BASIC Einplatinen-Computer auf Basis von einem STM32F429i-Discovery-Board

Autor : Uwe Becker
Web : www.Mikrocontroller-4U.de
EMail : mc-4u@t-online.de

Beschreibung :

Wer noch den guten alten C64 von Commodore kennt, wird sich an die einfache Art und Weise der Bedienung erinnern können. Das Gerät einschalten, ein paar Zeilen Programm eingeben "RUN" eintippen und sehen was passiert.

Ich wollte so ein System mit "neuer" Technik wieder ins Leben rufen.

Das Discovery-Board von ST mit dem STM32F429 bringt alles mit, was so ein Minimalsystem an Hardware braucht. Ein Grafik-Display als Ausgabemedium, ein USB-Port zum Anschluss einer USB-Tastatur als Eingabemedium, ein paar freie GPIO-Pins als Schnittstelle zur Außenwelt für Sensoren/Aktoren.

Wer will kann als "externes Speichermedium" eine SD-Karte benutzen, um von dort BASIC-Programme zu laden oder zu sichern.

Die Software auf dem System stellt für den User einen Full-Screen-Editor und einen Inline-BASIC-Interpreter zur Verfügung. Dadurch kann nach dem einschalten direkt mit dem programmieren losgelegt werden.

Vom Interpreter werden die "Standard" BASIC-Befehle unterstützt und einige die ich speziell auf die eingesetzte Hardware zugeschnitten habe.

Für Programmier-Anfänger sind die Einschränkungen vom System eher ein Gewinn, da sie komplizierte Befehlsverschachtelungen gar nicht zulassen. Und da "BASIC" von sich aus eine sehr intuitive Sprache ist, kann sie auch innerhalb kurzer Zeit erlernt werden.

Das Grundgerüst vom BASIC-Interpreter ist nicht von mir sondern von 'Adam Dunkels' und kann hier eingesehen werden : http://dunkels.com/adam/ubasic/

Allerdings habe ich einiges abgeändert und verbessert um die Geschwindigkeit zu erhöhen und um den Sprachumfang zu erweitern.

Hinweise :

Diese Doku ist sicher voll von Rechtschreibfehlern (die mir herzlich egal sind) aber falls jemand Inhaltliche Fehler findet (oder Punkte die ich vergessen habe) kann er mir gerne eine Mail schreiben.

Die Light-Version ist als Source-Code verfügbar. Dort gibt es nur die "standard" Basic Befehle und die zwei Befehle "ABS" und "RIGHT$" zur Demo einer Integer und einer String Bearbeitung. Im Editor-Mode gibt es keine Einschränkungen.

Jetzt viel Spass mit dem BASIC-Computer per STM32F429i-Discovery-Board
Gruß Uwe

Inhaltsverzeichnis

1 System Überblick :

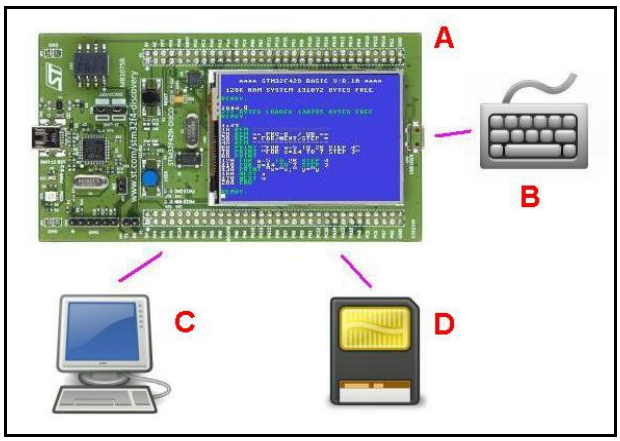
CPU	STM32F429 (32bit ARM Cortex-4) @ 168 MHz
LCD	320 x 240 Pixel (16bit Farbe)
Monitor*	VGA-Mode : 640 x 480 @ 60Hz Screen : 320 x 240 Pixel (16bit Farbe)
Font	5 verschiedene Schriftarten (8x8 Pixel bis 10x15 Pixel)
Farben	Editor-Mode : 16 Farben (für Text, Hintergrund, Systemmeldungen) BASIC-Mode : 65536 Farben
Tastatur	Standard USB-Tastatur (kann auch vom BASIC-Programm abgefragt werden)
Touch	kann vom BASIC-Programm abgefragt werden
RAM	128k Byte für BASIC-Programme 128k Byte für Daten und Variabeln
Schnittstelle	UART-Verbindung zum PC, zur alternativen Steuerung vom System (und zum laden/speichern von BASIC-Programmen)
SD-Karte	optional kann extern eine SD-Karte angeschlossen werden (zum laden/speichern von BASIC-Programmen)
USB-Drive*	optional kann extern ein USB-Stick angeschlossen werden (zum laden/speichern von BASIC-Programmen)
I/O-Pins	vom BASIC-Programm aus können 25 I/O-Pins benutzt werden (davon 3 als Analog-Eingänge)
I/O-Ports	vom BASIC-Programm aus können mehrere I/O-Ports benutzt werden : ▸ 1x UART (fest mit 115200 Baud) ▸ 1x UART (einstellbar mit 2400 Baud bis 115200 Baud) ▸ 1x I2C ▸ 1x SPI (fest mit Mode-0, MSB und Frq=5,2MHz) ▸ 1x SPI (einstellbar Mode-0 bis Mode-3, MSB/LSB, Frq=330kHz bis 32MHz)

(*VGA-Monitor und LCD funktionieren nicht gleichzeitig)

(* USB-Drive als Speichermedium ist noch nicht implementiert)

2 Hardware Versionen :

2.1 Hardware Typ-I (Keyboard-Version) :



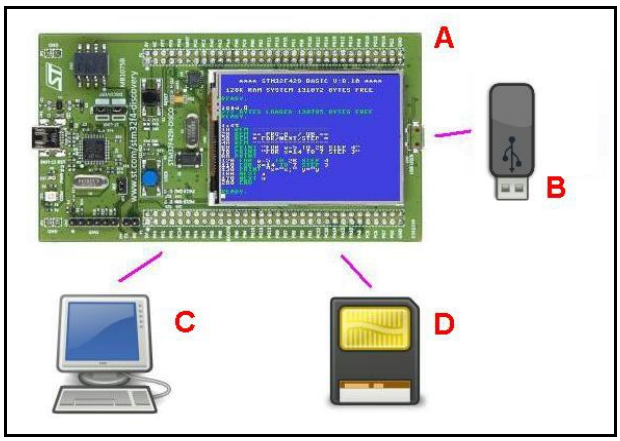
A ▸ STM32F429-BASIC-System

B ▸ USB-Tastatur
zur Bedienung vom System

C ▸ PC-Terminal (optional)
zur Bedienung vom System und
speichern/laden von BASIC-Programmen

D ▸ SD-Karte (optional)
zum speichern/laden von
BASIC-Programmen

2.2 Hardware Typ-II (USB-Drive-Version) * *noch nicht implementiert :*



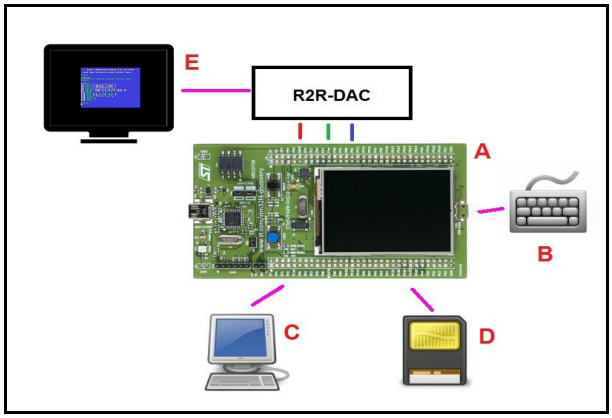
A ▸ STM32F429-BASIC-System

B ▸ USB-Drive (optional)
zum speichern/laden von BASIC-Programmen

C ▸ PC-Terminal
zur Bedienung vom System und speichern/laden
von BASIC-Programmen

D ▸ SD-Karte (optional)
zum speichern/laden von BASIC-Programmen

2.3 Hardware Typ-III (VGA-Version) :



A ▸ STM32F429-BASIC-System

B ▸ USB-Tastatur
zur Bedienung vom System

C ▸ PC-Terminal (optional)
zur Bedienung vom System und
speichern/laden von BASIC-Programmen

D ▸ SD-Karte (optional)
zum speichern/laden von
BASIC-Programmen

E ▸ VGA-Monitor als Bildschirm

Hinweis : im VGA-Mode ist das interne LCD vom Discovery-Board deaktiviert.

3 Hardware Belegung :

3.1 COM-1 Pinbelegung :

PA9 = TxD PA10 = RxD

3.2 SD-Karte Pinbelegung :

PC8 = DAT0 PC12 = CLK PD2 = CMD CD (per 4k7 an Vcc)

3.3 VGA Pinbelegung :

ROT	**GRÜN**	**BLAU**	**SYNC**
R7=PG6	G7=PD3	B7=PB9	HSync = PC6
R6=PB1	G6=PC7	B6=PB8	VSync = PA4
R5=PA12	G5=PB11	B5=PA3	
R4=PA11	G4=PB10	B4=PG12	
R3=PB0	G3=PG10	B3=PG11	
R2=PC10	G2=PA6	B2=PD6	

3.4 COM-6 Pinbelegung :

PG14 = TxD PG9 = RxD

3.5 I2C-3 Pinbelegung :

PA8 = SCL PC9 = SDA

3.6 SPI-5 Pinbelegung :

PF7 = SCK PF9 = MOSI PF8 = MISO

3.7 SPI-4 Pinbelegung :

PE2 = SCK PE6 = MOSI PE5 = MISO

3.8 Digital-Pins :

PA0	PB2	PC3	PD4	PE2*	PF6	PG2
PA5	PB4	PC11	PD5	PE3		PG3
PA7	PB7	PC13	PD7	PE4		PG9*
		PC14		PE5*		PG13
		PC15		PE6*		PG14*

(* = Doppelbelegung mit einer Schnittstelle)

3.9 Analog-Pins :

PA5 PA7 PC3

4 Editor-Mode :

4.1 Full-Screen-Editor :

Nach dem starten vom System ist der Editor-Mode aktiv (erkennbar am blinkenden Cursor).

In diesem Mode werden BASIC-Programme eingegeben/angezeigt/geladen/gespeichert usw.

Zum editieren kann der komplette Screen benutzt werden. Der Cursor kann mit den Cursor-Tasten bewegt werden und per "RETURN" wird die Zeile in der der Cursor gerade steht übernommen.

Im Editor-Mode können direkt sogenannte "Editor-Befehle" eingegeben und per "RETURN" ausgeführt werden. (siehe Liste der Editor-Befehle)

Basic-Zeilen beginnen immer mit einer Zeilen-Nummer und können auf dem gesamten Screen eingegeben und editiert werden.

Das System kann anstelle von der USB-Tastatur auch mit einem PC über die COM-1 bedient werden. (siehe Terminal-Betrieb)

4.2 Bildschirm :

Die "normal-Version" benutzt das interne LCD vom Discovery-Board als Anzeige.

Es gibt aber auch die Möglichkeit über einen zusätzlichen R2R-DAC ein VGA-Monitor anzuschließen.Dafür gibt es ein extra HEX-File bei dem das interne LCD abgeschaltet ist.

Die Auflösung beträgt jeweils 320 x 240 Pixel in 16bit Farben.
Wenn eine Schriftart von 8x8 Pixel eingestellt ist, passen 30 Zeilen zu je 40 Zeichen auf den Screen.

Die linke obere Bildschirmecke hat die Koordinate 0,0.

4.3 Sondertasten :

'Cursor-Tasten'	▸ zum navigieren vom Cursor
'BackSpace'	▸ löscht das Zeichen links der CursorPos und verschiebt den Cursor
'Entfernen'	▸ löscht das Zeichen an CursorPos
'Einfügen'	▸ fügt an CursorPos ein 'Space' ein
'Pos1'	▸ verschiebt den Cursor an den linken Rand
'Ende'	▸ verschiebt den Cursor auf das letzte Zeichen der Zeile
'F9'	▸ macht ein 'ClearScreen'
'ALT+F1'	▸ sendet den Bildschirminhalt als BMP-File per COM-1
'STRG+ALT+ENTF'	▸ macht einen 'Reset' vom System

4.4 Schriftarten :

Auf dem System sind 5 Schriftarten installiert.
Nach dem starten ist "Font-0 = 8x8 Pixel C64" aktiv.

Mit der Taste 'F12' kann zwischen den Schriftarten durch gewechselt werden :

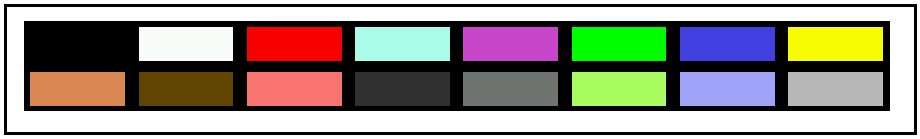
Font-0	▸ 8x8 Pixel C64	Font-3	▸ 10x15 Pixel Arial
Font-1	▸ 8x13 Pixel Arial	Font-4	▸ 10x15 Pixel Arial (bold)
Font-2	▸ 8x13 Pixel Arial (bold)		

4.5 Farben :

Im Editor-Mode kann zwischen 16 verschiedene Farben gewählt werden.

Die Farben für Text, Hintergrund und Systemmeldungen können getrennt voneinander eingestellt werden und gelten immer für ein komplettes Zeichen.

Textfarbe	▸ Taste : 'STRG+1 bis 8' bzw. 'STRG+SHIFT+1 bis 8'
Hintergrundfarbe	▸ Taste : 'ALT+1 bis 8' bzw. 'ALT+SHIFT+1 bis 8'
Systemfarbe	▸ Taste : 'ALTGR+1 bis 8' bzw. 'ALTGR+SHIFT+1 bis 8'



Im BASIC-Mode können 65535 Farben für jedes Pixel benutzt werden.

Der Farbwert entspricht RGB565 : 0x0000 = schwarz und 0xFFFF = weiß.

4.6 Terminal Betrieb :

An die COM-1 vom System kann ein PC mit Terminal-Programm angeschlossen werden.

Über diese Schnittstelle kann das System bedient werden genauso wie wenn man die Befehle über die USB-Tastatur eingeben würde.

Die Einstellungen der COM-1 sind fix auf 115200 Baud 8N1. Es dürfen nur Ascii-Zeichen gesendet werden und als Endekennung von jedem Befehl muss ein 0x0D (CarriageReturn) angehängt werden.

Um die COM-1 auch zur Ausgabe zu nutzen, kann der Editor-Befehl 'TERMINAL' benutzt werden. (siehe Editor-Befehl : TERMINAL)

Ein Laufendes BASIC-Programm wird mit dem UART-Befehl "ESC" abgebrochen oder mit dem UART-Befehl "EXIT" sofort beendet.

Das STM32F4-Terminal-Programm von mir für den PC hat eine eingebaute Registerkarte in dem BASIC-Programme geschrieben und dann zum STM32F4-BASIC-System übertragen werden können.

4.7 Laufwerke :

Zum laden/speichern von BASIC-Programmen gibt es 4 Laufwerke :

DEVICE-0	▸ UART (COM-1 : PA9=TX, PA10=RX, 115200 Baud, 8N1)
DEVICE-1	▸ FLASH (nur für LOAD)
DEVICE-2	▸ SD-Karte (1bit SDIO : PC8=DAT0, PC12=CLK, PD2=CMD)
DEVICE-3	▸ USB-Drive* (USB-MSC-Host : User-USB)

(* USB-Drive als Speichermedium ist noch nicht implementiert)

4.8 Filesystem :

Das Filesystem der SD-Karte (und USB-Drive) muss FAT16 sein.

Die Filenamen müssen im Format 8.3 sein. z.B. "Testprg1.bas"

Unterverzeichnisse werden im Moment nicht unterstützt.

5 Editor-Befehle :

Die Editor-Befehle werden im Editor-Mode über die Tastatur eingegeben oder über die COM-1 von einem Terminal-Programm empfangen.

Das System führt den Befehl aus und zeigt eine Meldung an, falls der Befehl unbekannt war oder es einen Syntaxfehler gab. Die meisten Befehle werden mit einem „READY“ bestätigt.

5.1 CLS :

'CLS' steht für 'ClearScreen' und löscht den kompletten Bildschirm mit der aktuellen Hintergrundfarbe. Zusätzlich wird der Cursor auf die Koordinate 0,0 gesetzt
(also die linke obere Ecke)

5.2 VERSION :

Mit diesem Befehl wird die Versions-Nummer und das Datum der Firmware vom STM32F429-BASIC angezeigt.

5.3 RESET :

Der Befehl 'RESET' macht das gleiche wie der Reset-Button. Er setzt das komplette System auf die Grundeinstellungen zurück und löscht den Speicher.

5.4 FREE :

Mit 'FREE' wird angezeigt wie viel Bytes vom Speicher das aktuelle BASIC-Programm belegt und wie viele Bytes noch frei sind. Es wird auch angezeigt wie hoch der Speicherbedarf der einzelnen Arrays ist.

5.5 NEW :

Der Befehl 'NEW' löscht das aktuelle BASIC-Programm. Alle anderen Einstellungen bleiben erhalten

5.6 TERMINAL :

Um alle Meldungen vom System zusätzlich zum LCD auch über die COM-1 auszugeben, muss der Befehl 'TERMINAL' benutzt werden.

'TERMINAL 1' ▸ aktiviert den Terminalbetrieb über COM-1
'TERMINAL 0' ▸ deaktiviert den Terminalbetrieb über COM-1

5.7 DELETE :

Um eine Zeile aus einem BASIC-Programm zu löschen, muss 'DELETE' mit anschließender Zeilen-Nummer aufgerufen werden.

(Hinweis die Zeilen-Nummer muss im BASIC-Programm existieren)

'DELETE 115' ▸ löscht die Zeile im BASIC-Programm mit der Zeilen-Nummer '115'

5.8 PARSE :

Der Befehl 'PARSE' startet den Pre-Parser. Dieser Testet das BASIC-Programm auf Syntax-Fehler. Bei einem Fehler, wird der Pre-Parser abgebrochen und eine Fehlermeldung mit Angabe der Zeilen-Nummer mit dem Fehler wird ausgegeben.
Hinweis : der Pre-Parser wird auch automatisch gestartet wenn der Befehl 'RUN' benutzt wird.

5.9 LIST :

Mit 'LIST' wird das Listing vom BASIC-Programm angezeigt.

(Das Syntax-Highlighting ist erst nach dem Befehl "PARSE" oder "RUN" aktiv)

Der Befehl kann in 5 verschiedenen Varianten aufgerufen werden :

'LIST'	▸ listet das komplette Programm von der ersten bis zur letzten Zeile
'LIST 100'	▸ listet nur die Zeile 100
'LIST 100 -'	▸ listet von der Zeile 100 bis zur letzten Zeile
'LIST - 200'	▸ listet von der ersten Zeile bis zur Zeile 200
'LIST 100 - 200'	▸ listet von der Zeile 100 bis zur Zeile 200

Mit der Taste 'ESC' kann das Listing abgebrochen werden.

Mit der Taste 'STRG' kann das Listing verlangsamt werden.

5.10 RUN :

Der Befehl 'RUN' startet das BASIC-Programm und wechselt in den BASIC-Mode.
(Falls der Pre-Parser einen Fehler entdeckt wird das Programm nicht gestartet)

Das BASIC-Programm wird solange abgearbeitet, bis der BASIC-Befehl 'END' oder das Fileende erreicht ist oder ein Fehler im Programm erkannt wurde.

Die Taste 'ESC' beendet das BASIC-Programm vorzeitig.

Wenn das BASIC-Programm beendet ist, leuchtet die 'grüne LED'.
Mit einer beliebigen Taste kann dann wieder in den Editor-Mode gewechselt werden.

Die Tastenkombination 'ALT+F4' beendet das BASIC-Programm und wechselt sofort in den Editor-Mode zurück.

5.11 DEVICE :

Es gibt 4 verschiedene Laufwerke von denen BASIC-Programme geladen/gespeichert werden können. Es ist immer nur ein Laufwerk aktiv (nach PowerOn = 'FLASH').

Mit dem Befehl 'DEVICE' und anschließender Laufwerks-Nummer kann zwischen den Laufwerken umgeschaltet werden.

'DEVICE 0'	aktives Laufwerk	▸ UART (LOAD / SAVE)
'DEVICE 1'	aktives Laufwerk	▸ FLASH (LOAD / DIR)
'DEVICE 2'	aktives Laufwerk	▸ SD-Karte (LOAD / SAVE / DIR / REMOVE)
'DEVICE 3'	aktives Laufwerk	▸ USB-Drive* (LOAD / SAVE / DIR / REMOVE)

(* USB-Drive als Speichermedium ist noch nicht implementiert)

5.12 DIR :

Mit 'DIR' wird das Directory vom aktuellen Laufwerk angezeigt.
(Vom Laufwerk 'UART' kann kein Directory angezeigt werden)

Mit der Taste 'ESC' kann die Auflistung abgebrochen werden.

Mit der Taste 'STRG' kann die Auflistung verlangsamt werden.

5.13 CD :

Mit dem Befehl 'CD "Directoryname"' wird in ein anderes Verzeichnis gewechselt.

Das Verzeichnis muss existieren und wird per <DIR> gekennzeichnet.
Per 'CD ".."' wird eine Verzeichnis-Ebene zurück gewechselt.

Diese Funktion wird nur im Laufwerk 'FLASH' unterstützt.

5.14 LOAD :

Mit dem Befehl 'LOAD "Filename"' wird ein BASIC-Programm vom aktiven Laufwerk in den internen Speicher vom System geladen.

Der Filename muss auf dem Laufwerk existieren.

Beim Laufwerk 'UART' ist kein Filename notwendig (LOAD ""). Nach dem Aufruf wird kurz gewartet ob Daten an COM-1 eintreffen, falls nicht wird das Laden abgebrochen.

'LOAD "Test1.bas"' ▸ das File 'Test1.bas' wird vom aktiven Laufwerk geladen.

5.15 SAVE :

Der Befehl 'SAVE "Filename"' speichert das BASIC-Programm vom internen Speicher in das aktuelle Laufwerk.

In das Laufwerk 'FLASH' kann nicht gespeichert werden.

Beim Laufwerk 'UART' ist kein Filename notwendig (SAVE ""). Nach dem Aufruf wird das Programm per COM-1 gesendet.

'SAVE "Test1.bas"' ▸ das BASIC-Programm wird als 'Test1.bas' gespeichert.

5.16 REMOVE :

Um ein File vom aktuellen Laufwerk zu löschen, muß 'REMOVE "Filename"' eingegeben werden.

Der Filename muss auf dem Laufwerk existieren.
(Vom Laufwerk 'FLASH' und 'UART' kann nicht gelöscht werden)

'REMOVE "Test1.bas"' ▸ das File 'Test1.bas' wird vom aktiven Laufwerk gelöscht.

6 BASIC-Mode :

Damit ein BASIC-Programm abgearbeitet werden kann, muss es sich im RAM Speicher vom System befinden.Das kann entweder durch direktes eingeben über den Editor oder durch laden von einem Speichermedium gemacht werden. Der Befehl 'RUN' startet den Pre-Parser und wenn dieser keinen Fehler findet, wird das BASIC-Programm ab der ersten Zeile vom Interpreter abgearbeitet.

Wird während dem Programmlauf ein Fehler erkannt, wird das Programm abgebrochen.

Das Programm wird solange ausgeführt bis die letzte Programm-Zeile erreicht ist oder der Interpreter auf den BASIC-Befehl 'END' stößt oder der User das Programm unterbricht.
(per Tastatur oder über die COM-1)

6.1 Einschränkungen vom BASIC :

Jede BASIC-Zeile muss mit einer eindeutigen (aufsteigenden) Zeilen-Nummer beginnen
(von 1 bis 99999)

Gültige integer Variablen sind 'a' bis 'z'
Gültige integer Arrays sind 'A(n)' bis 'Z(n)' wobei 'n' die Indexnummer ist
Gültige String Variablen sind 'a$' bis 'z$' (je max 85 Zeichen lang)
Gültige String Arrays sind 'A$(n)' bis 'Z$(n)' wobei 'n' die Indexnummer ist
Gültige Daten Arrays sind 'A#(n)' bis 'Z#(n)' wobei 'n' die Indexnummer ist

Als Sprungziele von GOTO und GOSUB dürfen keine Variablen benutzt werden
und die Zeilen-Nummer vom Sprungziel muss existieren.

Zahlenwerte dürfen nur ganzzahlig im Bereich von -99999 bis 99999 liegen.
Hexadezimale Schreibweise für positive Zahlen ist erlaubt von '0x0' bis '0xFFFF'

6.2 Funktionen :

Einige BASIC-Befehle sind in der Doku als "Funktionen" deklariert. Diese Befehle liefern beim Aufruf einen Rückgabewert zurück (entweder integer oder String).

Dieser Rückgabewert kann in eine Variable gespeichert werden oder er kann als Parameter für einen anderen BASIC-Befehl benutzt werden.

Zum Beispiel die Funktion 'ABS' die den Absolutwert einer Zahl zurückliefert kombiniert mit dem Befehl 'SQR' der die Wurzel einer Zahl bildet :

```
10 a=SQR(ABS(-9))          ▸ 'a' = 3
```

Bei dieser Schreibweise muss darauf geachtet werden, das alle notwendigen Klammern vorhanden sind.

6.3 Bildschirm :

Alle Ausgabe Befehle die den Bildschirm betreffen benutzen als „Standard“ Farbe die aktuelle Vordergrund- und Hintergrundfarbe vom Editor-Mode.

Falls die Farbe geändert werden soll, kann das mit den zwei BASIC-Befehlen 'INK' und 'PAPER' während dem Programmlauf gemacht werden.

Bei einigen Befehlen kann die Zeichenfarbe auch als Parameter mit angegeben werden.

6.4 Tastatur :

Falls keine Tastatur angeschlossen ist, liefert eine Abfrage immer den Wert 0 zurück.
(für : 'keine Taste gedrückt')

Es können maximal zwei gedrückte Tasten gleichzeitig erkannt werden.
Die Shift-Tasten (SHIFT,ALT,STRG usw. können unabhängig davon alle erkannt werden)

6.5 GPIOs und PORTs :

Beim initialisieren von Ports im BASIC-Programm werden festgelegte GPIO-Pins dafür benutzt.
(siehe 'Hardware Schnittstellen')

Der User muss sich selbst darum kümmern, das im Programm diese Pins nicht an anderer Stelle im Programm neu definiert oder doppelt belegt werden.

Der User muss auch selbst darauf achten, das durch seine angeschlossene Hardware keine Kurzschlüsse mit Ausgangsportpins entstehen können.

6.6 Variablen :

Das System unterstützt zwei Arten von Variablen :

- Integer Variablen
- String Variablen

6.6.1 Integer Variablen :

Integer Variablen werden mit einem einzelnen Kleinbuchstaben gekennzeichnet.
z.B. 'a' oder 'f' usw. Daraus folgt das maximal 26 verschiedene Integer Variablen im BASIC-Programm benutzt werden können ('a' bis 'z').

Jede Integer Variable kann einen ganzzahligen Wert von -99999 bis +99999 speichern.
Integer Variablen müssen vor dem benutzen nicht initialisiert werden.

6.6.2 String Variablen :

String Variablen werden mit einem einzelnen Kleinbuchstaben gefolgt von einem '$' gekennzeichnet.
zB. 'a$' oder 'f$' usw. Daraus folgt das maximal 26 verschiedene String Variablen im BASIC-Programm benutzt werden können ('a$' bis 'z$').

Jede String Variable kann einen String mit der maximalen Länge von 85 Zeichen speichern.
String Variablen müssen vor dem benutzen nicht initialisiert werden.

6.7 Arrays :

Das System unterstützt drei Arten von Arrays :

- Integer Arrays max 8192 Integer Werte
- String Arrays max 400 Strings (zu je 85 Zeichen)
- Daten Arrays max 65536 Byte Werte

6.7.1 Integer Arrays :

Integer Arrays werden mit einem einzelnen Großbuchstaben gefolgt von einer Klammer mit der Index-Nummer gekennzeichnet. z.B. 'A(3)' oder 'F(8)' usw. Daraus folgt das maximal 26 verschiedene Integer Arrays im BASIC-Programm benutzt werden können ('A(n)' bis 'Z(n)').

Vor der Benutzung von einem Integer Array muss dieses zunächst 'dimensioniert' werden.
Das bedeutet der Interpreter muss wissen wie viele Elemente das Array groß sein soll.

Das dimensionieren geschieht im Programm durch den Basic-Befehl 'DIM'

z.B. '10 DIM C(10)' ▸ dimensioniert das Integer Array 'C' für 10 integer Elemente

Nach der Dimensionierung kann das Array wie eine 'normale' Integer Variable benutzt werden.
Der Zugriff auf ein Element erfolgt durch die Index-Nummer innerhalb der Klammer.

z.B. um in das erste Element im Array 'C' einen Wert zu speichern :

'20 C(0)=15' ▸ speichert den Wert '10' in das Element '0' vom Array 'C'

das letzte Element von diesem Array wäre in dem Fall das mit der Index-Nummer 9 :

'30 C(9)=45' ▸ speichert den Wert '45' in das Element '9' vom Array 'C'

Ein einmal dimensioniertes Array darf nicht noch ein zweites mal dimensioniert werden.

6.7.2 String Arrays :

String Arrays werden mit einem einzelnen Großbuchstaben und '$' gefolgt von einer Klammer mit der Index-Nummer gekennzeichnet. z.B. 'A$(3)' oder 'F$(8)' usw. Daraus folgt das maximal 26 verschiedene String Arrays im BASIC-Programm benutzt werden können ('A$(n)' bis 'Z$(n)').

Vor der Benutzung von einem String Array muss dieses zunächst 'dimensioniert' werden.
Das bedeutet der Interpreter muss wissen wie viele Elemente das Array groß sein soll.

Das dimensionieren geschieht im Programm durch den Basic-Befehl 'DIM'

z.B. '10 DIM C$(10)' ▸ dimensioniert das String Array 'C$' für 10 String Elemente

Nach der Dimensionierung kann das Array wie eine 'normale' String Variable benutzt werden.
Der Zugriff auf ein Element erfolgt durch die Index-Nummer innerhalb der Klammer.

z.B. um in das erste Element im Array 'C$' einen Text zu speichern :

'20 C$(0)="Hallo"' ▸ speichert "Hallo" in das Element '0' vom Array 'C$'

das letzte Element von diesem Array wäre in dem Fall das mit der Index-Nummer 9 :

'30 C$(9)="Test"' ▸ speichert "Test" in das Element '9' vom Array 'C$'

Ein einmal dimensioniertes Array darf nicht noch ein zweites mal dimensioniert werden.

6.7.3 Daten Arrays :

Daten Arrays werden mit einem einzelnen Großbuchstaben und '#' gefolgt von einer Klammer mit der Index-Nummer gekennzeichnet. z.B. 'A#(3)' oder 'F#(8)' usw. Daraus folgt das maximal 26 verschiedene Daten Arrays im BASIC-Programm benutzt werden können ('A#(n)' bis 'Z#(n)').

Vor der Benutzung von einem Daten Array muss dieses zunächst 'dimensioniert' werden.
Das bedeutet der Interpreter muss wissen wie viele Elemente das Array groß sein soll.

Das dimensionieren geschieht im Programm durch den Basic-Befehl 'DIM'

z.B. '10 DIM C#(10)' ▸ dimensioniert das Daten Array 'C#' für 10 Daten Elemente

Nach der Dimensionierung kann das Array wie eine 'normale' Integer Variable benutzt werden.
Der Zugriff auf ein Element erfolgt durch die Index-Nummer innerhalb der Klammer.

z.B. um in das erste Element im Array 'C#' einen Wert zu speichern :

'20 C#(0)=15' ▸ speichert den Wert '15' in das Element '0' vom Array 'C#'

das letzte Element von diesem Array wäre in dem Fall das mit der Index-Nummer 9 :

'30 C#(9)=45' ▸ speichert den Wert '45' in das Element '9' vom Array 'C#'

Ein einmal dimensioniertes Array darf nicht noch ein zweites mal dimensioniert werden.

Der Unterschied zu Integer Arrays ist, das die einzelnen Elemente nur Werte zwischen '0' und '255' aufnehmen können und das Daten Arrays für das senden/empfangen und Sprites benutzt werden können.

Daten-Arrays können mit Integer und mit Strings beschrieben und gelesen werden.
Bei "PRINT" und "IF" werden Datenarrays immer als Integer gelesen.
(bei Strings muss die Endekennung 0x00 enthalten sein)

Beispiele :

```
10 DIM C#(100)        ▸ dimensioniert das Daten Array 'C#' für 100 Bytes
20 C#(2)=13           ▸ speichert den Wert '13' in das Element '2' vom Array 'C#'
30 PRINT C#(2)        ▸ Anzeige : '13'
40 C#(2)="Hallo"      ▸ speichert den String "Hallo" in Element 2 bis 7
50 a$=C#(2)           ▸ kopiert den String ab Element 2 in die Variable a$
60 PRINT a$           ▸ Anzeige : 'Hallo'
```

6.8 Sprites :

Sprite sind Grafik-Objekte die während dem Programmlauf erzeugt werden. Anders als Texte, Kreise und Linien können Sprites nach dem Zeichnen durch BASIC-Befehle auf dem Bildschirm verschoben werden.

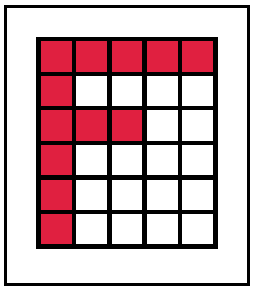
Damit lassen sich bewegte Objekte darstellen. Auch sind Kollisionsabfragen mit dem Bildschirmrand oder anderen Sprites möglich.

Um die Sprites zu unterscheiden, wird jedes mit einer eindeutigen Sprite-ID gekennzeichnet. Beim initialisieren von einem Sprite muss zusätzlich zur Sprite-ID noch eine Daten-ID angegeben werden. In diesem Datenarray stehen dann die eigentlichen Daten vom Sprite also Typ, Größe, Farbe, Pixeldaten.

Sprites sind als Rechteck-Matrix aufgebaut wobei die maximale Größe 16x16 Pixel ist und können je nach Sprite-Typ entweder einfarbig oder bis zu 16 Farben besitzen.

6.8.1 Onecolor Sprites :

Diese Sprites besitzen nur eine Farbe (im RGB565 Farbraum) die Pixel ohne Farbe sind Transparent.



Beispiel :
- Sprite-Typ : 0 (Onecolor, max 8Pixel breit)
- Breite : 5 Pixel
- Höhe : 6 Pixel
- Farbe : 0xF800 = rot

Sprite-Typ : 0	Onecolor, max 8Pixel breit
Sprite-Typ : 1	Onecolor, max 16Pixel breit

Die Pixeldaten für ein OneColor-Sprite beginnen mit der ersten Zeile der Matrix.
Jede Zeile entspricht einem Datenwert und die linke Kante ist das MSB.
Am einfachsten initialisiert man den Typ-0 mit 8bit HEX-Werten und den Typ-1 mit 16bit HEX-Werten.

Pixeldaten vom Beispiel (Typ-0) :

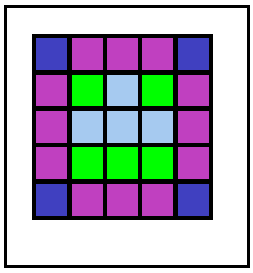
```
DATA 0xF8,0x80,0xC0,0x80,0x80,0x80
```

Pixeldaten von einem Typ-1 Sprite mit dem gleichen aussehen wie im Beispiel :

```
DATAW 0xF800,0x8000,0xC000,0x8000,0x8000,0x8000
```

6.8.2 Multicolor Sprites :

Jedes Pixel dieser Sprites kann eine der 16 Farben vom Editor-Mode besitzen. Eine Farbe kann als Transparenz-Wert bestimmt werden.



Beispiel :
- Sprite-Typ : 2 (Multicolor, max 8Pixel breit)
- Breite : 5 Pixel
- Höhe : 5 Pixel
- Transparenzfarbe : 6 = blau

Sprite-Typ : 2	Multicolor , max 8Pixel breit
Sprite-Typ : 3	Multicolor , max 16Pixel breit

Die Pixeldaten für ein Multicolor-Sprite beginnen mit dem linken Pixel der ersten Zeile der Matrix.
Jedes Pixel besteht aus 4bit Datenwerten und es werden immer Daten für 4Pixel auf einmal geladen.
Aus dem Grund ist es am einfachsten die Daten als 16bit HEX-Werte zu initialisieren.
Jedes Pixel entspricht dann einem Zeichen von '0' bis 'F'.
Es müssen 'Füllnibble' eingefügt werden um die Datenanzahl auf 4,8,12,16 Pixel aufzurunden.
Die Farbwerte kann man mit der Farbtabelle vom Editor-Mode bestimmen.

Pixeldaten vom Beispiel :

Das Sprite ist 5 Pixel breit, zum Aufrunden auf 8 müssen also 3 Füllnibble eingefügt werden.

DATAW 0x6444,0x6000	Daten der ersten Zeile (5 Farbwerte + 3 Füllnibble)
DATAW 0x45E5,0x4000	Daten der zweiten Zeile (5 Farbwerte + 3 Füllnibble)
DATAW 0x4EEE,0x4000	Daten der dritten Zeile (5 Farbwerte + 3 Füllnibble)
DATAW 0x4555,0x4000	Daten der vierten Zeile (5 Farbwerte + 3 Füllnibble)
DATAW 0x6444,0x6000	Daten der fünften Zeile (5 Farbwerte + 3 Füllnibble)

'6'=blau, '4'=violett, '5'=grün, 'E'=hellblau, '0'=Füllnibble

6.9 Operatoren :

Innerhalb vom Basic-Programm können verschiedene Operatoren zur Berechnung benutzt werden.

'+'	Addition	▸ 28 + 85 = 113
'-'	Subtraktion	▸ 28 - 85 = -57
'*'	Multiplikation	▸ 28 * 85 = 2380
'/'	Division	▸ 85 / 28 = 3
'%'	Modulo	▸ 85 % 28 = 1
'&'	logisches UND	▸ 28 & 85 = 20
'|'	logisches ODER	▸ 28 | 85 = 93
'^'	logisches XOR	▸ 28 ^ 85 = 73

6.10 Syntax Hinweise :

Zahlen :

<int_var>	▸ steht für eine beliebige integer Variable : 'a' bis 'z'
<int_array>	▸ steht für ein beliebiges integer Array : 'A(n)' bis 'Z(n)'
<data_array>	▸ steht für ein beliebiges Daten Array : 'A#(n)' bis 'Z#(n)'
<uint>	▸ steht für einen unsigned Zahlenwert : '0' bis '99999'
<int>	▸ steht für einen signed Zahlenwert : '-99999' bis '99999'
<int_cmd>	▸ steht für ein Basic-Befehl der einen Integer Wert zurückliefert z.B. 'ABS(n)'

Texte :

<str_var>	▸ steht für eine beliebige String Variable : 'a$' bis 'z$'
<str_array>	▸ steht für ein beliebiges String Array : 'A$(n)' bis 'Z$(n)'
<str>	▸ steht für einen String : z.B. "Hallo"
<str_cmd>	▸ steht für ein Basic-Befehl der einen String zurückliefert z.B. 'LEFT$(n)'

Sonst :

<tr>	▸ steht für ein Trennungszeichen : ';' oder ','
<rel>	▸ steht für ein Relationszeichen : '=' oder '<' oder '>'
<cmd>	▸ steht für einen beliebigen BASIC-Befehl z.B. : 'PRINT'
<int_exp>	▸ steht für einen beliebigen integer Ausdruck
<str_exp>	▸ steht für einen beliebigen string Ausdruck
<io_exp>	▸ steht für einen Ausdruck vom Typ : <int_exp>,<str_exp>
'[...]'	▸ symbolisiert einen optionalen Teil

7 BASIC-Befehle :

7.1 Standard-Befehle :

7.1.1 REM :

Syntax : REM<anything>
Return : <void>

dient zum hinzufügen von einer Kommentarzeile in das Programm.
Die Zeile wird vom Interpreter komplett ignoriert.

z.B. um den Namen vom Basic-Programm zu notieren :

```
10 REM Programm : PONG          ▸ Interpreter überspringt die Zeile
```

7.1.2 LET :

Syntax : [LET]<int_var>=<int_exp> oder [LET]<int_array>=<int_exp>
oder : [LET]<str_var>=<str_exp> oder [LET]<str_array>=<str_exp>
oder : [LET]<data_array>=<int_exp> oder [LET]<data_array>=<str_exp>
Return : <void>

Wertzuweisung einer Variable oder einem Array.

z.B. um der Variable 'a' den Wert '15' zuzuweisen :

```
10 LET a=15                     ▸ 'a' = 15
```

Hinweis : Der Befehl "LET" ist optional, es funktioniert auch :

```
10 a=15                         ▸ 'a' = 15
```

weitere Beispiele :

```
 2 DIM C(10)
 5 DIM C$(10)
 7 DIM C#(10)
10 b=15+3                       ▸ 'b' = 18
15 c=b+4                        ▸ 'c' = 22
20 d=ABS(-3)                    ▸ 'd' = 3
25 C(3)=-5                      ▸ 'C(3)' = -5
30 C(4)=b+4                     ▸ 'C(4)' = 22
35 C(5)=ABS(-3)                 ▸ 'C(5)' = 3
40 a$="Hallo"                   ▸ 'a$' = "Hallo"
45 b$=LEFT$("Hallo",3)          ▸ 'b$' = "Hal"
50 C$(3)="Hallo"                ▸ 'C$(3)' = "Hallo"
55 C$(4)=LEFT$("Hallo",3)       ▸ 'C$(4)' = "Hal"
60 C$(5)=a$                     ▸ 'C$(5)' = "Hallo"
70 C#(4)=b+4                    ▸ 'C#(4)' = 22
75 C#(5)=ABS(-3)                ▸ 'C#(5)' = 3
80 C#(0)="Hi"                   ▸ 'C#(0)' = 'H' , 'C#(1)' = 'i' , 'C#(2)' = 0x00
```

7.1.3 PRINT :

Syntax : PRINT<io_exp>[<tr><io_exp>...]
Return : <void>

Zum Anzeigen von Texten, Zahlen, Variablen und Basic-Befehlen mit Rückgabewert

z.B. um den Text "Hallo" auszugeben :

```
10 PRINT "Hallo"
```
▸ Anzeige : "Hallo"

oder um den Inhalt der Variable 'a' anzuzeigen :

```
10 a=15
20 PRINT a
```
▸ Anzeige : "15"

oder um den Rückgabewert von einem Befehl anzuzeigen :

```
10 PRINT LEFT$("Hallo",3)
```
▸ Anzeige : "Hal"

ein Semikolon verhindert einen Zeilenumbruch :

```
10 a=15
20 PRINT "wert=";a
```
▸ Anzeige : "wert=15"

ein Komma fügt einen Tabulatorsprung ein :

```
10 PRINT "länge","höhe","breite"
```
▸ Anzeige : "länge höhe breite"

Hinweis : ein Tabulator entspricht 80 Pixel

7.1.4 IF/THEN/ELSE :

Syntax : IF<int_exp><rel><int_exp>[THEN]<cmd>[ELSE<cmd>]
oder : IF<str_exp><rel><str_exp>[THEN]<cmd>[ELSE<cmd>]
Return : <void>

Prüft eine Bedienung auf "=" , "<", ">"

z.B. um zwei Variablen zu vergleichen :

```
10 IF a>b THEN PRINT "a ist größer als b"
```

Hinweis : "THEN" ist optional, es funktioniert auch :

```
10 IF a>b PRINT "a ist größer als b"
```

Hinweis : "ELSE" ist optional und wird ausgeführt falls die Bedingung nicht zutrifft :

```
10 IF a>b PRINT "a ist größer als b" ELSE PRINT "a ist nicht größer als b"
```

Hinweis : Bei Stringvergleichen wird Buchstabe für Buchstabe verglichen wobei 'A'<'B'

7.1.5 FOR/TO/STEP/NEXT :

Syntax : FOR<int_var>=<int_exp>TO<int_exp>[STEP<int_exp>]
Return : <void>

Erzeugt eine Schleife mit einer Zähl-Variable

z.B. um die Variable 'n' von 5 auf 10 zu zählen :

```
10 FOR n=5 TO 10
20 PRINT n                    ▸ Anzeige : "5","6","7","8","9","10"
30 NEXT n
```

es kann auch rückwärts gezählt werden :

```
10 FOR n=10 TO 5
20 PRINT n                    ▸ Anzeige : "10","9","8","7","6","5"
30 NEXT n
```

mit "STEP" kann eine Schrittweite >1 angegeben werden :

```
10 FOR n=5 TO 10 STEP 2
20 PRINT n                    ▸ Anzeige : "5","7","9"
30 NEXT n
```

7.1.6 GOTO :

Snytax : GOTO<uint>
Return : <void>

Springt zu einer anderen Basic-Zeile
Hinweis : Zeilennummer muss existieren

z.B. um zur Zeile 100 zu springen :

```
10 GOTO 100                   ▸ springt auf Zeile 100
```

7.1.7 GOSUB/RETURN :

Syntax : GOSUB<uint>
Return : <void>

Springt zu einer anderen Basic-Zeile und kehrt beim Befehl "RETURN" wieder zurück
Hinweis : Zeilennummer muss existieren

```
10 GOSUB 100                  ▸ springt auf Zeile 100
20 PRINT "Zeile 20"
30 END                        ▸ Ende vom BASIC-Programm
100 PRINT "Zeile 100"
110 RETURN                    ▸ springt auf Zeile 20 zurück
```

7.1.8 END :

Syntax : END
Return : <void>

Beendet das BASIC-Programm.

Beispiel :

```
10 END
```

7.2 Array-Befehle :

7.2.1 DIM :

Syntax : DIM<int_array>
oder : DIM<str_array>
oder : DIM<data_array>
Return : <void>

Reserviert Speicherplatz für ein eindimensionales Integer- String- oder Daten-Array.

Beispiele :

```
10 DIM A(100)
20 DIM A$(100)
30 DIM A#(100)
```

- reserviert 100 Integer Werte für Array 'A'
- reserviert 100 String Werte für Array 'A$'
- reserviert 100 Byte Werte für Array 'A#'

Hinweis : im System sind 128kByte RAM für die Arrays reserviert. Aufgeteilt auf die 3 Array-Typen. Wenn der User versucht mehr Speicher zu dimensionieren als möglich ist, erfolgt eine Fehlermeldung.

Mit dem Editor-Befehl 'FREE' kann der aktuelle Speicherbedarf angezeigt werden.

7.3 System-Befehle :

7.3.1 CMD :

Syntax : CMD(<int_exp>)
Return : <void>

Der 'CMD' Befehl ändert die Standard Ausgabe für den PRINT-Befehl
entweder auf LCD oder auf COM-1
Aufruf : gültiger Wertebereich : [0 bis 1]

Beispiele :

```
10 CMD(0)
20 PRINT "Hallo"          ▸ Anzeige : "Hallo" auf dem Display
30 CMD(1)
40 PRINT "Hallo"          ▸ sendet per COM-1 : "Hallo"
```

7.3.2 RND : (INTEGER FUNKTION)

Syntax : RND(<int_exp>)
Return : <int>

Liefert einen Zufallswert innerhalb von einem Maximalwert zurück.
Aufruf : gültiger Wertebereich : [1 bis n]

z.B. um einen Zufallswert von 0 bis 99 in die Variable 'a' zu speichern :

```
10 a=RND(100)             ▸ 'a' = Zufallszahl von 0 bis 99
```

7.3.3 RANDOMIZE :

Syntax : RANDOMIZE(<int_exp>)
Return : <void>

Diese Funktion initialisiert den Zufallszahlengenerator mit einem "Seed"-Wert
Aufruf : gültiger Wertebereich : [0 bis n]

Parameter = 0 ▸ als "Seed" wird der aktuelle TIC-Wert benutzt
Parameter >0 ▸ der Parameterwert wird als "Seed" benutzt

Hinweis : Mit dem gleichen "Seed" erhält man immer die gleiche Abfolge von Zufallszahlen.

```
10 RANDOMIZE(5)           ▸ 'Seed' = 5
20 a=RND(100)             ▸ 'a' = 22
```

Hinweis : mit dem TIC-Wert als "Seed" können zufällige Abfolgen hergestellt werden.

```
10 RANDOMIZE(5)           ▸ 'Seed' = 5
20 a=RND(100)             ▸ 'a' = 22
30 RANDOMIZE(0)           ▸ 'Seed' = TIC
40 a=RND(100)             ▸ 'a' = Zufallszahl von 0 bis 99
```

7.3.4 PAUSE :

Syntax : PAUSE(<int_exp>)
Return : <void>

Der Befehl fügt eine Pause von "x" ms ein.
Aufruf : gültiger Wertebereich : [0 bis n]

z.B. um eine Pause von 100 ms einzulegen :

```
10 PAUSE(100)
```
▸ macht eine Pause von 100ms

7.3.5 CLRTIC :

Syntax : CLRTIC
Return : <void>

setzt den Basic-Timer auf den Wert 0 zurück
(der Basic-Timer wird jede ms um 1 inkrementiert)

Beispiel :

```
10 CLRTIC
```
▸ Timerwert = 0

7.3.6 GETTIC : (INTEGER FUNKTION)

Syntax : GETTIC()
Return : <int>

liefert den aktuellen Inhalt vom Basic-Timer.
Also die Zeit in ms die seit dem letzten 'CLRTIC' vergangen sind.

Beispiel :

```
10 a=GETTIC()
```
▸ 'a' = Timerwert [in ms]

7.4 Mathe-Befehle :

7.4.1 SIN : (INTEGER FUNKTION)

Syntax : SIN(<int_exp>[,<int_exp>])
Return : <int>

berechnet den Sinus eines Winkels (multipliziert mit 10000)
Aufruf : gültiger Wertebereich vom Winkel : [0 bis 359]

z.B. um den Sinus von 30° zu berechnen :

```
10 a=SIN(30)                    ▸ 'a' = 5000
```

Mit dem zweiten Parameter kann die Hypotenuse übergeben werden dadurch wird direkt die Gegenkathete berechnet

z.B. um die GK von 37° und H=400 zu berechnen :

```
10 a=SIN(37,400)                ▸ 'a' = 240
```

Hinweis : Winkel-Werte <0 oder >359 werden auf 0-359 umgerechnet.

7.4.2 COS : (INTEGER FUNKTION)

Syntax : COS(<int_exp>[,<int_exp>])
Return : <int>

berechnet den Kosinus eines Winkels (multipliziert mit 10000)
Aufruf : gültiger Wertebereich vom Winkel : [0 bis 359]

z.B. um den Kosinus von 30° zu berechnen :

```
10 a=COS(30)                    ▸ 'a' = 8660
```

Mit dem zweiten Parameter kann die Hypotenuse übergeben werden dadurch wird direkt die Ankathete berechnet

z.B. um die AK von 37° und H=400 zu berechnen :

```
10 a=COS(37,400)                ▸ 'a' = 319
```

Hinweis : Winkel-Werte <0 oder >359 werden auf 0-359 umgerechnet.

7.4.3 ABS : (INTEGER FUNKTION)

Syntax : ABS(<int_exp>)
Return : <int>

ermittelt den Betragswert eines Ausdrucks

z.B. um den Betragswert von -13 zu berechnen :

```
10 a=ABS(-13)                   ▸ 'a' = 13
```

7.4.4 SQR : (INTEGER FUNKTION)

Syntax : SQR(<int_exp>)
Return : <int>

berechnet die Quadratwurzel einer Zahl
Aufruf : gültiger Wertebereich : [0 bis n]

z.B. um die Wurzel von 16 zu berechnen :

```
10 a=SQR(16)
```

▸ 'a' = 4

7.5 STRING-Befehle :

7.5.1 LEN : (INTEGER FUNKTION)

Syntax : LEN(<str_exp>)
Return : <int>

ermittelt die Anzahl der Zeichen eines Strings

Beispiel :

```
10 a$=("Hallo")          ▸ 'a$' = "Hallo"
20 a=LEN(a$)             ▸ 'a' = 5
```

Hinweis : bei einem leeren String wird '0' als Länge zurückgeliefert

7.5.2 ASC : (INTEGER FUNKTION)

Syntax : ASC(<str_exp>)
Return : <int>

ermittelt den Ascii-Wert des ersten Zeichens eines Strings

Beispiel :

```
10 a$=("Hallo")          ▸ 'a$' = "Hallo"
20 a=ASC(a$)             ▸ 'a' = 72
```

Hinweis : falls der String leer ist, wird eine '0' zurückgeliefert

7.5.3 CHR$: (STRING FUNKTION)

Syntax : CHR$(<int_exp>)
Return : <str>

wandelt einen Ascii-Wert in einen String um (der String enthält nur ein Zeichen)
Aufruf : gültiger Wertebereich : [0 bis 255]

Beispiel :

```
10 a$=CHR$(72)           ▸ 'a$' = "H"
```

7.5.4 VAL : (INTEGER FUNKTION)

Syntax : VAL(<str_exp>)
Return : <int>

wandelt einen String (in dezimaler Schreibweise) in eine Zahl um

Beispiel :

```
10 a=VAL("72")
```
▸ 'a' = '72'

Hinweis : falls der String keine gültige Zahl enthält, wird '0' zurückgeliefert.

7.5.5 STR$: (STRING FUNKTION)

Syntax : STR$(<int_exp>)
Return : <str>

wandelt eine Zahl in einen String um (in dezimaler Schreibweise)

Beispiel :

```
10 a$=STR$(72)
```
▸ 'a$' = "72"

7.5.6 HEX$: (STRING FUNKTION)

Syntax : HEX$(<int_exp>[,<int_exp>])
Return : <str>

wandelt eine Zahl in einen String um (in hexadezimaler Schreibweise)
Aufruf : gültiger Wertebereich : [0 bis n]

Beispiel :

```
10 a$=HEX$(123)
```
▸ 'a$' = "7B"

Mit dem zweiten Parameter kann die Anzahl der Ziffern mit angegeben werden.

Beispiel :

```
10 a$=HEX$(123,4)
```
▸ 'a$' = "007B"

7.5.7 LEFT$: (STRING FUNKTION)

Syntax : LEFT$(<str_exp>,<int_exp>)
Return : <str>

kopiert eine bestimmte Anzahl an Zeichen von einem String von links beginnend

Beispiel :

```
10 a$=LEFT$("STM32F429",5)          'a$' = "STM32"
```

7.5.8 RIGHT$: (STRING FUNKTION)

Syntax : RIGHT$(<str_exp>,<int_exp>)
Return : <str>

kopiert eine bestimmte Anzahl an Zeichen von einem String von rechts beginnend

Beispiel :

```
10 a$=LEFT$("STM32F429",5)          'a$' = "2F429"
```

7.5.9 MID$: (STRING FUNKTION)

Syntax : MID$(<str_exp>,<int_exp>,<int_exp>)
Return : <str>

kopiert eine bestimmte Anzahl an Zeichen von einem String ab einer Position

Beispiel :

```
10 a$=MID$("STM32F429",2,4)         'a$' = "M32F"
```

Hinweis : es werden u.U. Weniger Zeichen kopiert, wenn der Quell-String kleiner ist als die Anzahl
Hinweis : falls der Quell-String kleiner ist als die Startposition, wird ein leerer String zurückgegeben

7.6 LCD-Befehle :

7.6.1 CLS :

Syntax : CLS
Return : <void>

löscht den Bildschirm mit der aktuellen Hintergrundfarbe

Beispiel :

```
10 CLS
```
▸ Bildschirm wird gelöscht

7.6.2 CURSOR :

Syntax : CURSOR(<int_exp>,<int_exp>)
Return : <void>

setzt den Cursor für den nächsten PRINT-Befehl auf die x,y Koordinate
Aufruf : gültiger Wertebereich für x : [0 bis 319] für y : [0 bis 239]

Beispiel um den Cursor auf 20,53 zu setzen :

```
10 CURSOR(20,53)
20 PRINT "Hallo"
```
▸ Anzeige : "Hallo" an Position 20,53

7.6.3 INK :

Syntax : INK(<int_exp>)
Return : <void>

stellt die Vordergrundfarbe auf einen Wert ein
Aufruf : gültiger Wertebereich : [0 bis 65535] als RGB565

z.B. um die Vordergrundfarbe auf 'rot' zu setzen :

```
10 INK(0xF800)
20 PRINT "Hallo"
```
▸ Anzeige : "Hallo" mit Schriftfarbe 'rot'

7.6.4 PAPER :

Syntax : PAPER(<int_exp>)
Return : <void>

stellt die Hintergrundfarbe auf einen Wert ein
Aufruf : gültiger Wertebereich : [0 bis 65535] als RGB565

z.B. um die Hintergrundfarbe auf 'rot' zu setzen :

```
10 PAPER(0xF800)
20 PRINT "Hallo"
```
▸ Anzeige : "Hallo" mit Hintergrundfarbe 'rot'

7.6.5 SETPX :

Syntax : SETPX(<int_exp>,<int_exp>[,<int_exp>])
Return : <void>

setzt einen Pixel auf dem LCD an x,y Koordinate
Aufruf : gültiger Wertebereich für x : [0 bis 319] für y : [0 bis 239]

z.B. um den Pixel an 10,39 zu setzen :

10 SETPX(10,39) ▸ Anzeige : Pixel an Position 10,39

Mit dem dritten Parameter kann die Farbe übergeben werden
Aufruf : gültiger Wertebereich : [0 bis 65535] als RGB565

z.B. um den Pixel an 10,39 in 'rot' zu setzen

10 SETPX(10,39,0xF800) ▸ Anzeige : 'rotes' Pixel an Position 10,39

7.6.6 GETPX : (INTEGER FUNKTION)

Syntax : GETPX(<int_exp>,<int_exp>)
Return : <int>

liefert die Farbe von einem Pixel auf dem LCD
Aufruf : gültiger Wertebereich für x : [0 bis 319] für y : [0 bis 239]

z.B. um die Farbe vom Pixel an 10,39 auszulesen :

10 a=GETPX(10,39) ▸ 'a' = Farbwert von Pixel an 10,39

Rückgabe : Wertebereich : [0 bis 65535] als RGB565

7.6.7 LINE :

Syntax : LINE(<int_exp>,<int_exp>,<int_exp>,<int_exp>[,<int_exp>])
Return : <void>

zeichnet eine Linie von x1,y1 nach x2,y2 auf dem LCD
Aufruf : gültiger Wertebereich für x : [0 bis 319] für y : [0 bis 239]

z.B. um eine Linie von 10,20 nach 30,40 zu zeichnen :

10 LINE(10,20,30,40) ▸ Anzeige : Linie von 10,20 nach 30,40

Mit dem fünften Parameter kann die Farbe übergeben werden
Aufruf : gültiger Wertebereich : [0 bis 65535] als RGB565

z.B. um eine 'rote' Linie von 10,20 nach 30,40 zu zeichnen :

10 LINE(10,20,30,40,0xF800) ▸ Anzeige : 'rote' Linie von 10,20 nach 30,40

7.6.8 RECT :

Syntax : RECT(<int_exp>,<int_exp>,<int_exp>,<int_exp>[,<int_exp>,<int_exp>])
Return : <void>

zeichnet ein Rechteck von x1,y1 nach x2,y2 auf dem LCD
Aufruf : gültiger Wertebereich für x : [0 bis 319] für y : [0 bis 239]

z.B. um ein Rechteck von 10,20 nach 30,40 zu zeichnen :

```
10 RECT(10,20,30,40)
```
▸ Anzeige : Rechteck von 10,20 nach 30,40

Mit dem fünften Parameter kann ein gefülltes Rechteck gezeichnet werden
Aufruf : gültiger Wertebereich : [0 bis 1]

z.B. um ein gefülltes Rechteck von 10,20 nach 30,40 zu zeichnen :

```
10 RECT(10,20,30,40,1)
```
▸ Anzeige : gefülltes Rechteck

Mit dem sechsten Parameter kann die Farbe übergeben werden
Aufruf : gültiger Wertebereich : [0 bis 65535] als RGB565

z.B. um ein gefülltes 'rotes' Rechteck von 10,20 nach 30,40 zu zeichnen :

```
10 RECT(10,20,30,40,1,0xF800)
```
▸ Anzeige : gefülltes 'rotes' Rechteck

7.6.9 CIRCLE :

Syntax : CIRCLE(<int_exp>,<int_exp>,<int_exp>[,<int_exp>,<int_exp>])
Return : <void>

zeichnet einen Kreis an x,y mit dem Radius r auf dem LCD
Aufruf : gültiger Wertebereich für x : [0 bis 319] für y : [0 bis 239]

z.B. um einen Kreis an 10,20 mit Radius 8 zu zeichnen :

```
10 CIRCLE(10,20,8)
```
▸ Anzeige : Kreis an 10,20 mit Radius 8

Mit dem vierten Parameter kann ein gefüllter Kreis gezeichnet werden
Aufruf : gültiger Wertebereich : [0 bis 1]

z.B. um einen gefüllten Kreis an 10,20 mit Radius 8 zu zeichnen :

```
10 CIRCLE(10,20,8,1)
```
▸ Anzeige : gefüllter Kreis

Mit dem fünften Parameter kann die Farbe übergeben werden
Aufruf : gültiger Wertebereich : [0 bis 65535] als RGB565

z.B. um einen gefüllten 'roten' Kreis an 10,20 mit Radius 8 zu zeichnen :

```
10 CIRCLE(10,20,8,1,0xF800)
```
▸ Anzeige : gefüllter 'roter' Kreis

7.7 Tastatur-Befehle :

7.7.1 GETKEY :

Syntax : GETKEY <int_var>[,<int_var>,<int_var>]
oder : GETKEY <str_var>[,<str_var>,<int_var>]
Return : <void>

ließt die Tastatur aus und speichert den Tastencode oder das Ascii-Zeichen in Variablen
Rückgabe-Tastencode <int>: Wertebereich : [1 bis 128]
Rückgabe-Asciizeichen <str> : Wertebereich : [32 bis 126]
Hinweis : falls keine Taste gedrückt : Wert = 0 bzw leerer String

Beispiel um nur die erste gedrückte Taste auszulesen :

```
10 GETKEY a
```
- 'a' = Tastencode von Taste-1

Die zweite Variable enthält den Tastencode der zweiten gedrückten Taste :

```
10 GETKEY a,b
```
- 'a' = Tastencode von Taste-1
- 'b' = Tastencode von Taste-2

Die dritte Variable enthält die Statusbits der Shift-Tasten :

```
10 GETKEY a,b,c
```
- 'a' = Tastencode von Taste-1
- 'b' = Tastencode von Taste-2
- 'c' = Shift-Status-Bits :

Shift-Statusbits :

Bit0 = left SHIFT	[0=unbetätigt, 1=betätigt]
Bit1 = right SHIFT	[0=unbetätigt, 1=betätigt]
Bit2 = left STRG	[0=unbetätigt, 1=betätigt]
Bit3 = ALT	[0=unbetätigt, 1=betätigt]
Bit4 = ALT-GR	[0=unbetätigt, 1=betätigt]

Die gedrückte Taste kann auch als String ausgelesen werden :

Beispiel um nur die erste gedrückte Taste auszulesen :

```
10 GETKEY a$
```
- 'a$' = Ascii-Zeichen von Taste-1

Die zweite Variable enthält das Ascii-Zeichen der zweiten gedrückten Taste :

```
10 GETKEY a$,b$
```
- 'a$' = Ascii-Zeichen von Taste-1
- 'b$' = Ascii-Zeichen von Taste-2

Die beiden Mode können auch gemischt werden :

```
10 GETKEY a$,b,c
```
- 'a$' = Ascii-Zeichen von Taste-1
- 'b' = Tastencode von Taste-2
- 'c' = Shift-Status-Bits :

Hinweis : falls die gedrückte Taste kein Ascii-Zeichen ist, wird ein leerer String zurückgeliefert.

7.7.2 INPUT :

Syntax : INPUT [<str>,]<int_var>[,<tr>]
oder : INPUT [<str>,]<str_var>[,<tr>]
Return : <void>

ließt einen String von der Tastatur ein (bis zur RETURN-Taste) und speichert ihn entweder als Integer- oder als String-Variable.

Hinweis : Als erster Parameter kann ein "Anzeigetext" übergeben werden, falls dieser nicht angegeben wird, wird ein '?' ausgegeben zur Info das auf eine Eingabe gewartet wird.
Hinweis : per Trennungszeichen kann der LineFeed unterdrückt oder ein TAB eingefügt werden

Beispiel : um den Namen vom User abzufragen :

```
10 INPUT "input name :",a$      ▸ Ausgabe von einem Text und warten auf einem String
20 PRINT "Hallo : ";a$          ▸ Ausgabe vom String
```

Beispiel : um das Alter vom User abzufragen :

```
10 INPUT "input age :",a        ▸ Ausgabe von einem Text und warten auf eine Zahl
20 PRINT "you are : ";a         ▸ Ausgabe der Zahl
```

Hinweis : falls der eingegebene Text keine Zahl war, wird eine 0 übergeben.

7.8 Touch-Befehle :

7.8.1 GETTOUCH :

Syntax : GETTOUCH <int_var>,<int_var>
Return : <void>

ließt die aktuelle Touchposition [x,y] ein und speichert sie in Variablen.
Rückgabe : Wertebereich für x : [-1 bis 319] für y : [-1 bis 239]

Beispiel :

```
10 GETTOUCH x,y
```

- 'x' = Touchposition in X-Richtung
- 'y' = Touchposition in Y-Richtung

Hinweis : Wenn der Touch nicht betätigt ist, wird '-1' als Position übergeben

7.8.2 CHKTOUCH : (INTEGER FUNKTION)

Syntax : CHKTOUCH(<int_exp>,<int_exp>,<int_exp>,<int_exp>)
Return : <int>

test ob der Touch innerhalb eines Rechtecks (von x1,y1 nach x2,y2) betätigt ist
Aufruf : gültiger Wertebereich für x : [0 bis 319] für y : [0 bis 239]

z.B. um zu prüfen ob der Touch innerhalb von 10,20 nach 30,40 betätigt ist :

```
10 a=CHKTOUCH(10,20,30,40)
```

- 'a' = Statuswert von Rechteck 10,20 nach 30,40

Rückgabe : Wertebereich : [0 bis 1] 0 = nicht betätigt
1 = betätigt

7.9 GPIO-Befehle

7.9.1 INITPINT :

Syntax : INITPIN(<int_exp>,<int_exp>[,<int_exp>])
Return : <void>

Initialisiert einen GPIO Pin der CPU [pin, typ, mode]
Aufruf : gültiger Wertebereich für pin : [0x00 bis 0x6F]

pin :	0x00 = PA0	bis	0x0F = PA15
	0x10 = PB0	bis	0x1F = PB15
	0x20 = PC0	bis	0x2F = PC15
	0x30 = PD0	bis	0x3F = PD15
	0x40 = PE0	bis	0x4F = PE15
	0x50 = PF0	bis	0x5F = PF15
	0x60 = PG0	bis	0x6F = PG15

Aufruf : gueltiger Wertebereich für typ : [0 bis 2]

typ :	0 = Digital Eingang	mode : 0 = NoPull 1 = PullUp 2 = PullDown
typ :	1 = Digital Ausgang	mode : 0 = Lo_Pegel 1 = Hi_Pegel
typ :	2 = Analog Eingang	mode : nc

z.B. um den Pin PA0 auf Eingang mit PullDown zu schalten :

```
10 INITPIN(0x00,0,2)
```
▸ PA0 = Digital Eingang mit PullDown

z.B. um den Pin PG13 auf Ausgang zu schalten :

```
10 INITPIN(0x6D,1)
```
▸ PG13 = Digital Ausgang

z.B. um den Pin PA5 auf Analog-Eingang zu schalten :

```
10 INITPIN(0x05,2)
```
▸ PA5 = Analog Eingang

7.9.2 IN : (INTEGER FUNKTION)

Syntax : IN(<int_exp>)
Return : <int>

Ließt den Pegel von einem Eingangspin ein
Aufruf : gültige Werte für pin : PA0,5,7 / PB2,4,7 / PC3,11,13,14,15
PD4,5,7 / PE2,3,4,5,6 / PF6 / PG2,3,8,13,14

z.B. um den Pegel von PA0 einzulesen :

```
10 INITPIN(0x00,0,2)
20 a=IN(0x00)
```
▸ PA0 = Digital Eingang
▸ 'a' = Pegel von PA0

Rückgabe : Wertebereich : [0 bis 1]

7.9.3 OUT :

Syntax : OUT(<int_exp>,<int_exp>)
Return : <void>

Setzt den Pegel von einem Ausgangspin
Aufruf : gültige Werte für pin : PA0,5,7 / PB2,4,7 / PC3,11,13,14,15
PD4,5,7 / PE2,3,4,5,6 / PF6 / PG2,3,8,13,14

Aufruf : gültige Werte für pegel [0 bis 2] 0 = Lo_Pegel
1 = Hi_Pegel
2 = Pegel toggeln

z.B. um den Pegel von PG13 auf Hi zu setzen :

```
10 INITPIN(0x6D,1)
20 OUT(0x6D,1)
```

- PG13 = Digital Ausgang
- PG13 = HI_Pegel

7.9.4 ADC : (INTEGER FUNKTION)

Syntax : ADC(<int_exp>[,<int_exp>])
Return : <int>

Ließt den ADC-Wert von einem Analogpin.
Aufruf : gültige Werte für pin : PA5 / PA7 / PC3

z.B. um den ADC Wert von PA5 einzulesen :

```
10 INITPIN(0x05,2)
20 a=ADC(0x05)
```

- PA5 = Analog Eingang
- 'a' = ADC-Wert von PA5

Rückgabe : Wertebereich : [0 bis 4095]

Mit dem zweiten Parameter kann die Mittelwertbildung eingestellt werden
Aufruf : gültiger Wertebereich : [n=0 bis 7] : Messwerte = 2^n

z.B. um den Mittelwert von 32 Messungen an PA5 einzulesen :

```
10 INITPIN(0x05,2)
20 a=ADC(0x05,5)
```

- PA5 = Analog Eingang
- 'a' = ADC-Wert von PA5

Rückgabe : Wertebereich : [0 bis 4095]

7.10 DATA-Befehle :

7.10.1 DATA/DATAW :

Syntax : DATA <int_exp>[,<int_exp>...]
oder : DATAW<int_exp>[,<int_exp>...]
Return : <void>

Füllt direkt nach dem dimensionieren von einem Datenarray das Array mit Werten ab Adresse 0

z.B. um die ersten 8 Bytes vom Array 'C#' zu beschreiben :

```
10 DIM C#(100)          ▸ reserviert 100 Bytes für Data-Array 'C#'
20 DATA 10,20,30,40     ▸ Adr 0=10, Adr 1=20, Adr 2=30, Adr 3=40
30 DATA 50,60,70,80     ▸ Adr 4=50, Adr 5=60, Adr 6=70, Adr 7=80
```

Der Befehl 'DATAW' funktioniert genauso, nur werden 16bit Werte geschrieben

z.B. um die ersten 8 Bytes vom Array zu beschreiben :

```
10 DIM C#(100)          ▸ reserviert 100 Bytes für Data-Array 'C#'
20 DATAW 10,20          ▸ Adr 0=00, Adr 1=10, Adr 2=00, Adr 3=20
30 DATAW 0x1234,1234    ▸ Adr 4=18, Adr 5=52, Adr 6=04, Adr 7=210
```

7.11 PORT-Befehle :

7.11.1 INITPORT :

Syntax : INITPORT(<int_exp>[,<int_exp>,<int_exp>])
Return : <void>

Initialisiert einen PORT der CPU [port-id, v1, v2]
Aufruf : gültiger Wertebereich für port-id : [0 bis 4]

port-id :
- 0 = COM-1* — TX = PA9 / RX = PA10 / 15200 Baud / 8N1
  - v1 = nc
  - v2 = nc
- 1 = COM-6 — TX = PG14 / RX = PG9 / 8N1
  - v1 = Baudrate [0 bis 7]: 0=2400 / 1=4800 / 2=9600 / 3=14400 / 4=19200 / 5=38400 / 6=57600 / 7=115200
  - v2 = nc
- 2 = I2C-3* — SCL=PA8 / SDA=PC9
  - v1 = nc
  - v2 = nc
- 3 = SPI-5* — SCK=PF7 / MOSI=PF9 / MISO=PF8 / MODE-0_MSB / 5 MHz
  - v1 = nc
  - v2 = nc
- 4 = SPI-4 — SCK=PE2 / MOSI=PE6 / MISO=PE5
  - v1 = Mode [0 bis 7]: 0=Mode-0_MSB / 1=Mode-1_MSB / 2=Mode-2_MSB / 3=Mode3_MSB / 4=Mode-0_LSB / 5=Mode-1_LSB / 6=Mode-2_LSB / 7=Mode-3_LSB
  - v2 = Frq [0 bis 7]: 0=330 kHz / 1=660 kHz / 2=1,3 MHz / 3=2,6 MHz / 4=5,2 MHz / 5=10,5 MHz / 6=21 MHz / 7=42 MHz

Hinweis* : COM-1 / I2C-3 / SPI-5 müssen nicht initialisiert werden.

z.B. um COM-6 mit 19200 Baud zu initialisieren :

```
10 INITPORT(1,4)          ▸ init von Port-ID Nr. 1 (COM-6) mit 19200 Baud
```

7.11.2 SEND :

Syntax : SEND(<int_exp>,<data_array>,<int_exp>)
Return : <void>

sendet eine Anzahl Bytes aus einem Datenarray über einen PORT [port-id, data-array, anzahl]
Aufruf : gültiger Wertebereich für port-id : [0 bis 4]

z.B. um 10 Bytes aus Array 'C#' über Port-ID Nr. 1 (COM-6) zu senden :

```
10 DIM C#(100)
20 DATA 1,2,3,4,5,6,7,8,9,10
30 INITPORT(1,4)
40 SEND(1,C#(0),10)
```

- reserviert 100 Bytes für Data-Array 'C#'
- füllt die ersten 10 Bytes mit Daten
- init von Port-ID Nr. 1 (COM-6)
- sendet 10 Bytes vom Array nach Port-1

port-id : 0 = COM-1
1 = COM-6
2 = I2C-3
3 = SPI-5
4 = SPI-4

Hinweis : bei Port-ID 3+4 muss das SlaveSelect-Signal vom BASIC-Programm erzeugt werden.

Hinweis zum senden über I2C (Port-2) :

Byte-0	muss die Slave-Adresse (8bit) enthalten
Byte-1	muss die Register Adresse enthalten
Byte-2 bis n	müssen die zu sendenden Daten enthalten

7.11.3 RECEIVE : (INTEGER FUNKTION)

Syntax :
Port0+1 : RECEIVE(<int_exp>,<data_array>,<int_exp>)
Port2+3+4 : RECEIVE(<int_exp>,<data_array>,<int_exp>,<data_array>)
Return : <int>

empfängt eine Anzahl Bytes von einem PORT, speichert sie in ein Datenarray und übergibt die Anzahl der empfangenen Bytes. [port-id, data-array, anzahl]
Aufruf : gültiger Wertebereich für port-id : [0 bis 4]

z.B. um Bytes von Port-ID Nr. 1 (COM-6) zu empfangen und in Array 'C#' zu speichern :

```
10 DIM C#(100)                  ▸ reserviert 100 Bytes für Data-Array 'C#'
20 INITPORT(1,4)                ▸ init von Port-ID Nr. 1 (COM-6)
30 a=RECEIVE(1,C#(0),10)        ▸ empfängt Bytes vom Port-1 in Array 'C#'
                                ▸ 'a' = Anzahl der Daten die empfangen wurden
```

port-id : 0 = COM-1
1 = COM-6
2 = I2C-3
3 = SPI-5
4 = SPI-4

Hinweis : bei Port-ID 3+4 muss das SlaveSelect-Signal vom BASIC-Programm erzeugt werden.

Hinweis zum empfangen über I2C (Port-2) :

es muss noch ein zweites Daten-Array übergeben werden mit der Slave-Adr und Register-Adr

Byte-0 muss die Slave-Adresse (8bit) enthalten
Byte-1 muss die Register Adresse enthalten

z.B. um 4 Bytes von Port-ID Nr. 2 (I2C-3) zu empfangen und in Array 'C#' zu speichern :

```
10 DIM C#(100)                  ▸ reserviert 100 Bytes für Data-Array 'C#'
20 DIM D#(2)                    ▸ reserviert 2 Bytes für Data-Array 'D#'
30 DATA 0x80,0x10               ▸ setzt SlaveAdr auf 0x80 und RegAdr auf 0x10
40 INITPORT(2)                  ▸ init von Port-ID Nr. 2 (I2C-3)
50 a=RECEIVE(2,C#(0),4,D#(0))   ▸ empfängt 4 Bytes vom Port-2 in Array 'C#'
                                ▸ 'a' = Anzahl der Daten die empfangen wurden
```

Hinweis zum empfangen über SPI (Port-3 und Port-4) :

es muss noch ein zweites Daten-Array übergeben werden mit den zu sendenden Daten

Byte-0 bis n muss die zu sendenden Daten enthalten

z.B. um 10 Bytes von Port-ID Nr. 3 (SPI-5) zu empfangen und in Array 'C#' zu speichern :

```
10 DIM C#(100)                  ▸ reserviert 100 Bytes für Data-Array 'C#'
20 DIM D#(10)                   ▸ reserviert 10 Bytes für Data-Array 'D#'
20 DATA 1,2,3,4,5,6,7,8,9,10    ▸ füllt 10 Bytes mit Daten
40 INITPORT(3)                  ▸ init von Port-ID Nr. 3 (SPI-5)
50 a=RECEIVE(3,C#(0),10,D#(0))  ▸ empfängt 10 Bytes vom Port-3 in Array 'C#'
                                ▸ 'a' = Anzahl der Daten die empfangen wurden
```

7.12 SPRITE-Befehle :

7.12.1 INITSPRITE :

Syntax : INITSPRITE(<int_exp>,<data_array>)
Return : <void>

Initialisiert ein SPRITE mit einer eindeutigen Sprite-ID und weißt ihm ein Datenarray zu.
Aufruf : gültiger Wertebereich für sprite-id : [1 bis 99]

z.B. um ein SRITE mit der Sprite-ID 7 dem Datenarray 'C#' zuzuordnen :

```
10 DIM C#(100)                ▸ reserviert 100 Bytes für Data-Array 'C#'
20 INITSPRITE(7,C#(0))        ▸ weißt Sprite Nr. 7 das Datenarray 'C#' zu
```

Hinweis : das Datenbyte-0 im Datenarray bestimmt den Typ vom SPRITE (byte-0 = sprite-typ)

sprite-typ : 0 = ▸ bit per pixel : 1/ color : 1 / Breite : max 8 pixel

```
byte-1      = Breite vom SPRITE in pixel [1 bis 8]
byte-2      = Höhe vom SPRITE in pixel [1 bis n]
byte-3+4    = Farbe vom SPRITE als RGB565 Wert [0 bis 65535]
byte-5 bis n = Pixelwerte vom SPRITE (1bit = 1Pixel)
```

1 = ▸ bit per pixel : 1/ color : 1 / Breite : max 16 pixel

```
byte-1      = Breite vom SPRITE in pixel [9 bis 16]
byte-2      = Höhe vom SPRITE in pixel [1 bis n]
byte-3+4    = Farbe vom SPRITE als RGB565 Wert [0 bis 65535]
byte-5 bis n = Pixelwerte vom SPRITE (1bit = 1Pixel)
```

2 = ▸ bit per pixel : 4/ color : max 16 / breite : max 8 pixel

```
byte-1      = Breite vom SPRITE in pixel [1 bis 8]
byte-2      = Höhe vom SPRITE in pixel [1 bis n]
byte-3      = Transparenzfarbe vom SPRITE als 4bit Wert [0 bis 15]
byte-4 bis n = Farbwerte vom SPRITE (4bit = 1Pixel)
```

3 = ▸ bit per pixel : 4/ color : max 16 / breite : max 16 pixel

```
byte-1      = Breite vom SPRITE in pixel [9 bis 16]
byte-2      = Höhe vom SPRITE in pixel [1 bis n]
byte-3      = Transparenzfarbe vom SPRITE als 4bit Wert [0 bis 15]
byte-4 bis n = Farbwerte vom SPRITE (4bit = 1Pixel)
```

z.B. um ein SPRITE vom sprite-typ:0 und 5x6 Pixel zu erstellen :

```
10 DIM C#(100)                ▸ reserviert 100 Bytes für Data-Array 'C#'
20 DATA 0                     ▸ setzt Sprite-Typ auf 0
30 DATA 5,6                   ▸ setzt Größe vom Sprite auf 5x6 Pixel
40 DATAW 0xF800               ▸ setzt Farbe vom Sprite auf 'rot'
50 DATA 0xF8,0x80,0xE0        ▸ setzt Pixeldaten für ein 'F'
60 DATA 0x80,0x80,0x80
70 INITSPRITE(7,C#(0))        ▸ weißt Sprite Nr. 7 das Datenarray 'C#' zu
```

7.12.2 SPRITEMODE :

Syntax : SPRITEMODE(<int_exp>,<int_exp>)
Return : <void>

stellt den Mode von einem SPRITE mit einer bestimmten Sprite-ID ein
Aufruf : gültiger Wertebereich für sprite-id : [1 bis 99]

Spritemode : Bit0 : visible [0=unsichtbar, 1=sichtbar]
Bit1 : mirror-X [0=normal, 1=gespiegelt]
Bit2 : mirror-Y [0=normal, 1=gespiegelt]
Bit3 : zoom-X [0=normal, 1=vergrößert]
Bit4 : zoom-Y [0=normal, 1=vergrößert]
Bit5 : direction [0=landscape, 1=portrait]
Bit6 : sprite collision [0=inaktiv, 1=aktiv]

z.B. um den Spritemode von SPRITE Nr. 7 auf den Wert '17' zu stellen :

```
10 SPRITEMODE(7,17)
```
▸ Sprite ist sichtbar und in Y-Richtung vergrößert

Hinweis : Mirrox-X/Y funktioniert nur bei Sprite-Typ 0+1

7.12.3 DRAWSPRITE :

Syntax : DRAWSPRITE(<int_exp>,<int_exp>,<int_exp>)
Return : <void>

zeichnet ein SPRITE mit einer bestimmten Sprite-ID auf den Bildschirm an x,y Position
Aufruf : gültiger Wertebereich für sprite-id : [1 bis 99]
Aufruf : gültiger Wertebereich für x : [0 bis 319] für y : [0 bis 239]

z.B. um das SPRITE Nr. 7 an Position 59,68 zu zeichnen

```
10 DRAWSPRITE(7,59,68)
```
▸ Sprite Nr. 7 an Koordinate 59,68 zeichnen

Hinweis : der 'DRAWSPRITE' Befehl setzt automatisch das Bit0 vom Spritemode 'visible'

7.12.4 MOVESPRITE : (INTEGER FUNKTION)

Syntax : MOVESPRITE(<int_exp>,<int_exp>,<int_exp>)
Return : <int>

bewegt ein SPRITE mit einer bestimmten Sprite-ID um x,y Pixel auf dem Bildschirm
und gibt den Kollisionswert zurück.
Aufruf : gültiger Wertebereich für sprite-id : [1 bis 99]
Aufruf : gültiger Wertebereich für x : [-319 bis 319] für y : [-239 bis 239]

z.B. um das SPRITE Nr. 7 um 10 Pixel nach rechts zu verschieben und 0 Pixel nach unten :

```
10 a=MOVESPRITE(7,10,0)
```

- Sprite Nr. 7 um 10 Pixel nach rechts bewegen
- 'a' = Kollisionswert

Kollisionswert :	Bit0 : screen left	[1=Kollision mit linkem Bildschirmrand]
	Bit1 : screen right	[1=Kollision mit rechtem Bildschirmrand]
	Bit2 : screen top	[1=Kollision mit oberem Bildschirmrand]
	Bir3 : screen bottom	[1=Kollision mit unterem Bildschirmrand]
	Bit4 : sprite left	[1=Kollision an linker Kante mit anderem Sprite]
	Bit5 : sprite right	[1=Kollision an rechter Kante mit anderem Sprite]
	Bit6 : sprite top	[1=Kollision an oberer Kante mit anderem Sprite]
	Bit7 : sprite bottom	[1=Kollision an unterer Kante mit anderem Sprite]

Hinweis : für eine Kollisionserkennung mit einem anderem SPRITE (Bit4 bis Bit7) muss bei beiden SPRITES das Bit6 vom Spritemode gesetzt sein 'sprite collision'

7.12.5 GETSPRITECOLL : (INTEGER FUNKTION)

Syntax : GETSPRITECOLL(<int_exp>)
Return : <int>

wenn per 'MOVESPRITE' eine Kollision mit einem anderen SPRITE erkannt wurde, kann dessen Sprite-ID per 'GETSPRITECOLL' ausgelesen werden.
Aufruf : gültiger Wertebereich für sprite-id : [1 bis 99]

z.B. um das SPRITE Nr. 7 um 10 Pixel nach links zu verschieben und dann die Sprite-ID nach einer Kollision zu ermitteln :

```
10 a=MOVESPRITE(7,-10,0)
20 b=GETSPRITECOLL(7)
```

- Sprite Nr. 7 um 10 Pixel nach links bewegen
- bei Spritekollision die Sprite-ID auslesen
- 'b' = Sprite-ID vom anderen Sprite

Hinweis : falls keine Kollision mit einem Sprite stattgefunden hat, wird '0' zurückgeliefert.

7.12.6 GETSPRITEPOS :

Syntax : GETSPRITEPOS <int_exp>,<int_var>,<int_var>
Return : <void>

ließt die aktuelle Spriteposition von einem Sprite ein [x,y] und speichert sie in Variablen.
Aufruf : gültiger Wertebereich für sprite-id : [1 bis 99]
Rückgabe : Wertebereich für x : [0 bis 319] für y : [0 bis 239]

Beispiel um die Position vom SPRITE Nr. 7 auszulesen :

```
10 GETSPRITEPOS 7,x,y
```

- 'x' = Spriteposition in X-Richtung
- 'y' = Spriteposition in Y-Richtung