statement
\$94	TK_WAIT	WAIT	Statement
\$95	TK_LOAD	LOAD	Statement
\$96	TK_SAVE	SAVE	Statement
\$97	TK_DEF	TROFF	Statement (also accepts DEF; LIST always prints TROFF)
\$98	TK_POKE	POKE	Statement
\$99	TK_RESERVED_99	-	Reserved
\$9A	TK_RESERVED_9A	-	Reserved
\$9B	TK_CALL	CALL	Statement
\$9C	TK_DO	DO	Statement
\$9D	TK_LOOP	LOOP	Statement
\$9E	TK_PRINT	PRINT	Statement
\$9F	TK_CONT	CONT	Statement
\$A0	TK_LIST	LIST	Statement
\$A1	TK_CLEAR	CLEAR	Statement
\$A2	TK_NEW	NEW	Statement
\$A3	TK_WIDTH	BLOAD	Statement
\$A4	TK_GET	GET	Statement
\$A5	TK_SWAP	SWAP	Statement

Hex	Name	Keyword	Kind
\$A6	TK_BITSET	BITSET	Statement
\$A7	TK_BITCLR	BITCLR	Statement
\$A8	TK_TAB	TAB	Keyword
\$A9	TK_TO	TO	Keyword
\$AA	TK_FN	FN	Keyword
\$AB	TK_ELSE	ELSE	Keyword
\$AC	TK_THEN	THEN	Keyword
\$AD	TK_NOT	NOT	Operator
\$AE	TK_STEP	STEP	Keyword
\$AF	TK_UNTIL	UNTIL	Keyword
\$B0	TK_WHILE	WHILE	Keyword

## A.5 Function tokens

Hex	Name	Keyword	Returns
\$BE	TK_SGN	SGN	Number
\$BF	TK_INT	INT	Number
\$C0	TK_ABS	ABS	Number
\$C1	TK_USR	USR	Number
\$C2	TK_FRE	FRE	Number
\$C3	TK_POS	POS	Number
\$C4	TK_SQR	SQR	Number
\$C5	TK_RND	RND	Number
\$C6	TK_LOG	LOG	Number
\$C7	TK_EXP	EXP	Number
\$C8	TK_COS	COS	Number
\$C9	TK_SIN	SIN	Number
\$CA	TK_TAN	TAN	Number
\$CB	TK_ATN	ATN	Number
\$CC	TK_PEEK	PEEK	Number
\$CD	TK_RESERVED_CD	-	Reserved
\$CE	TK_RESERVED_CE	-	Reserved
\$CF	TK_SADD	SADD	Number
\$D0	TK_LEN	LEN	Number

Hex	Name	Keyword	Returns
\$D1	TK_STRS	STR\$	String
\$D2	TK_VAL	VAL	Number
\$D3	TK_ASC	ASC	Number
\$D4	TK_UCASES	UCASE\$	String
\$D5	TK_LCASES	LCASE\$	String
\$D6	TK_CHRS	CHR\$	String
\$D7	TK_HEXS	HEX\$	String
\$D8	TK_BINS	BIN\$	String
\$D9	TK_BITTST	BITTST	Number
\$DA	TK_MAX	MAX	Number
\$DB	TK_MIN	MIN	Number
\$DC	TK_PI	PI	Number
\$DD	TK_TWOPI	TWOPI	Number
\$DE	TK_VPTR	VARPTR	Number
\$DF	TK_LEFTS	LEFT\$	String
\$E0	TK_RIGHTS	RIGHT\$	String
\$E1	TK_MIDS	MID\$	String

VARPTR is a function. Using VARPTR in statement position is reported as an UNKNOWN STATEMENT error.

## A.6 Extended tokens (\$92, subtoken)

TK\_EXT is followed by one subtoken byte. Subtokens \$0C-\$7F are reserved for future IE64 BASIC extensions; \$80-\$FF are invalid in this token stream.

Subtoken	Name	Keyword	Kind
\$00	EXT_TRON	TRON	Statement
\$01	EXT_TROFF	TROFF	Statement
\$02	EXT_MEMALLOC	MEMALLOC	Function
\$03	EXT_POKE8	POKE8	Statement
\$04	EXT_POKE16	POKE16	Statement
\$05	EXT_POKE32	POKE32	Statement
\$06	EXT_POKE64	POKE64	Statement
\$07	EXT_PEEK8	PEEK8	Function
\$08	EXT_PEEK16	PEEK16	Function
\$09	EXT_PEEK32	PEEK32	Function

Subtoken	Name	Keyword	Kind
\$0A	EXT_PEEK64	PEEK64	Function
\$0B	EXT_MIDI	MIDI	Statement

POKE and PEEK remain one-byte aliases for byte-width POKE8 and PEEK8.

## A.7 Hardware-extension tokens (\$E2-\$FF)

The IE-specific commands live here. They begin a statement; the bytes that follow are arguments. See the chapters listed in the right-hand column for the syntax of each one.

Hex	Name	Keyword	See chapter
\$E2	TK_SCREEN	SCREEN	5
\$E3	TK_CLS	CLS	5
\$E4	TK_PLOT	PLOT	5
\$E5	TK_PALETTE	PALETTE	3
\$E6	TK_VSYNC_CMD	VSYNC	3
\$E7	TK_LOCATE	LOCATE	5
\$E8	TK_COLOR	COLOR	5
\$E9	TK_LINE_CMD	LINE	5
\$EA	TK_CIRCLE	CIRCLE	5
\$EB	TK_BOX	BOX	5
\$EC	TK_SCROLL_CMD	SCROLL	5
\$ED	TK_COPPER	COPPER	4
\$EE	TK_BLIT	BLIT	4
\$EF	TK_SOUND	SOUND	11
\$F0	TK_ENVELOPE	ENVELOPE	11
\$F1	TK_GATE	GATE	11
\$F2	TK_ULA	ULA	8
\$F3	TK_TED_CMD	TED	6
\$F4	TK_ANTIC	ANTIC	7
\$F5	TK_GTIA	GTIA	7
\$F6	TK_VOODOO	VOODOO	9
\$F7	TK_PSG_CMD	PSG	13
\$F8	TK_SID_CMD	SID	15
\$F9	TK_POKEY_CMD	POKEY	17
\$FA	TK_AHX	AHX	18

Hex	Name	Keyword	See chapter
\$FB	TK_SAP	SAP	17
\$FC	TK_ZBUFFER	ZBUFFER	9
\$FD	TK_VERTEX	VERTEX	9
\$FE	TK_TRIANGLE	TRIANGLE	9
\$FF	TK_TEXTURE	TEXTURE	9

## A.8 What the tokeniser does with words it does not know

Any word that is not in the table above is left as literal characters. The body of a quoted string is always copied byte-for-byte. The body of a DATA statement is copied byte-for-byte from the DATA keyword up to the next colon or end-of-line. The body of a REM statement is copied byte-for-byte to the end of the line.

This is why a variable named BAND cannot be confused with AND and a variable named BORDER cannot be confused with OR: the tokeniser checks the character before and after a match and rejects the match if either is a letter.

## A.8 Reading a tokenised line

Every stored program line begins with a sixteen-byte header followed by the tokenised content of the line. The layout is:

```
+0 (8 bytes) next-line pointer
+8 (4 bytes) line number
+12 (4 bytes) reserved
+16 (n bytes) tokenised content, terminated by a $00 byte
```

After the terminator, IE64 BASIC aligns the next line on an eight-byte boundary, so there may be one to seven padding bytes between the \$00 and the next stored line's header.

The next-line pointer is an eight-byte little-endian value. The line number and reserved field are four-byte little-endian values.

The end of the program is a single eight-byte qword containing zero, called the **terminator qword**. The next-line pointer of the last real program line is the address of this terminator qword, *not* zero. Walking the list, you read eight bytes through each header's next pointer; you reach the terminator when the eight bytes you read are themselves zero. The terminator has no line number and no content.

Suppose you type:

```
10 FOR I=1 TO 10:PRINT I:NEXT
```

Inspecting the line in memory with PEEK8 byte reads from the start of its header shows, in hex:

```

NN NN NN NN NN NN NN NN next-line pointer (8 bytes, little-endian)
0A 00 00 00                line number 10 (4 bytes, little-endian)
00 00 00 00                reserved
81                          FOR
20 49 BC 31 20             space "I" EQUAL "1" space
A9 20 31 30                T0 space "1" "0"
3A                          colon
9E 20 49                   PRINT space "I"
3A                          colon
82                          NEXT
00                          end-of-line null
                             (padding to 8-byte boundary, if any)

```

The eight NN bytes hold the address of the next line's header. If this is the last real program line, that address is the location of the terminator qword, not zero. The terminator qword itself reads as eight bytes of zero.

Tokens are emitted with no surrounding space. Spaces between keywords and their arguments survive tokenisation as the literal byte \$20. The null at offset N+16 terminates the stored line; the next-line pointer field of the following line begins at the next eight-byte boundary after that null.

When you LIST the line, IE64 BASIC walks back through the bytes, prints the line number in decimal, prints FOR for \$81, copies the literal characters, prints TO for \$A9, copies the literal characters, and so on. The form you see on screen is the same as the form you typed.