LINE\_SCAN

// edit line address

int charAddress = E\_LINE;

int lineNumber = E\_LINE\_NO(ref charAddress);

int statementNumber = 0;

while( )

{

statementNumber++;

if(statementNumber > 127) // 'Nonsense in BASIC'

{ ThrowErrorMessage(0x0B, lineNumber, statementNumber); }

STMT\_NEXT

char chr = GET\_CHAR(ref charAddress);

if(chr == 0xD) break to LINE\_END(); // carriage return ? i.e. an empty statement.

if(chr == 0x3A) // statement end marker ':' ? i.e. another type of empty statement.

{ NEXT\_CHAR(ref charAddress); continue; }

NEXT\_CHAR(ref charAddress); // advance pointer to a position after command

int commandIndex = chr – 0xCE; // subtract 'DEF FN' - first command

if(commandIndex < 0) // if less than a command raising 'Nonsense in BASIC'

{ ThrowErrorMessage(0x0B, lineNumber, statementNumber); }

Command command = basicCommands[commandIndex];

|  |  |
| --- | --- |
|  | **GET\_PARAM** |

foreach(byte param in command.Parameters)

{

If(param >= ‘ ‘)

{

// check that correct separator appears in statement

SEPARATOR(param);

}

Else

{

chr = GET\_CHAR(ref charAddress);

ParamClassRoutines[param]();

}

}

NEXT\_CHAR(ref charAddress);

}

Void SEPARATOR(char separator)

{

Char chr = GET\_CHAR(ref charAddress);

If(chr != separator) // 'Nonsense in BASIC'

{ ThrowErrorMessage(0x0B, lineNumber, statementNumber); }

Else { NEXT\_CHAR(ref charAddress); }

}

// When a command may be followed by an optional numeric expression e.g. RUN.

// If the end of statement has been reached then zero is used as the default.

int FETCH\_NUM(char chr)

{

If(chr == 0x0D || // *is character a carriage return ?*

chr == 0x3A) // *is it ':' ?*

{ calculator.Push(0); }

Else

{ ParamClass08(); }

}

// *End of statement?*

Void CHECK\_END()

{

Jump to STMT\_NEXT

}

// *class-00 e.g. CONTINUE ; no operand*

void ParamClass00()

{

ParamClass05(true)

}

// *class-01 e.g. LET A = 2\*3 ; a variable is required*

*// This class routine is also called from INPUT and READ to find the*

*// destination variable for an assignment*

void ParamClass01()

{

}

// CLASS-02

void ParamClass02()

{

}

// *class-03 e.g. RUN or RUN 200 ; optional operand*

void ParamClass03()

{

FETCH\_NUM(chr)

ParamClass00();

}

// CLASS-04

void ParamClass04()

{

}

// *class-05 e.g. PRINT ; variable syntax checked by routine*

void ParamClass05(bool endOfStatement)

{

Break scan loop

If(endOfStatement)

{ CHECK\_END }

ExecuteCommand(command.Parameters.Last)

}

// CLASS-06

void ParamClass06()

{

}

// CLASS-07

void ParamClass07()

{

}

// CLASS-08

void ParamClass08()

{

}

// CLASS-09

void ParamClass09()

{

}

// CLASS-0A

void ParamClass0A()

{

}

// CLASS-0B

void ParamClass0B()

{

}

int E\_LINE\_NO(ref int charAddress) // return value in BC

{

// GET-CHAR skips any noise and white-space to point exactly at the first digit.

char firstDigit = GET\_CHAR(ref charAddress);

INT\_TO\_FP // routine INT-TO-FP will read digits till a non-digit found.

int lineNumber = FP\_TO\_BC // routine FP-TO-BC will retrieve number from stack at membot.

if(lineNumber > 9999) // 'Nonsense in BASIC'

{ ThrowErrorMessage(0x0B, lineNumber, statementNumber); }

return lineNumber;

}

int INT\_TO\_FP\_TO\_BC (char chr, ref int charAddress)

{

int number = 0;

while( 0x30 <= chr <= 0x39 )

{

int digit = (chr – 0x30); // convert from ASCII to digit

number = 10 \* number + digit;

chr = memory[charAddress++];

}

return number;

}

char NEXT\_CHAR(ref int charAddress) // return value in A,

{

charAddress++;

return GET\_CHAR(charAddress);

}

char GET\_CHAR(ref int charAddress)

{

char chr = memory[charAddress];

while(SKIP\_OVER(chr, ref charAddress))

{

chr = memory[charAddress++];

}

return chr;

}

bool SKIP\_OVER(char chr, ref int charAddress)

{

if(chr >= 0x21) return false; // Higher than space

if(chr == 0x0D) return false; // Carriage return

if(chr < 0x10) return true; // 0-15d

if(chr >= 0x18) return true; // 24-32d

// now leaves 16d-23d

charAddress++; // all above have at least one extra character to be stepped over

if(chr >= 0x16) // controls 22d ('at') and 23d ('tab') have two

{ charAddress++; } // step over the second character of 'at'/'tab'.

return true;

}

// The BASIC interpreter has found a command code $CE - $FF which is then reduced to range $00 - $31.

// Parameter table where a list of class codes, separators and addresses relevant to the command exists.

// For each command there exists a variable list of parameters.

// If the character is greater than a space it is a required separator.

// If less, then it is a command class in the range 00 - 0B.

// Note that classes 00, 03 and 05 will fetch the addresses from this table.

// Some classes e.g. 07 and 0B have the same address in all invocations

// and the command is re-computed from the low-byte of the parameter address.

// Some e.g. 02 are only called once so a call to the command is made from

// within the class routine rather than holding the address within the table.

// Some class routines check syntax entirely and some leave this task for the

// command itself.

// Others for example CIRCLE (x,y,z) check the first part (x,y) using the

// class routine and the final part (,z) within the command.

// The last few commands appear to have been added in a rush but their syntax

// is rather simple e.g. MOVE "M1","M2"

// Class-00 - No further operands.

// Class-01 - A variable is required.

// Class-02 - An expression, numeric or string, must follow.

// Class-03 - A numeric expression may follow else default to zero.

// Class-04 - A single character variable must follow.

// Class-05 - Variable syntax checked by routine.

// Class-06 - A numeric expression must follow.

// Class-07 - Offset address is converted to colour code.

// Class-08 - Two comma-separated numeric expressions required.

// Class-09 - Two comma-separated numeric expressions required with optional colour items.

// Class-0A - A string expression must follow.

// Class-0B - Offset address converted to tape command.

static Command[] basicCommands = new Command[] {

// P\_DEF\_FN

new Command(0xCE, "DEF FN", 0x05, DEF\_FN),

// P\_CAT

new Command(0xCF, "CAT", 0x00, CAT\_ETC),

// P\_FORMAT

new Command(0xD0, "FORMAT", 0x0A, 0x00, CAT\_ETC),

// P\_MOVE

new Command(0xD1, "MOVE", 0x0A, 0x2C, 0x0A, 0x00, CAT\_ETC),

// P\_ERASE

new Command(0xD2, "ERASE", 0x0A, 0x00, CAT\_ETC),

// P\_OPEN

new Command(0xD3, "OPEN #", 0x06, 0x2C, 0x0A, 0x00, OPEN),

// P\_CLOSE

new Command(0xD4, "CLOSE #", 0x06, 0x00, CLOSE),

// P\_MERGE

new Command(0xD5, "MERGE", 0x0B),

// P\_VERIFY

new Command(0xD6, "VERIFY", 0x0B),

// P\_BEEP

new Command(0xD7, "BEEP", 0x08, 0x00, beep),

// P\_CIRCLE

new Command(0xD8, "CIRCLE", 0x09, 0x05, CIRCLE),

// P\_INK

new Command(0xD9, "INK", 0x07),

// P\_PAPER

new Command(0xDA, "PAPER", 0x07),

// P\_FLASH

new Command(0xDB, "FLASH", 0x07),

// P\_BRIGHT

new Command(0xDC, "BRIGHT", 0x07),

// P\_INVERSE

new Command(0xDD, "INVERSE", 0x07),

// P\_OVER

new Command(0xDE, "OVER", 0x07),

// P\_OUT

new Command(0xDF, "OUT", 0x08, 0x00, OUT\_),

// P\_LPRINT

new Command(0xE0, "LPRINT", 0x05, LPRINT),

// P\_LLIST

new Command(0xE1, "LLIST", 0x05, LLIST),

// P\_STOP

new Command(0xE2, "STOP", 0x00, STOP),

// P\_READ

new Command(0xE3, "READ", 0x05, READ),

// P\_DATA

new Command(0xE4, "DATA", 0x05, DATA\_),

// P\_RESTORE

new Command(0xE5, "RESTORE", 0x03, RESTORE),

// P\_NEW

new Command(0xE6, "NEW", 0x00, NEW),

// P\_BORDER

new Command(0xE7, "BORDER", 0x06, 0x00, BORDER),

// P\_CONT

new Command(0xE8, "CONTINUE", 0x00, CONTINUE),

// P\_DIM

new Command(0xE9, "DIM", 0x05, INPUT),

// P\_REM

new Command(0xEA, "REM", 0x05, REM),

// P\_FOR

new Command(0xEB, "FOR", 0x04, 0x3D /\*’=’\*/, 0x06, 0xCC /\*’TO’\*/, 0x06, 0x05, FOR),

// P\_GO\_TO

new Command(0xEC, "GO TO", 0x06, 0x00, GO\_TO),

// P\_GO\_SUB

new Command(0xED, "GO SUB", 0x06, 0x00, GO\_SUB),

// P\_INPUT

new Command(0xEE, "INPUT", 0x05, INPUT),

// P\_LOAD

new Command(0xEF, "LOAD", 0x0B),

// P\_LIST

new Command(0xF0, "LIST", 0x05, LIST),

// P\_LET

new Command(0xF1, "LET", 0x01, 0x3D /\*’=’\*/, 0x02),

// P\_PAUSE

new Command(0xF2, "PAUSE", 0x06, 0x00, PAUSE),

// P\_NEXT

new Command(0xF3, "NEXT", 0x04, 0x00, NEXT),

// P\_POKE

new Command(0xF4, "POKE", 0x08, 0x00, POKE),

// P\_PRINT

new Command(0xF5, "PRINT", 0x05, PRINT),

// P\_PLOT

new Command(0xF6, "PLOT", 0x09, 0x00, PLOT),

// P\_RUN

new Command(0xF7, "RUN", 0x03, RUN),

// P\_SAVE

new Command(0xF8, "SAVE", 0x0B),

// P\_RANDOM

new Command(0xF9, "RANDOMIZE", 0x03, RANDOMIZE),

// P\_IF

new Command(0xFA, "IF", 0x06, 0xCB /\*’THEN’\*/, 0x05, IF),

// P\_CLS

new Command(0xFB, "CLS", 0x00, CLS),

// P\_DRAW

new Command(0xFC, "DRAW", 0x09, 0x05, DRAW),

// P\_CLEAR

new Command(0xFD, "CLEAR", 0x03, CLEAR),

// P\_RETURN

new Command(0xFE, "RETURN", 0x00, RETURN),

// P\_COPY

new Command(0xFF, "COPY", 0x00, COPY)

}

string MAIN\_4/ThrowErrorMessage(byte errorCode, int lineNumber, int statemenNumber)

{

string errorMessage;

errorCode++; // increment to give true code.

if(errorCode >= 10) // is it a print-ready digit 0-9 ?

{ errorCode += 7; } // if not add ASCII offset to letters (errorCode 10 => letter A)

errorMessage += (char)(errorCode + 48); // add 48 decimal to give the ASCII char

errorMessage += " "; // followed by a space.

errorMessage += errorMessages[errorCode];

errorMessage += ", "; // comma and space message

errorMessage += lineNumber; // fetch PPC the current line number.

errorMessage += ":";

errorMessage += statementNumber; // then SUBPPC for statement, limited to 127

}

static string[] errorMessages = new string[]

{

"OK", // 0

"NEXT without FOR", // 1

"Variable not found", // 2

"Subscript wrong", // 3

"Out of memory", // 4

"Out of screen", // 5

"Number too big", // 6

"RETURN without GOSUB", // 7

"End of file", // 8

"STOP statement", // 9

"Invalid argument", // A

"Integer out of range", // B

"Nonsense in BASIC", // C

"BREAK - CONT repeats", // D

"Out of DATA", // E

"Invalid file name", // F

"No room for line", // G

"STOP in INPUT", // H

"FOR without NEXT", // I

"Invalid I/O device", // J

"Invalid colour", // K

"BREAK into program", // L

"RAMTOP no good", // M

"Statement lost", // N

"Invalid stream", // O

"FN without DEF", // P

"Parameter error", // Q

"Tape loading error" // R

}

CH\_ADD will be addressing somewhere -

1) in the BASIC program area during line execution.

2) in workspace if evaluating, for example, a string expression.

3) in the edit buffer if parsing a direct command or a new BASIC line.

4) in workspace if accepting input but not that from INPUT LINE.

CH\_ADDplus1

HL = CH\_ADD++; // chrpointer++

A = (HL); // chr = \*chrpointer