User Manual

Version 07/08 E BA001b ProgRef

Revision 1.1

CE

Programmier Referez

MICRO DIP
MICRO PCB

Control Units

INHALT

EINFÜHRUNG

BASIC PROGRAMMIERBARE CONTROL UNITS

Kleine Microcontroller haben die Welt erobert und sind überall zu finden. Microcontroller werden vorwiegend in Assembler programmiert, was Erfahrung in dieser Programmiersprache und genaue Kenntnis der Prozessorarchitektur erfordert. Nicht selten ist allein das Manual zum Prozessor mehrere hundert Seiten lang und von Laien praktisch nicht zu verstehen. Auch wer Assembler programmieren möchte, ohne diese Kenntnisse bereits zu haben ist auf Bücher angewiesen deren Seitenumfang ganz sicher noch größer ist BASIC++ ist der BASIC-Dialekt, der zur Programmierung der MICRO Control Units verwendet wird und keine der genannten Spezialkenntnisse erfordert. Die Syntax entspricht in etwa der des Standard-BASIC. Um den Kern des Standard-BASIC sind zahlreiche neue Funktionen verfügbar.

Speicher
Die Computer der MICRO Serie haben ein komplettes Betriebssystem auf dem Chip und lassen sich in einem BASIC-Dialekt programmieren. BASIC ist nicht nur leicht zu erlernen, sondern auch sehr platzsparend. Ein BASIC- Program braucht nur etwa ein Fünftel des Speichers, das ein Asseblerprogramm gleicher Funktion benötigen würde. Grundsätzlich gilt, je länger das BASIC - Programm ist, desto grösser ist der Speicher-Vorteil gegenüber einer Assembler Programmierung. Die MICRO Serie bietet dem Anwender etwa 2kB Programmspeicher und offiziell 24 Byte Speicher für Variablen (tatsächlich sind aber bis zu 40 Byte nutzbar),

Geschwindigkeit
Der Ausspruch "BASIC ist langsam" stammt aus der Zeit, als BASIC auf dem Computer als Quelltext gespeichert war und der gesamte Text bei der Ausführung erst gelesen und interpretiert werden musste. Bei BASIC wird der Quelltext von einem Compiler gelesen und in die von Betriebssystem lesbaren sog. Token verwandelt, welche als eigentliches Programm in die Control Units geladen wird. Das Betriebssystem muss also nicht die vielen Buchstaben einer Programmzeile lesen sonder nur das dafür repräsentative Token. Und das geht sehr schnell. Die MICRO Unit benötigt nur etwa 100us für die Ausführung eines BASIC-Befehls.

I/O Ports:
Die Control Units MICRO sind dafür konzipiert kleinere Steuerfunktionen mit minimalem Bauteileaufwand und auf kleinstem Raum zu realisieren. Sechs Ports die (auch innerhalb eines Programms) mehrere verschiedene Funktionen haben können, lassen eine vielseitige Verwendung zu. Der 1uA Low Power Mode ist ideal für Anwendungen in denen Batteriebetrieb mit langen Standzeiten erforderlich ist (z.B. Datenlogger, Temperaturwächter)

Standard digital I/O Ports
8Bit A/D Ports
PLM D/A Ports
Beep Ports
Frequency Count Input
Serial RX/TX Ports

1=5V
2=P6/AD6/BEEP
3=P5/AD5/BEEP
4=P4/FREQ/BEEP
5=P3(INPUT ONLY)
6=P2/AD2/DA2/RXD/BEEP
7=P1/AD1/DA1/TXD/BEEP
8=GND

Pin-Belegung u. Funktion

ALLGEMEIN ZU BASIC++

Programmieren in BASIC++
Wenn man bedenkt, dass ein einziger BASIC-Befehl Routinen des Betriebssystems aufruft, die meistens mehrere hundert Bytes lang sind, ist leicht einzusehen, dass es eine enorme Zeiterspartnis darstellt ein Programm in BASIC zu programmieren. Ein Programm in BASIC zu programmieren dürfte nur etwa ein Zehntel der Zeit beanspruchen, welches ein Assemblerprogramm gleicher Funktion zum Entwurf benötigt.

Nötige Programmierkenntnisse
Basic++ ist sowohl für den Einstieg (einfache Kernfunktionen) als auch für die professionelle Programmierung der Control Units optimiert. Der Einstieg wird durch viele Beispiele sowohl zu BASIC++ als auch zu angebotenem Zubehör erleichtert.
Sie werden sehr schnell erkennen wie einfach die Handhabung ist und wie übersichtlich sich Ihre Programme gestalten lassen. Ausserden finden Sie in der LIBRARY eine Auswahl von Programmen die Sie in Ihren Anwendungen unterstützen
BASIC++ und dieMICRO Control Units sind also auch für Entwickler und Programmierer, welche die

Kenntnisse über Prozessorarchitektur und Assemblerprogrammierung haben, ein leistungsfähiges Tool, komplexe Programme in kürzester Zeit zu entwickeln und eine flexibles Controller-System für die Steuerung in Ihrer Anwendung.

Bitte besuchen Sie www.Spiketronics.com

ALLGEMEIN ZU DIESER REFERENZ

Diese Entwickler Referenz beschreibt die Aspekte der Programmierung in BASIC++ speziell für die Control Units MICRO. Die MICRO Units haben gegenüber den anderen Control Units derMICRO Serie einige Einschränkungen.

Bei der MICRO sind grundsätzlich alle Datentypen auf 8 Bit beschränkt (ohne Vorzeichen). Wortvariablen (signed Integer) werden also nicht unterstützt, was aber für den vorgesehenen Einsatzzweck nicht ins Gewicht fällt. Außerdem werden einige Funktionen wie z.B. die Datumsfunktionen nicht unterstützt. Allgemein kann man aber sagen, dass sich fast alle im Betriebssystem fehlenden Funktionen leicht in BASIC++ nachbilden lassen, wenn es erforderlich sein sollte. So ist z.B. auch die fehlende MIN/MAX Funktion mit zwei Zeilen BASIC Programm nachbildbar.

Nicht vorhanden sind z.B. die Datei-Funktionen zur Datenspeicherung, die aber prinzipiell als spezielle Systemerweiterung nachgesrüstet werden könnten und im Rahmen dieser Referenz auch beschrieben werden.

Im Rahmen dieser Referenz werden die für die MICRO anwendbaren Instruktionen und Definitionen beschrieben, soweit es sinnvoll ist, wird jeweils eine allgemein gültige Syntax angegeben.

Syntax: *Variable = value1 MOD value2*

Variable: Variable Byte type
value1 Variable, value oder Konstante Byte type
value2 Variable, value oder Konstante Byte type

Die Werte (hier *value1* and *value2*) können auch Terme in Klammern sein.

Variable= *(SQR(value*value))* MOD *value2*

Das ist gültig, solange Rechenergebnisse den Zahlenbereich eine Bytes nicht überschreiten.

Zusätzlich zur reinen Syntaxbeschreibung werden in dieser Referenz jeweils auch Beispiele gezeigt, bei denen die notwendige Definition der verwendeten Variablen aber aus Platzgründen fehlt.

Definiton:

```
DEFINE MyByte1 as byte
DEFINE  MyByte2 as byte
```

Beispiel:

```
MyByte=MyByte1 MOD MyByte2
```

DATA TYPES

Die MICRO unterstützt nur Bits und Bytes. Bits nehmen die Zustände ON und OFF bzw. TRUE und FALSE ein. Bytes können einen vorzeichenlosen Wert von 0 bis 255 annehmen. Die Zuweisung kann dabei in verschiedenen Zahlensystemen erfolgen.

Beispiele verschiedener Zahlensysteme

01011101b	binary system
123o	oktal system
1AFh	hexadecimal system
1000	decimal system
ON	boolean true (numerical 255 at byte values)
OFF	**boolean false(numerical 0 for byte and word values)**

DEKLARIERUNGEN UND DEFINITIONEN

OPTION

Das **OPTION** Schlüsselwort ist ein Compilerpräprozessorbefehl, der dazu dient für unterschiedliche Controller Versionen einen angepassten Befehlscode zu erzeugen.

Generell sollte am Anfang des Programms immer die Zielplattform durch OPTION angegeben werden.

Wenn Sie zum Beispiel mit der MICRO arbeiten sollten Sie mit dem OPTION Befehl die Zielplattform CCMICRO wählen; da die Micro nur mit BYTE Variablen arbeitet muss die systemeigene Rückgabe-Variable vom Typ BYTE sein.

Hier eine Lister der OPTIONS:

- **CC2.0**: 64/140 Byte User Variablen, Pointer-Unterstützung
- **CCMICRO**: 24 Byte User Variablen
- **CCADV**: 240 Byte User Variablen, Gleitkomma-Unterstützung (nur ADVANCED)
- **CCADVNOFLOAT**: 240 Byte User Variablen, keine Gleitkomma-Unterstützung (nur ADVANCED)
- **FLOAT**: 140 Byte User Variablen, Gleitkomma-Unterstützung (nur ADVANCED)

```
OPTION MICRO
```

Beispiel: Zielplattform MICRO

Variablen und Ports sind Speicherstellen im Controller die in der Regel gelesen und beschrieben werden können. Während Anwendervariablen frei verwendbar und benennbar sind, sind Ports in ihrer Funktion festgelegt. Sie sind zu Laufzeit veränderbar und deren Inhalt abrufbar. Dazu muss der Speicherplatz oder der Port zuerst vom Anwender mit einem symbolischen Namen versehen werden

```
DEFINE MyBitport1  as PORT[1]
DEFINE MyByte as 1
```

Definition mit symbolischen Namen

Sind Speicherstellen oder Ports einmal definiert werden sie im Programm ausschliesslich mit ihrem symbolischen Namen verwendet um z.B.neue Werte zuzuweisen oder Inhalte zu lesen.

```
MyBitport1 = ON
MyByte1 = MyByte1*10
```

Zuweisungen

PROGRAM MEMORY

Die MICRO hat 2KB Programmspeicher. Speicher der nicht für das Anwenderprogramm genutzt wird, steht für nicht flüchtige Datenspeicherung zur Verfügung. Dafür ist allerdings ein Erweiterungsmodul des Betriebssystems erforderlich, das selbst 256 Bytes Speicher benötigt.

USER VARIABLEN - COMPILER DEFINIERT

Die MICRO Control Units haben 24 Variablen die dem Anwender für sein Programm zur freien Verfügung stehen. Allerdings lassen sich tatsächlich bis zu 40 benutzen.
Bei BASIC++ hat man bei der Variablen-Definition zwischen lokalen und globalen Varianblen zu unterscheiden die, wie der Name sagt nur innerhalb einer Function oder innerhalb des ganzen Programms Gültigkeit haben. Weiterhin gibt es referenzierte Variablen welche jeweils eine einzige Speicherstelle im definierten Format Word,Byte,Float darstellen, aber unter verschiedenen symbolischen Namen benutzt werden können.

Diese Variablen müssen definiert werden, bevor sie im Programm verwendet werden können. In der Regel überlässt man dem Compiler die Aufteilung der Variablen auf den Speicherbereich. Für spezielle Anwendungen kann aber der Anwender die Variablen auf bestimmte Speicherplätze selbst verteilen.

BIT Variablen
Acht Bit Varaibles(ON oder OFF) belegen 1 Byte im Speicher

```
define MyBit as bit
```

BYTE Variablen
Byte (value 0 ... 255) 1 Byte = 8 Bit

```
define MyByte as byte
```

Die MICRO kennt keine anderen Datentypen als Bit und Byte Variablen

Referenzierte Variablen
Um Variablenspeicher zu sparen kann man verschiedene Variablen auf eine einzige Variable referenzieren, d.h einer einzigen Variablen verschiedene Namen geben. Das ist für Bit Byte und Word Variablen zulässig

```
define MyByte as byte
define Byte1 Ref MyByte
define Byte2 Ref MyByte
```

Byte1 und Byte2 benutzen den Variablenspeicher von MyByte

Mit der Referenzierung hat man auch leicht Zugriff auf die einzelnen Bits innerhalb einer Variable

```
define DATA  as byte

define DB0   ref DATA at bit[1]
define DB1   ref DATA at bit[2]
define DB2   ref DATA at bit[3]
define DB3   ref DATA at bit[4]
define DB4   ref DATA at bit[5]
define DB5   ref DATA at bit[6]
define DB6   ref DATA at bit[7]
define DB7   ref DATA at bit[8]
```

Im Programm manupiliert dann z.B.

DB7=OFF

Das höchste Bit der Variablen DATA. Wichtig ist in diesem Zusammenhang dass die Referenzierung auf bit [1] bis bit [8] und nicht auf bit [0] bis bit [8] erfolgt

USER VARIABLEN - USER DEFINIERT

In der Regel überlässt man dem Compiler die Aufteilung der Variablen auf den Speicherbereich. Für spezielle Anwendungen kann aber der Anwender die Variablen auf bestimmte Speicherplätze selbst verteilen. Der Anwender hat selbst Sorge zu tragen dass gemeinsame Speicherzellen nicht ungewollt verändert werden, d.h BIT[8] teilt sich z.B.eine Speicherzelle mit BYTE[1]

In der Regel ist das nur erforderlich wenn Systemtreiber geladen werden welche bestimmte Variablen als Schnittstelle zum BASIC Teil des Programms benötigen. Dann werden sie Speicherzellen für den Treiber vom Anwender dafür reserviert und der verbleibende Rest dem Compiler zu Vergabe überlassen.

{BYTEOFFSET}

Der Präprozessor Befehl {BYTEOFFSET} legt den Startpunkt zur automatischen Vergabe des Index für Byte und Word Variablen fest. Bei Byte und Word Variablen wird der Index immer inkrementiert.

Beispiel für die Vergabe von Variablen Für einen Systemtreiber und Compileranweisung zur Variablen-reservierung. Byte [1] und Byte [2] sind hier für einen Treiber reserviert

```
define MyByte1   as Byte[1]
define MyByte2   as Byte[2]
{BYTEOFFSET 3}
define MyByte3 as Byte
```

Der Compiler beginnt hier mit der Vergabe der Variablen ab dem dritten Byte. MyByte3 wäre also Byte[3]

BIT Variablen
Die Speicherplätze können auf BIT Variablen verteilt werden. Da aber maximal 256 BIT Variablen zulässig sind, kann nicht der ganze Variablenspeicher an BIT Variablen vergeben werden. Die höchste BIT Variable ist BIT[256] und liegt im Speicher im höchsten Bit des Memory Bytes 32

Acht BIT Variablen (Zustand ON oder OFF) belegen 1 Byte im Speicher.

```
define MyBit as bit[1]
```

zulässig ist BIT[1] bis BIT[256]

BYTE Variablen
Bytes (Wertebereich 0 ... 255) ist der kleinste numerische Datentyp 1 Byte = 8 Bit

```
define MyByte as byte[1]
```

zulässig ist BYTE[1] bis BYTE[40]

KONSTANTEN

Mit Konstanten werden Zahlen, implementierte Konstanten und durch den Benutzer erstellte Konstanten bezeichnet. Konstanten (außer in BASIC++ schon vorimplementierte Konstanten) können nur während der Entwicklungsphase verändert werden. Während der Laufzeit ist der Wert im Gegensatz zu Eigenschaften fest definiert.

Numerische Konstanten werden in BASIC++ in der Regel im Dezimalsystem angegeben. Zusätzlich unterstützt BASIC++ auch das Binär- , Oktal- , und Hexadezimalsystem. Um eine solche Zahl zu markieren muss die Zahl entweder durch Endung "b", "o" oder "h" hervorgehoben werden.

```
const MyConstant = 122
```

EXTERNE DATEIEN

IMPORT

Es ist möglich externe Dateien in Form von Tabellen oder Programmen zu importieren. BASIC++ stellt für den Anwender eine ganze Reihe von Bibliotheksdateien zur Verfügung, die es dem Anwender ersparen allgemein gebräuchliche Funktionen (wie z.B. Tastatureingabe) selbst zu programmieren. Darüber hinaus machen sie ein Programm wesentlich besser lesbar. Es kann jedes beliebige File importiert werden, allerdings muss es natürlich vom Compiler lesbar sein.

Syntax: Import [File]

```
IMPORT "lib\MYFILE.BAS"
```

SYSCODE

Die Control Units sind in der Lage Teiber oder auch Erweiterungen des Betriebssystems zu laden. Dafür stehen zwei Seiten (je 256 Bytes) am Ende des Programmspeichers zur Verfügung. Die Files haben immer die Extension
.S19

Syntax: SYSCODE [File]

```
SYSCODE "lib\MYFILE.S19"
```

STANDARD DIGITAL I/O PORTS - P1,P2,P3,P4,P5P6

Die standard digital Input/Output - Funktion steht dem Anwender an allen 6 Ports der MICRO zur Verfügung (Ausnahme: P3 ist nur Input). Nach Anlegen der Betriebsspannung (Reset) sind alle Ports im Modus Digitalport - Eingang. Jeder Digitalport wird zu Anfang des Programms mit einem Symbolischen Namen versehen. Sehen Sie im Kapitel INSTRUKTIONEN UND SCHLÜSSELWÖRTER wie ein Digitalport gesetzt, gelesen oder deaktiviert wird.

```
define MyBitPort1          as PORT[1]
```

Zulässig ist die Definition für PORT[1] bis PORT[6] aber P[3] ist ausschliesslich Eingang.

DIGITAL INPUT PORT

Digitaleingänge werden zur Abfrage von Schaltzuständen verwendet. Wird ein Digitalport als Eingang benutzt, führt er im unbeschalteten Zustand einen nicht definierten Pegel, der vom Anwender mittels eines Pullup Widerstandes festgelegt werden sollte. Ist beispielsweise ein Reedkontakt an diesem Port angeschlossen, wird dann bei offenem Schalter eine logische Eins (,,wahr") vom Port gelesen, bei geschlossenem Schalter eine logische Null (,,falsch"). Achten Sie bitte unbedingt darauf, dass je nach Beschaltung des Ports und der logischen Aussage, die Ihr Programm beinhalten soll, der eingelesene Wert eventuell invertiert werden muss (NOT- Operator).
Nach dem Zuschalten der Betriebsspannung oder nach einem Reset verhalten sich alle Digitalports zunächst elektrisch als Eingang, sie führen also über den Pullup- Widerstand High- Pegel oder undefinierten Pegel, wenn nichts angeschlossen ist.

DIGITAL OUTPUT PORT

Wird ein Digitalport als Ausgang verwendet, können daran nachfolgende ICs, Transistoren oder Low-Current- Leuchtdioden über Widerstände betrieben werden. Der maximal zulässige Laststrom für einen einzelnen Port beträgt 10 mA. In jedem Fall ist eine ausreichende Strombegrenzung, zum Beispiel durch einen Widerstand, zu gewährleisten, da es sonst zur Zerstörung des Mikrocontrollers kommen kann! Innerhalb des Mikrocontrollers erfolgt die interne Beschaltung eines Digitalports als Ausgang oder Eingang beim Ausführen des Anwenderprogramms. Eine Schreibanweisung auf einen Port veranlasst diesen die Funktion eines Ausgangs anzunehmen.

ANALOGPORTS - P1,P2,P5,P6

Als alternative Funktion der Ports P1,P2,P5,P6 können alle die Funktion eines A/D ports und zwei davon die Funktion eines D/A-ports annehmen.Die maximale Eingangsspannung an den AD Eingängen ist durch die Betriebsspannung bestimmt welche auch gleichzeitig als Referenzspannung dient. Die Maximale

Ausgangsspannung der DA Ausgänge entspricht der Betriebsspannung.
Sehen Sie im Kapitel INSTRUKTIONEN UND SCHLÜSSELWÖRTER wie ein Analogport gelesen oder geschrieben wird.

Referenz-Spannung:
Der angelegte Spannungswert der Betriebsspannung wird dem internen Referenzspannungseingang zugeführt. Diese Spannung gilt als Obergrenze des Messbereiches der A/D-Wandlung und entspricht dem Wandlungswert 255 (FF hexadezimal). Meistens kann die Betriebsspannung aus einem standard Spannungsregler direkt benutzt werden. Bei größeren Ansprüchen an Genauigkeit und Stabilität muss die Betriebsspannung mit einer entsprechenden Präzision erzeugt werden. Als Referenz für das untere Ende des Messbereiches der A/D-Wandlung dient stets das Groundpotenzial (Masse, -, GND, Minus) der Betriebsspannung.

A/D CONVERTER PORTS - P1,P2,P5,P6

An den A/D-Ports können Sensoren aller Art angeschlossen werden, die eine Ausgangsspannung von 0 bis 5 Volt liefern. In den meisten Fällen werden hier aktive Sensoren zur Anwendung kommen, um das Signal des eigentlichen Sensorelementes zu verstärken und den Ansprüchen an Auflösung, Linearität und Driftverhalten zu genügen. Die AD- Eingänge haben eine Auflösung von 8 Bit, was einem Spannungswert von 19,6 mV pro Digit entspricht. Schützen sie die Analogeingänge mit einem Serienwiderstand (10k) wenn Sie nicht sicherstellen können, dass die Spannung niemals größer als 5V ist. Der Widerstand beeinflusst die Genauigkeit der Messung absolut unwesentlich.

Zulässig ist die Definition für AD[1] [2] [5] [6]

```
define MyAnalogIn          as AD[1]
```

D/A CONVERTER PORTS - P2,P1

Die zwei 8-Bit-D/A-Wandler arbeiten nach dem Prinzip der Pulsweitenmodulation. In einem Zeitabschnitt (Modulationsintervall), der aus 256 Teilabschnitten besteht, wird ein D/A-Ausgang für die Dauer von so vielen Teilabschnitten high- gepulst, wie es dem 8-Bit-Wert entspricht, der zur Ausgabe bestimmt ist. Die Dauer eines Teilabschnittes beträgt 78µs, die des gesamten Modulationsintervalls 20ms.
Die max. Ausgangsspannung der DA-Ports ist immer so groß (und ebenso genau) wie die 5V Betriebsspannung. Absoluter Fehler +/- 1 Digit (= 1/256 vom Messbereichsendwert) zzgl. Fehler der Betriebsspannung.
Zur Demodulation, also Wandlung in ein echtes Analogsignal genügt meist ein einfaches RC- Glied. Beachten Sie dabei jedoch die Restwelligkeit und den erzielbaren Maximalwert des Ausgangssignals. Beides ist abhängig von der Last, die nach dem RC- Glied folgt.

Beachten Sie bitte, dass der DA-Ausgang auch bei einem Wert von Null immer eine Restspannung am RC-Glied erzeugt. Das liegt zum einen daran, dass der Port im LO-Zustand etwa 50mV Spannung hat, und zum anderen daran, das auch bei einem DA-Wert von Null ein 78us langer Nadelimpuls erzeugt wird. Die Ausgangsspannung nach dem RC- Filter wird daher in diesem Fall etwa 70mV betragen.
Für den Betrieb von LED´s oder Lämpchen am DA-Port (zur Einstellung der Helligkeit) benötigen Sie kein RC- Filter, da das Auge zu träge ist dem schnellen hell/dunkel- Wechsel zu folgen.
Alternativ können die DA-Ausgänge für die Ansteuerung zweier Servos benutzt werden. Ein DA Wert von 12 entspricht 1ms und damit der Ruhestellung eines Servos. Ein Wert von 24 entspricht etwa 2ms und damit der Endstellung. Damit lassen sich die Servos mit 12 Stufen über den gesamten Winkelbereich bewegen, was für viele Anwendunge durchaus ausreichend sein kann.

Zulässig ist die Definition für DA[1] und DA[2]

```
define MyAnalogOut         as DA[1]
```

BEEP PORTS - P1,P2,P4,P5,P6

Die BEEP Instruktion erzeugt an einem ausgewählten Port eine Rechteckspannung. Dieser Prt wird dan vorübergehend ein Ausgang und verbleibt als solcher solange bis er eine andere Funktion z.B Analogport zugewiesen bekommt.

Zulässig ist die Definition für PORT[1] [2] [4] [5] [6]

```
DEFINE MyBeepPort          port[2]
```

SYSTEM PROPERTIES

System Properties - System Eigenschaften oder auch System Variablen - sind spezielle Ressourcen die mit festen, reservierten Bezeichnern versehen sind, die nicht deklariert werden müssen.

RTC - HOUR,MINUTE,SECOND

Die Echtzeituhr, Real Time Clock (RTC) beinhaltet die Uhrzeit. Sie kann manuell durch Beschreiben der zugehörigen Variablen gestellt werden, oder auch automatich mit der gesetzlich gültigen Zeit synchronisiert werden, wenn ein entsprechended Empfangsmodul angeschlossen und die Systemerweiterung dafür geladen wurde. Alle Variablen sind Byte Werte.

Für die RTC sind die Variablen HOUR, MINUTE, SECOND reserviert. Sie sind lesbar und beschreibbar.

Beispiel: Abfrage der Sekunden.

```
IF SECOND = 10 THEN GOTO X
```

Beispiel: Stellen der Sekunden.

```
SECOND = 0
```

Beim Beschreiben der Variablen SECOND wird gleichzeitig der 20ms TIMER auf null gestellt.

Systembedingt ist die Genauigkeit der RTC sehr schlecht.
Das Beispiel TOOL_CLOCK_CHECK.BAS gibt weitere Hinweise darauf und zeigt wie man die Genauigkeit verbessern kann.

TIMER

Auch TIMER ist eine System Variable. Sie enthält eine Kurzzeitmesser, der alle 20ms inkrementiert wird. Der TIMER nimmt einen maximalen Wert von 49 an und rollt dann auf Null. Er zählt also 50 Ticks und ist damit für Zeitmessungen (z.B. Dauer eines Tastendrucks) bis max. 1 Sekunde geeignet.
Der TIMER kann nur gelesen werden. Er wird aber gelöscht wenn die System Variable SECOND beschrieben wird.

Eine Instruktion wie TIMER=10 ist daher nicht zulässig.

Beispiel für eine TIMER Abfrage in einer 1/2 Sekunden Warteschleife.

```
SECOND=0
#WAITLOOP
IF TIMER < 25 THEN GOTO WAITLOOP
```

FREQ

Die System Variable FREQ hat zwei unterschiedliche Bedeutungen, je nach dem ob sie beschrieben oder gelesen wird.

Schreiben auf FREQ
Schreiben auf FREQ setzt die Gate Time der Frequenzzählung. Die Gate Time ist die Zeit, in welcher die Ereignisse an Port FREQ - P4 gezählt und aufsummiert werden. P$ muss dazu ein Input sein. Die Gate Time wird dazu auf den zu messenden Frequenzbereich eingestellt. Grosse Zeiten für niedrige Frequenz, kleine Zeiten für hohe Frequenzen.
Bei einer Gate Time von z.B. 20ms ergibt sich ein Zählerstand (siehe Lesen von FREQ) von 200 wenn die Frequenz an P4 10KHz ist.

0.02*10000 = 200.

Weil FREQ eine Byte Variable ist, kann sie max. den Wert 255 annehmen. Die Gate Time ist dann 5100ms. Damit können auch niedrige Frequenzen im Bereich von wenigen Hertz gemessen werden. Eine neu eingestellte Gate Time wird gültig, wenn die vorherige abgelaufen ist.

Beispiel um die Gate Time auf 100ms zu stellen:

```
FREQ=5
```

Lesen von FREQ
Das Lesen von FREQ liefert den Zählerstand der während der Gate Time aufsummierten, an P4 gezählten Ereignisse.

Beispiel zum Lesen von FREQ

```
x=FREQ
```

Berechnung der Frequenz
Die Berechnung der Frequenz erfolgt in Abhängigkeit der Gate Time : FREQUENCY = x / GATE TIME.
Ist die Gate Time z.B. auf 100ms eingestellt und das Zählergebnis (x=FREQ) 200, dann berechnet sich die an P4 angelegte Frequenz zu:

F=200 / 0.1= 2000 Hz

Frequenz Bereich
Weil FREQ eine Byte Variable ist, können maximal 255 Ereignisse bei einer Gate Time zwischen 20ms und 5100ms gezählt werden. Die Gate Time muss daher auf die zu verarbeitende Frequenz eingestellt werden. Die kürzeste Zeit (20ms) verarbeitet Frequenzen bis ca. 12KHz. Die längste Gate Time Frequenzen von nur wenigen Hertz. Wenn man unbekannte Frequenzen verarbeiten möchte kann man eine automatische Gate Time Einstellung programmieren. Diese beginnt mit der kürzesten Gate Time und erhöht diese solange, bis sich ein nennenswertes Zählerergebnis beim Lesen von FREQ eistellt.

Einschränkungen:
Währen der Ausführung von PAUSE oder BEEP ist die Zählung vorübergehend unterbrochen, wird aber danach weitergeführt.

INSTRUKTIONEN UND SCHLÜSSELWÖRTER

INPUT/OUTPUT

Die MICRO Control Units haben nur eingeschränke Resourssen, sind aber denoch in der Lage mit externen Geräten über eine serielle Schnittstelle zu kommunizieren. Dies und die BEEP Funktion können wertvolle Hilfen beim Debuggen von Programmen sein.

BAUD

Die BAUD Instruktion stellt die serielle Schnittstelle auf 9600 Bd oder auch wahlweise auf 19200 Bd ein. Das Format ist in beiden Fällen 8N1, 9600 Bd ist der default Wert.

Syntax: Baud Rate

```
BAUD R9600
```

BAUD R9600
BAUD R19200
Stellt Sender (TX) und Empfänger (RX) auf die jeweilige Baudrate ein. Das Format 8N1 kann nicht geändert werden. Gleichzeitig wird damit auch TXD - P1 als Output geschalten.

Eine PAUSE 5 Instruktion sollte nach der Initialisierung der Schnittstelle mit BAUD folgen, um die Pegel an den entsprechenden Ports zu stabilisieren.

Beschränkungen:
Systembedingt durch den RC Oszillator, der den Systemtakt erzeugt, ist auch die Baudrate sehr ungenau. Diese wird deshalb beim Programm Download calibriert und ist auch über den zulässigen Temperaturbereich hinreichend stabil um den Betrieb bei 9600 Bd zu gewährleisten. Der Betrieb bei 19200 Bd wird jedoch mit grossen Spannungs und Temperaturschwankungen zunehmend unsicherer und die Betriebssicherheit kann in extremen Fällen nicht garantiert werden.
Allerdings gibt es die Möglichkeit hier korrigierend einzugreifen.

Das Program TOOL_19200_CALIBRATE.BAS zeigt welche Möglichkeiten der Anwender hier hat.

GET

Die GET Instruktion wartet auf ein einzelnes Byte von der seriellen Schnittstelle.

Syntax: Get Variable

```
GET MyByte
```

PRINT

Die PRINT Instruktion sendet eine Zeichenkette (String), Variablen oder Konstanten über die serielle Schnittstelle. PRINT sendet als letze Zeichen CR, LF (SCII 13,10). Ein Semikolon nach PRINT unterdrückt CR LF. Mehrere Argumente können mit & zu einer ainzigen Ausgabe verbunden werden.

Syntax: Print Argument [& Argument [& ...]] [;]

```
PRINT "WERT :" &  MvBvte & " V"
```

PUT

Die PUT Instruktion sendet ein einen einzelnen Wert zur seriellen Schnittstelle.

Syntax: Put Variable
*variable B*yte Variable oder Konstante

```
PUT MyByte
```

BEEP

Eine der alternativen Port-Funktionen ist die eines Beep-Ports. Die alternative Funktion wird aktiviert, sobald der Port in der betreffenden Weise (hier das aktivieren eines als Beep-Port definierten Ports) angesprochen wird. Die BEEP Instruktion erzeugt an einem definierten Port eine Rechteckspannung. Der Port ist dann ein Ausgang und verbleibt als solcher solange, bis er für eine alternative Funktion benutzt wird, z.B. als AD Eingang.

Syntax: *Beep Ton, Dauer, Port*

```
BEEP 10,100,MyBeepPort
```

Ton
Ton ist eine Konstante welche die Frequenz der Rechteckspannung festlegt. Es sind Werte von 1 (etwa 7000 Hz) bis 255 (etwa 600 Hz) zulässig..

Dauer
Dauer ist in 20ms Stufen angegeben und leggt die Dauer des Tons fest.

Port
Einer von 5 möglichen Ports der als Ausgangs-Port für den BEEP definiert wurde.

INTERNE DATENSPEICHERUNG

Diese Datenspeicher Funktionen sind nur verfügbar, wenn das entsprechende Erweiterungsmodul PAGE0_EXT1 geladen wurde. Das Modul selbst benötight 256 Bytes Programmspeicher.

Die Dateifunktionen erlauben das Aufzeichnen von Messwerten oder anderen Daten oder können zum Abspeichern von Information benutzt werden, die nach Ausfall der Betriebsspannung wieder in die Programmvariablen geladen werden sollen. Der Speicherbereich im FLASH nach dem Anwenderprogramm -meist der größte Teil - steht für diesen Zweck zur Verfügung. Der Speicherbereich wird als eine Datei verwaltet, auf die lesend oder schreibend zugegriffen werden kann, nachdem sie mit dem entsprechenden Attribut READ/WRITE geöffnet wurde.

OPEN# FOR READ - OPEN# FOR WRITE

Damit wird eine Datei zum sequentiellen Lesen oder Schreiben geöffnet und der Wert für FILEFREE initialisiert.
Dabei bedeutet WRITE das Öffnen zum Schreiben mit Überschreiben eventueller alter Aufzeichnungen. READ das Öffnen zum Auslesen der Aufzeichnungen. Gleichzeitig setzt OPEN den Datenzeiger (gemeinsam für READ und WRITE) auf den Anfangswert. Nach einem Reset ist der Dateizeiger gelöscht, es kann also nicht mit einer begonnenen Aufzeichnung fort gefahren werden. Gespeicherte Werte können natürlich gelesen werden aber der Zeiger muss auf jeden Fall wieder mit OPEN initialisiert werden.

INPUT#

Liest ein Byte von der durch den Datenzeiger adressierte Speicherstelle und inkrementiert den Datenzeiger. Es gibt keine EOF Funktion. Um also nur gültige Werte aus dem gesamten Speicherbereich zu lesen, muss der Anwender eine Variable für die Anzahl der gültigen Werte im Speicher anlegen. Damit ist es dann auch möglich gespeicherte Werte zu lesen während die Datenaufzeichnung läuft.

Syntax: INPUT# Variable
Variable: Byte Variable

```
INPUT# MyByte
```

PRINT#

Schreibt ein Byte an die durch den WRITE Datenzeiger adressierte Speicherstelle und inkrementiert den Datenzeiger.
Vor jedem Schreiben sollte geprüft werden, ob noch genügend Platz im FLASH zur Aufnahme der Daten vorhanden ist. Dafür kann die Funktion FILEFREE abgefragt werden, die als Ergebnis die Größe des noch freien Speichers liefert (in Blöcken zu feweils 64 Byte).

Syntax: PRINT# Variable
*Variable: B*yte oder Konstante

```
PRINT# MyByte
```

FILEFREE

Stellt fest wie viele Blöcke a 64 Byte aktuell zur freien Verfügung stehen. Direkt nach einem OPEN-Befehl ist das Ergebnis der maximale Wert, der sich verringert, je mehr Blöcke geschrieben werden.
FILEFREE berücksichtigt, dass die PAGE0 geladen ist (sie erweitert ja das Betriebssystem um die Funktion der Datenspeicherung). Die Auskunft von FILEFREE = 10 bedeutet dass Block 0 bis Block 10 verfügbar ist, es also 11 Blöcke sind (der erste Block ist Block 0). FILEFREE wird mit OPEN# initialisiert.

Beispiel für FILEFREE:

```
IF FILEFREE = 0 THEN GOTO X
```

ACHTUNG:
Das FLASH hat nur eine begrenzete Anzahl von zulässigen Schreibzyklen (100.000). Permanentes Schreiben auf das Flash wird also sehr schnell zur vollständigen Zerstörung des Programm-speichers führen.

LOOKTAB - TABLE

Konstante Daten können in Form von Tabellen abgelegt werden. Jede Tabelle bekommt einen Bezeichner (table) zugewiesen und kann beliebig viele Einträge enthalten, soweit der Programmspeicher Platz bietet. Jeder Dateneintrag wird als Integerwert abgelegt und belegt somit zwei Bytes. Dabei können die Daten direkt im Quelltext aufgeführt werden, dürfen aber nur Werte bis max. 255 annehmen, da auch Tabellen nur Byteweise verarbeitet werden, auch wenn sie der Compiler als Word ablegt.
Die Tabellendefinitionen müssen stets am Ende eines Programms, hinter dem END Befehl stehen, da die Daten nahtlos hinter den vorangehenden Codebytes im Speicher abgelegt werden. Die Programmabarbeitung darf nie über Tabellendaten laufen, da die Daten sonst als BASIC Befehle interpretiert werden würden, was sicher zum Absturz des Systems führt.

MyTab bezeichnet eine gültige Tabelle, für Myindex kann ein beliebiger Term stehen. Die Variable MyByte

bezeichnet die Speicherzelle, in der das Ergebnis abgespeichert werden soll. Der berechnete Wert des Index-Terms darf muss in Byte Grösse sein. Der erste Tabelleneintrag hat den Index 0.

Syntax: Looktab(Table, Index, Variable)
Variable = Looktab(Table, Index)

Index: Byte Variable oder Konstante
Variable: Byte Variable

```
LOOKTAB   MyTab,MyIndex,MyByte

TABLE myTab
1 2 3 4 5 6 7
END TABLE
```

LOOKTAB ist auch ohne System Erweiterung nutzbar.

STANDARD DIGITAL PORTS

Nach einem RESET d.h dem Einschalten der Betriebsspannung sind alle Ports Inputs. Sie verbleiben als Digitalpots solange bis sie mit einer entsprechenden Instruktion einer Alternativen Verwendung zugeführt werden.

Ein Port sollte niemals direkt, ohne Vorwiderstand, mit einer niederohmigen Spannungsquelle verbunden werden, auch nicht wenn er als AD-Port verwendet wird. Ein Programmfehler könnte den Port als Digitalport - Ausgang LO schalten und ihn damit zerstören.

PORT READ/WRITE

Bevor ein Port benutzt werden kann muss er mit einem symbolischen Namen als solcher definieret werden (siehe Kapitel DEKLARATIONEN and DEFINITIONEN). Danach wird er im Programm ausschliesslich mit diesem Namen angesprochen.
Das Lesen von einem Digital Bitport liefert einen Boolschen Wert, also TRUE (ON) oder FALSE (OFF).
ON ergibt sich wenn der Eingang log 1 ist, OFF wenn er log 0 ist.
Wenn ein Port vorher eine alternative Funktion hatte z.B als AD Eingang, so wird diese Funktion beendet und dem Port die neue Funktion digitaler Bitport zugewiesen.

PORT READ
Liest den Zustand am Porteingang. Der Port muss dazu ein Eingang sein. Hatte der Port vorher eine andere Funktion wird er automatisch richtig als Digitalport-Eingang konfiguriert. War der Port vorher ein Digitalport-Ausgang muss er vor der Benutzung als Eingang mit DEACT in den Modus Eingang geschaltet werden.

Beispiel Port lesen:

```
IF  MyBitPort1=OFF THEN GOTO X
IF MyBitPort1=ON THEN GOTO X
```

Das Lesen von einem Bitport Ausgang liefert den Ausgangszustand. DEACT schaltet den Port als Eingand zurück.

PORT WRITE
Setzt den Ausgangszustand eines Digitalports. Hatte der Port vorher eine alternative Funktion wird er automatisch als Digitalport mit dem entsprechenden Aushangswert geschaltet.

Beispiel Bitport Schreiben:

```
MyBitPort1=OFF
MyBitPort1=ON
```

TOG

TOG (toggle) ist eine Instruktion die den gegenwärtigen Zustand eines Bitport-Ausgangs umkehrt.
TOG hat keine Wirkung wenn der Port kein Digitalport ist, oder wenn er ein Digitalport-Eingang ist.

Syntax: TOG(bitport)

Beispiel: Bitport toggle.

```
TOG MyBitPort
```

DEACT

DEACT schaltet einen Digitalport-Ausgang als Eingang

Syntax: DEACT(bitport)

Beispiel Deaktivieren eine Digitalport Ausgangs

```
DEACT MyBitPort1
```

AD PORTS

Eine der Alternativen Port-Funktionen ist die eines AD-Wandlers.
Bevor ein AD-Port benutzt werden kann muss er mit einem symbolischen Namen als solcher definieret werden (siehe Kapitel DEKLARATIONEN and DEFINITIONEN). Danach wird er im Programm ausschliesslich mit diesem Namen angesprochen.
Die alternative Funktion wird aktiviert, sobald der Port in der betreffenden Weise (hier das Lesen eines als AD-Wandler definierten Ports) angesprochen wird. Das Lesen liefert einen Byte Wert zurück, welcher der Eingangsspannung des AD-Wandlers entspricht.
Der Port verbeleibt in seiner Funktion als AD-Wandler solange nicht eine seiner alternativen Funktionen aktiviert wird

Beispiel zum Lesen des AD Wandlers:

```
IF MyAnalogIn = 100 THEN GOTO X
MyByte=MyAnalogIn / 10
```

DA PORTS

Eine der alternativen Port-Funktionen ist die eines DA-Wandlers.
Bevor ein DA-Port benutzt werden kann muss er mit einem symbolischen Namen als solcher definieret werden (siehe Kapitel DEKLARATIONEN and DEFINITIONEN). Danach wird er im Programm ausschliesslich mit diesem Namen angesprochen.
Das Schreiben eine Byte-Wertes setzt den DA Ausgang auf den entsprechenden durchschnittlichen Spannungswert.Der Port verbeleibt in seiner Funktion als DA-Wandler solange nicht eine seiner alternativen Funktionen aktiviert wird

Beispiel zum Schreiben des DA Port:

```
MyAnalogOut =10
```

MATH FUNCTIONS

Alle Mathematischen Funktionen und Operationen sind auf Byte-Grösse beschränkt. Das bedeutet, dass selbst wenn Einggangs und Ausgangsgrössen im Wertebereich eine Byte liegen, dafür Sorge getragen werden muss, dass Zwischenergebnisse bei Berechnungen keinen Überlauf erzeugen.

Beispiel: x=100*3/2

Das Ergebnis ist 150 und ist im gültigen Wertebereich. Die Berechnung 100*3 jedoch führt zu einem Überlauf und damit zu einem falschen Ergebnis. Hier sieht man deutlich dass die Reihenfolge der Berechnung eines Term durchaus von Bedeutung ist.

x=100/2*3 führt zu einem richtigen Ergebnis

GRUNDRECHENARTEN

Syntax: *Variable = value1 [operator] value2*

MyByte=Value1 / Value2
MyByte=Value1 * Value2
MyByte=Value1 + Value2
MyByte=Value1 - Value2

Variable: Variable Byte typ
value1: Variable, Wert oder Konstante, Byte Typ
value2: Variable, Wert oder Konstante, Byte Typ
operator: division /
multiplikation *
addition +
subtraktion

RAND

Die RAND Funktion liefert mit jedem Aufruf einen zufälligen Wert. Die Reihenfolge der Werte ist jedoch immer gleich. Ausserdem wiederholen sich die Werte nach kurzer Zeit.

Syntax: Variable = RAND

MyByte=RAND

Variable: Variable Byte Typ

MOD

Die MOD (Modulo) Funktion liefert den Rest einer Division.
Beispiel: 10 MOD 3 liefert z.B. 1 als Ergebnis.

Syntax: *Variable = value1 MOD value2*

MyByte=MyByte1 MOD MyByte2)

Variable: Variable - Byte Typ
value1 Variable, Wert oder Konstante - Byte Typ
value2 Variable, Wert oder Konstante - Byte Typ

MATHEMATISCHE UND BOOLSCHE OPERATOREN

Bei der Berechnung von Termen ist die Hirarchie der Operatoren sehr wichtig. Hier eine Aufstellung dazu:

RANK	**OPERATOR**
8	( )
7	MATH FUNCTIONS
6	MULTIPLY DIVISION MOD SHL SHR
5	PLUS MINUS
4	COMPARES
3	NOT
2	AND
1	OR

COMPARES - VERGLEICHE

> (grösser) <(kleiner) >=(grösser oder gleich) <=(kleiner oder gleich) =(gleich) <>(ungleich)

Vergleiche sind zulässig für Byte-Werte, Terme und Funktionen.

```
IF MyByte1 <= MyByte2 THEN GOTO X
IF  MyByte1*10>100/(MyByte2) THEN GOTO X
```

SHL

Bitweises Linksschieben
SHL (logical shift left) schiebt alle 8 Bit eines Bytes um eine Stelle nach links und entspricht einer Multiplikation des Wertes mit zwei.

Syntax: *Variable = value1 SHL value2*

```
MyByte1 = MyByte2 SHL 8
```

Variable: Variable - Byte Typ
value1 Variable, Wert oder Konstante - Byte Typ
value2 Variable, Wert oder Konstante - Byte Typ

SHR

Bitweises Rechtssschieben
SHR (logical shift right) schiebt alle 8 Bit eines Bytes um eine Stelle nach rechts und entspricht einer Division des Wertes durch zwei.

Syntax: *Variable = value1 SHR value2*

```
MyByte1 = MyByte2 SHR 8
```

Variable: Variable - Byte Typ
value1 Variable, Wert oder Konstante - Byte Typ
value2 Variable, Wert oder Konstante - Byte Typ

AND, OR, XOR, NOT

sind boolsche Operatoren und dienen zur bitweisen Manipulation von Werten

Syntax: *Variable = value1 AND value2*

```
MyByte=MyByte1 AND MyByte2
```

Variable: Variable - Byte Typ
value1 Variable, Wert oder Konstante - Byte Typ
value2 Variable, Wert oder Konstante - Byte Typ

Bei der Verwendung innerhalb von IF THEN entspricht das der logisschen Verknüpfung von TRUE/FALSE welche wiederum TRUE/FALSE als Ergenis liefert.

```
IF (MyByte=1) OR (MyByte=2) THEN GOTO X
```

PROGRAMMFLUSS KONTROLLE

PAUSE

Stoppt die Programmausführung für eine bestimmte Zeit. Systeminterrupts z.B. die der RTC werden weiterhin ausgeführt. Die Zeit für die PAUSE wird in Einheiten von 20ms angegeben. PAUSE 50 hält das Programm demnach für 1 Sekunde an.

Syntax: PAUSE = *value1*
V*alue1:* Variable, Wert oder Konstante
- Byte Typ.

```
PAUSE 25
```

FOR, TO, NEXT, STEP, EXIT FOR

Wie fast jeder Programmiersprache verfügt auch Basic++ über eine Zählerschleife. Eine Zählerschleife weist einer Variable am Anfang einen Wert zu und addiert zu der Variable bei jedem Schleifendurchgang einen festgelegten Wert bis die Abbruchbedingung erfüllt ist. Solange innerhalb der Zählerschleife (FOR...NEXT) die Zählervariable nicht manipuliert wird kann es auch nicht zu einer Endlosschleife kommen. Der Nachteil der FOR Schleife ist jedoch, dass Sie ohne Manipulation der Zählervariable nicht so flexibel ist wie die DO...LOOP Schleife.

Mit STEP nach der Abbruchbedingung können Sie festlegen inwiefern die Zählervariable bei jedem Durchlauf inkrementiert werden soll. Wenn Sie STEP
weglassen wird die Variable standardmäßig um 1 inkrementiert. Eine Zähler-schleife sollte immer dann benutzt werden, wenn die Anzahl der Schleifen-durchläufe entweder als Konstante oder Variable bekannt ist.

EXIT FOR führt zu einem vorzeitigen verlassen der Schleife.
FOR TO NEXT Schleifen können verschachtelt werden.

Syntax: For [Counter] = [Start] To [End] [Step [Incr]]
[Exit For]
Next

```
FOR MyByte=MyStart TO MyEND STEP  MyStep
PRINT MyByte
IF MyPort=OFF THEN EXIT FOR
NEXT MyByte
```

Counter: Variable als Schleifenzähler
Start: Variable, Wert oder Konstante die zum Start in den Zähler geschrieben wird
End: Variable, Wert or Konstante, mit welcher der Zähler zur Feststellung der Endbedingung verglichen wird
Incr: Erforderlich wenn die Schrittweite grösser als 1 ist. Ein Wert, Variable oder Konstante.

DO, LOOP UNTIL, EXIT DO

Die DO...LOOP Schleife ist eine sog. Wiederholungsschleife, wobei die Schleifenabbruchbedingung nach der Ausführung des Codes innerhalb des Schleifenblocks geprüft wird. Die Schleife wird also mindestens einmal ausgeführt. Die DO...LOOP Schleife ist blockorientiert aufgebaut und kann somit mehrzeilige Anweisungen enthalten und Verschachtelt verwendet werden.
EXIT DO Führt zu einem vorzeitigen Abbruch der Schleife.
DO LOOP Schleifen können verschachtelt werden.

Syntax: Do
[instruction]
[Exit Do]
Loop (Until [expression])

```
DO
Mybyte=Mybyte+1
IF MyPort=OFF THEN EXIT DO
LOOP  UNTIL MyByte=5
```

IF, THEN, ELSE END IF

Die IF Bedingung ist die am meisten verwendete Kontrollstruktur in Basic. Mittels IF können Sie abfragen, ob eine Bedingung erfüllt ist oder nicht und abhängig davon weitere Befehle ausführen.Beim IF Befehl handelt es sich um einen mehrzeiligen Befehl einem sogenannten Block. Das Blockende wird durch das Schlüsse-wort END IF markiert.
Zwischen IF und END IF können Sie beliebig viele Anweisungen einfügen, die ausgeführt werden, wenn die Bedingungen erfüllt sind. Optional können sich zwischen IF und END IF noch ELSE einfügen. Alle Anweisungen zwischen ELSE und END IF werden ausgeführt wenn die IF Bedingungen nicht erfüllt sind. Natürlich können Sie auch IF Anweisungen ineinander verschachteln. Achten Sie hier jedoch besonders darauf, dass jeder IF Block mit END IF geschlossen wird. Sollte ein IF Block offen bleiben oder eine END IF Markierung ohne passende IF Anweisung auftreten meldet der Compiler einen Fehler.

Einzeilige Syntax bei Einzelanweisungen:

Syntax1: *If [expression] Then [instruction] [Else] [instruction]*

```
IF MyByte=10 THEN GOTO X ELSE GOTO Y
```

Mehrzeilige Syntax bei Anweisungsblöcken:

Syntax2: *If [expression] Then*
[instructions]
[Else]
[elseinstructions]
End If

```
IF MyByte=10 THEN
    PRINT "This is "
    PRINT MyByte
ELSE
    PRINT "No Match"
    PRINT "found"
END IF
```

SELECT CASE, CASE, CASE ELSE

Neben dem IF Befehl können Sie noch den SELECT Befehl verwenden um eine Selektion in Ihren Programmablauf einzufügen. Der SELECT Befehl eignet sich
besonders immer dann, wenn mehrere unterschiedliche Bedingungen abgefragt
werden sollen. Wie der IF Befehl ist auch der SELECT Befehl mehrzeilig angelegt. Den SELECT Block schließt man mit END SELECT.

Select Case matchexpression
[Case expression]
[instructions]
[Case Else expression]
[elseinstructions]

End Select

```
SELECT CASE i
   CASE 1
      PRINT "1"
   CASE 2
      PRINT "2"
   CASE ELSE
      PRINT "No Match"
END SELECT
```

Die einzelnen Bedingungen werden mit dem Schüsselwort CASE (jeweils zeilenweise) eingeschoben. Analog zum IF Befehl kann man optional auch mit CASE ELSE einen Zweig einfügen, der ausgeführt wird, wenn keine andere der bisher gelisteten CASE Bedingungen erfüllt war. Ebenso wie IF können auch SELECT Anweisungen verschachtelt werden. Auch hier müssen Sie wieder darauf achten, dass jeder SELECT Block durch END SELECT geschlossen wird.

WAIT

Eine wesentliche Voraussetzung für ein Programm ist eine Steuerung des Programmflusses, z.B. Verzweigungen. Die WAIT Anweisug ist die kurze Ausführung einer Bedingungsabfrage mit anschliessendem GOTO. Sie wartet bis die Bedingung erfüllt, also TRUE ist

Syntax: WAIT [expression]

```
WAIT (Mybitport = ON)
```

Weil ein Bitport immer Typ Bool ist kann man auch schreiben:

```
WAIT (MyBitPort)
```

GOTO

Sprungmarken sog. „Labels“ werden Ihnen beim Programmieren von Basic++ nicht so häufig begegnen, wie beim klassischen BASIC. BASIC++ ist viel modularer strukturiert und damit übersichtlicher. Hin und wieder sind Sprungmarken dennoch ganz nützlich. Sprungmarken werden mit dem Hash-Zeichen („#“) deklariert, wobei beim Aufruf der Sprungmarke das Hash-Zeichen nicht Bestandteil des Namens ist.
Achten Sie darauf, dass ein Label niemals lokal, sondern nur global initialisiert wird. Sie können auf eine Sprungmarke verweisen, die noch nicht deklariert wurde. Wenn nach der Kompilierung die Sprungmarke jedoch nicht gefunden wurde erscheint ein Fehler.
Mit Hilfe des GOTO Befehls gelangen Sie zu einer Sprungmarke, es wird keine Rücksprungadresse gesichert.

Syntax: GOTO Label

```
MyLabel:    IF (MyBitPort=OFF  THEN  GOTO MyLabel
```

Das kleine Beispiel entspricht: WAIT MyBitPort.

FUNCTION

FUNCTION - ein Unterprogramm leiten Sie mit dem Befehl FUNCTION ein. Jedes Unterprogramm benötigt einen eindeutigen Namen, wobei der Name den allgemeinen Namenskonventionen entspricht, die Sie auch schon für Variablen- und Sprung-markennamen kennen gelernt haben.Wie Sie an der Syntaxdefinition schon erkennen können müssen auch Funktionen mit END FUNCTION geschlossen werden.
Mit RETURN können Sie einer Funktion einen Rückgabewert zuweisen. Verwechseln Sie das RETURN nicht mit dem RETURN des standard BASIC
Eine FUNCTION kann vorzeitig mit EXIT FUNCTION verlassen werden.
Zwischen der geöffneten und geschlossenen Klammer hinter dem Funktionsname können Sie optional Parameter listen, die beim Aufruf der Funktion gesetzt werden müssen. Parameter sind lokale Variablen, die nur innerhalb der Funktion verwendet werden können. Der Umgang und die Gebrauchskonventionen von Parameter entspricht dem von normalen Variablen, die mit DEFINE erstellt wurden.

Syntax: Function [Name] ([[Parameter] As [Data type] | [Parameter] Ref [Variable], ...])
[instructions]
Return [expression]
End Function

```
FUNCTION MyFunction(X as byte Y as byte)
      X=2*Y
      RETURN 2*X
END FUNCTION
-------------------------------------------------------------
PRINT MyFunction(1,2)
```

Name: Erforderlich, Gültiger Bezeichner
Parameter: Optional, Gültiger Bezeichner für eine Parametervariable
Data type: Erforderlich, wenn kein Ref bei Parametervariable. Gültiger Datentyp
Variable: Erforderlich, wenn Ref bei Parametervariable. Bezeichner einer schon deklarierten Variable
Expression Optional, Term, Variable, Zahl oder Konstante als Rückgabewert

SPECIAL PORTS UND FUNKTIONEN

Spezielle Digitale Ports haben gegenüber den regulären Funktionen wie z.B. AD-Port oder BEEP-Port zusätzliche, einzigartige Funktion

RESET

Die MICRO hat keinen RESET-Anschluss. Der Reset wird als sogenannter POWER ON Reset unmittelbar nach Anlegen der Betriebsspannung durchgeführt. Der Hersteller des Controllers fordert, dass für einen RESET die Spannung am Controller unter 100mV fallen muss. Es ist daher erforderlich für einen Reset die MICRO komplett von der Stromversorgung zu trennen, ebenso bedeutet dies dass alle Ports ebenfalls spannungslos sein müssen, um zu verhindern, dass Spannung über die Ports zum Controller zurück gespeist wird.

Ausserdem hat die MICRO eine LOW VOLTAGE INHIBIT (LVI) Funktion, die sicherstellt, dass der Controller unterhalb einer Spannung (4.8V) bei welcher er nicht mehr sicher arbeiten kann, in den RESET geht.

Es muss also sichergestellt werden, dass die Betriebsspannung im normalen Betrieb nicht unter diesen Wert fällt.

DOWNLOAD - P3

P3 ist in seiner Funktion als Digitalport als einziger auf eine Input - Funktion beschränkt und hat auch keine weiteren alternativen Funktionen. P3 ist der Kontroll-Port der den Download einleitet wenn er beim Anlegen der Betriebsspannung auf LO liegt. Danach kann er als regulärer Digitalport (nur Eingang !) betrieben werden. Der Anwender muss aber dafür sorgen, dass der Port zu diesem Zeitpunkt, bis 100ms nach Anlegen der Betriebsspannung, HI ist.

Verwenden Sie Port 3 niemals in Anwendungen bei welchen nicht sichergestellt ist, dass Port 3 beim Anlegen der Betriebsspannung log. high ist, da sonst der Download-Modus aktiv ist und das Anwender-programm nicht läuft.

FREQ - P4

Dieser Port hat, neben weiteren allgemeinen, alternativen Funktionen, als einziger Port die Möglichkeit Frequenzen zu messen

RXD-P2, TXD-P1

P1 und P2 stellen die serielle Schnittstelle dar, über welche in Download Modus das Programm geladen wird. In der Anwenderschaltung können diese Ports regulär mit ihren alternativen Funktionen oder eben als serieller Eingang (RXD) oder Ausgang (TXD) zum austausch von Daten mit anderen Geräten benutzt werden.

SPECIAL FEATURES

Für viele Anwendungen ist es wichtig die Stromaufnahme auf ein Minimum zu reduzieren. Allgemein geschieht dies durch die Reduzierung des Systemtaktes. Bei der MICRO ist das so in dieser Form nicht möglich. Hier bewirkt der Befehl SLOWMODE einen STOP des Systemtaktes für 20ms (während dieser Zeit ist die Stromaufnahme < 1uA) danach wird mit der normalen Abarbeitung des Programms fort-gefahren.Sinnvoll eingestzt kann dies immer werden, wenn vom Prozessor keine koninuierliche Rechenleistung verlangt wird.
Der STOP Modus ist einzigartig und wird vom Compiler nicht direkt unterstützt. Die notwendigen Token werden so erzeugt:

```
ADDTOKEN 9
ADDTOKEN 255
```

Ein typisches Beispiel ist die Temperaturüberwachung mit Alarmfunktion. Hier verbringt die CPU die Zeit mit dem Warten auf den Alarmfall. Das Beispiel demonstriert so eine Anwendung in einer einfachen Form
Eine grobe Berechnung (Annahme 100us pro Instruktion) ergibt dass die CPU jeweils 500us läuft und 20ms im Stop Modus ist. Das lässt eine Absenkung der Stromaufnahme (von 5mA im regulären RUN Modus) auf 120 uA (1/41) erwarten.

```
DEFINE TEMP         as AD[1]
DEFINE ALERT  as PORT[2]

#WATCH_ADC
if ADC6 > 100 then goto ALERT else goto NOALERT
ADDTOKEN 9
ADDTOKEN 255
goto WATCH_ADC

#ALERT
ALERT=ON
goto WATCH_ADC

#NOALERT
ALERT=OFF
goto WATCH_ADC
```

GB

Imprint