Interpreter Command Documentation

Table of Contents

Variables

VAR - Declaring Variables

Syntax:

```
VAR name1, name2, name3
VAR Name = Value
VAR array[size]
VAR array[size] = {value1, value2, ...}
VAR array[size] = {index: value1, value2, ...}        Filling from a position
VAR matrix[rows, columns]
VAR matrix[rows, columns] = {{value1, value2}, {value3, value4}, ...}
VAR matrix[rows, columns] = {row: {value1, value2}, ...}     From line N
VAR matrix[rows, columns] = {{column: value1, ...}, ...}          With column N in each line
VAR matrix[rows, columns] = {row: {column: value1, ...}, ...}  Combined option
```

Description:
Declaration of scalar variables, one-dimensional and two-dimensional arrays. Supports comma-separated multiple declarations.

All arrays (1D and 2D) are allocated from a total memory pool of 32KB (8192 float elements).

Examples:

```
Simple Announcement
VAR A, B, C, DD, D2S

With Initialization
VAR A=1, B=2, C=3

Mixed
VAR A=5, B, C=10

One-dimensional arrays
VAR A[10], B, C[5]

One-dimensional arrays with initialization
VAR A[3]={1,2,3}, B=5, C

Two-dimensional arrays
VAR MATRIX[3, 4] // Matrix 3x4 (3 rows, 4 columns)
VAR GRID[5, 5] // 5x5 grid

Two-dimensional arrays with initialization
VAR MAT[2, 3] = {{1, 2, 3}, {4, 5, 6}}
VAR IDENT[3, 3] = {{1, 0, 0}, {0, 1, 0}, {0, 0, 1}}

Mixed ad
VAR A[5], MATRIX[3, 3], B = 10

One-dimensional arrays initialized from an arbitrary position
VAR A[10] = {5: 1,2,3,4,5}           // A[0..4]=0, A[5..9]={1,2,3,4,5}
VAR B[20] = {10: 100,200,300}        // B[0..9]=0, B[10..12]={100,200,300}, B[13..19]=0
VAR C[15] = {12: 99,88,77}           // C[0..11]=0, C[12..14]={99,88,77}

Two-dimensional arrays initialized from line N
VAR M[5,5] = {2: {1,2,3}, {4,5,6}} // Start from the 2nd line
M[0..1,*]=0, M[2,0..2]={1,2,3}, M[3,0..2]={4,5,6}, remainder=0

Two-dimensional arrays initialized with column N on each line
VAR M[5,5] = {{2: 1,2,3}, {1: 4,5,6}} // 0th row from column 2, 1st from column 1
// M[0,0..1]=0, M[0,2..4]={1,2,3}
// M[1,0]=0, M[1,1..3]={4,5,6}, M[1,4]=0

Two-dimensional arrays: combination (start with line N and column M on each line)
VAR M[5,5] = {2: {1:10,20}, {0:30,40,50}} // From the 2nd line, each from its own column
// M[0..1,*]=0
// M[2,0]=0, M[2,1..2]={10,20}, M[2,3..4]=0
// M[3,0..2]={30,40,50}, M[3,3..4]=0
// M[4,*]=0
```

Working with two-dimensional arrays:

```
Announcement
VAR MATRIX[3, 4]

Recording an Item
MATRIX[0, 0] = 10
MATRIX[1, 2] = 25.5
MATRIX[2, 3] = 100

Read an item
VAR X = MATRIX[1, 2]

Output of the entire matrix (by rows)
PRINT MATRIX

Single Line Output
PRINT MATRIX[1]

Single Item Output
PRINT MATRIX[1, 2]

Use in Expressions
VAR SUM = MATRIX[0, 0] + MATRIX[1, 1] + MATRIX[2, 2]
```

Full example: Working with a temperature matrix

```
Declaring a 4x4 Temperature Storage Matrix
VAR TEMP[4, 4]

Matrix Filling
VAR ROW = 0
WHILE ROW < 4
    VAR COL = 0
    WHILE COL < 4
        TEMP[ROW, COL] = 20 + RND(0, 10) // Temperatures from 20 to 30
        COL = COL + 1
    WEND
    ROW = ROW + 1
WEND

Output of the entire matrix
PRINT "Temperature matrix:"
PRINT TEMP

Find the maximum temperature
VAR MAX_TEMP = TEMP[0, 0]
VAR MAX_ROW = 0
VAR MAX_COL = 0

ROW = 0
WHILE ROW < 4
    VAR COL = 0
    WHILE COL < 4
        IF TEMP[ROW, COL] > MAX_TEMP THEN
            MAX_TEMP = TEMP[ROW, COL]
            MAX_ROW = ROW
            MAX_COL = COL
        ENDIF
        COL = COL + 1
    WEND
    ROW = ROW + 1
WEND

PRINT "Max. Temperature:", MAX_TEMP.1, "°C"
PRINT "Position: [", MAX_ROW, ",", MAX_COL, "]"

Output diagonal
PRINT "Diagonal:"
VAR I = 0
WHILE I < 4
    PRINT TEMP[I, I].1
```

```
    I = I + 1
WEND
```

Important Notes:

- Two-dimensional arrays cannot be automatically created by assignment - they must be declared via VAR
- Indexes start at 0: for array [3, 4], indexes [0..2, 0..3] are allowed
- The 'PRINT MATRIX' output outputs the matrix line by line (each line on a new line)
- The 'PRINT MATRIX[row]' output outputs a single row of the matrix
- Two-dimensional arrays are stored in the same memory pool as one-dimensional arrays (32KB, 8192 elements)
- The size of the NxM matrix occupies N×M elements from the general pool
- All numbers are stored as float (32-bit, ~6-7 digit precision, range ±3.4×10³⁸)

COPY - Copying Arrays

Syntax:

```
COPY(src, src_pos, dst, dst_pos, count)
```

Options:

- 'src' - the name of the source array (1D or 2D)
- 'src_pos' - initial position in the source array (index with 0)
- 'dst' is the name of the target array (1D or 2D)
- 'dst_pos' - initial position in the target array (index with 0)
- 'count' - the number of elements to be copied

Description:
Copies a specified number of elements from one array to another. It works with both one-dimensional and two-dimensional arrays (the elements of 2D arrays are referenced linearly).

All parameters can be expressions (including variables).

Examples:

```
One-dimensional arrays
VAR A[10] = {1,2,3,4,5,6,7,8,9,10}
VAR B[10]

Copying 3 items from A[5] to B[0]
COPY(A, 5, B, 0, 3)
Result: B[0..2] = {6,7,8}, B[3..9] = 0

Copying using variables
VAR SRC_POS = 2
VAR DST_POS = 5
VAR CNT = 4
COPY(A, SRC_POS, B, DST_POS, CNT)
Result: B[5..8] = {3,4,5,6}

Copying a Part of an Array to Itself
VAR DATA[20] = {0: 10,20,30,40,50}
COPY(DATA, 0, DATA, 5, 5)
Result: DATA[0..4] and DATA[5..9] contain {10,20,30,40,50}

Two-dimensional arrays (linear inversion)
VAR M1[3, 4] = {{1,2,3,4}, {5,6,7,8}, {9,10,11,12}}
VAR M2[3, 4]

Copy the second line M1 (items 4-7) to the first line M2 (items 0-3)
COPY(M1, 4, M2, 0, 4)
Result: M2[0,*] = {5,6,7,8}
```

```
Copy the entire matrix
COPY(M1, 0, M2, 0, 12) // 3×4 = 12 elements
```

Important Notes:

- Both arrays must be pre-declared via VAR
- Bounds check: copying should not go beyond arrays
- An error message is displayed when going beyond the boundaries
- For 2D arrays, the elements are numbered line by line: [0,0], [0,1], ..., [0,cols-1], [1,0], ...
- You can copy data from the array to yourself (if the ranges do not overlap)

Populating Array Ranges

There are two ways to populate ranges: a single value and a list of values.

Syntax:

```
Single Value Filling
array[start-end] = value // 1D array
array[row, column_start-column_end] = value // 2D: single row, range of columns
array[row_start-row_end, column] = value // 2D: range of rows, single column
array[row_start-row_end, column_start-column_end] = value // 2D: rectangular area

Populating with a list of values
array[start-end] = {value1, value2, ...}           1D Array
array[row, column_start-column_end] = {value1, value2, ...}   2D: Column Range
array[row_start-row_end, column] = {value1, value2, ...}    2D: Range of Rows
array[row_start-row_end, column_start-column_end] = {value1, value2, ...}  2D: Rectangular area
```

Description:
Fills the specified range of array elements. When filling in the list of values, the values are assigned sequentially from left to right, from top to bottom (for 2D arrays).

Examples: Single Value Filling

```
One-dimensional arrays
VAR A[100]
A[0-99] = 0 // Fill the entire array with zeros
A[10-19] = 100 // Fill A[10..19] with 100
A[5-14] = 42 // Fill A[5..14] with 42

With variables
VAR START = 10
VAR END = 20
A[START-END] = 255

Two-dimensional arrays
VAR M[5, 10]
M[0-4, 0-9] = 0 // Fill the entire matrix with zeros
M[1-3, 2-7] = 10 // Fill the 3x6 rectangle with the value 10
M[2, 0-9] = 5 // Fill in the third line in its entirety
M[0-4, 5] = 7 // Fill the sixth column in its entirety

Creating a Frame Around the Matrix
VAR GRID[10, 10]
GRID[0-9, 0-9] = 0 // Clear the entire grid
GRID[0, 0-9] = 1 // Upper bound
GRID[9, 0-9] = 1 // Lower bound
GRID[0-9, 0] = 1 // Left border
GRID[0-9, 9] = 1 // Right border (turns out to be a frame)
```

Examples: Populating with a list of values

```
One-dimensional arrays
VAR MAS[20]

Fill the range with a list of values
MAS[5-10] = {4, 6, 8, 0, 1, 2}
Result: MAS[5]=4, MAS[6]=6, MAS[7]=8, MAS[8]=0, MAS[9]=1, MAS[10]=2

Range with expressions in the list
VAR BASE = 100
MAS[0-4] = {BASE, BASE+10, BASE+20, BASE+30, BASE+40}
Result: MAS[0..4] = {100, 110, 120, 130, 140}

Two-dimensional arrays
VAR M[10, 10]

Fill a range of columns in a single row
M[1, 5-7] = {2, 5, 6}
Result: M[1,5]=2, M[1,6]=5, M[1,7]=6

Populate a range of rows in a single column
M[0-2, 3] = {10, 20, 30}
Result: M[0,3]=10, M[1,3]=20, M[2,3]=30

Fill rectangular area (3 rows × 3 columns = 9 items)
M[1-3, 1-3] = {1, 2, 3, 4, 5, 6, 7, 8, 9}
Result:
// M[1,1]=1  M[1,2]=2  M[1,3]=3
// M[2,1]=4  M[2,2]=5  M[2,3]=6
// M[3,1]=7  M[3,2]=8  M[3,3]=9

Creating a Chessboard (Part)
VAR BOARD[8, 8]
BOARD[0, 0-7] = {1, 0, 1, 0, 1, 0, 1, 0} // First line
BOARD[1, 0-7] = {0, 1, 0, 1, 0, 1, 0, 1} // Second line
```

Case Studies

```
Initializing a lookup table
VAR LUT[16]
LUT[0-15] = {0, 15, 31, 47, 63, 79, 95, 111, 127, 143, 159, 175, 191, 207, 223, 255}

Filling the buffer with a template
VAR BUFFER[100]
BUFFER[0-9] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 0}
BUFFER[10-19] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 0}

Creating a Gradient on the Screen (Luminance Matrix)
VAR BRIGHTNESS[5, 10]
BRIGHTNESS[0, 0-9] = {0, 28, 56, 84, 112, 140, 168, 196, 224, 255}
BRIGHTNESS[1, 0-9] = {0, 28, 56, 84, 112, 140, 168, 196, 224, 255}
BRIGHTNESS[2, 0-9] = {0, 28, 56, 84, 112, 140, 168, 196, 224, 255}
```

Important Notes:

- Indexes start at 0
- Range includes both ends: 'A[5-10]' will fill elements 5, 6, 7, 8, 9, 10 (6 elements)
- The beginning of the range must be <= end
- The array must be pre-declared via VAR
- Bounds check: the range must not extend beyond the array
- An error message is displayed when going beyond the boundaries
- **When filling in the list of values:**
  - The number of values may be less than the range size (the remaining elements will not change)
  - Values are assigned sequentially: for 1D from left to right, for 2D from left to right, from top to bottom
  - All values in the list can be expressions (constants, variables, calculations)

- For 2D arrays, when filling a rectangular area, the values are line-by-line

---

Number systems

The interpreter supports working with numbers in various number systems: decimal, hexadecimal, and binary.

Number literals

Syntax:

```
Decimal numbers (regular)
VAR A = 255
VAR B = 100

Hexadecimal numbers (prefix 0x or 0X)
VAR C = 0xFF // 255 in decimal
VAR D = 0x10 // 16 in decimal
VAR E = 0xABCD // 43981 in decimal

Binary numbers (prefix 0b or 0B)
VAR F = 0b1111 // 15 in decimal
VAR G = 0b1010 // 10 in decimal
VAR H = 0b111111111 // 255 in decimal
```

Description:

Literals allow you to write numbers in a convenient number system directly in the code:

- **Decimals** - regular numbers: '123', '255', '1000'
- **Hexadecimal** - prefixed with '0x': '0xFF', '0x10', '0xABCD'
- **Binary** - prefixed with '0b': '0b1010', '0b11111111'

All numbers are internally stored in decimal form, but you can choose a convenient format for writing.

Examples of use:**

```
Working with colors (hexadecimal format is more convenient)
VAR RED = 0xF800 // RGB565: red
VAR GREEN = 0x07E0 // RGB565: green
VAR BLUE = 0x001F // RGB565: Blue

DISPLAY_FILL_RECT(0, 0, 100, 100, 0xF800)

Working with bitmasks
VAR MASK1 = 0b00001111 // Low 4 bits
VAR MASK2 = 0b11110000 // Low 4 bits
VAR FLAGS = 0b10100101 // Flag Set

Mixed use
VAR DEC = 100 // Decimal
VAR HEX = 0x64 // Same number in hex (100)
VAR BIN = 0b1100100 // Same number in binary (100)

IF DEC == HEX THEN
    PRINT "Equal!"      It will bring out "Equals!"
ENDIF
```

Output numbers in different formats

Use with the PRINT command

Syntax:

```
PRINT variable // Decimal format (default)
PRINT variable.h // Hexadecimal format (0xXXXX)
PRINT variable.b // Binary format (0bXXXXXXXX)
```

Examples:

```
VAR NUM = 255

Output in different formats
PRINT NUM // Outputs: 255
PRINT NUM.h // Outputs: 0xFF
PRINT NUM.b // Outputs: 0b11111111

Combined Output
PRINT "Dec:", NUM, "Hex:", NUM.h, "Bin:", NUM.b
Output: Dec: 255 Hex: 0xFF Bin: 0b111111111

Working with Variables
VAR COLOR = 0xF800
PRINT "Color value:", COLOR      // 63488
PRINT "Color hex:", COLOR.h      // 0xF800

Arrays
VAR DATA[3] = {10, 255, 0b1010}
PRINT "Array[0]:", DATA[0].h     // 0xA
PRINT "Array[1]:", DATA[1].b     // 0b11111111
PRINT "Array[2]:", DATA[2]       // 10
```

Formatting features:

- **Decimal** (default): Prints the number as is ('255')
- Hex (.h): Adds the '0x' prefix and outputs in uppercase ('0xFF')
- **Binary** (.b): adds the '0b' prefix and outputs only significant bits ('0b1010' instead of '0b00001010')

Case Studies

Example 1: Debugging ADC values

```
VAR V1 = ADC_IN1

PRINT "ADC Channel 1:"
PRINT "  Decimal: ", V1
PRINT "  Hex: ", V1.h
PRINT "  Binary: ", V1.b

On display
DISPLAY_TEXT(0, 0, "ADC1: ", V1.2, " V", Font_7x10, 1, 0xFFFF)
DISPLAY_TEXT(0, 20, "Raw: ", V1.h, Font_7x10, 1, 0xFFFF)
```

Example 2: Working with Color Codes

```
Creating Color
VAR R=255, G=128, B=64
VAR COLOR = RGB565(R, G, B)

Color Information Output
PRINT "RGB:", R, G, B
PRINT "RGB565:", COLOR.h // Hex is convenient for colors
PRINT "RGB565 bits:", COLOR.b // Show bit structure

Display
```

```
DISPLAY_FILL_RECT(10, 10, 100, 50, COLOR)
DISPLAY_TEXT(10, 70, "Color: ", COLOR.h, Font_7x10, 1, 0xFFFF)
```

Example 3: Bitmasks

```
Defining Bit Flags
VAR FLAG_ENABLE = 0b00000001
VAR FLAG_READY  = 0b00000010
VAR FLAG_ERROR  = 0b00000100
VAR FLAG_DONE   = 0b00001000

VAR STATUS = 0b00000101  // ENABLE + ERROR

PRINT "Status register:"
PRINT "  Binary: ", STATUS.b
PRINT "  Hex: ", STATUS.h

Checking Flags (Simple Comparison)
IF STATUS == FLAG_ENABLE THEN
    PRINT "Device enabled"
ENDIF
```

Example 4: Number System Converter

```
VAR INPUT = 42

DISPLAY_CLEAR()
DISPLAY_TEXT(10, 10, "Number converter", Font_11x18, 1, RGB565(255, 255, 0))

DISPLAY_TEXT(10, 50, "Input: ", INPUT, Font_7x10, 1, 0xFFFF)
DISPLAY_TEXT(10, 70, "Decimal: ", INPUT, Font_7x10, 1, RGB565(255, 255, 255))
DISPLAY_TEXT(10, 90, "Hex: ", INPUT.h, Font_7x10, 1, RGB565(0, 255, 0))
DISPLAY_TEXT(10, 110, "Binary: ", INPUT.b, Font_7x10, 1, RGB565(0, 255, 255))

You can also use literals to validate
VAR TEST1 = 0x2A // 42 in hex
VAR TEST2 = 0b101010 // 42 in binary

IF INPUT == TEST1 AND INPUT == TEST2 THEN
    DISPLAY_TEXT(10, 140, "All equal!", Font_7x10, 1, RGB565(0, 255, 0))
ENDIF
```

Example 5: Conversion Table

```
DISPLAY_CLEAR()
DISPLAY_TEXT(10, 0, "DEC  HEX   BINARY", Font_7x10, 1, RGB565(255, 255, 0))

FOR I = 0 TO 15
    VAR Y = 20 + I * 15
    DISPLAY_TEXT(10, Y, I, Font_7x10, 1, 0xFFFF)
    DISPLAY_TEXT(50, Y, I.h, Font_7x10, 1, 0xFFFF)
    DISPLAY_TEXT(100, Y, I.b, Font_7x10, 1, 0xFFFF)
NEXT

Displays a table:
// 0   0x0   0b0
// 1   0x1   0b1
// 2   0x2   0b10
// ...
// 15  0xF   0b1111
```

Notes

- The literals '0x' and '0b' can be used wherever a number is expected
- '.h' and '.b' formats work only for output (PRINT, DISPLAY_TEXT)
- The case of the prefix is not important: '0xFF' = '0Xff' = '0XFF'
- Binary format outputs only significant bits (no leading zeros)
- Hexadecimal format outputs uppercase (A-F)
- All numbers are internally stored as a double (floating-point)
- When inferring to hex/binary, the fractional part is discarded

---

Color Functions

The interpreter supports three color formats for different types of devices:

Function	Bits	Format	Application
RGB565()	16-bit	R5G6B5	ILI9341, ILI9488, ST7789, ST7735 Displays
RGB666()	18-bit	R6G6B6	ST7796 Display
RGB888()	24-bit	R8G8B8	WS2812 LED, matrices

RGB565(red, green, blue)

Syntax:

```
RGB565(red, green, blue)
```

Description:
Converts RGB888 (0-255) to RGB565 (16-bit) for ILI9341/ILI9488/ST7789/ST7735 displays.

Options:

- 'red' (0-255) - red component
- 'green' (0-255) - green component
- 'blue' (0-255) - blue component

Format:
RGB565 uses 16 bits to store color:

- Red: 5-bit (0-31)
- Green: 6 bits (0-63)
- Blue: 5-bit (0-31)

Examples:

```
Primary colors
VAR RED = RGB565(255, 0, 0)
VAR GREEN = RGB565(0, 255, 0)
VAR BLUE = RGB565(0, 0, 255)
VAR WHITE = RGB565(255, 255, 255)

Use with 16-bit displays
DISPLAY_INIT(SPI, ILI9341, 320, 240)
DISPLAY_FILL_RECT(0, 0, 100, 100, RGB565(255, 128, 64))

With variables
VAR R=255, G=128, B=64
```

```
VAR COLOR = RGB565(R, G, B)
DISPLAY_FILL_CIRCLE(160, 120, 50, COLOR)

With arrays
VAR COLORS[3] = {255, 128, 64}
VAR C = RGB565(COLORS[0], COLORS[1], COLORS[2])
```

Note:

- RGB565 is the most common format for TFT displays
- Saves memory compared to RGB888
- Green uses 6 bits (more precision) for better eye perception

---

RGB888(red, green, blue)

Syntax:

```
RGB888(red, green, blue)
```

Description:
Converts RGB components to RGB888 (24-bit): 0xRRGGBB

Options:

- 'red' (0-255) - red component
- 'green' (0-255) - green component
- 'blue' (0-255) - blue component

Format:
RGB888 uses the full 8 bits per color channel:

- Red: 8-bit (0-255)
- Green: 8 bits (0-255)
- Blue: 8-bit (0-255)

Application:

- Addressable LED strips WS2812
- LED matrices
- Color operations with maximum precision

Examples:

```
Primary colors
VAR RED = RGB888(255, 0, 0)
VAR GREEN = RGB888(0, 255, 0)
VAR BLUE = RGB888(0, 0, 255)

For WS2812 LED Strip
VAR LEDS[10]
FOR I = 0 TO 9
    LEDS[I] = RGB888(255, I*25, 0)
NEXT
WS2812_SEND LEDS[]

For LED matrix
MATRIX_INIT(16, 16)
VAR COLOR = RGB888(100, 50, 200)
MATRIX_SET(8, 8, COLOR)
MATRIX_UPDATE()
```

```
With expressions
VAR BRIGHTNESS = 128
VAR WARM_WHITE = RGB888(BRIGHTNESS, BRIGHTNESS*0.9, BRIGHTNESS*0.7)
```

Note:

- RGB888 delivers maximum color quality (16.7 million colors)
- Takes up more memory than RGB565 or RGB666
- The result in the 0xRRGGBB format can be used with hex literals

---

RGB666(red, green, blue)

Syntax:

```
RGB666(red, green, blue)
```

Description:
Converts RGB888 (0-255) to RGB666 (18-bit) for the ST7796 display.

Options:

- 'red' (0-255) - red component
- 'green' (0-255) - green component
- 'blue' (0-255) - blue component

Format:
RGB666 uses 6 bits per color channel:

- Red: 6-bit (0-63)
- Green: 6 bits (0-63)
- Blue: 6-bit (0-63)

Examples:

```
Primary colors
VAR RED = RGB666(255, 0, 0)
VAR GREEN = RGB666(0, 255, 0)
VAR BLUE = RGB666(0, 0, 255)
VAR WHITE = RGB666(255, 255, 255)

Use with ST7796 Display
DISPLAY_INIT(SPI, ST7796, 480, 320)
DISPLAY_FILL_RECT(0, 0, 100, 100, RGB666(255, 128, 64))

With variables
VAR R=200, G=100, B=50
VAR COLOR = RGB666(R, G, B)
DISPLAY_FILL_CIRCLE(240, 160, 80, COLOR)

With arrays
VAR PALETTE[3] = {255, 128, 64}
VAR C = RGB666(PALETTE[0], PALETTE[1], PALETTE[2])
```

Note:

- RGB666 is only used for ST7796 display (18-bit color)
- For ILI9341/ILI9488/ST7789/ST7735 (16-bit) use RGB565()
- For WS2812/matrices (24-bit), use RGB888()

Conditional Statements

IF / ELSE IF / ELSE / ENDIF

Syntax:

```
IF condition THEN
    Command
ENDIF

IF condition THEN
    Command
ELSE
    Command
ENDIF

IF condition1 THEN
    Teams1
ELSE IF condition2 THEN
    Teams2
ELSE
    Teams3
ENDIF
```

Description:
The IF construct allows commands to be executed depending on the condition. The condition is evaluated as true (not 0) or false (0).

Branching:

- `IF ... THEN ... ENDIF' - execute commands if the condition is true
- `IF ... THEN ... ELSE ... ENDIF' - execute some commands if true, others if false
- `IF ... THEN ... ELSE IF ... THEN ... ENDIF' - check several conditions in order
- 'ELSE IF' allows you to check for additional conditions if the previous ones were false
- It is possible to use multiple 'ELSE IF' in a row for multiple selection
- The last 'ELSE' (optional) is met if all conditions were false

Comparison operators:

- '=' or '==' - equals
- '<>' is not equal to
- '<' - less
- '>' - more
- '<=' - less than or equal to
- '>=' - greater than or equal to

Boolean operators:**

- 'AND' - logical AND (high priority)
- 'OR' - Boolean OR (low priority)
- Parentheses '()' - change the priority of the calculation

Features of the conditions:

- Any non-zero value is considered true: 'IF A THEN' is equal to 'IF A <> 0 THEN'
- Mathematical expressions can be used in conditions: 'IF (A + B) > (C * 2) THEN'
- In conditions, you can use the functions: 'IF ABS(X) > 10 THEN', 'IF ADC_IN1 > 2.5 THEN'
- In conditions, you can use array elements: 'IF DATA[I] > 100 THEN'

Examples:

```
' Simple Condition
IF A > 10 THEN
    PRINT "A is greater than 10"
ENDIF

' IF-ELSE IF-ELSE (Multiple Choice)
' Conditions are checked from top to bottom
Once one condition is true, the others are not checked
IF A > 100 THEN
    PRINT "High"
ELSE IF A > 50 THEN
    PRINT "Medium"
ELSE
    PRINT "Low"
ENDIF

' Multiple ELSE IFs in a row
IF TEMP < 10 THEN
    PRINT "Cold"
ELSE IF TEMP < 20 THEN
    PRINT "Cool"
ELSE IF TEMP < 30 THEN
    PRINT "Comfortable"
ELSE IF TEMP < 40 THEN
    PRINT "Teplo"
ELSE
    PRINT "Zharko"
ENDIF

' ELSE IF with Boolean Operators
IF VOLTAGE > 12.6 THEN
    PRINT "Battery fully charged"
ELSE IF VOLTAGE > 12.0 AND VOLTAGE <= 12.6 THEN
    PRINT "Battery partially charged"
ELSE IF VOLTAGE > 11.5 THEN
    PRINT "Battery is low"
ELSE
    PRINT "Charging Required!"
ENDIF

' ELSE IF to define a range of values
VAR ADC_VAL = ADC_IN1
IF ADC_VAL < 0.5 THEN
    PRINT "Range 1"
    OUT(1, 1)
ELSE IF ADC_VAL < 1.0 THEN
    PRINT "Range 2"
    OUT(2, 1)
ELSE IF ADC_VAL < 1.5 THEN
    PRINT "Range 3"
    OUT(3, 1)
ELSE IF ADC_VAL < 2.0 THEN
    PRINT "Range 4"
    OUT(4, 1)
ELSE
    PRINT "Range 5"
    OUT(5, 1)
ENDIF

' Operator is not equal to
IF A <> 0 THEN
    PRINT "A is not equal to zero"
ENDIF

› Boolean operators
IF A > 10 AND B < 20 THEN
    PRINT "A in the range of 10-20"
ENDIF
```

```
IF X < 0 OR X > 100 THEN
    PRINT "X out of the 0-100 range"
ENDIF

> Difficult conditions with brackets
IF (A > 5 AND B > 5) OR (A < -5 AND B < -5) THEN
    PRINT "Both are greater than 5 or both are less than -5"
ENDIF

' Expression Conditions
IF (A + B) / 2 > 50 THEN
    PRINT "Average is greater than 50"
ENDIF

> Terms with features
IF ABS(TEMP - 25) < 5 THEN
    PRINT "The temperature is close to 25 degrees"
ENDIF

> ADC Terms
IF ADC_IN1 > 2.5 AND ADC_IN2 < 1.0 THEN
    PRINT "Voltage on IN1 > 2.5V, on IN2 < 1.0V"
ENDIF

> Conditions with arrays
IF DATA[I] > THRESHOLD THEN
    PRINT "Value", I, "Exceeded Threshold"
ENDIF

' Nested IF
IF A > 0 THEN
    IF B > 0 THEN
        PRINT "Both positive"
    ELSE
        PRINT "A positive, B negative"
    ENDIF
ELSE
    PRINT "A negative"
ENDIF
```

Loops

FOR / NEXT

Syntax:

```
FOR variable = start TO end
    Command
NEXT

FOR variable = start TO end STEP step
    Command
NEXT
```

Description:
A FOR loop executes commands a specified number of times by changing the loop variable from start to end in specified increments.

Options:

- 'variable' - the name of the loop counter variable
- 'beginning' - initial meaning (can be an expression)
- 'end' - final value (can be an expression)

- 'step' - change step (default 1, can be negative)

Features:

- Start, end, and step can be fractional numbers
- Start, end, and step are calculated only once when entering a loop
- To count backwards, use a negative STEP
- Loop variable is available inside and after the loop
- You can nest cycles inside each other (up to 5 levels)
- Expressions in parameters can contain variables and functions

Examples:

```
' Simple cycle from 1 to 10
FOR I = 1 TO 10
    PRINT I
NEXT

' Increment Cycle
FOR I = 0 TO 100 STEP 10
    PRINT I
NEXT

› Cycle in reverse
FOR I = 10 TO 1 STEP -1
    PRINT I
NEXT

' Fractional Step Cycle
FOR X = 0 TO 1 STEP 0.1
    PRINT X
NEXT

' Loop with variables in parameters
VAR N = 10
FOR I = 1 TO N
    PRINT I
NEXT

' Loop with expressions in all parameters
VAR START = 5
VAR END = 20
VAR STEP_SIZE = 3
FOR I = START TO END STEP STEP_SIZE
    PRINT I
NEXT

' Expressions are evaluated once when entering a loop
VAR N = 10
FOR I = 1 TO N * 2 STEP 2
    PRINT I
    N = N + 1 // A change in N does not affect the end of the cycle
NEXT

' Using Functions in Settings
VAR MAX_VAL = ABS(-100)
FOR I = 0 TO MAX_VAL STEP 10
    PRINT I
NEXT

Using ADC for a Dynamic Number of Iterations
VAR ITERATIONS = ADC_IN1 * 100 // 0-330 iterations depending on voltage
FOR I = 1 TO ITERATIONS
    PRINT I
NEXT

' Calculating a Range Based on an Array
VAR DATA[10] = {5, 8, 12, 3, 15, 9, 20, 6, 11, 7}
```

```
VAR MIN_VAL = 0
VAR MAX_VAL = 0
› Find the minimum and maximum
FOR I = 0 TO 9
    IF DATA[I] < MIN_VAL OR I = 0 THEN
        MIN_VAL = DATA[I]
    ENDIF
    IF DATA[I] > MAX_VAL THEN
        MAX_VAL = DATA[I]
    ENDIF
NEXT
' Use the found values in a new loop
FOR VAL = MIN_VAL TO MAX_VAL
    PRINT VAL
NEXT

' Using a loop variable in expressions
FOR ANGLE = 0 TO 360 STEP 30
    VAR RAD = ANGLE * 3.14159 / 180
    VAR X = 160 + COS(RAD) * 100
    VAR Y = 120 + SIN(RAD) * 100
    DISPLAY_CIRCLE(X, Y, 5, RGB565(255, 0, 0))
NEXT

' Nested loops (multiplication table)
FOR I = 1 TO 10
    FOR J = 1 TO 10
        PRINT I * J, " "
    NEXT
    PRINT // Line feed
NEXT

' Loop over an array
VAR DATA[10]
FOR I = 0 TO 9
    DATA[I] = I * I
NEXT

' A Condition Loop Inside
FOR I = 1 TO 100
    IF I % 2 = 0 THEN
        PRINT I, " - even"
    ENDIF
NEXT

' Cycle to fill a 2D array
VAR MATRIX[5, 5]
FOR ROW = 0 TO 4
    FOR COL = 0 TO 4
        MATRIX[ROW, COL] = ROW * 10 + COL
    NEXT
NEXT
```

WHILE / WEND

Syntax:

```
WHILE (condition)
    Command
WEND
```

Description:
The WHILE loop executes commands as long as the condition is true. The condition is checked before each iteration.

Features:

- Condition **mandatory** is indicated in parentheses '()'

- Condition is checked before each execution of the loop body
- If the condition is false from the start, the loop body will never be executed
- All comparison operators and Boolean operators can be used in the condition
- You can use variables, expressions, functions, arrays in the condition
- You can nest WHILE loops inside each other and in FOR loops (up to 5 levels)
- Be careful with infinite loops – always change the condition variables inside the loop

Examples:

```
' Simple Loop with Counter
VAR I = 0
WHILE (I < 10)
    PRINT I
    I = I + 1
WEND

' Conditional Loop
VAR SUM = 0
VAR N = 1
WHILE (SUM < 100)
    SUM = SUM + N
    N = N + 1
WEND
PRINT "Sum:", SUM

' Loop with Boolean Operators
VAR X = 0
VAR Y = 0
WHILE (X < 10 AND Y < 10)
    X = X + 1
    Y = Y + 2
    PRINT X, Y
WEND

' Loop with expressions in the condition
VAR A = 5
VAR B = 10
WHILE ((A + B) / 2 < 20)
    A = A + 1
    B = B + 2
    PRINT "Average:", (A + B) / 2
WEND

' Loop with function in condition
VAR ANGLE = 0
WHILE (ABS(SIN(ANGLE)) < 0.9)
    ANGLE = ANGLE + 0.1
    PRINT ANGLE, SIN(ANGLE)
WEND

› Cycle with ADC (Threshold Exceeding Wait)
WHILE (ADC_IN1 < 2.5)
    PRINT "Waiting for a signal..."
    PAUSE 100
WEND
PRINT "Signal detected!"

' Multi-sensor condition
WHILE (ADC_IN1 > 1.0 AND ADC_IN2 < 2.0)
    PRINT "ADC1:", ADC_IN1, "ADC2:", ADC_IN2
    PAUSE 50
WEND

› Nested loops
VAR I = 0
WHILE (I < 5)
    VAR J = 0
    WHILE (J < 5)
```

```
        PRINT I * 10 + J, " "
        J = J + 1
    WEND
    PRINT // Line feed
    I = I + 1
WEND

' Loop with array in condition
VAR DATA[10]
VAR I = 0
WHILE (I < 10 AND DATA[I] <> 0)
    PRINT DATA[I]
    I = I + 1
WEND

' Loop with Early Exit Through Condition
VAR FOUND = 0
VAR I = 0
WHILE (I < 100 AND FOUND = 0)
    IF DATA[I] = TARGET THEN
        FOUND = 1
        PRINT "Found in position", I
    ENDIF
    I = I + 1
WEND

> Sensor data cycle
VAR STABLE = 0
VAR PREV = 0
VAR COUNT = 0
WHILE (STABLE = 0)
    VAR CURR = ADC_IN1
    IF ABS(CURR - PREV) < 0.01 THEN
        COUNT = COUNT + 1
        IF COUNT > 10 THEN
            STABLE = 1
        ENDIF
    ELSE
        COUNT = 0
    ENDIF
    PREV = CURR
    PAUSE 10
WEND
PRINT "Value stabilized:", CURR

' Condition with variables
VAR MIN_VAL = 10
VAR MAX_VAL = 100
VAR VALUE = 50
WHILE (VALUE >= MIN_VAL AND VALUE <= MAX_VAL)
    VALUE = VALUE + RND(-10, 10)
    PRINT VALUE
    PAUSE 100
WEND
PRINT "The value is out of range"
```

Mathematical and logical expressions

Expressions are used in assignments, IF/WHILE conditions, and command parameters.

Arithmetic Operators

Binary operators:**

- '+' - addition
- '-' - subtraction

- '*' - multiplication
- '/' - division (fractional)
- '%' is an integer division

Priority of operations (from high to low):

1. Parentheses '()'
2. Multiplication '*', division '/', '%'
3. Addition '+', subtraction '-'

Examples:

```
VAR A = 5 + 3          // 8
VAR B = 10 - 4         // 6
VAR C = 3 * 4          // 12
VAR D = 15 / 4 // 3.75 (fractional division)
VAR E = 15 % 4 // 3 (integer division)

› Priority of operations
VAR X = 2 + 3 * 4 // 14 (multiplication first)
VAR Y = (2 + 3) * 4 // 20 (parentheses change priority)

' Compound expressions
VAR RESULT = (A + B) / (C - D)
VAR AVG = (X + Y + Z) / 3
```

Number Formats

```
VAR DEC = 255 // Decimal Number
VAR HEX = 0xFF // Hexadecimal (hex)
VAR BIN = 0b111111111 // Binary
VAR FLOAT = 3.14159 // Fractional number

' All formats are equal
IF DEC = HEX AND HEX = BIN THEN
    PRINT "All three numbers are 255"
ENDIF
```

Math Functions

```
› Modulus of Number
VAR X = ABS(-5)              // 5

' Trigonometry (argument in radians)
VAR S = SIN(1.57)            // ~1.0 (sin(π/2))
VAR C = COS(3.14)            // ~-1.0 (cos(π))
VAR T = TAN(0.785)           // ~1.0 (tan(π/4))

' Rounding
VAR R1 = ROUND(3.7) // 4 (to the nearest integer)
VAR R2 = FLOOR(3.7) // 3 (down)
VAR R3 = CEIL(3.2) // 4 (up)

' Use in expressions
VAR ANGLE_DEG = 45
VAR ANGLE_RAD = ANGLE_DEG * 3.14159 / 180
VAR DISTANCE = 100
VAR X_POS = 160 + COS(ANGLE_RAD) * DISTANCE
VAR Y_POS = 120 + SIN(ANGLE_RAD) * DISTANCE
```

Random Numbers

```
VAR R1 = RND // Random 0.0..1.0
VAR R2 = RND(10) // Random 0..10
VAR R3 = RND(5, 15) // Random 5..15

› Application
VAR DICE = RND(1, 6) // Dice roll
VAR COLOR_R = RND(0, 255)
VAR COLOR_G = RND(0, 255)
VAR COLOR_B = RND(0, 255)
VAR COLOR = RGB565(COLOR_R, COLOR_G, COLOR_B)
```

Bit Operations

Bit operations work with integers and perform bitwise logical operations.

Syntax:

```
AND(a, b) // bitwise and
OR(a, b) // Bitwise OR
XOR(a, b) // Bitwise exclusive OR
NOT(a) // Bitwise NOT (inversion)
```

Description:

- 'AND(a, b)' - returns the result of bitwise AND (1 only if both bits are equal to 1)
- 'OR(a, b)' - returns the result of a bitwise OR (1 if at least one bit is 1)
- 'XOR(a, b)' - returns the result of the exclusive OR (1 if the bits are different)
- 'NOT(a)' - returns bitwise inversion (all bits are inverted)

Examples:

```
Basic operations
VAR A = 0b1100 // 12 in binary
VAR B = 0b1010 // 10 in binary

VAR C1 = AND(A, B)    // 0b1000 = 8
VAR C2 = OR(A, B)     // 0b1110 = 14
VAR C3 = XOR(A, B)    // 0b0110 = 6
VAR C4 = NOT(A) // Inversion of all bits

PRINT "A AND B = ", C1
PRINT "A OR B = ", C2
PRINT "A XOR B = ", C3

Working with masks
VAR FLAGS = 0b00000000
VAR FLAG_RED = 0b00000001 // bit 0
VAR FLAG_GREEN = 0b00000010 // bit 1
VAR FLAG_BLUE = 0b00000100 // bit 2

Setting flags
FLAGS = OR(FLAGS, FLAG_RED)
FLAGS = OR(FLAGS, FLAG_BLUE)
PRINT "Flags: ", FLAGS.b      // 0b00000101

Flag check
VAR HAS_RED = AND(FLAGS, FLAG_RED)
IF HAS_RED > 0 THEN
    PRINT "Red flag is set"
ENDIF

Clearing the flag (resetting the bit)
FLAGS = AND(FLAGS, NOT(FLAG_RED))
PRINT "After clear: ", FLAGS.b  // 0b00000100
```

```
Flag toggle
FLAGS = XOR(FLAGS, FLAG_GREEN)
PRINT "After toggle: ", FLAGS.b  // 0b00000110

Bitfield extraction
VAR DATA = 0x1A5      // 0001 1010 0101
VAR NIBBLE = AND(DATA, 0x0F) // Low 4 bits: 0101 = 5
VAR BYTE = AND(DATA, 0xFF) // Low Byte

Combination of operations
VAR MASK1 = 0b11110000
VAR MASK2 = 0b00001111
VAR FULL_MASK = OR(MASK1, MASK2)  // 0b11111111

Working with RGB565 Colors
VAR COLOR = 0xF800 // Red in RGB565
VAR RED_BITS = AND(COLOR, 0xF800) // Extract Red Channel
VAR GREEN_BITS = AND(COLOR, 0x07E0) // Extract green channel
VAR BLUE_BITS = AND(COLOR, 0x001F) // Extract Blue Channel
```

Practical Application:

```
GPIO Pin Management via Bitmasks
VAR PORT_STATE = 0
VAR PIN0 = 0b00000001
VAR PIN1 = 0b00000010
VAR PIN7 = 0b10000000

Enable PIN0 and PIN7
PORT_STATE = OR(PORT_STATE, PIN0)
PORT_STATE = OR(PORT_STATE, PIN7)

Check PIN1 status
IF AND(PORT_STATE, PIN1) > 0 THEN
    PRINT "PIN1 is HIGH"
ELSE
    PRINT "PIN1 is LOW"
ENDIF

Creating Bitfields
VAR SENSOR_DATA = 0
Write temperature (12 bits) to high bits
VAR TEMP_VALUE = 850    // 850 = 0x352
SENSOR_DATA = OR(SENSOR_DATA, TEMP_VALUE)
Extract Temperature
VAR TEMP_READ = AND(SENSOR_DATA, 0xFFF) // Low 12 bits
```

Notes:

- All operations work with integers (conversion via '(long)')
- Fractional part is discarded
- The result is returned as a float for compatibility with the variable system
- The NOT operation inverts all bits (for 32-bit long)

Variables and Arrays in Expressions

```
' Scalar Variables
VAR A = 10
VAR B = 20
VAR C = A + B              // 30

› 1D arrays
VAR DATA[10]
DATA[0] = 100
DATA[1] = DATA[0] + 50     // 150
VAR SUM = DATA[0] + DATA[1] // 250
```

```
› 2D arrays
VAR MATRIX[3, 3]
MATRIX[0, 0] = 1
MATRIX[0, 1] = 2
VAR TOTAL = MATRIX[0, 0] + MATRIX[0, 1]  // 3

' Indices can be expressions
VAR I = 5
VAR X = DATA[I * 2]         // DATA[10]
VAR Y = MATRIX[I / 2, I % 3]  // MATRIX[2, 2]
```

Time and RTC Functions

```
' Timer (milliseconds since last TIMER_RESET)
VAR ELAPSED = TIMER_GET
IF ELAPSED > 5000 THEN
    PRINT "5 seconds passed"
ENDIF

' RTC (Real-Time Clock)
VAR YEAR = RTC_Y // Year (2025)
VAR MONTH = RTC_M // Month (1-12)
VAR DAY = RTC_D // Day (1-31)
VAR HOUR = RTC_H // Hour (0-23)
VAR MINUTE = RTC_MIN // Minute (0-59)
VAR SECOND = RTC_S // Second (0-59)

› Time Condition
IF RTC_H >= 9 AND RTC_H < 18 THEN
    PRINT "Working Hours"
ENDIF
```

Sensors in Expressions

```
› ADC inputs (0-3.3V zoom voltage)
VAR V1 = ADC_IN1 // Input Voltage 1
VAR V2 = ADC_IN2

› ADC Terms
IF ADC_IN1 > 2.5 THEN
    PRINT "High Voltage"
ENDIF

› Computing with ADC
VAR AVG = (ADC_IN1 + ADC_IN2 + ADC_IN3) / 3
VAR DIFF = ABS(ADC_IN1 - ADC_IN2)

› Temperature sensors DS18B20
VAR TEMP1 = DS1820[1] // Fahrenheit Temperature
VAR TEMP2 = DS1820C[2] // Temperature in Celsius

IF DS1820C[1] > 25 THEN
    PRINT "Temperature above 25°C"
ENDIF
```

Color Functions

```
' RGB565 (16-bit color for most TFT displays)
VAR RED = RGB565(255, 0, 0)
VAR GREEN = RGB565(0, 255, 0)
VAR BLUE = RGB565(0, 0, 255)
VAR WHITE = RGB565(255, 255, 255)
VAR CUSTOM = RGB565(128, 200, 50)

' RGB888 (24-bit color)
```

```
VAR COLOR24 = RGB888(255, 128, 64)

' RGB666 (18-bit color for ST7796)
VAR COLOR18 = RGB666(255, 255, 0)

Using variables in RGB
VAR R = 255
VAR G = RND(0, 255)
VAR B = 0
VAR RANDOM_RED = RGB565(R, G, B)

' Solid Wood Color
VAR COLORS[3] = {255, 128, 64}
VAR MIXED_COLOR = RGB565(COLORS[0], COLORS[1], COLORS[2])
```

Examples of complex expressions

```
' Calculating the Average with Range Check
VAR SUM = 0
VAR COUNT = 0
FOR I = 0 TO 9
    IF DATA[I] >= 0 AND DATA[I] <= 100 THEN
        SUM = SUM + DATA[I]
        COUNT = COUNT + 1
    ENDIF
NEXT
VAR AVG = SUM / COUNT

' Normalizing the value to the range 0-100
VAR RAW = ADC_IN1
VAR MIN_VAL = 0.5
VAR MAX_VAL = 2.5
VAR NORMALIZED = (RAW - MIN_VAL) / (MAX_VAL - MIN_VAL) * 100
IF NORMALIZED < 0 THEN
    NORMALIZED = 0
ENDIF
IF NORMALIZED > 100 THEN
    NORMALIZED = 100
ENDIF

' Calculating Brightness from a Sine Wave
FOR T = 0 TO 360 STEP 10
    VAR RAD = T * 3.14159 / 180
    VAR BRIGHTNESS = (SIN(RAD) + 1) / 2 * 255
    VAR COLOR = RGB565(BRIGHTNESS, BRIGHTNESS, BRIGHTNESS)
    OUT(1, BRIGHTNESS)
    PAUSE 50
NEXT

› Checking for a point in a circle
VAR CENTER_X = 160
VAR CENTER_Y = 120
VAR RADIUS = 50
VAR POINT_X = 180
VAR POINT_Y = 140
VAR DX = POINT_X - CENTER_X
VAR DY = POINT_Y - CENTER_Y
VAR DIST = (DX * DX + DY * DY) // Distance Square
IF DIST <= (RADIUS * RADIUS) THEN
    PRINT "Point inside a circle"
ENDIF
```

Graphical Commands

DISPLAY_INIT

Syntax:

```
DISPLAY_INIT(interface, controller, width, height, [rotation], [cs_out, dc_out, rst_out])
```

Options:

- 'interface' - I2C or SPI
- 'controller' - controller name: ST7735, ILI9341, ILI9488, ST7796, ST7789, SSD1306, etc.
- 'width' - the width of the display
- 'height' - display height
- 'rotation' (optional) - rotate (0, 90, 180, 270)
- 'cs_out, dc_out, rst_out' (optional for SPI) - OUT output numbers

Examples:

```
' ST7735 (128x160, RGB565)
DISPLAY_INIT(SPI, ST7735, 160, 128, 0)

' ILI9341 (320x240, RGB565)
DISPLAY_INIT(SPI, ILI9341, 320, 240, 0)

' ILI9488 (480x320, RGB565)
DISPLAY_INIT(SPI, ILI9488, 480, 320, 0)

' ST7796 (480x320, RGB666)
DISPLAY_INIT(SPI, ST7796, 480, 320, 0)

' ST7789 (240x320, RGB565)
DISPLAY_INIT(SPI, ST7789, 240, 320, 0)

' SSD1306 OLED (128x64, I2C)
DISPLAY_INIT(I2C, SSD1306, 128, 64, 0)

' With custom pins for SPI (CS=OUT10, DC=OUT11, RST=OUT12)
DISPLAY_INIT(SPI, ILI9341, 320, 240, 0, 10, 11, 12)
```

DISPLAY_CLEAR

Syntax:

```
DISPLAY_CLEAR([color])
```

Description:
Clears the display. If no color is specified, use black (0x0000).

Examples:

```
DISPLAY_CLEAR()
DISPLAY_CLEAR(RGB565(255, 255, 255))
```

DISPLAY_PIXEL

Syntax:

```
DISPLAY_PIXEL(x, y, color)
```

Description:

Draws a single point on the display.

Examples:

```
DISPLAY_PIXEL(100, 100, RGB565(255, 0, 0))
```

DISPLAY_LINE

Syntax:

```
Format 1: Plain Line (Backward Compatibility)
DISPLAY_LINE(x0, y0, x1, y1, color)

Format 2: Polyline on an array of points
DISPLAY_LINE(array[], color)

Format 3: Polyline with coordinates (up to 30 points)
DISPLAY_LINE(x0, y0, x1, y1, x2, y2, ..., color)
```

Description:
Draws a line from point to point. Supports three formats:

1. **Simple line** - connects two points (classic version)
2. **Point Array** - draws a polyline through all the points in the array
3. **Enumeration** - draws a polyline through the listed points

For array and enumeration: minimum 2 points, maximum 30 points.

Examples:

```
Simple Line
DISPLAY_LINE(0, 0, 320, 240, RGB565(255, 255, 255))

Grid
FOR I = 0 TO 320 STEP 20
    DISPLAY_LINE(I, 0, I, 240, RGB565(100, 100, 100))
NEXT

Polyline on a solid wood
VAR PATH[10]
PATH[0-9] = {10, 10, 50, 100, 100, 50, 150, 120, 200, 60}
DISPLAY_LINE(PATH[], RGB565(0, 255, 0))

A polyline with a list of coordinates
DISPLAY_LINE(10, 10, 50, 100, 100, 50, 150, 120, 200, 60, RGB565(0, 255, 0))

Sine Wave Graph
VAR POINTS[100]
FOR I = 0 TO 49
    POINTS[I*2] = I * 5
    POINTS[I*2+1] = 120 + SIN(I * 0.2) * 50
NEXT
DISPLAY_LINE(POINTS[], RGB565(255, 128, 0))
```

DISPLAY_RECT

Syntax:

```
DISPLAY_RECT(x, y, width, height, color)
```

Description:
Draws the outline of the rectangle.

DISPLAY_FILL_RECT

Syntax:

```
DISPLAY_FILL_RECT(x, y, width, height, color)
```

Description:
Draws a filled rectangle.

Examples:

```
DISPLAY_FILL_RECT(10, 10, 100, 50, RGB565(255, 0, 0))

VAR X=10, Y=20, W=100, H=50
DISPLAY_FILL_RECT(X, Y, W, H, RGB565(0, 255, 0))
```

DISPLAY_CIRCLE

Syntax:

```
DISPLAY_CIRCLE(x, y, radius, color)
DISPLAY_CIRCLE(x, y, radius, start_angle, end_angle, color)
```

Description:
Draws a circle outline or arc.

- In the first version, he draws a full circle
- In the second option, he draws an arc from start_angle to end_angle
- Angles are set in degrees, 0° corresponds to the top point (12 hours)
- 90° - right (3 hours), 180° - bottom (6 hours), 270° - left (9 hours)

Examples:

```
' Full Circle
DISPLAY_CIRCLE(160, 120, 50, RGB565(255, 255, 255))

' Arc from 0° to 90° (upper right quarter)
DISPLAY_CIRCLE(160, 120, 50, 0, 90, RGB565(255, 0, 0))

' Arc from 270° to 360° (upper left quarter)
DISPLAY_CIRCLE(160, 120, 50, 270, 360, RGB565(0, 255, 0))

> Clock Face
FOR I = 0 TO 330 STEP 30
    DISPLAY_CIRCLE(160, 120, 80, I, I+10, RGB565(255, 255, 255))
NEXT
```

DISPLAY_FILL_CIRCLE

Syntax:

```
DISPLAY_FILL_CIRCLE(x, y, radius, color)
DISPLAY_FILL_CIRCLE(x, y, radius, start_angle, end_angle, color)
```

Description:
Draws a filled circle or sector.

- In the first option, draws a full filled circle
- In the second option, he draws a painted sector (like a piece of cake) from start_angle to end_angle
- Angles are set in degrees, 0° corresponds to the top point (12 hours)
- 90° - right (3 hours), 180° - bottom (6 hours), 270° - left (9 hours)

Examples:

```
' Full Filled Circle
DISPLAY_FILL_CIRCLE(160, 120, 50, RGB565(0, 0, 255))

' Sector from 0° to 120° (one-third from the top)
DISPLAY_FILL_CIRCLE(160, 120, 50, 0, 120, RGB565(255, 0, 0))

› Pie chart
DISPLAY_FILL_CIRCLE(160, 120, 60, 0, 90, RGB565(255, 0, 0)) ' 25% red
DISPLAY_FILL_CIRCLE(160, 120, 60, 90, 180, RGB565(0, 255, 0)) ' 25% green
DISPLAY_FILL_CIRCLE(160, 120, 60, 180, 270, RGB565(0, 0, 255)) ' 25% Blue
DISPLAY_FILL_CIRCLE(160, 120, 60, 270, 360, RGB565(255, 255, 0)) ' 25% Yellow

' Concentric circles with gradient
FOR R = 10 TO 100 STEP 10
    DISPLAY_FILL_CIRCLE(160, 120, R, RGB565(R*2, 255-R*2, 128))
NEXT
```

DISPLAY_FILL_POLYGON

Syntax:

```
Format 1: Polygon by Point Array
DISPLAY_FILL_POLYGON(array[], color)

Format 2: Polygon with a list of coordinates (from 3 to 30 points)
DISPLAY_FILL_POLYGON(x0, y0, x1, y1, x2, y2, ..., color)
```

Description:
Draws a shaded polygon at specified points. The team supports from 3 to 30 vertices. Uses the scanline fill algorithm for efficient rendering.

Format 1: Dot Array

- 'array[]' - an array of coordinates in the format [x0, y0, x1, y1, x2, y2, ...]
- 'color' - fill color (RGB565 for color displays)
- The number of points is determined automatically from the size of the array (array_size / 2)

Format 2: Enumeration of coordinates

- 'x0, y0, x1, y1, ...' - coordinates of the vertices of the polygon
- 'color' - fill color (last parameter)
- Minimum 3 points (triangle), maximum 30 points

Examples:

```
' Triangle through the enumeration of coordinates
DISPLAY_FILL_POLYGON(160, 50, 100, 150, 220, 150, RGB565(255, 0, 0))

' Pentagon (pentagon)
```

```
DISPLAY_FILL_POLYGON(
    160, 40,
    220, 100,
    190, 170,
    130, 170,
    100, 100,
    RGB565(0, 255, 0)
)

' Star (8 peaks)
VAR STAR[16]
STAR[0-15] = {
    160, 40, ' top
    175, 100,
    235, 100, ' right top
    185, 140,
    210, 200, ' right bottom
    160, 160,
    110, 200, ' left bottom
    135, 140,
    85, 100, ' left top
    145, 100
}
DISPLAY_FILL_POLYGON(STAR[], RGB565(255, 215, 0))

' Hexagon on Array
VAR HEX[12]
FOR I = 0 TO 5
    ANGLE = I * 60 * 3.14159 / 180
    HEX[I*2] = 160 + COS(ANGLE) * 60
    HEX[I*2+1] = 120 + SIN(ANGLE) * 60
NEXT
DISPLAY_FILL_POLYGON(HEX[], RGB565(128, 0, 255))

' Custom polygon (10 vertices)
DISPLAY_FILL_POLYGON(
    50, 50,
    100, 40,
    150, 60,
    180, 100,
    170, 150,
    140, 180,
    100, 190,
    60, 170,
    40, 130,
    45, 80,
    RGB565(255, 128, 0)
)

' Rhombus
DISPLAY_FILL_POLYGON(160, 50, 220, 120, 160, 190, 100, 120, RGB565(0, 200, 200))

> Trapeze
DISPLAY_FILL_POLYGON(100, 80, 220, 80, 250, 160, 70, 160, RGB565(200, 100, 50))
```

DISPLAY_BITMAP

Syntax:

```
For one-dimensional arrays:
DISPLAY_BITMAP(x, y, array[], width, height)
DISPLAY_BITMAP(x, y, array[], width, height, scale)

For two-dimensional arrays:
DISPLAY_BITMAP(x, y, array2d[])
DISPLAY_BITMAP(x, y, array2d[], scale)
```

Description:
Outputs the bitmap from the array to a scalable display. Each element of the array is the color of a pixel. Supports both one-dimensional and two-dimensional arrays.

Important - interpretation of colors:

- E-paper displays: '0' = white, anything greater than '0' = black
- **OLED displays**: '0' = black, anything greater than '0' = white
- **Color Displays (TFT/LCD)**: Values are used as-is in RGB565/RGB888 format

Options:

- 'x, y' - coordinates on the display (upper left corner of the bitmap)
- 'array[]' - the name of the one-dimensional array with pixel data
- 'array2d[]' - the name of the two-dimensional array with pixel data
- 'width, height' - bitmap dimensions in pixels (only for 1D arrays)
- 'scale' (optional) - zoom scale (1-16, default 1)

Examples:

```
Creating a Simple 8x8 Bitmap
VAR SPRITE[64]

Filling in an array (emoticon)
FOR I = 0 TO 63
    SPRITE[I] = RGB565(0, 0, 0) // Background black
NEXT

Eyes (red dots)
SPRITE[2*8 + 2] = RGB565(255, 0, 0)
SPRITE[2*8 + 5] = RGB565(255, 0, 0)

Mouth (yellow arc)
SPRITE[5*8 + 2] = RGB565(255, 255, 0)
SPRITE[5*8 + 3] = RGB565(255, 255, 0)
SPRITE[5*8 + 4] = RGB565(255, 255, 0)
SPRITE[5*8 + 5] = RGB565(255, 255, 0)

Output without scaling (8x8 pixels)
DISPLAY_BITMAP(0, 0, SPRITE[], 8, 8)

Output at x2 scale (16x16 pixels)
DISPLAY_BITMAP(20, 0, SPRITE[], 8, 8, 2)

Output at x4 scale (32x32 pixels)
DISPLAY_BITMAP(50, 0, SPRITE[], 8, 8, 4)

Creating a 16x16 gradient
VAR GRADIENT[256]
FOR Y = 0 TO 15
    FOR X = 0 TO 15
        VAR R = X * 16
        VAR G = Y * 16
        GRADIENT[Y*16 + X] = RGB565(R, G, 128)
    NEXT
NEXT
DISPLAY_BITMAP(100, 100, GRADIENT[], 16, 16, 2)

Animating a Sprite Movement
VAR X=0
WHILE X < 200
    DISPLAY_CLEAR()
    DISPLAY_BITMAP(X, 50, SPRITE[], 8, 8, 3)
    X = X + 5
    PAUSE 50
WEND
```

```
Creating an icon using variables
VAR ICON[16] // 4x4 pixels
VAR RED = RGB565(255, 0, 0)
VAR GREEN = RGB565(0, 255, 0)
VAR BLUE = RGB565(0, 0, 255)
VAR YELLOW = RGB565(255, 255, 0)

ICON[0] = RED
ICON[1] = GREEN
ICON[2] = BLUE
ICON[3] = YELLOW
ICON[4] = GREEN
ICON[5] = RED
ICON[6] = YELLOW
ICON[7] = BLUE
ICON[8] = BLUE
ICON[9] = YELLOW
ICON[10] = RED
ICON[11] = GREEN
ICON[12] = YELLOW
ICON[13] = BLUE
ICON[14] = GREEN
ICON[15] = RED

Output with different scales
DISPLAY_BITMAP(10, 10, ICON[], 4, 4, 1)    // 4x4
DISPLAY_BITMAP(30, 10, ICON[], 4, 4, 2)    // 8x8
DISPLAY_BITMAP(60, 10, ICON[], 4, 4, 4)    // 16x16
DISPLAY_BITMAP(100, 10, ICON[], 4, 4, 8)   // 32x32
```

Examples with two-dimensional arrays:

```
Creating a 2D 8x8 Bitmap (More Intuitive Syntax)
VAR SPRITE2D[8, 8]

Filling in an array (emoticon)
FOR Y = 0 TO 7
    FOR X = 0 TO 7
        SPRITE2D[Y, X] = RGB565(0, 0, 0) // Background black
    NEXT
NEXT

Eyes (red dots)
SPRITE2D[2, 2] = RGB565(255, 0, 0)
SPRITE2D[2, 5] = RGB565(255, 0, 0)

Mouth (yellow arc)
SPRITE2D[5, 2] = RGB565(255, 255, 0)
SPRITE2D[5, 3] = RGB565(255, 255, 0)
SPRITE2D[5, 4] = RGB565(255, 255, 0)
SPRITE2D[5, 5] = RGB565(255, 255, 0)

Output without scaling - dimensions are taken automatically!
DISPLAY_BITMAP(0, 0, SPRITE2D[])

Output with a scale of x2 - width/height is not needed!
DISPLAY_BITMAP(20, 0, SPRITE2D[], 2)

Creating a 16x16 Gradient in a 2D Pattern
VAR GRADIENT2D[16, 16]
FOR Y = 0 TO 15
    FOR X = 0 TO 15
        VAR R = X * 16
        VAR G = Y * 16
        GRADIENT2D[Y, X] = RGB565(R, G, 128)
    NEXT
NEXT
DISPLAY_BITMAP(100, 100, GRADIENT2D[], 2)
```

```
Animating a Sprite Move with a 2D Pattern
VAR X=0
WHILE X < 200
    DISPLAY_CLEAR()
    DISPLAY_BITMAP(X, 50, SPRITE2D[], 3)
    X = X + 5
    PAUSE 50
WEND

Creating an 8x8 Chessboard
VAR CHESS[8, 8]
VAR WHITE = RGB565(255, 255, 255)
VAR BLACK = RGB565(0, 0, 0)

FOR Y = 0 TO 7
    FOR X = 0 TO 7
        IF (X + Y) % 2 = 0 THEN
            CHESS[Y, X] = WHITE
        ELSE
            CHESS[Y, X] = BLACK
        ENDIF
    NEXT
NEXT
DISPLAY_BITMAP(10, 10, CHESS[], 4) // 8x8 pixels -> 32x32 on the screen

Creating a Simple 5x5 Heart Icon - Initialization on Announcement
VAR RED = RGB565(255, 0, 0)
VAR BG = RGB565(0, 0, 0)

VAR HEART[5, 5] = {
    {BG, RED, BG, RED, BG},
    {RED, RED, RED, RED, RED},
    {RED, RED, RED, RED, RED},
    {BG, RED, RED, RED, BG},
    {BG, BG, RED, BG, BG}
}

DISPLAY_BITMAP(50, 50, HEART[], 5) // Increase by 5 times

Create a 10x10 rainbow gradient
VAR RAINBOW[10, 10]
FOR Y = 0 TO 9
    FOR X = 0 TO 9
        VAR HUE = (X + Y) * 25 // Value from 0 to 450
        IF HUE < 256 THEN
            Red -> Green
            RAINBOW[Y, X] = RGB565(255 - HUE, HUE, 0)
        ELSE
            Green -> Blue
            VAR VAL = HUE - 256
            RAINBOW[Y, X] = RGB565(0, 255 - VAL, VAL)
        ENDIF
    NEXT
NEXT
DISPLAY_BITMAP(100, 10, RAINBOW[], 3)

Creating a 16x16 Circle (Pixel Art)
VAR CIRCLE[16, 16]
VAR CENTER_X = 7.5
VAR CENTER_Y = 7.5
VAR RADIUS = 7

FOR Y = 0 TO 15
    FOR X = 0 TO 15
        VAR DX = X - CENTER_X
        VAR DY = Y - CENTER_Y
        VAR DIST = SQR(DX * DX + DY * DY)

        IF DIST <= RADIUS THEN
            CIRCLE[Y, X] = RGB565(255, 255, 0) // Yellow circle
        ELSE
```

```
            CIRCLE[Y, X] = RGB565(0, 0, 128) // Blue Background
        ENDIF
    NEXT
NEXT
DISPLAY_BITMAP(200, 100, CIRCLE[], 2)

Function for creating an arrow
FUNCTION CREATE_ARROW(ARROW2D[], DIR)
    DIR: 0=up, 1=right, 2=down, 3=left
    VAR ARROW_COLOR = RGB565(0, 255, 0)
    VAR BG_COLOR = RGB565(0, 0, 0)

    Clear the array
    FOR Y = 0 TO 6
        FOR X = 0 TO 6
            ARROW2D[Y, X] = BG_COLOR
        NEXT
    NEXT

    IF DIR = 0 THEN // Up
        ARROW2D[0, 3] = ARROW_COLOR
        ARROW2D[1, 2] = ARROW_COLOR
        ARROW2D[1, 3] = ARROW_COLOR
        ARROW2D[1, 4] = ARROW_COLOR
        ARROW2D[2, 1] = ARROW_COLOR
        ARROW2D[2, 3] = ARROW_COLOR
        ARROW2D[2, 5] = ARROW_COLOR
        ARROW2D[3, 3] = ARROW_COLOR
        ARROW2D[4, 3] = ARROW_COLOR
        ARROW2D[5, 3] = ARROW_COLOR
        ARROW2D[6, 3] = ARROW_COLOR
    ENDIF

    IF DIR = 1 THEN // Right
        ARROW2D[3, 6] = ARROW_COLOR
        ARROW2D[2, 5] = ARROW_COLOR
        ARROW2D[3, 5] = ARROW_COLOR
        ARROW2D[4, 5] = ARROW_COLOR
        ARROW2D[1, 4] = ARROW_COLOR
        ARROW2D[3, 4] = ARROW_COLOR
        ARROW2D[5, 4] = ARROW_COLOR
        ARROW2D[3, 3] = ARROW_COLOR
        ARROW2D[3, 2] = ARROW_COLOR
        ARROW2D[3, 1] = ARROW_COLOR
        ARROW2D[3, 0] = ARROW_COLOR
    ENDIF

    You can add DIR=2 and DIR=3 in the same way
ENDFUNC

Using the
VAR ARROW_UP[7, 7]
CREATE_ARROW(ARROW_UP[], 0)
DISPLAY_BITMAP(50, 150, ARROW_UP[], 4)

VAR ARROW_RIGHT[7, 7]
CREATE_ARROW(ARROW_RIGHT[], 1)
DISPLAY_BITMAP(100, 150, ARROW_RIGHT[], 4)

Flashing 2D Sprite Animation
VAR STAR[6, 6]
VAR YELLOW = RGB565(255, 255, 0)
VAR BLACK = RGB565(0, 0, 0)

Creating a star
FOR Y = 0 TO 5
    FOR X = 0 TO 5
        STAR[Y, X] = BLACK
    NEXT
NEXT
STAR[0, 3] = YELLOW
```

```
STAR[1, 2] = YELLOW
STAR[1, 3] = YELLOW
STAR[1, 4] = YELLOW
STAR[2, 1] = YELLOW
STAR[2, 2] = YELLOW
STAR[2, 3] = YELLOW
STAR[2, 4] = YELLOW
STAR[2, 5] = YELLOW
STAR[3, 2] = YELLOW
STAR[3, 3] = YELLOW
STAR[3, 4] = YELLOW
STAR[4, 3] = YELLOW
STAR[5, 3] = YELLOW

Flashing Animation
FOR I = 1 TO 10
    DISPLAY_BITMAP(150, 150, STAR[], 6)
    PAUSE 300
    DISPLAY_CLEAR()
    PAUSE 300
NEXT

Copying and modifying a 2D array
VAR SPRITE_ORIGINAL[4, 4]
VAR SPRITE_COPY[4, 4]

Filling out the original
FOR Y = 0 TO 3
    FOR X = 0 TO 3
        SPRITE_ORIGINAL[Y, X] = RGB565(X * 64, Y * 64, 128)
    NEXT
NEXT

Copy and invert colors
FOR Y = 0 TO 3
    FOR X = 0 TO 3
        VAR COLOR = SPRITE_ORIGINAL[Y, X]
        Simple inversion (not exact for RGB565, but demonstrative)
        SPRITE_COPY[Y, X] = RGB565(255, 255, 255) - COLOR
    NEXT
NEXT

DISPLAY_BITMAP(10, 200, SPRITE_ORIGINAL[], 8)
DISPLAY_BITMAP(100, 200, SPRITE_COPY[], 8)

Vertical bitmap reflection
VAR ORIGINAL[6, 6]
VAR FLIPPED[6, 6]

Creating an Asymmetrical Pattern
FOR Y = 0 TO 5
    FOR X = 0 TO 5
        ORIGINAL[Y, X] = RGB565(X * 40, Y * 40, 100)
    NEXT
NEXT

Flip vertically
FOR Y = 0 TO 5
    FOR X = 0 TO 5
        FLIPPED[Y, X] = ORIGINAL[5 - Y, X]
    NEXT
NEXT

DISPLAY_BITMAP(10, 300, ORIGINAL[], 5)
DISPLAY_BITMAP(80, 300, FLIPPED[], 5)
```

Note:

- For 2D arrays, dimensions (width, height) are taken automatically from the dimension of the array
- For 1D arrays, you need to explicitly specify width and height

- Zoom allows you to zoom in on the image: scale=2 makes each pixel 2x2, scale=4 - 4x4, etc.
- The array must contain the width × height of the elements
- The colors in the array must be in RGB565 format (use the RGB565() function)
- Maximum scale: 16x
- Automatically calls Display_Update() after rendering
- Useful for sprites, icons, pixel art

DISPLAY_ARC

Syntax:

```
DISPLAY_ARC(center_x, center_y, radius, thickness, start_angle, end_angle, color, bg_color)
```

Description:
Draws an arc with a specified thickness.

Options:

- 'center_x, center_y' - coordinates of the center
- 'radius' - radius
- 'thickness' - the thickness of the line
- 'start_angle, end_angle' - angles in degrees (0-360)
- 'color' - arc color
- 'bg_color' - background color

DISPLAY_TEXT

Syntax:

```
DISPLAY_TEXT(x, y, arg1, arg2, ..., FONT, SCALE, color)
DISPLAY_TEXT(x, y, arg1, arg2, ..., FONT, SCALE, color, bg_color)
```

Description:
Displays text with support for combining string literals and variables. Supports integer scaling of fonts to achieve the desired size.

Options:

- 'x, y' - coordinates of the beginning of the text
- 'arg1, arg2, ...' - parts of the text (quoted strings and/or variables/expressions)
- 'FONT' - font name (see list below)
- 'SCALE' - scaling factor (1, 2, 3, ... 8)
- 'color' - text color (RGB565)
- 'bg_color' (optional) - background color (RGB565)

Number formatting:

For variables and expressions, you can specify formatting options separated by a dot:

- 'variable. N' - rounding to N decimal places
  - Example: 'TEMP.2' → "23.45" (2 decimal places)
  - Example: 'TEMP.0' → "23" (without fractional part)
  - Example: 'PI.4' → "3.1416" (4 decimal places)

- 'variable. Nz' is rounding with the addition of leading zeros
  - Example: 'COUNT.2z' at COUNT=5 → "05.00" (leading zero in the integer part)
  - Example: 'VOLTAGE.3z' at VOLTAGE=1.5 → "01.500" (nulls added)
  - Format: at least N+2 characters in an integer part filled with zeros
- 'variable.h' - output in hexadecimal format
  - Example: 'NUM.h' at NUM=255 → "0xFF"
  - Example: 'COLOR.h' at COLOR=65535 → "0xFFFF"
- 'variable.b' - output in binary format
  - Example: 'NUM.b' at NUM=10 → "0b1010"
  - Example: 'MASK.b' at MASK=7 → "0b111"

Formatting examples:**

```
VAR TEMP = 5.6789
DISPLAY_TEXT(0, 0, TEMP, Font_7x10, 1, 0xFFFF)      // "5.678900"
DISPLAY_TEXT(0, 20, TEMP.1, Font_7x10, 1, 0xFFFF)   // "5.7"
DISPLAY_TEXT(0, 40, TEMP.2, Font_7x10, 1, 0xFFFF)   // "5.68"
DISPLAY_TEXT(0, 60, TEMP.2z, Font_7x10, 1, 0xFFFF)  // "05.68"

VAR COUNT = 7
DISPLAY_TEXT(0, 80, COUNT.0, Font_7x10, 1, 0xFFFF)   // "7"
DISPLAY_TEXT(0, 100, COUNT.0z, Font_7x10, 1, 0xFFFF) // "07"
DISPLAY_TEXT(0, 120, COUNT.3z, Font_7x10, 1, 0xFFFF) // "07.000"

Hex and binary formatting
VAR NUM = 255
DISPLAY_TEXT(0, 140, "Dec: ", NUM, Font_7x10, 1, 0xFFFF)    // "Dec: 255"
DISPLAY_TEXT(0, 160, "Hex: ", NUM.h, Font_7x10, 1, 0xFFFF)  // "Hex: 0xFF"
DISPLAY_TEXT(0, 180, "Bin: ", NUM.b, Font_7x10, 1, 0xFFFF)  // "Bin: 0b11111111"

VAR MASK = 0b1010
DISPLAY_TEXT(0, 200, "Mask: ", MASK.h, Font_7x10, 1, 0xFFFF) // "Mask: 0xA"
```

Examples:

```
Plain text without scaling (original size)
DISPLAY_TEXT(10, 10, "Hello World", Font_7x10, 1, RGB565(255, 255, 255))

Combination of text and variables
VAR TEMP = 23.456
DISPLAY_TEXT(10, 30, "Temperature: ", TEMP, " C", Font_11x18, 1, RGB565(255, 0, 0))

Formatting numbers, small text
VAR PI = 3.14159
DISPLAY_TEXT(10, 50, "Pi = ", PI.2, Font_6x8, 1, RGB565(0, 255, 0))
Outputs: "Pi = 3.14"

With Background Color
DISPLAY_TEXT(10, 70, "Status: OK", Font_16x26, 1, RGB565(0, 255, 0), RGB565(0, 0, 0))

2x Zoom
DISPLAY_TEXT(10, 100, "Scale 2x", Font_7x10, 2, RGB565(255, 255, 0))

3x Scaling
DISPLAY_TEXT(10, 130, "Scale 3x", Font_6x8, 3, RGB565(255, 255, 255))

Zero-filled
VAR COUNT = 5
DISPLAY_TEXT(10, 150, "Count: ", COUNT.2z, Font_7x10, 2, RGB565(255, 255, 255))
Displays: "Count: 05.00"

Multiple Variables
```

```
VAR X = 100, Y = 200
DISPLAY_TEXT(10, 170, "Position: (", X, ", ", Y, ")", Font_6x8, 1, RGB565(128, 128, 128))

Decorative font Baksheesh12pt (28px basic)
DISPLAY_TEXT(10, 190, "Beautiful", Baksheesh12pt, 1, RGB565(100, 200, 255))

Baksheesh20pt (46px Basic) with 2x scale = 92px
DISPLAY_TEXT(10, 230, "Elegant", Baksheesh20pt, 2, RGB565(255, 200, 100))

Baksheesh40pt (92px Basic)
DISPLAY_TEXT(10, 340, "Large", Baksheesh40pt, 1, RGB565(255, 100, 100))

VCRSCapsSSK Monospaced Font (Retro Style)
DISPLAY_TEXT(10, 10, "RETRO 8pt", VCRSCapsSSK8pt, 1, 0xFFFF) // 20px base
DISPLAY_TEXT(10, 35, "RETRO 12pt", VCRSCapsSSK12pt, 1, 0xFFFF) // 31px base
DISPLAY_TEXT(10, 70, "RETRO 24pt", VCRSCapsSSK24pt, 1, 0xFFFF) // 61px base

Scaling VCRSCapsSSK
DISPLAY_TEXT(10, 140, "RETRO 2x", VCRSCapsSSK8pt, 2, 0xFF00)      // 20px * 2 = 40px
DISPLAY_TEXT(10, 185, "RETRO 3x", VCRSCapsSSK8pt, 3, 0x00FF)      // 20px * 3 = 60px
```

Available fonts:

Fixed width (monospaced):

- 'Font_6x8' - 8px base height, most compact
- 'Font_7x10' - 10px base height, versatile
- 'Font_11x18' - 18px base height, medium
- 'Font_16x26' - 26px base height, large
- 'Font_SBIG' - special large font

Decorative (variable width) - Baksheesh:

- 'Baksheesh' or 'Baksheesh12pt' - 28px base height, elegant
- 'Baksheesh20pt' - 46px base height, medium
- 'Baksheesh40pt' - 92px base height, large

Retro Style (Variable Width) - VCRSCapsSSK:

- 'VCRSCapsSSK' or 'VCRSCapsSSK8pt' - 20px base height, shallow
- 'VCRSCapsSSK12pt' - 31px base height, medium
- 'VCRSCapsSSK24pt' - 61px base height, large
- 'VCRSCapsSSK40pt' - 102px base height, extra large

Zoom Note:

- The SCALE parameter applies integer scaling (1x, 2x, 3x, etc.)
- Total Height = Base Height × SCALE
- For example: 'Font_7x10' with SCALE=2 will give a height of 20px (10px × 2)
- For example: 'Baksheesh20pt' with SCALE=2 will give a height of 92px (46px × 2)
- Maximum SCALE = 8
- GFX fonts (Baksheesh, VCRSCapsSSK) have variable character widths - they look more beautiful
- FontDef fonts (Font_6x8, Font_7x10, etc.) have a fixed width - render faster
- Large zoom (>4x) can result in visible pixelation
- Use bg_color to highlight text (e.g. for buttons or statuses)

Note:

- Maximum length of the final line: 128 characters
- Font scale can be from 1 to 8

- Display automatically updates after text is displayed

Widgets

BUTTON_INIT

Syntax:

```
BUTTON_INIT(x, y, width, height, color, "label", is_toggle, x_var, y_var, touched_var, press_handler,
release_handler)
```

Description:
Creates an interactive button on the display with automatic touch handling. When the button is pressed and released, handlers are automatically invoked via the label mechanism and 'GOSUB/RETURN'.

Options:

- 'x, y' - coordinates of the upper left corner of the button (pixels)
- 'width, height' - button dimensions (pixels)
- 'color' - the color of the button in RGB565 format
- '"label"' - text on the button (line in quotes)
- 'is_toggle' - working hours:
  - '0' = normal button (press and release)
  - '1' = switch (each press changes the state)
- 'x_var, y_var, touched_var' - variable names for touch and state coordinates (must be pre-initialized with FT6336U_INIT)
- 'press_handler' - **label** (name) of the click handler (called automatically when pressed)
- 'release_handler' - **label** (name) of the release handler (can be empty if not needed)

How handlers work:

Handlers are labels in your code. When the button is pressed, the interpreter automatically performs 'GOSUB press_handler', when released, 'GOSUB release_handler'. Handlers must:

1. Be declared as labels (name with colon)
2. End with the 'RETURN' command to return to the main code
3. Can change variables, draw on display, control peripherals

Examples:

```
Example 1: A simple START button with two handlers
FT6336U_INIT(TX, TY, TOUCHED, 50)
BUTTON_INIT(10, 10, 100, 50, RGB565(0, 255, 0), "START", 0, TX, TY, TOUCHED, OnStart, OnEnd)

WHILE 1
    PAUSE 100
WEND

OnStart:
    PRINT "Button pressed!"
    DISPLAY_FILL_CIRCLE(160, 120, 30, RGB565(255, 0, 0))
    RETURN

OnEnd:
    PRINT "Button released!"
    DISPLAY_FILL_CIRCLE(160, 120, 30, RGB565(0, 255, 0))
    RETURN
```

```
Example 2: A button without a release handler
FT6336U_INIT(TX, TY, TOUCHED, 50)
BUTTON_INIT(10, 10, 100, 50, RGB565(255, 165, 0), "CLICK", 0, TX, TY, TOUCHED, OnClick, )

VAR COUNTER = 0

WHILE 1
    DISPLAY_TEXT(10, 100, "Clicks: ", COUNTER, Font_7x10, 1, 0xFFFF)
    PAUSE 100
WEND

OnClick:
    COUNTER = COUNTER + 1
    PRINT "Click #", COUNTER
    RETURN
```

```
Example 3: Toggle button
FT6336U_INIT(TX, TY, TOUCHED, 50)
BUTTON_INIT(10, 10, 120, 50, RGB565(100, 100, 255), "TOGGLE", 1, TX, TY, TOUCHED, OnToggle, )

VAR LED_STATE = 0

WHILE 1
    IF LED_STATE == 1 THEN
        DISPLAY_TEXT(10, 100, "LED: ON ", Font_7x10, 1, RGB565(0, 255, 0))
    ELSE
        DISPLAY_TEXT(10, 100, "LED: OFF", Font_7x10, 1, RGB565(255, 0, 0))
    ENDIF
    PAUSE 100
WEND

OnToggle:
    LED_STATE = 1 - LED_STATE // Invert State
    PRINT "LED state:", LED_STATE
    RETURN
```

```
Example 4: Managing PWM via Buttons
FT6336U_INIT(TX, TY, TOUCHED, 50)

VAR BRIGHTNESS = 50

BUTTON_INIT(10, 10, 80, 50, RGB565(0, 255, 0), "+", 0, TX, TY, TOUCHED, BtnPlus, )
BUTTON_INIT(100, 10, 80, 50, RGB565(255, 0, 0), "-", 0, TX, TY, TOUCHED, BtnMinus, )

WHILE 1
    PWM1 CH1 1000, BRIGHTNESS, 0
    DISPLAY_TEXT(10, 80, "PWM: ", BRIGHTNESS.0, "%  ", Font_7x10, 1, 0xFFFF)
    PAUSE 100
WEND

BtnPlus:
    IF BRIGHTNESS < 100 THEN
        BRIGHTNESS = BRIGHTNESS + 10
    ENDIF
    RETURN

BtnMinus:
    IF BRIGHTNESS > 0 THEN
        BRIGHTNESS = BRIGHTNESS - 10
    ENDIF
    RETURN
```

```
Example 5: Multi-button menus
FT6336U_INIT(TX, TY, TOUCHED, 50)

VAR MODE = 0

BUTTON_INIT(10, 10, 100, 40, RGB565(255, 0, 0), "MODE 1", 0, TX, TY, TOUCHED, SelectMode1, )
BUTTON_INIT(10, 60, 100, 40, RGB565(0, 255, 0), "MODE 2", 0, TX, TY, TOUCHED, SelectMode2, )
```

```
BUTTON_INIT(10, 110, 100, 40, RGB565(0, 0, 255), "MODE 3", 0, TX, TY, TOUCHED, SelectMode3, )

WHILE 1
    IF MODE == 1 THEN
        DISPLAY_TEXT(120, 20, "Current: MODE 1", Font_7x10, 1, 0xFFFF)
        Mode 1 logic
    ELSE IF MODE == 2 THEN
        DISPLAY_TEXT(120, 20, "Current: MODE 2", Font_7x10, 1, 0xFFFF)
        Mode 2 logic
    ELSE IF MODE == 3 THEN
        DISPLAY_TEXT(120, 20, "Current: MODE 3", Font_7x10, 1, 0xFFFF)
        Mode 3 logic
    ENDIF
    PAUSE 100
WEND

SelectMode1:
    MODE = 1
    PRINT "Switched to MODE 1"
    RETURN

SelectMode2:
    MODE = 2
    PRINT "Switched to MODE 2"
    RETURN

SelectMode3:
    MODE = 3
    PRINT "Switched to MODE 3"
    RETURN
```

```
Example 6: Visual feedback button
FT6336U_INIT(TX, TY, TOUCHED, 50)

VAR BTN_COLOR = RGB565(0, 200, 0)
BUTTON_INIT(100, 100, 120, 60, BTN_COLOR, "PRESS", 0, TX, TY, TOUCHED, OnPress, OnRelease)

WHILE 1
    PAUSE 100
WEND

OnPress:
    Change the color of the button when pressed
    BTN_COLOR = RGB565(0, 100, 0) // Darker
    DISPLAY_FILL_RECT(100, 100, 120, 60, BTN_COLOR)
    DISPLAY_TEXT(120, 120, "PRESS", Font_7x10, 1, 0xFFFF)
    RETURN

OnRelease:
    Return to original color
    BTN_COLOR = RGB565(0, 200, 0) // Lighter
    DISPLAY_FILL_RECT(100, 100, 120, 60, BTN_COLOR)
    DISPLAY_TEXT(120, 120, "PRESS", Font_7x10, 1, 0xFFFF)
    RETURN
```

Important Notes:**

- Handlers are labels, not functions: These are called via 'GOSUB' and must end with 'RETURN'
- Be sure to initialize touch variables with a 'FT6336U_INIT' before creating the buttons
- Release handler can be omitted: If you don't need release handle, leave the parameter blank or specify a comma without a value
- **Toggle buttons**: At 'is_toggle=1', the button changes state with each press (ON/OFF switch)
- Handlers can change any variables, draw on the display, control PWM, DAC, LED, etc.
- Multiple buttons: You can create multiple buttons with different handlers to build interfaces
- **Auto Rendering**: Buttons are automatically redrawn when the state changes

STEPPER_INIT

Syntax:

```
STEPPER_INIT(x, y, width, height, x_var, y_var, touched_var, value_var, min, max, step)
```

Description:
Creates a stepper widget (+/- buttons to change the value).

Options:

- 'x,y' coordinates
- 'width, height' - dimensions
- 'x_var, y_var, touched_var' - touch variables
- 'value_var' - variable name with the value
- 'min, max' - minimum and maximum
- 'step' - change step

Examples:

```
VAR VOLUME = 50
STEPPER_INIT(10, 10, 150, 60, TX, TY, TOUCHED, VOLUME, 0, 100, 5)
```

PROGRESS_INIT

Syntax:

```
PROGRESS_INIT(x, y, width, height, value_var, min, max, progress_color)
```

Description:
Creates a progress bar widget.

Options:

- 'x,y' coordinates
- 'width, height' - dimensions
- 'value_var' - variable name with the value
- 'min, max' - minimum and maximum
- 'progress_color' - fill color (RGB565)

Examples:

```
VAR PROGRESS = 0
PROGRESS_INIT(10, 100, 300, 30, PROGRESS, 0, 100, RGB565(0, 255, 0))

FOR PROGRESS = 0 TO 100
    PAUSE 50
NEXT
```

GAUGE_INIT

Syntax:

```
GAUGE_INIT(x, y, width, height, value_var, min, max, label, gauge_color)
```

Description:
Creates a gauge widget with segments and a label.

Options:

- 'x, y' - coordinates of the center
- 'width, height' - the dimensions of the area (define the radius)
- 'value_var' - variable name with the value
- 'min, max' - minimum and maximum of the range
- 'label' - signature (text in quotation marks, up to 4 characters), displayed at the bottom between the arcs
- 'gauge_color' - segment color (RGB565)

Examples:

```
VAR TEMP = 25
GAUGE_INIT(160, 120, 150, 150, TEMP, 0, 100, "°C", RGB565(0, 255, 0))

Pulse oximeter example
VAR HR = 0
VAR SPO2 = 0
GAUGE_INIT(120, 140, 150, 150, HR, 0, 120, "BPM", RGB565(255, 0, 0))
GAUGE_INIT(360, 140, 150, 150, SPO2, 0, 100, "%", RGB565(0, 0, 255))
```

SLIDER_INIT

Syntax:

```
SLIDER_INIT(x, y, width, height, x_var, y_var, touched_var, value_var, min, max, slider_color)
```

Description:
Creates a slider widget.

Options:

- 'x,y' coordinates
- 'width, height' - dimensions
- 'x_var, y_var, touched_var' - touch variables
- 'value_var' - variable name with the value
- 'min, max' - minimum and maximum
- 'slider_color' - slider color (RGB565)

Examples:

```
VAR BRIGHTNESS = 50
SLIDER_INIT(10, 180, 300, 40, TX, TY, TOUCHED, BRIGHTNESS, 0, 100, RGB565(255, 165, 0))
```

GRAPH_INIT

Syntax:

```
GRAPH_INIT(x, y, width, height, data_array, min, max, update_interval_sec, graph_color)
```

Description:
Creates a graph widget to display an array of data as a line graph.

Options:

- 'x,y' coordinates
- 'width, height' - dimensions
- 'data_array' - **array name** with data to display
- 'min, max' - minimum and maximum for scaling along the Y axis
- 'update_interval_sec' - chart update interval in seconds
- 'graph_color' - graph line color (RGB565)

Examples:

```
Creating an array for data
VAR DATA[100]

Initializing the Graph
GRAPH_INIT(10, 10, 300, 200, DATA, 0, 100, 0.5, RGB565(0, 255, 0))

Populating an array with data
FOR I = 0 TO 99
    DATA[I] = 50 + SIN(I * 0.1) * 30
NEXT

Temperature sensor data graph
VAR TEMPS[50]
DHT_INIT(T, H, 1000)
GRAPH_INIT(10, 10, 300, 150, TEMPS, 0, 50, 1.0, RGB565(255, 0, 0))

VAR INDEX = 0
WHILE 1
    TEMPS[INDEX] = T
    INDEX = (INDEX + 1) % 50
    PAUSE 1000
WEND
```

Note:

- Data must be in an array, scalar variable is not supported
- The graph displays all the elements of the array
- Data is scaled along the Y axis in the range [min, max]
- The chart is automatically redrawn at a specified interval

WIDGET_REDRAW

Syntax:

```
WIDGET_REDRAW
```

Description:
Redraws all widgets on the display.

Sensors

DHT_INIT

Syntax:

```
DHT_INIT(temp_var, hum_var, delay_ms)
```

Description:
Initializes a DHT sensor (DHT11/DHT22) to read temperature and humidity periodically.

Options:

- 'temp_var' is the variable name for the temperature
- 'hum_var' is the variable name for humidity
- 'delay_ms' - polling interval in milliseconds

Examples:

```
DHT_INIT(TEMP, HUM, 2000)

PRINT "Temperature: ", TEMP, " C"
PRINT "Humidity: ", HUM, " %"
```

HX711_INIT

Syntax:

```
HX711_INIT(var_name, delay_ms, dout_out, sck_out, gain)
```

Description:
Initializes the HX711 load cell.

Options:

- 'var_name' - variable name for the value
- 'delay_ms' - polling interval in milliseconds
- 'dout_out' is the OUT output number for DOUT
- 'sck_out' is the OUT output number for SCK
- 'gain' - gain (usually 128)

Examples:

```
HX711_INIT(WEIGHT, 100, 1, 2, 128)
PRINT "Weight: ", WEIGHT
```

DS1820_INIT

Syntax:

```
DS1820_INIT(array_name, delay_ms)
```

Description:
Initializes temperature sensors DS18B20 (1-Wire).

Options:

- 'array_name' is the name of the temperature storage array
- 'delay_ms' - polling interval in milliseconds

Examples:

```
DS1820_INIT(TEMPS, 1000)
```

```
PRINT "Sensor 1: ", TEMPS[0], " C"
PRINT "Sensor 2: ", TEMPS[1], " C"
```

APDS_INIT

Syntax:

```
APDS_INIT(als_var, prox_var, delay_ms)
```

Description:
Initializes the APDS9930 (light and proximity) sensor.

Options:

- 'als_var' - Ambient Light variable
- 'prox_var' is a variable for proximity (Proximity)
- 'delay_ms' - polling interval in milliseconds

Examples:

```
APDS_INIT(LIGHT, PROX, 500)
PRINT "Light: ", LIGHT
PRINT "Proximity: ", PROX
```

FFT_INIT

Syntax:

```
FFT_INIT(array_name[], adc_channel, smoothing)
```

Description:
Initializes the FFT (Fast Fourier Transform) module to automatically analyze the signal spectrum from a specified ADC channel with EMA anti-aliasing support for smooth music visualization.

Options:

- 'array_name' - the name of the array to store the results of FFT (automatically created with 128 elements)
- 'adc_channel' - ADC channel for signal capture (ADC_IN1... ADC_IN8)
- 'smoothing' - anti-aliasing level 0-100 (optional, default 0):
  - '0' = no anti-aliasing (instant response, sharp rendering)
  - '50' = Medium smoothness (optimal for music)
  - '100' = maximum smoothness (cinematic effect, slow fade)

Features:

- FFT is performed on 256 samples, the result is 128 frequency bins (spectral components)
- Automatic 16 kHz signal capture (allows you to analyze frequencies up to 8 kHz)
- **Improved Frequency Resolution: 62.5 Hz per bin** (16000 / 256) - 2 times better!
- EMA (Exponential Moving Average) with asymmetric filtration:
  - **Fast Attack** - The bars jump up quickly when the signal rises
  - **Slow decay** - the columns smoothly fall down when they fade out
- Once initialized, use FFT_START() to trigger automatic capture
- The array is automatically updated after each FFT calculation (~60 times/sec)

What is contained in the array:

- Each element of the array contains the magnitude of the frequency component
- Formula: 'magnitude = EMA(√(real² + imag²))' - absolute value with EMA filtering
- Values are raw amplitudes in ADC units (0-4095)
- The higher the value, the stronger this frequency is present in the signal
- 'array[0]' - constant component (DC, 0 Hz)
- 'array[i]' - amplitude at 'i * 62.5' Hz (e.g. array[16] = 1000 Hz)

Examples:

```
' Initialize and start FFT with a ADC_IN1 channel
FFT_INIT(SPECTRUM[], ADC_IN1)
FFT_START()

' Main Spectrum Display Cycle (Updates Automatically)
WHILE 1
    • Spectrum display on LED matrix (first 32 bins)
    FOR i = 0 TO 31
        VAR HEIGHT = SPECTRUM[i] / 10
        MATRIX_SET(i, 0, RGB888(0, 255, 0))
    NEXT i
    MATRIX_UPDATE()

    DELAY(50)
WEND
```

```
' Audio Spectrum Visualization with Auto Update
FFT_INIT(AUDIO[], ADC_IN2)
FFT_START()

WHILE 1
    ' Color spectrum: low frequencies are red, high frequencies are blue
    FOR i = 0 TO 31
        VAR INTENSITY = AUDIO[i]
        VAR RED = 255 - i * 8
        VAR BLUE = i * 8
        MATRIX_PRINT(i, 0, INTENSITY.0z, Font_6x8, RGB888(RED, 0, BLUE), 0x000000)
    NEXT i
    MATRIX_UPDATE()

    DELAY(100)
WEND
```

```
› Analysis of specific frequencies
FFT_INIT(FREQ[], ADC_IN1)
FFT_START()

WHILE 1
    ' Indexes for specific frequencies:
    ' array[0] = 0 Hz (DC)
    ' array[16] = 1000 Hz (1 kHz) - 16 * 62.5 = 1000
    ' array[32] = 2000 Hz (2 kHz) - 32 * 62.5 = 2000
    ' array[80] = 5000 Hz (5 kHz) - 80 * 62.5 = 5000

    VAR AMP_1KHZ = FREQ[16] ' Amplitude at 1 kHz
    VAR AMP_2KHZ = FREQ[32] ' Amplitude at 2 kHz
    VAR AMP_5KHZ = FREQ[80] ' Amplitude at 5 kHz

    DISPLAY_TEXT(0, 0,  "1kHz: ", AMP_1KHZ.0z, Font_7x10, 0xFFFF)
    DISPLAY_TEXT(0, 10, "2kHz: ", AMP_2KHZ.0z, Font_7x10, 0xFFFF)
    DISPLAY_TEXT(0, 20, "5kHz: ", AMP_5KHZ.0z, Font_7x10, 0xFFFF)

    DELAY(100)
WEND
```

FFT_START

Syntax:

```
FFT_START()
```

Description:
Starts automatic signal capture and processing for FFT. The timer starts capturing samples at 16 kHz, automatically calculates FFT every 128 samples, and updates the array of results.

IMPORTANTLY:
After calling FFT_START(), the array will be constantly updated in the background (via a timer interrupt) about 125 times per second (every 8 ms) until you call FFT_STOP(). You don't need to call FFT_START() in a loop - just call once, and the array will automatically contain fresh spectrum data.

Requirements:

- FFT must be pre-initialized with FFT_INIT()

Examples:

```
› Basic Spectrum Analyzer
FFT_INIT(FREQ[], ADC_IN1)
FFT_START()

WHILE 1
    'Find the dominant frequency
    VAR MAX_VAL = 0
    VAR MAX_IDX = 0
    FOR i = 1 TO 127
        IF FREQ[i] > MAX_VAL THEN
            MAX_VAL = FREQ[i]
            MAX_IDX = i
        ENDIF
    NEXT i

    ' Calculate Frequency in Hz (62.5 Hz per bin)
    VAR PEAK_FREQ = MAX_IDX * 62.5
    DISPLAY_TEXT(0, 0, "Peak: ", PEAK_FREQ, " Hz", Font_7x10, 0xFFFF)
    DELAY(100)
WEND
```

```
› Start/stop FFT by button
FFT_INIT(SPECTRUM[], ADC_IN1)

VAR RUNNING = 0
WHILE 1
    IF BUTTON(1) == 1 THEN
        IF RUNNING == 0 THEN
            FFT_START()
            RUNNING = 1
            DISPLAY_TEXT(0, 0, "FFT: ON", Font_7x10, 0x00FF00)
        ELSE
            FFT_STOP()
            RUNNING = 0
            DISPLAY_TEXT(0, 0, "FFT: OFF", Font_7x10, 0xFF0000)
        ENDIF
        DELAY(300)  ' Debounce
    ENDIF
WEND
```

FFT_STOP

Syntax:

```
FFT_STOP()
```

Description:
Stops automatic capture and processing of FFT. The timer stops, and the array of results stores the last calculated values.

Requirements:

- FFT must be pre-initialized with FFT_INIT()

Examples:

```
' Spectrum capture on demand
FFT_INIT(SPECTRUM[], ADC_IN1)

WHILE 1
    ' Capture & Analyze
    FFT_START()
    DELAY(100) ' Allow time for multiple FFT cycles
    FFT_STOP()

    ' Display captured spectrum (first 64 bins on a 32x2 matrix)
    FOR i = 0 TO 63
        VAR HEIGHT = SPECTRUM[i] / 10
        MATRIX_SET(i MOD 32, i / 32, RGB888(0, HEIGHT, 0))
    NEXT i
    MATRIX_UPDATE()

    DELAY(1000)
WEND
```

AHT21_INIT

Syntax:

```
AHT21_INIT(temp_var, hum_var, delay_ms)
```

Description:
Initializes the AHT21/AHT20 sensor (I2C temperature and humidity).

Examples:

```
AHT21_INIT(T, H, 2000)
```

SI7021_INIT

Syntax:

```
SI7021_INIT(temp_var, hum_var, delay_ms)
```

Description:
Initializes the SI7021 (I2C temperature and humidity) sensor.

BMP280_INIT

Syntax:

```
BMP280_INIT(temp_var, press_var, delay_ms)
```

Description:
Initializes the BMP280 (temperature and pressure) sensor.

Options:

- 'temp_var' is a variable for temperature
- 'press_var' is a variable for pressure
- 'delay_ms' - polling interval

Examples:

```
BMP280_INIT(TEMP, PRESS, 1000)
PRINT "Pressure: ", PRESS, " hPa"
```

SHT21_INIT

Syntax:

```
SHT21_INIT(temp_var, hum_var, delay_ms)
```

Description:
Initializes the SHT21 (I2C Temperature and Humidity) sensor.

CCS811_INIT

Syntax:

```
CCS811_INIT(co2_var, tvoc_var, delay_ms)
```

Description:
Initializes the CCS811 sensor (CO2 and TVOC).

Options:

- 'co2_var' is the variable for CO2 (ppm)
- 'tvoc_var' is a variable for TVOC (ppb)
- 'delay_ms' - polling interval

Examples:

```
CCS811_INIT(CO2, TVOC, 2000)
PRINT "CO2: ", CO2, " ppm"
PRINT "TVOC: ", TVOC, " ppb"
```

CCS811_STATUS

Syntax:

```
CCS811_STATUS
```

Description:
Displays the status of the CCS811 sensor.

CCS811_BASELINE

Syntax:

```
CCS811_BASELINE <value>
```

Description:
Sets the baseline for the CCS811 sensor.

CCS811_ENV

Syntax:

```
CCS811_ENV <temp> <humidity>
```

Description:
Sets environmental data to compensate for CCS811 readings.

VL53_INIT

Syntax:

```
VL53_INIT(distance_var, delay_ms)
```

Description:
Initializes the VL53L0X laser rangefinder (I2C ToF distance sensor).

Options:

- 'distance_var' - variable name for distance (in millimeters)
- 'delay_ms' - polling interval in milliseconds

Examples:

```
VL53_INIT(DIST, 100)
PRINT "Distance: ", DIST, " mm"

Automatic Distance Response
VL53_INIT(RANGE, 200)
WHILE 1
    IF RANGE < 100 THEN
        PRINT "Object detected!"
        DISPLAY_FILL_CIRCLE(160, 120, 30, RGB565(255, 0, 0))
    ELSE
        DISPLAY_FILL_CIRCLE(160, 120, 30, RGB565(0, 255, 0))
    ENDIF
    PAUSE 100
WEND
```

Note:

- The sensor works on an I2C bus with an address of 0x29
- Measuring range: 30-2000 mm
- After initialization, the variable is automatically updated in the background

FT6336U_INIT

Syntax:

```
FT6336U_INIT(x_var, y_var, touched_var, delay_ms)
```

Description:
Initializes the FT6336U (I2C capacitive touch controller) to track touch on the display.

Options:

- 'x_var' - variable name for the X coordinate of the touch (pixels)
- 'y_var' is the variable name for the Y coordinate of the tangent (pixels)
- 'touched_var' - variable name for touch state (1 = touches, 0 = no)
- 'delay_ms' - polling interval in milliseconds

Examples:

```
A simple example of touch tracking
FT6336U_INIT(TX, TY, TOUCHED, 50)

WHILE 1
    IF TOUCHED == 1 THEN
        PRINT "Touch at: ", TX, ", ", TY
        DISPLAY_FILL_CIRCLE(TX, TY, 10, RGB565(255, 0, 0))
    ENDIF
    PAUSE 100
WEND

Interactive drawing
FT6336U_INIT(X, Y, T, 20)
DISPLAY_CLEAR()

WHILE 1
    IF T == 1 THEN
        Draw a point at the point of touch
        DISPLAY_FILL_CIRCLE(X, Y, 5, RGB565(0, 255, 0))
    ENDIF
    PAUSE 20
WEND

Virtual Buttons
FT6336U_INIT(PX, PY, PRESS, 30)

Drawing buttons
VAR BTN1_COLOR = RGB565(0, 255, 0)
VAR BTN2_COLOR = RGB565(255, 0, 0)

DISPLAY_FILL_RECT(10, 10, 100, 50, BTN1_COLOR)
DISPLAY_TEXT(20, 25, "Button 1", Font_7x10, 1, 0xFFFF)

DISPLAY_FILL_RECT(120, 10, 100, 50, BTN2_COLOR)
DISPLAY_TEXT(130, 25, "Button 2", Font_7x10, 1, 0xFFFF)

WHILE 1
    IF PRESS == 1 THEN
        Check for button 1
        IF PX > 10 AND PX < 110 AND PY > 10 AND PY < 60 THEN
            PRINT "Button 1 pressed!"
        ENDIF

        Checking for button 2
        IF PX > 120 AND PX < 220 AND PY > 10 AND PY < 60 THEN
            PRINT "Button 2 pressed!"
        ENDIF
```

```
    ENDIF
    PAUSE 50
WEND
```

Note:

- The sensor runs on the I2C bus (usually the address is 0x38)
- Supports simultaneous recognition of multiple touches (multi-touch)
- Coordinates are automatically scaled to match the display resolution
- Variables are updated automatically in the background
- Recommended polling interval: 20-50ms for smooth operation

MAX30102_INIT

Syntax:

```
MAX30102_INIT(hr_var, spo2_var, delay_ms)
```

Description:
Initializes the MAX30102 pulse oximeter sensor (I2C heart rate and oxygen saturation sensor). The sensor uses an optical measurement method with red and infrared LEDs to detect heart rate (HR) and blood oxygen level (SpO2).

Options:

- 'hr_var' is the variable name for Heart Rate in beats per minute (BPM, 0-200)
- 'spo2_var' is the name of the variable for oxygen saturation (SpO2) as a percentage (0-100%)
- 'delay_ms' - polling interval in milliseconds (100-500 ms recommended)

Examples:

```
Easy heart rate and SpO2 monitoring
MAX30102_INIT(HR, SPO2, 200)

WHILE 1
    PRINT "Heart Rate: ", HR.0, " BPM"
    PRINT "SpO2: ", SPO2.0, " %"
    PAUSE 1000
WEND
```

```
Display visualization with GAUGE widgets
MAX30102_INIT(PULSE, OXYGEN, 200)

Creating Circular Indicators with Captions
GAUGE_INIT(120, 140, 150, 150, PULSE, 0, 120, "BPM", RGB565(255, 0, 0))
GAUGE_INIT(360, 140, 150, 150, OXYGEN, 0, 100, "%", RGB565(0, 100, 255))

WHILE 1
    Displaying values in text
    DISPLAY_TEXT(10, 10, "HR: ", PULSE.0, " BPM", Font_11x18, 1, RGB565(255, 0, 0))
    DISPLAY_TEXT(10, 40, "SpO2: ", OXYGEN.0, " %", Font_11x18, 1, RGB565(0, 100, 255))

    Checking Normal Values
    IF PULSE > 0 AND PULSE < 200 THEN
        DISPLAY_TEXT(10, 70, "Status: OK", Font_7x10, 1, RGB565(0, 255, 0))
    ELSE
        DISPLAY_TEXT(10, 70, "Status: --", Font_7x10, 1, RGB565(128, 128, 128))
    ENDIF

    PAUSE 100
WEND
```

```
Color status indication
MAX30102_INIT(BPM, O2, 200)

WHILE 1
    Determine the color of the indication by the pulse rate
    VAR HR_COLOR
    IF BPM < 60 THEN
        HR_COLOR = RGB565(0, 150, 255) // Blue - low heart rate
    ELSE IF BPM < 100 THEN
        HR_COLOR = RGB565(0, 255, 0) // Green - Normal
    ELSE
        HR_COLOR = RGB565(255, 150, 0) // Orange - High
    ENDIF

    Oxygen Level Indication Color
    VAR O2_COLOR
    IF O2 < 90 THEN
        O2_COLOR = RGB565(255, 0, 0) // Red - Low
    ELSE IF O2 < 95 THEN
        O2_COLOR = RGB565(255, 200, 0) // Yellow - Medium
    ELSE
        O2_COLOR = RGB565(0, 255, 0) // Green is excellent
    ENDIF

    DISPLAY_FILL_RECT(10, 100, 200, 40, HR_COLOR)
    DISPLAY_TEXT(20, 115, "HR: ", BPM.0, " BPM", Font_7x10, 1, 0xFFFF)

    DISPLAY_FILL_RECT(10, 150, 200, 40, O2_COLOR)
    DISPLAY_TEXT(20, 165, "SpO2: ", O2.0, " %", Font_7x10, 1, 0xFFFF)

    PAUSE 500
WEND
```

```
Recording Measurement History to the Array
MAX30102_INIT(HEARTRATE, OXYGEN, 200)

VAR HISTORY[50]
VAR INDEX = 0

GRAPH_INIT(10, 10, 460, 150, HISTORY, 0, 120, 1.0, RGB565(255, 0, 0))

WHILE 1
    Save the heart rate value to an array
    HISTORY[INDEX] = HEARTRATE
    INDEX = (INDEX + 1) % 50 // Cyclic buffer

    DISPLAY_TEXT(10, 180, "Current HR: ", HEARTRATE.0, " BPM", Font_11x18, 1, 0xFFFF)
    DISPLAY_TEXT(10, 210, "SpO2: ", OXYGEN.0, " %", Font_11x18, 1, 0xFFFF)

    PAUSE 1000
WEND
```

Note:

- The sensor works on an I2C bus with an address of 0x57
- Requires finger-to-sensor contact
- Variables are updated automatically in the background
- If there is no contact or insufficient signal, the variables are set to 0
- Sensor automatically adjusts the brightness of the LEDs for optimal signal (AGC)
- Typical values:
  - HR (heart rate): 40-200 BPM, normal 60-100 BPM at rest
  - SpO2 (oxygen): 90-100%, normal above 95%
- Recommended polling interval: 100-500 ms for stable readings
- For accurate measurements, the sensor and finger must be stationary

Peripherals

ADC_READ

Syntax:

```
ADC_READ
```

Description:
Displays the values of all available ADC channels in comma-separated volts, in order ADC_IN1, ADC_IN2, ADC_IN3, etc.

Examples:

```
ADC_READ
Output: 2.458,1.234,3.012,0.567,1.890,2.345
```

Note:

- Only available channels defined in mainconfig.h are displayed
- The first value corresponds to ADC_IN1, the second to ADC_IN2, etc.
- Number of channels depends on board configuration

ADC_IN1, ADC_IN2, ... (ADC read functions)

Syntax:

```
ADC_IN1
ADC_IN2
ADC_IN3
...
```

Description:
Functions for reading ADC channel values in volts. Used in expressions that are not commands.

Examples:

```
Reading to a variable
VAR V1 = ADC_IN1
VAR V2 = ADC_IN2

Value Output
PRINT "Voltage: ", ADC_IN1, " V"

Condition
IF ADC_IN3 > 2.5 THEN
    PRINT "Voltage above 2.5V"
ENDIF

Use in Expressions
VAR AVG = (ADC_IN1 + ADC_IN2 + ADC_IN3) / 3

ADC brightness control
FOR I = 0 TO 100
    VAR BRIGHTNESS = ADC_IN1 * 30
    PWM1 CH1 1000, BRIGHTNESS, 0
    PAUSE 100
NEXT
```

Note:

- Functions return voltage based on calibration (scaling)
- Channels are numbered with 1: ADC_IN1, ADC_IN2, ADC_IN3, ADC_IN4, ADC_IN5, ADC_IN6
- The number of available channels depends on the board version

---

PWM

Syntax:

```
PWM<timer_id> CH<channel> <freq>, <duty>, <count>
```

Description:
Sets up and starts PWM on the specified timer and channel.

Options:

- 'timer_id' - timer number (1 or 2)
- 'channel' - channel number (1 or 2)
- 'freq' - frequency in Hertz
- 'duty' - duty duty cycle in percentage (0-100)
- 'count' - number of pulses (0 = infinite)

Examples:

```
PWM1 CH1 1000, 50, 0
PWM2 CH2 500, 25, 100
```

DAC1 / DAC2

Syntax:

```
DAC1 <voltage_mv>
DAC2 <voltage_mv>
```

Description:
Sets voltage to external DACs (I2C MCP4725).

Options:

- 'voltage_mv' is the voltage in millivolts (0-3300 mV)

Address:

- DAC1 - I2C address 0x60
- DAC2 - I2C address 0x61

Examples:

```
DAC1 1650 // Set 1.65V to DAC1
DAC2 3300 // Set 3.3 V (maximum) to DAC2
DAC1 0 // Set 0 V to DAC1
DAC2 2500 // Set 2.5V to DAC2

Use with variables
VAR VOLTAGE = 1800
DAC1 VOLTAGE // Set 1.8V

Smooth variation
```

```
FOR V = 0 TO 3300 STEP 100
    DAC1 V
    PAUSE 50
NEXT
```

Note:

- Maximum voltage: 3.3 V (3300 mV)
- The value is automatically converted to a 12-bit DAC code using the formula: 'dac_value = (voltage_mv * 4095) / 3300'
- Values greater than 3300 mV are automatically limited to 3300 mV

I2C_SCAN

Syntax:

```
I2C_SCAN
```

Description:
Scans the I2C bus and displays the addresses of all found devices.

Examples:

```
I2C_SCAN
Will:
// [I2C] Scanning I2C bus...
// [I2C] Device found at address 0x3C
// [I2C] Device found at address 0x48
// [I2C] Scan complete. Found 2 devices
```

RTC_SET

Syntax:

```
RTC_SET <year>, <month>, <day>, <hour>, <minute>, <second>
```

Description:
Sets the time and date of the RTC.

Options:

- 'year' - year (full, e.g. 2025)
- 'month' - month (1-12)
- 'day' - day (1-31)
- 'hour' - hour (0-23)
- 'minute' - minute (0-59)
- 'second' - second (0-59)

Examples:

```
RTC_SET 2025, 10, 13, 15, 30, 0
```

RTC_READ

Syntax:

```
RTC_READ
```

Description:
Outputs the current time and date from the RTC.

Examples:

```
RTC_READ
Output: 2025,10,13 15:30:45
```

LED matrices

All matrix commands use the **RGB888** (24-bit) color format. You can use:

- 'RGB888(r, g, b)' function - to create color from components
- Variables - 'VAR COLOR = RGB888(255, 0, 0)' → 'MATRIX_FILL COLOR'
- Hex literals - '0xFF0000', '0x00FF00', '0x0000FF'
- Expressions - 'RGB888(BRIGHTNESS*2, 128, 64)'

MATRIX_INIT

Syntax:

```
MATRIX_INIT(width, height, layout, corner)
```

Description:
Initializes the LED matrix of the specified size and topology.

Options:

- 'width' - matrix width (number of columns)
- 'height' - matrix height (number of rows)
- 'layout' - connection topology (optional, default "H"):
  - '"H"' - horizontal snake (lines alternate from left to right / right to left)
  - ''V'' - vertical snake (columns alternately go from bottom to top / top to bottom)
- 'corner' - the corner where the first physical LED/data input is located (optional, default 0):
  - '0' - Bottom-Left (bottom left)
  - '1' - Bottom-Right (bottom right)
  - '2' - Top-Right (top right)
  - '3' - Top-Left (top left)

Note: The logical origin (0,0) is always in the lower-left corner, regardless of physical location.

Examples:

```
16x16 Horizontal Snake Matrix, Data Input Bottom Left
MATRIX_INIT(16, 16, "H", 0)

8x8 vertical snake sensor, data input top right
MATRIX_INIT(8, 8, "V", 2)

No optional parameters (default "H", 0)
```

```
MATRIX_INIT(16, 16)

Only by specifying layout (default corner 0)
MATRIX_INIT(16, 16, "V")

With variables
VAR W=16, H=16
MATRIX_INIT(W, H, "V", 3)
```

MATRIX_FILL

Syntax:

```
MATRIX_FILL(color)
```

Description:
Fills the entire matrix with the specified color (RGB888).

Examples:

```
With a constant
MATRIX_FILL(0xFF0000) // Krasny

With RGB888 function
MATRIX_FILL(RGB888(255, 0, 0))

With a variable
VAR RED = RGB888(255, 0, 0)
MATRIX_FILL(RED)

With hex literal
MATRIX_FILL(0x00FF00) // Green

With the expression
VAR BRIGHTNESS = 128
MATRIX_FILL(RGB888(BRIGHTNESS, BRIGHTNESS, BRIGHTNESS))
```

MATRIX_SET

Syntax:

```
MATRIX_SET(x, y, color)
```

Description:
Sets the color of a single pixel of the sensor.

Examples:

```
With RGB888 function
MATRIX_SET(8, 8, RGB888(255, 0, 0))

With a variable
VAR COLOR = RGB888(0, 255, 0)
MATRIX_SET(10, 10, COLOR)

With hex literal
MATRIX_SET(5, 5, 0x0000FF)

Drawing a Line
FOR I = 0 TO 15
    MATRIX_SET(I, I, RGB888(255, 255, 0))
```

```
NEXT
MATRIX_UPDATE()

With coordinate and color variables
VAR X=8, Y=8
VAR BLUE = RGB888(0, 0, 255)
MATRIX_SET(X, Y, BLUE)
```

MATRIX_BITMAP

Syntax:

```
MATRIX_BITMAP(x, y, array[], width, height)
```

Description:
Outputs the bitmap from the array to the matrix. The colors in the array must be in the RGB888 format.

Examples:

```
Simple 4x4 Bitmap
VAR IMG[16]
IMG[0] = RGB888(255, 0, 0) // Red
IMG[1] = RGB888(0, 255, 0) // Green
IMG[2] = RGB888(0, 0, 255) // Blue
IMG[3] = RGB888(255, 255, 0) // Yellow
// ... Fill in the remaining pixels
MATRIX_BITMAP(0, 0, IMG[], 4, 4)

Creating a pattern
VAR COLORS[4] = {0xFF0000, 0x00FF00, 0x0000FF, 0xFFFF00}
VAR PATTERN[16]
FOR I = 0 TO 15
    PATTERN[I] = COLORS[I % 4]
NEXT
MATRIX_BITMAP(4, 4, PATTERN[], 4, 4)
MATRIX_UPDATE()

With hex literals
VAR SMILE[9] = {0x000000, 0xFF0000, 0x000000,
                0xFF0000, 0x000000, 0xFF0000,
                0x000000, 0xFF0000, 0x000000}
MATRIX_BITMAP(0, 0, SMILE[], 3, 3)
```

MATRIX_CHAR

Syntax:

```
MATRIX_CHAR(x, y, "char", font_name, color)
```

Description:
Outputs a single character per matrix. RGB888 color.

Available fonts:

- Font_6x8 - 6x8 px
- Font_7x10 - 7x10 pixels
- Font_11x18 - 11x18 pixels

Examples:

```
With RGB888 function
MATRIX_CHAR(0, 0, "A", Font_7x10, RGB888(255, 0, 0))

With variable color
VAR GREEN = RGB888(0, 255, 0)
MATRIX_CHAR(8, 0, "B", Font_7x10, GREEN)

With hex literal
MATRIX_CHAR(0, 8, "!", Font_6x8, 0xFF00FF)

Output of all letters
VAR X=0, Y=0
VAR COLOR = RGB888(255, 255, 0)
MATRIX_CHAR(X, Y, "H", Font_7x10, COLOR)
X = X + 8
MATRIX_CHAR(X, Y, "i", Font_7x10, COLOR)
MATRIX_UPDATE()
```

MATRIX_PRINT

Syntax:

```
MATRIX_PRINT(x, y, part1, part2, ..., font_name, color)
```

Description:
Prints text to a matrix (can combine strings and variables). RGB888 color.

Examples:

```
Plain text with RGB888
MATRIX_PRINT(0, 0, "Hello", Font_7x10, RGB888(255, 0, 0))

With variable color
VAR COLOR = RGB888(0, 255, 0)
MATRIX_PRINT(0, 8, "World", Font_6x8, COLOR)

Combination of text and variables
VAR COUNT = 42
MATRIX_PRINT(0, 0, "Count: ", COUNT, Font_7x10, RGB888(255, 255, 0))

With hex literal
MATRIX_PRINT(0, 0, "Status: OK", Font_6x8, 0x00FF00)

Formatting numbers
VAR TEMP = 23.456
MATRIX_PRINT(0, 0, "T:", TEMP.1, "C", Font_7x10, RGB888(255, 128, 0))

Counter animation
VAR I=0, COL = RGB888(0, 255, 255)
WHILE I < 100
    MATRIX_FILL 0x000000
    MATRIX_PRINT(0, 0, "Count:", I, Font_7x10, COL)
    MATRIX_UPDATE
    I = I + 1
    PAUSE 50
WEND
```

MATRIX_UPDATE

Syntax:

```
MATRIX_UPDATE()
```

Description:
Updates the contents of the matrix (sends data to the device).

Special Commands

WS2812_SEND

Syntax:

```
WS2812_SEND <array>[]
```

Description:
Sends data to the WS2812 addressable LED strip.

Options:

- 'array' - an array with RGB888 colors (0xRRGGBB)

Examples:

```
VAR LEDS[10]
FOR I = 0 TO 9
    LEDS[I] = RGB888(255, 0, 0)
NEXT
WS2812_SEND LEDS[]
```

SHIFT_LEFT

Syntax:

```
SHIFT_LEFT <array>, <count>
```

Description:
Shifts the elements of the array to the left by 'count' positions (cyclically).

Examples:

```
VAR A[5] = {1, 2, 3, 4, 5}
SHIFT_LEFT A, 2
Result: {3, 4, 5, 1, 2}

Animation of a creeping line of LEDs
VAR LEDS[10]
FOR I = 0 TO 9
    LEDS[I] = RGB888(255 - I*25, I*25, 0)
NEXT
WHILE 1
    WS2812_SEND LEDS[]
    SHIFT_LEFT LEDS, 1
    PAUSE 100
WEND
```

SHIFT_RIGHT

Syntax:

```
SHIFT_RIGHT <array>, <count>
```

Description:
Shifts the elements of the array to the right by 'count' positions (cyclically).

Examples:

```
VAR A[5] = {1, 2, 3, 4, 5}
SHIFT_RIGHT A, 2
Result: {4, 5, 1, 2, 3}

Reverse Creeping Line Animation
VAR LEDS[10]
FOR I = 0 TO 9
    LEDS[I] = RGB888(I*25, 255 - I*25, 128)
NEXT
WHILE 1
    WS2812_SEND LEDS[]
    SHIFT_RIGHT LEDS, 1
    PAUSE 100
WEND
```

STUSB_INIT

Syntax:

```
STUSB_INIT()
```

Description:
Initializes the USB Power Delivery STUSB4500 controller.

STUSB_SET_VOLTAGE

Syntax:

```
STUSB_SET_VOLTAGE(<voltage>[, <current>])
```

Description:
Requests the specified voltage from the USB-C PD power supply.

Options:

- 'voltage' - voltage in volts (e.g. 5.0, 9.0, 12.0, 15.0, 20.0)
- 'current' - current in amperes (default 1.5A)

Examples:

```
STUSB_SET_VOLTAGE(12.0, 3.0)
STUSB_SET_VOLTAGE(20.0)
```

STUSB_GET_SOURCE

Syntax:

```
STUSB_GET_SOURCE
```

Description:
Lists the available voltages from the USB-C PD power supply.

DEBUG_ON

Syntax:

```
DEBUG_ON [<delay_ms>]
```

Description:
Enables debugging mode with optional delay between lines.

Examples:

```
DEBUG_ON
DEBUG_ON 100
```

DEBUG_OFF

Syntax:

```
DEBUG_OFF
```

Description:
Disables debug mode.

END

Syntax:

```
END
```

Description:
Stops the execution of the program.

GOSUB / RETURN

Syntax:

```
GOSUB <label>
...
<label>:
    Command
    RETURN
```

Description:
Call a routine on a check-in label.

Examples:

```
GOSUB DrawCircle
END

DrawCircle:
```

```
    DISPLAY_FILL_CIRCLE(160, 120, 50, RGB565(255, 0, 0))
    RETURN
```

VARS_READ

Syntax:

```
VARS_READ(name1, name2, name3, ...)
```

Description:
Starts a background task to periodically poll and output the values of the specified variables. Used inside a BASIC script to monitor variables.

Options:

- 'nameN' - names of variables to be monitored (separated by commas)

Examples:

```
Initializing sensors
DHT_INIT(TEMP, HUM, 2000)
BMP280_INIT(PRESS, TEMP2, 1000)

Run variable monitoring
VARS_READ(TEMP, HUM, PRESS)

The program continues to work
WHILE 1
    Variables are automatically polled in the background
    DISPLAY_TEXT(0, 0, "T:", TEMP.1, " C", Font_7x10, 1, 0xFFFF)
    DISPLAY_TEXT(0, 20, "H:", HUM.1, " %", Font_7x10, 1, 0xFFFF)
    PAUSE 500
WEND

Example with stopping monitoring
VARS_READ(X, Y, Z)
PAUSE 5000
STOP_VARS_READ
```

Note:

- The survey task runs in the background
- Values are automatically output via USB/UART
- To stop, use the STOP_VARS_READ command
- Different from the VARIABLES_READ system command (which is called externally via USB/UART)

STOP_VARS_READ

Syntax:

```
STOP_VARS_READ
```

Description:
Stops a background variable monitoring task started by the VARS_READ command.

Examples:

```
Start monitoring
VARS_READ(TEMP, HUM)
```

```
Work for 10 seconds
PAUSE 10000

Stop monitoring
STOP_VARS_READ

PRINT "Monitoring stopped"
```

Math Functions

Trigonometric Functions

```
SIN(x) // Sine (x in radians)
COS(x) // Cosine (x in radians)
TAN(x) // Tangent (x in radians)
ASIN(x) // Arx sine
ACOS(x) // Arccosine
ATAN(x) // Arctangent
```

Rounding and absolute value functions

```
ROUND(x) // Rounding to the nearest integer
FLOOR(x) // Rounding down
CEIL(x) // Round up
ABS(x) // Absolute value
```

Degree and Root

```
POW(x, y) // x to the power of y
SQRT(x) // Square root
EXP(x) // e to the power of x
LOG(x) // Natural logarithm
LOG10(x) // Decimal Logarithm
```

Random Numbers

```
RND() // Random number from 0.0 to 1.0
```

Sample Programs

Example 1: Color-coded thermometer

```
DHT_INIT(TEMP, HUM, 2000)

WHILE 1
    VAR COLOR
    IF TEMP > 30 THEN
        COLOR = RGB565(255, 0, 0)
    ELSE IF TEMP > 20 THEN
        COLOR = RGB565(255, 165, 0)
    ELSE
        COLOR = RGB565(0, 255, 0)
    ENDIF
```

```
    DISPLAY_FILL_RECT(10, 10, 100, 50, COLOR)
    PAUSE 1000
WEND
```

Example 2: Interactive button

```
BUTTON_INIT(10, 10, 100, 50, RGB565(0, 255, 0), "START", 0, TX, TY, TOUCHED, OnPress, )

END

OnPress:
    PRINT "Button pressed!"
    DISPLAY_FILL_CIRCLE(160, 120, 30, RGB565(255, 0, 0))
    RETURN
```

Example 3: Animated Progress

```
VAR PROGRESS = 0
PROGRESS_INIT(10, 100, 300, 30, PROGRESS, 0, 100, RGB565(0, 255, 0))

FOR PROGRESS = 0 TO 100
    PAUSE 50
NEXT

PRINT "Complete!"
```

Example 4: Sprites and animations on a display with DISPLAY_BITMAP

```
Creating an 8x8 Player Sprite
VAR PLAYER[64]

Drawing a simple man
VAR WHITE = RGB565(255, 255, 255)
VAR SKIN = RGB565(255, 200, 150)
VAR RED = RGB565(255, 0, 0)
VAR BLUE = RGB565(0, 0, 255)
VAR BLACK = RGB565(0, 0, 0)

Fill with a background (transparent = black)
FOR I = 0 TO 63
    PLAYER[I] = BLACK
NEXT

Head (skin)
PLAYER[2*8 + 3] = SKIN
PLAYER[2*8 + 4] = SKIN
PLAYER[3*8 + 3] = SKIN
PLAYER[3*8 + 4] = SKIN

Body (red shirt)
PLAYER[4*8 + 3] = RED
PLAYER[4*8 + 4] = RED
PLAYER[5*8 + 3] = RED
PLAYER[5*8 + 4] = RED

Legs (blue pants)
PLAYER[6*8 + 3] = BLUE
PLAYER[6*8 + 4] = BLUE

Creating a 4x4 coin
VAR COIN[16]
VAR GOLD = RGB565(255, 215, 0)
COIN[0] = BLACK
COIN[1] = GOLD
```

```
COIN[2] = GOLD
COIN[3] = BLACK
COIN[4] = GOLD
COIN[5] = GOLD
COIN[6] = GOLD
COIN[7] = GOLD
COIN[8] = GOLD
COIN[9] = GOLD
COIN[10] = GOLD
COIN[11] = GOLD
COIN[12] = BLACK
COIN[13] = GOLD
COIN[14] = GOLD
COIN[15] = BLACK

Game Variables
VAR PX = 50 // Player position
VAR PY = 100
VAR CX = 200 // Coin Position
VAR CY = 100
VAR SCORE = 0

Gameplay Loop
WHILE 1
    DISPLAY_CLEAR(RGB565(50, 150, 255)) // Sky

    Earth
    DISPLAY_FILL_RECT(0, 180, 320, 60, RGB565(100, 200, 50))

    Drawing a player with a scale of x3
    DISPLAY_BITMAP(PX, PY, PLAYER[], 8, 8, 3)

    Drawing a coin with a scale of x4
    DISPLAY_BITMAP(CX, CY, COIN[], 4, 4, 4)

    Account
    DISPLAY_TEXT(10, 10, "Score: ", SCORE, Font_7x10, 1, WHITE)

    Control (via ADC or buttons)
    VAR CONTROL = ADC_IN1 * 100
    IF CONTROL > 50 THEN
        PX = PX + 3
    ENDIF
    IF CONTROL < 30 THEN
        PX = PX - 3
    ENDIF

    Collision Check (Simple)
    IF PX > CX-20 AND PX < CX+20 AND PY > CY-20 AND PY < CY+20 THEN
        SCORE = SCORE + 1
        CX = RND() * 280 // New coin position
    ENDIF

    Traffic restriction
    IF PX < 0 THEN
        PX = 0
    ENDIF
    IF PX > 290 THEN
        PX = 290
    ENDIF

    PAUSE 50
WEND
```

Example 5: Animation on an LED matrix

```
16x16 matrix initialization
MATRIX_INIT(16, 16)
```

```
Defining Colors Using Variables and RGB888 Functions
VAR RED = RGB888(255, 0, 0)
VAR GREEN = RGB888(0, 255, 0)
VAR BLUE = RGB888(0, 0, 255)
VAR YELLOW = RGB888(255, 255, 0)

Or via hex literals
VAR CYAN = 0x00FFFF
VAR MAGENTA = 0xFF00FF
VAR WHITE = 0xFFFFFF

Ticker Point Animation
VAR X=0, Y=0
WHILE X < 16
    MATRIX_FILL 0x000000 // Clear Matrix (Black)
    MATRIX_SET X, Y, RED // Set red dot
    MATRIX_UPDATE // Refresh Display
    X = X + 1
    PAUSE 100
WEND

Gradient
FOR I = 0 TO 15
    VAR BRIGHTNESS = I * 16
    VAR COLOR = RGB888(BRIGHTNESS, 0, 255-BRIGHTNESS)
    FOR J = 0 TO 15
        MATRIX_SET I, J, COLOR
    NEXT
NEXT
MATRIX_UPDATE

Creating a pattern with an array of
VAR COLORS[4]
COLORS[0] = RGB888(255, 0, 0)
COLORS[1] = RGB888(0, 255, 0)
COLORS[2] = RGB888(0, 0, 255)
COLORS[3] = RGB888(255, 255, 0)

FOR I = 0 TO 15
    FOR J = 0 TO 15
        VAR C_INDEX = (I + J) % 4
        MATRIX_SET I, J, COLORS[C_INDEX]
    NEXT
NEXT
MATRIX_UPDATE

Displaying variable color text
VAR COUNTER = 0
WHILE COUNTER < 10
    MATRIX_FILL 0x000000
    VAR TEXT_COLOR = RGB888(255, COUNTER*25, 0)
    MATRIX_PRINT(0, 0, "Count:", COUNTER, Font_7x10, TEXT_COLOR)
    MATRIX_UPDATE
    COUNTER = COUNTER + 1
    PAUSE 500
WEND
```

System Control Commands

These commands are sent via USB or UART to control the device externally and are not part of the interpreter's BASIC script.

VARIABLE_SET

Syntax:

```
VARIABLE_SET(name1,value1:name2,value2:name3,value3...)
```

Description:
Sets the values of one or more variables from an external source (terminal, application PC). Variables are separated by a colon, within each pair, the name and value are separated by a comma.

Options:

- 'nameN' - variable name (created if it does not exist)
- 'valueN' - the value of the variable (can be a number or an expression)

Examples:

```
# Set a single variable
VARIABLE_SET(TEMP,25.5)

# Set multiple variables
VARIABLE_SET(X,100:Y,200:Z,50)

# With expressions
VARIABLE_SET(A,10:B,20:C,30.5:D,100)

# Mixed Value Types
VARIABLE_SET(VOLTAGE,3.3:COUNT,42:BRIGHTNESS,255)
```

Note:

- The command is processed before the interpreter starts
- If the variable does not exist, it will be created automatically
- Only scalar variables are supported (arrays are not supported)
- Expressions are evaluated via EvalExpression (support +, -, *, /, parentheses)
- Useful for remote control of device parameters

VARIABLES_READ

Syntax:

```
VARIABLES_READ(Name1,Name2,Name3,...)
```

Description:
Triggers periodic sending of specified variable values via USB/UART. Values are automatically sent when they change.

Examples:

```
# Monitoring a single variable
VARIABLES_READ(TEMP)

# Multivariable Monitoring
VARIABLES_READ(TEMP,HUM,PRESSURE)

# Sensor Monitoring
VARIABLES_READ(ADC1,ADC2,ADC3,VOLTAGE)
```

Note:

- Starts a background variable polling task
- To stop, use the command STOP_VARIABLES_READ

STOP_VARIABLES_READ

Syntax:

```
STOP_VARIABLES_READ
```

Description:
Stops the periodic sending of variables triggered by the VARIABLES_READ command.

Notes

- All angles in trigonometric functions are specified in radians
- The coordinates (0,0) are in the upper left corner of the screen
- Color values are automatically limited to 0-255
- Variable names can contain letters, numbers, and underscores
- Case is not case-sensitive in command names, but it is in variable names
- Comments start with '//' and continue to the end of the line