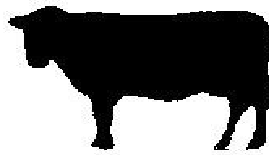


Great Cow BASIC

für

MICROCHIP

PIC - Microcontroller

GCBASIC – COMPILER

und

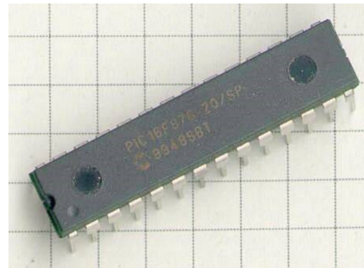
GCBIDE - EDITOR

Dokumentation

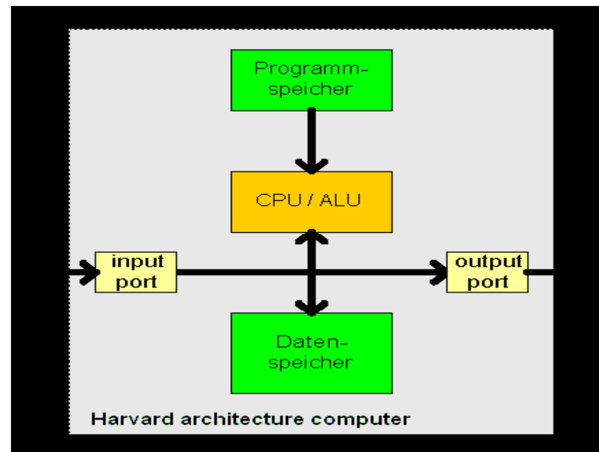
by Hugh Considine (Main developer)

Translation by LS – Baiersbronn

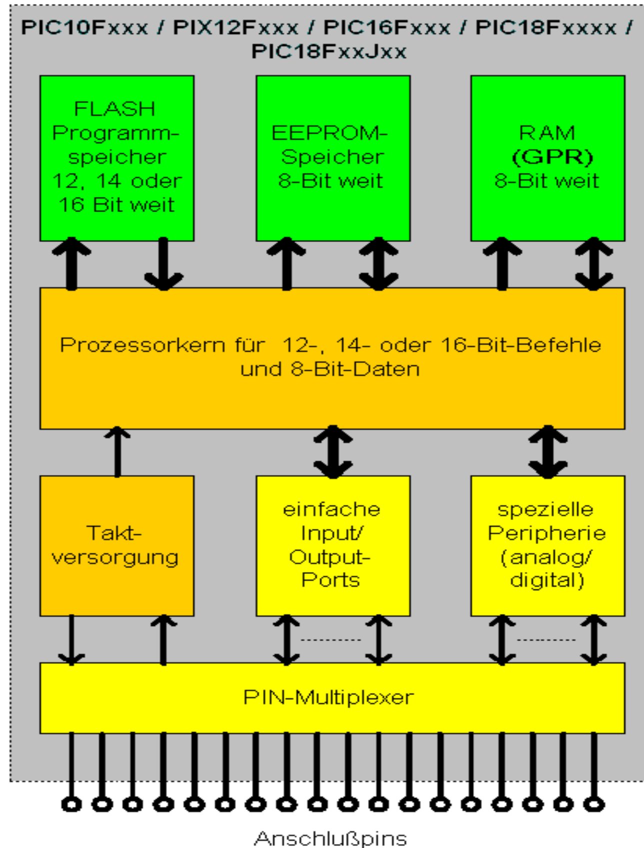
Der PIC



Seine Architektur

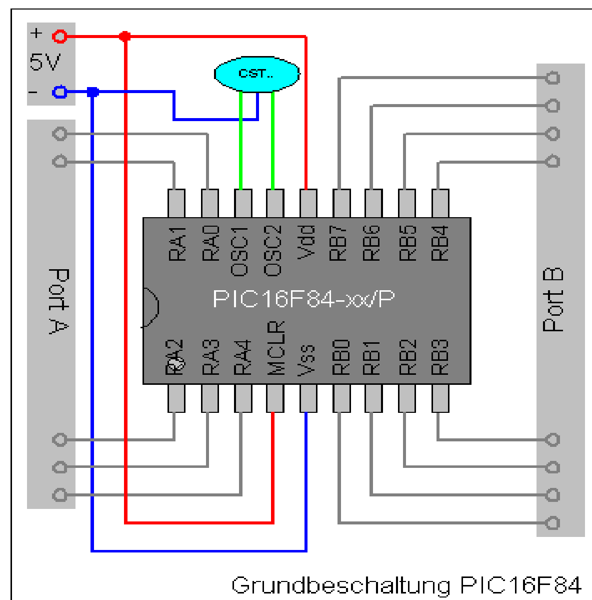


Das Blockschaltbild



Was braucht man an Pheripherie?

Minimalkonfiguration



Zum Programmieren wird wie nachfolgend erläutert **GCBASIC** und **GCBIDE** verwendet.

Zum Brennen des PIC kann unter www.sprut.de verschiedene kostenlose Brenner z.B. **Benner8P inkl. Brenn – Software** geladen werden. Diese Homepage ist gleichfalls auch ***sehr zu empfehlen*** um sich tiefer in die PIC Mikrocontroller und deren Programmierung in Assembler einzuarbeiten.

Gleichfalls kann natürlich auf der **GCBASIC Homepage** unter www.gcbasic.sourceforge.net die komplette Unterstützung (englisch) hierzu herangezogen werden.

Im WEB erhält man umfangreiche Unterstützung zu diesen Mikrocontrollern, so dass hier nun nicht mehr im Einzelnen darauf eingegangen wird. Viele User haben hier in vielfältigen Abhandlungen Ihre Projekte vorgestellt, so dass man meist immer etwas findet um seine eigenen Projekte erfolgreich zum Abschluss zu bringen. Man findet sich hier schnell in einem Pool von Anwendern.

Trotz sorgfältiger Bearbeitung der vorliegenden Dokumentation können Fehler nicht ganz ausgeschlossen werden.
Herausgeber und Autor können für fehlerhafte Angaben und deren Folgen weder eine juristische Verantwortung noch irgendeine Haftung übernehmen.

Syntax

Arrays

Über Arrays

Ein Array ist ein spezieller Typ von Variablen in welcher man verschieden Werte speichern kann. Es ist im Grund eine Liste von Nummern in welche jede von Ihnen durch einen Index adressiert werden kann. Der Index ist ein wert welcher eingeklammert hinter dem Namen des Array steht.

Examples of array names are:

Array/Index	Meaning
Fish(10)	Element 10 of an array called Fish
DataLog(2)	The second number in an array named DataLog
ButtonList(Temp)	An element in the array "ButtonList" that is selected according to the value in the variable "Temp"

Verwendung von Arrays

Beim verwenden von Arrays wird erst der Name bestimmt dann der Index. Arrays könne überall eingesetzt werden wo auch eine normale Variable angewendet wird.

For more help, see:
Declaring arrays with DIM

Conditions

In GCBASIC (und auch bei den meisten anderen Programmiersprachen) ist condition eine Aussage die entweder wahr oder falsch sein kann. Conditions werden verwendet wenn das Programm eine Unterscheidung machen muss.
Eine Condition wird generell als Wert oder Variable angegeben. Eine relative Operation (wie bei = or >) und andere Werte oder Variablen. Mehrere Conditions können zu einer kombiniert werden durch die Verwendung von logischen Operanden wie AND und OR.

GCBASIC supports these relative operators:

Symbol	**Meaning**
=	Equal
<>	Not Equal
<	Less Than
>	Greater Than
<=	Less than or equal to
>=	Equal to or greater than

In addition, these logical operators can be used to combine several conditions into one:

Name	**Abbreviation**	**Condition true if**
AND	&	both conditions are true
OR	|	at least one condition is true
XOR	#	one condition is true
NOT	!	the condition is not true

NOT ist etwas anderst als die anderen logischen Operanden in dem nur eine andere Codition verwendet wird. Andere arithmetische Opeaden können in Conditions kombiniert sein, so z.B. zum wechseln von Werten bevor diese verglichen werden.

GCBASIC hat zwei eingebaute Conditions – TRUE , welche immer wahr ist und FALSE, welche immer falsch ist. Dies kann beim erstellen von Endlosschleifen verwendet werden.

Es ist auch möglich mit Conditions einzelne Bits zu testen. Zum Durchführen werden die Bits beispielsweise auf 1 oder 0 (ON oder OFF) getestet. Zur Zeit ist es nicht möglich Bit Test mit anderen Conditions wie NOT, AND,OR; und XOR zu kombinieren.

Example conditions:

Condition	Comments
Temp = 0	Condition is true if Temp = 0
Sensor <> 0	Condition is true if Sensor is not 0
Reading1 > Reading2	True if Reading1 is more than Reading2
Mode = 1 AND Time > 10	True if Mode is 1 and Time is more than 10
Heat > 5 OR Smoke > 2	True if Heat is more than 5 or Smoke is more than 2
Light >= 10 AND (NOT Time > 7)	True if Light is 10 or more, and Time is 7 or less
Temp.0 ON	True if Temp bit 0 is on

Über Defines

Define ist eine Directive, welche dem Compiler mitteilt ein bestimmtes Word durch ein anderes oder eine Nummer zu ersetzen.

Es ist also möglich define zum spezifizieren von Ports zu verwenden – demnach kann defines benutzt werden zur Unterstützung beim erstellen von Codes, was dann leicht auch bei anderen PICs und verschiedenen Ports anwendbar ist.

In GCABSIC gibt es umfangreiche interne Anwendungen von defines. Zum Beispiel benutzt der LCD Code defines zum setzen der Ports für die Kommunikation mit dem LCD.

Anwendung Defines

Zum erstellen eines define ist es wichtig die #define Directrive zu verwenden. Im folgenden einige Beispiele.

```
#define Line 34
#define Light PORTB.0
#define LightOn Set PORTB.0 on
```

Line ist eine einfache Konstante – GCBASIC erkennt „Line“ im Programm und ersetzt es durch die Nummer 34. Das kann nun in der Zeile eines folgenden Programms verwendet werden, es macht es einfacher für spätere Einstellungen des Programms.

Light ist ein Port – es entspricht einem bestimmten Pin des PIC Chip.

LightOn ist ein define und macht das Programm leserlicher. Anstatt immer wieder zu schreiben „Set PORTB.0 on“ macht es dies möglich nur zu schreiben „LightOn“ und der Compiler macht den Rest.

Functions

Über Functions
Functions ist ein spezieller Typ eines Unterprogramms, welcher einen Wert zurückgeben kann. Das bedeutet, dass wenn der Name der Funktion bei einer Variablen verwendet wird, ruft GCBASIC die Funktion auf, erhält einen Wert davon und setzt dann den Wert in die Zeile des Code am Ort der Variablen.

Function beinhaltet also Parameter, die auf die gleiche Weise behandelt werden als Parameter für Unterprogramme. Die einzige Ausnahme ist das Klammern um einige Parameter sein müssen wenn eine Funktion aufgerufen wird.

Gebrauch von Functions
This program uses a function called AverageAD to take two analog readings, and then make a decision based on the average:

```
'Select chip
#chip 16F88, 20

'Define ports
#define LED PORTB.0
#define Sensor AN0

'Set port directions
dir LED out
dir PORTA.0 in

'Main code
Main:
	Set PORTB.0 Off
	If AverageAD > 128 Then Set PORTB.0 On
	wait 10 ms
goto Main

function AverageAD
'Get 2 readings
Temp = ReadAD(Sensor)
Temp = Temp + ReadAD(Sensor)

'Divide by 2 using rotate
rotate Temp right

'Return value
AverageAD = Temp

end function
```

See Also:
Subroutines

Interrupts

About Interrupts

Interrupts sind ein Bestandteil verschiedener Mikrocontroller. Sie ermöglichen eine temporäre Pause (Interrupt) der Code arbeitet und dann startet eine andere Einheit des Code wenn ein Ereignis ausgelöst wird. Ist das Ereignis beendet, kehrt es zurück wo es war und das Programm läuft weiter.

Viele Ereignisse können einen Interrupt triggern, wie z.B. der Timer erreicht seine Grenze, serielle Übertragung, oder ein spezieller Pin bekommt ein Signal.

Gebrauch des Interrupts

Es gibt zwei Möglichkeiten in GWBASIC Interrupts zu nutzen. Die erste Möglichkeit ist der Befehl On Interrupt. Dies erzeugt automatisch einen Interrupt und startet ein spezielles Unterprogramm wenn der Interrupt erscheint.

Die andere Art einen Interrupt zu erzeugen ist ein Unterprogramm mit dem Namen Interrupt zu schreiben. GCBASIC ruft dann immer dieses Unterprogramm auf wenn ein Interrupt erscheint und dann prüft der Code die „flag“ Bits um zu entscheide welcher Interrupt ausgeführt wird und was damit passiert. Wenn man diese Methode verwendet dann muss der gewählte Interrupt manuell freigegeben werden. Es ist wichtig, dass der Code die Flags löscht sonst wiederholt sich der Interrupt unendlich.

Einige Kombinationen dieser zwei Methoden ist möglich, der Code erzeugt bei On Interrupt eine Überprüfung ob ein Interrupt erkannt wurde. Wurde der Inerrupt erkannt, wird On Interrupt mit ihm arbeiten., wenn nicht, wird das Interrupt Unterprogramm den Interrupt weiter behandeln.

Durch einige Bereiche des Code ist es wünschenswert keine Interrupts zu verwenden. Wenn das der Fall ist dann wird der IntOff Befehl verwendet um den Interrupt abzuschalten am Anfang des Bereichs und IntOn um wieder am Ende zurück zuschalten. Wenn einige Interrupt Ereignisse eingetreten sind weil Interuppt abgeschaltet wird der Prozess weitergeführt sobald Interrupt zurück geschaltet ist. Wenn das Programm keinen Interrupt benutzt, dann wird von GCBASIC IntON und IntOff automatisch zurückgenommen.

See Also:
IntOff
IntOn
On Interrupt

Lookup Tables

Über Lookup Tables

Lookup Tabel ist eine Liste von Werten, die im Programmspeicher des Chip abgelegt sind und auf die mit dem ReadTable Befehl zugegriffen werden kann.

Der Vorteil von Lookup Tables ist da sie im Vergleich zu bestimmten gleichwertigen IF Aussagen effizienter arbeiten.

Using Lookup Tables

Zuerst muss die Tabelle erstellt werden, der Code zum erstellen eines Lookup Table ist einfach, eine Zeile mit „Table“ , dann der Name der Tabelle, eine Liste von Nummern (bis 255) und dann „End Table“.

Wenn dann die Tabelle erstellt ist können mit dem ReadTable Befehl die Daten davon gelesen werden. Der ReadTable Befehl verlangt den Namen dr Tabelle von der gelesen wird, den Ort von wo die Elemente abgerufen werden und eine Variable zum rückspeichern.

Element 0 der Lookup Table speichert die Dimension der Tabelle. Wenn der ReadTable Befehl über das Ende der Tabelle hinaus versucht zu lesen, werden unsinnige Resultate erzeugt.

For more help, see:
ReadTable

Scripts

Über Scripts

Ein Script ist ein kleiner Codebereich der durch GCBASIC gestartet wird wenn ein Programm compiliert wird. Die Hauptaufgabe besteht darin, Berechnungen bezüglich der verschiedenen Chipgeschwindigkeiten durchzuführen und das Programm entsprechend einzustellen.

Scripts werden nicht in den Mikrocontroller geladen – GCBASIC liest diese, startet es, entfernt es vom Programm und dann werden die jeweiligen Resultate als Konstante im Programm benutzt.

Intern werden die Script Konstanten wie Variablen behandelt. Script kann die Werte der Konstanten lesen und auch durch neue Werte ersetzen.

Verwendung von Script

Script wird mit „#script“ gestartet und mit „#endscript“ beendet. Intern, können sie aus 3 Befehlen bestehen:

- If
- Assignment (=)
- Error

Es ist gleichwertig ist zum IF Befehl im normalen GCBASIC Code, mit Ausnahme das es keine ELSE Anweisung hat.

= ist identisch zu dem im GCBASIC Programm. Die Konstante die gesetzt wird steht auf der linken Seite von = und der neue Wert steht rechts.

ERROR wird verwendet zur Anzeige einer Fehlerinformation. Alles nach dem ERROR Befehl wird am Ende der Compilierung angezeigt und im Fehlerprotokoll gespeichert.

Beispiel Script

This script is used in the pwm.h file. It takes the values of the constants PWM_Freq, PWM_Duty and ChipMHz, and uses the equations shown in the PIC datasheets to calculate the correct values for the relevant system variables.

```
#script
PR2Temp = int((1/PWM_Freq)/(4*(1/(ChipMHz*1000))))
T2PR = 1
If PR2Temp > 255 Then
        PR2Temp = Int((1 / PWM_Freq) / (16 * (1 / (ChipMHz * 1000))))
        T2PR = 4
        If PR2Temp > 255 Then
        PR2Temp = Int(( 1 / PWM_Freq) / (64 * (1 / (ChipMHz * 1000))))
        T2PR = 16
        If PR2Temp > 255 Then
                Error Invalid PWM Frequency value
                End If
        End If
End If

        DutyCycle = (PWM_Duty * 10.24) * PR2Temp / 1024
        DutyCycleH = (DutyCycle AND 1020) / 4
        DutyCycleL = DutyCycle AND 3
#endscript
```

After this script has run, the values PR2Temp, DutyCycleH and DutyCycleL are used as constants to set up the required variables.

Subroutines

Über Subroutines

Eine Subroutine (Unterprogramm) ist ein kleines Programm innerhalb des Hauptprogramms. Typische Anwendungen von Soubroutines sind wenn ein Task von verschiedenen Stellen und zu verschiedenen Zeiten vom Hauptprogramm aus wiederholt wird.

Es gibt zwei Hauptanwendungen von Subroutines:

- Übersichtliche und einfache Programmführung
- Reduzierung der Größe der Programme unter Berücksichtigung der Codeabschnitte.

Wenn der PIC auf eine Subroutine trifft, speichert er den aktuellen Programm – Stand bevor er auf Start springt oder diese ausführt. Sobald es das Ende der Subroutine erreicht hat, kehrt es zum Hauptprogramm zurück und führt das Programm an der Stelle fort wo es verlassen wurde.

Normalerweise können Subroutunes auch von anderen Soubroutines ausgeführt werden. Diese Verschachtelung ist jedoch von Chip zu Chip begrenzt:

PIC Family	Instruction Width	Number of subs called
10F*, 12C5*, 12F5*, 16C5*, 16F5*	12	1
12C*, 12F*, 16C*, 16F*, except those above	14	7
18F*, 18C*	16	31

Diese Grenzen sind durch die Größe des Speichers des PIC bestimmt, auf welchen gesprungen wird befor eine neue Subroutine startet. Einige GCBASIC Befehle sind Subroutines, also sollten immer 2 oder 3 Subroutines mehr erlaubt sein.

Eine andere Eigenschaft von Subroutines ist, dass sie Parameter verarbeiten können. Dies sind Werte die vom Hauptprogramm in die Subroutine übernommen werden und am Ende wieder zurück.

Gebrauch der Subroutines

Eine Subroutine aufzurufen ist sehr einfach, alles was man braucht ist der Name der Sub und eine Liste von Parametern. Dieser Code ruft eine Sub mit Namen Buzz auf ohne Parameter :

```
Buzz
```

Wenn die Sub Parameter enthält werden diese nach dem Namen der Sub gelistet.Dies würde der Befehl zum Aufruf einer Sub mit Namen „MoveArm“ sein mit 3 Parametern :

MoveArm NewX, NewY, 10

Wenn eine Sub Parameter hat, werden diese durch Klammern hervorgehoben, s.u.:

MoveArm (NewX, NewY, 10)

Es dient lediglich der Darstellung des Code, hierbei ist es egal zu was der Code da ist. Entscheide welche Art dir zusagt und arbeite damit.

Eine Sub zu erstellen ist geradezu einfach. Da muss eine Zeile am Start sein mit „Sub" und dann der Name der Subroutine. Am Ende der Sub muss noch eine Zeile mit „end sub" stehen. Zum erstellen der Sub mit dem Namen „Buzz" nachfolgender Code:

sub Buzz

'code for the subroutine goes here

end sub

Wenn die Sub Parameter hat müssen diese nach dem Namen gelistet sein. Zum Beispiel , zur Definition des „MoveArm" benutze den erweiterten sub, folgenden Code:

sub MoveArm(ArmX, ArmY, ArmZ)

'code for the subroutine goes here

end sub

In dem erweiterten Sub ArmX, ArmY und ArmZ sind alles Variablen. Wenn die Erweiterung arbeitet haben die Variablen den Werte am Anfang von der Sub:

- ArmX = NewX
- ArmY = NewY
- ArmZ = 10

Bei beenden der Subroutine wird GCBASIC die Werte wenn möglich zurückkopieren. NewX zu ArmX und NewY zu ArmY. GCBASIC wird nicht versuchen die Nummer 10 ArmZ zuzuordnen.

Es ist möglich GCBASIC anzuweisen, die Werte nach der Beendigung der Sub nicht zurück zu kopieren. Wenn wie nachfolgend die Sub definiert wird:

sub MoveArm(In ArmX, In ArmY, In ArmZ)

'code for the subroutine goes here

end sub

GCBASIC kopiert die Werte zur Sub aber nicht zurück.

GCBASIC kann also das zurück kopieren verhindern, bei hinzufügen von „Out“ vor dem Parameter Name. Dies wird verwendet in der EEProm lese Routine, es wird hierbei nicht direkt ein Data Wert in die laufende Sub kopiert, so das Out verwendet wir um das verschwenden von Zeit und Speicher zu vermeiden. Der „#NR“ bedeutet „No Return“ und bei der Verwendung den gleichen Zweck wie der Zusatz „In“ vor jedem Parameter. Der Gebrauch von „#NR“ ist nicht zu empfehlen, da er nicht das gleiche Steuerungsniveou hat.

```
sub MoveArm(ArmX As Word, ArmY As Word, ArmZ As Word)
...
end sub
```

See Also:
Functions

Variables

Über Variables

Eine Variable ist ein Bereich des Speichers des Mikrocontrollers welcher zum speichern von Nummern oder Serien von Buchstaben verwendet wird. Dies ist für viele Vorhaben sinnvoll, wie das einlesen eines Sensors und dessen Verwendung oder das Zählen der Zeitspanne welche ein Roboter für eine bestimmte Aktion benötigt.

Jede Variable muss einen Namen erhalten, wie „MyVariable“ oder „PieCounter“. Das auswählen des Namens einer Variablen ist einfach, man darf keine Spaces oder Symbole einfügen (other than_) und man muss sicher gehen, dass der Name länger als 2 Character (Buchstaben und /oder Nummern) ist .

Variable Types

Es gibt mehrere Verschiedener Arten von Variablen und jede kann verschiedene Arten von Informationen beinhalten. Nachfolgend die Arten der GCBASIC Variablen:

Variable type	Information that this variable can store	Example uses for this type of variable
Bit	A bit (0 or 1)	Flags to track whether or not a piece of code has run
Byte	A whole number between 0 and 255	General purpose storage of data, such as counters
Word	A whole number between 0 and 65535	Storage of extra large numbers
Array	A list of whole numbers ranging from 0 to 255	Logs of sensor readings
String	A series of letters, numbers and symbols.	Messages that are to be shown on a screen

Gebrauch der Variablen

Byte Variablen benötigen keine speziellen Befehle zum setzen. Man setzt den Namen der Variablen in den Befehl den die Variable benötigt.

Andere Arten von Variablen können auf sehr ähnliche Art verwendet werden, ausgenommen sie müssen erst „dimensioned“ werden. Dies benötigt dann den DIM Befehl damit GCBASIC mitgeteilt wird, dass es sich um etwas anderes als eine Byte Variable handelt.

Eine Schlüsseleigenschaft von Variablen besteht darin, dass es möglich ist einen Mikrocontroller Check auf die Variable zu machen und nur dann einen Codebereich zu starten wenn es einen Wert ergibt.

Variable Aliases

Einige Variablen sind Alias , welche zum Zuordnen von Speicherbereich bei der Verwendung von anderen Variablen benützt werden. Dies ist sinnvoll für anschließende festgelegte Byte Variablen um zusammen Word Variablen zu formen.

Alias sind nicht wie Zeiger in anderen Sprachen, sie müssen sich immer auf die gleiche Variable(n) beziehen und können sich nicht ändern.

For more help, see:
Declaring variables with DIM
Setting Variables

Doing things to individual bits of variables
SET
ROTATE

Checking variables and doing different things based on their value:
IF
DO
FOR
Conditions

GCBasic – Befehle

ReadAD

Syntax:

var = ReadAD *(PORT)*

Beschreibung

Mit **ReadAD** werden die Eingänge der AD Konverter eingelesen, welche einige PIC's benutzen. Hierbei ist ***PORT*** gleich dem Analogeingang AN0,AN1,AN2, usw. des jeweiligen PIC (siehe Datenblatt).

Beispiel:

```
'This program will turn on a led on PORTB.0 when the A/D converter

'on port AN0 returns a value of more than 120.

dir PORTA.0 in
dir PORTB.0 out

set PORTB.0 off
if ReadAD(AN0) > 120 then set PORTB.0 on
```

Messgenauigkeit

Eine Messung ist sinnlos, wenn man nichts über ihre Genauigkeit weiß. **Ein 10-Bit-ADC** kennt **1024 Abstufungen** zwischen Minimum und Maximum, was seine Meßgenauigkeit auf 0,1% limitiert. Da kein Bauteil perfekt ist, muß man noch mit 4..5 Stufen Linearitäts- und Offset-Fehler rechnen (näheres im Datenblatt). Damit kann der ADC aber immer noch 0,5% Genauigkeit im gesamten Meßbereich garantieren.

Die Messung kann nicht genauer sein, als die Referenzspannung, auf deren Erzeugung entsprechend Aufmerksanmkeit verwendet werden sollte. Wer die Referenzspannung aus der Betriebsspannung des PIC ableitet, sollte diese also gut stabilisieren. Ein 7805-Regulator kann 1% Genauigkeit erreichen.

Der Wert liegt also zwischen 0 und 1024

$Wert_{min} = U_{ref} / 1024$ mit U_{ref} = 5VDC ergibt sich 4,88mV / Step

ADOff

Syntax:

ADOff

Beschreibung:

ADCOff schält alle ADC Ports aus und setzt diese in den Digitalmodus.
Damit können die Ports nun als digitale IN/OUT Ports benutzt werden (siehe Beispiel Unten).

Dieser Befehl kann auch bei Störungen von Port A oder E gesetzt werden.

Beispiel:

```
'This program will turn on a LED on PORTA.1

#chip 16F877A, 20

'Set all pins on ports A and E to digital mode
ADOff

'Turn on port A bit 1
SET PORTA.1 on
```

EPRead

Syntax:

EPRead *(location,store)*

Mit **EPRead** werden Daten aus dem internen EEPROM - Speicher des PIC gelesen, welcher bei manchen PIC's zur Verfügung steht.
Location bezeichnet den Ort wo die Daten im EEPROM (Speicherzelle) gelesen werden und variiert hierbei entsprechend der verschiedenen Chips.
Store ist die Variable in welcher die Daten nach dem lesen des EEPROM's abgelegt werden.

Beispiel:

```
'This program will read and check a value from the EEPROM
'It will

EPRead(0, Temp)

IF Temp ! 255 THEN SET green ON
IF Temp = 255 THEN SET red ON
```

Viele PICs besitzen EEPROM-Zellen, in denen jeweils 1 Byte gespeichert werden kann. Im Unterschied zu den normalen Daten-Speicherzellen vergessen die EEPROM-Speicherzellen die in ihnen gespeicherten Informationen nicht beim Ausschalten der Stromversorgung. Hier lassen sich also Werte speichern, die immer wieder benötigt werden

Daten-EEPROM

Mit dem Ausschalten des PIC gehen sämtliche im Datenspeicher abgelegte Werte verloren. Oft ist es aber wünschenswert, einige Informationen oder Werte bis zum nächsten Einschalten zu speichern. Zum Beispiel Kalibrierdaten.

Für diesen Zweck besitzt der PIC einige EEPROM-Speicherzellen, die ihre Daten auch beim Abschalten der Stromversorgung halten. Jede Speicherzelle ist 1 Byte groß. Das Schreiben und Lesen dieser Speicherzellen erfolgt indirekt und benötigt deshalb mehrere Befehle. Auch ist das Schreiben langsam (4 ms).

Typ	Anzahl der EEPROM-Zellen (Bytes)
PIC12F629/675	128
PIC16F84	64
PIC16F873	128
PIC16F876	256

EPWrite

Syntax:

EPWrite *(location, data)*

Beschreibung:

Mit **EPWrite** werden Daten in den Datenspeicher des EEPROM geschrieben, so dass auf diese später mit dem Programm oder dem Befehl EPread zugegriffen werden kann.
Location bezeichnet den Ort wo die Daten im EEPROM (Speicherzelle) gelesen / geschrieben werden und variiert hierbei entsprechend der verschiedenen Chips.
Data sind die Daten welche in den EEPROM geschrieben werden und können Werte oder Variablen sein.

Beispiel:

```
'This program will take a measurement from Analog Input 2 every
'second, and then log the reading to EEPROM for future access.

DataPosition = 1

Main:

EPWrite(DataPosition, ReadAD(AN2))
DataPosition = DataPosition + 1

goto Main:
```

ProgramErase

Syntax:

ProgramErase *(location)*

Beschreibung:

ProgramEarse löscht Informationen aus dem Programmspeicher der Chips, welche diese Möglichkeit unterstützen.

Der größtmögliche Wert für ***location*** hängt von der Größe des Programmspeichers des jeweiligen PIC ab und kann im Datenblatt nachgelesen werden.
Dieser Befehl muss gesetzt sein bevor man in den Speicherblock schreibt. Es ist langsam im Vergleich zu anderen GCBasic Befehlen. Beachte, dass der Speicher in 32-Byte Blocks gelöscht wird, siehe Datenblatt.

Dies ist ein Befehl, welcher nur von fortgeschrittenen Programmierern verwendet werden sollte. Man muss vorsichtig mit dem Befehl umgehen, da es hierbei leicht zum löschen des PIC-Programms kommen kann.

ProgramRead

Syntax:

ProgramRead *(location, store)*

Beschreibung:

ProgramRead liest Informationen aus dem Programmspeicher der Chips, welche diese Möglichkeit unterstützen.
Location und ***store*** sind beides Word-Variablen, bedeutet das sie Werte über 255 speichern können.
Der größtmögliche Wert für ***location*** hängt von der Größe des Programmspeichers des jeweiligen PIC ab und kann im Datenblatt nachgelesen werden. ***Store*** ist 14 Bit breit wodurch Werte bis 16383 gespeichert werden können.

Dies ist ein Befehl, welcher nur von fortgeschrittenen Programmierern verwendet werden sollte.

ProgramWrite

Syntax:

ProgramWrite *(location, value)*

Beschreibung:

ProgramWrite schreibt Informationen in den Programmspeicher der Chips, welche diese Möglichkeit unterstützen. ***location*** und ***value*** sind beides Word-Variablen.

Der größtmögliche Wert für ***location*** hängt von der Größe des Programmspeichers des jeweiligen PIC ab und kann im Datenblatt nachgelesen werden. ***value*** ist 14 Bit breit wodurch Werte bis 16383 gespeichert werden können.

Dies ist ein Befehl, welcher nur von fortgeschrittenen Programmierern verwendet werden sollte.

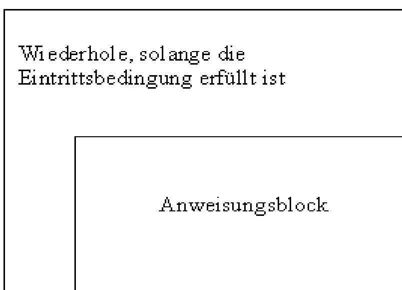
DO

Syntax:

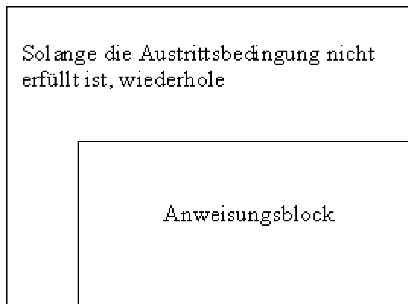
DO *((WHILE / UNTIL)* condition)

Beschreibung:

Der **DO** Befehl bewirkt, dass der Code zwischen dem DO und dem LOOP solange ausgeführt wird wie die ***WHILE*** oder ***UNTIL*** Bedingung war ist, abhängig von der Spezifikation.
Die Bedingungen für WHILE und UNTIL müssen angegeben werden, da sonst die Schleife endlos durchläuft.



DO ***While*** LOOP – Schleife: Sie wird solange ausgeführt wie eine Bedingung **erfüllt** ist



DO ***UNTIL*** LOOP – Schleife: Sie wird solange ausgeführt wie eine Bedingung **nicht erfüllt** ist.

Beispiel:

```
'This code will flash a light until the button is pressed

#define BUTTON PORTA.0
#define light PORTB.0

dir BUTTON in
dir light out

DO UNTIL BUTTON PRESSED
        SET light ON
        WAIT 1 s
        SET light OFF
        WAIT 1 s
LOOP
```

END

Syntax:

END

Beschreibung:

Stoppt das Programm.

Beispiel:

'This program will turn on the red light, but not the green light

```
SET red ON
END
SET green ON
```

EXIT SUB

Syntax:

EXIT SUB

Beschreibung:

Der Befehl beendet das Programm in dem momentanen Unterprogramm, so als würde es zum normalen Ende des Unterprogramms kommen.

Beispiel:

```
'Burglar Alarm subroutine

        SUB Burglar(sensor)
        IF sensor = clear THEN EXIT SUB
        SET buzzer ON
        SET LIGHT ON

END SUB
```

FOR

Syntax:

FOR *counter* = *start* TO *end* (*STEP increment*)
....
Programm Code
....
NEXT

Beschreibung:

Der **FOR** Befehl ist besonders bei Situationen geeignet wo ein Teil des Programm Codes eine bestimmte Anzahl mal wiederholt wird.
Der Programmcode innerhalb der Schleife wird so lange wiederholt bis die Bedingung erfüllt ist. Beim der ersten Ausführung ist der Wert ***start*** gesetzt. Beim erreichen von **Next** wird der ***counter*** wieder um 1 erhöht usw.

Wenn kein Wert für ***STEP*** angegeben ist, so wird der Wert jeweils um 1 erhöht.

Beispiel:

```
'This code will beep flash a green light 6 times.

#chip 16F88, 20

#define LED PORTB.0
dir LED out

FOR LoopCounter = 1 to 6

        SET LED ON
        WAIT 1 s
        SET LED OFF
        WAIT 1 s

NEXT
```

GOSUB

Syntax:

GOSUB *label*

Beschreibung:

Der **GOSUB** Befehl wird benutzt um zu einem ***label*** zu springen, in ähnlicher Art wie GOTO. Der Unterschied besteht darin, dass man den Befehl RETURN dazu benutzt um in das Programm nach GOTO zurückzukehren. Der Befehl ist im Aufbau gleich dem CALL.

Beispiel:

```
'This program will flash an LED on portb bit 0 and play a beep on
        'porta bit 4. until the robot is turned off.

#chip 16F628A, 4 'Change this to suit your circuit

#define Soundout PORTA.4

MainLoop:

        'Flash Light
        SET red ON
        WAIT 1 s
        SET red OFF
        WAIT 1 s

        'Beep
        GOSUB PlayBeep
GOTO MainLoop


PlayBeep:
        Tone 200, 10
        Tone 100, 10
RETURN
```

GOTO

Syntax:

GOTO *label*

Beschreibung:

Der **GOTO** Befehl kehr zurück zum angegebenen ***label***. Der GOTO Befehl wird auch für endlos Schleifen eingesetzt. Dem ***label*** wird ein ***(:)*** nachgestellt.

Beispiel:

```
'This program will flash PORTB bit 0 until the robot is turned
        'off. Notice the label named MainLoop.

#chip 16F628A, 4 'Change this line to suit your circuit

MainLoop:
        SET PORTB.0 ON
        WAIT 1 s
        SET PORTB.0 OFF
        WAIT 1 s
GOTO MainLoop
```

IF

Syntax:

Short form

IF *condition* THEN *command*

Long form

IF *condition* THEN
....
program code
....

END IF

Beschreibung:

Der **IF** Befehl ist der meist benutzte Befehl um Entscheidungen zu treffen. Wenn die IF ***condition*** **war** ist wird das ***command*** ausgeführt (short) oder der program code wird gestartet (long). Wenn sie **falsch** ist, so wird in die nächste Zeile gesprungen (short) oder in die Zeile nach END IF (long).

Beispiel:

```
'Burglar alarm code

Start:
        IF door = locked THEN SET green ON  'Short form
        IF door = broken THEN               'Long form
        SET green OFF
        SET red ON
        SET buzzer ON
        END IF
GOTO Start
```

REPEAT

Syntax:

REPEAT *times*
....
program code
....
END REPEAT

Beschreibung:

Der REPEAT Befehl ist ideal für Situationen wo der Programmcode für eine bestimmte Anzahl Wiederholt wird. Dieser benötigt weniger Speicher und ist schneller und kommt zur Anwendung wenn es nicht notwendig ist die Schleifendurchläufe zu zählen.

Beispiel:

```
'This code will beep flash a green light 6 times.

#chip 16F88, 20

#define LED PORTB.0
        dir LED out

REPEAT 6

        SET LED ON
        WAIT 1 s
        SET LED OFF
        WAIT 1 s

END REPEAT
```

SELECT

Syntax:

SELECT CASE *var*

Select Case Var1

Case Ausdruck1
' Variable Var1 hat den Ausdruck Ausdruck1

Case Ausdruck2
' Variable Var1 hat den Ausdruck Ausdruck2

Case Else
' Variable Var1 hat keinen der vorhandenen Ausdrücke

End Select

Beschreibung:

Eine **Select-Case-Anweisungen** ist eine elegantere Variante zu If-Then-Anweisungen, um mehrere hintereinander folgende Bedingungen abzufragen.

Es ist wichtig zu beachten, dass nur ein Bereich des Code ausgeführt wird bei der Verwendung von SELECT CASE.

Beispiel:

```
'Program to read a value from a potentiometer, and display a different word
' based on the result

#chip 18F4550, 20

'LCD connection settings
#define LCD_IO 4
#define LCD_DB4 PORTD.4
#define LCD_DB5 PORTD.5
#define LCD_DB6 PORTD.6
#define LCD_DB7 PORTD.7
#define LCD_RS PORTD.0
#define LCD_RW PORTD.1
#define LCD_Enable PORTD.2

DIR PORTA.0 IN

do
        Temp = ReadAD(AN0) / 20
cls
```

```
select case Temp

        case 0
        Print "None!"

        case 1
        Print "One"

        case 2
        Print "Two"

        case 3
        Print "Three"

        case 4
        Print "Four"

        case 5
        Print "Five"

        case else
        Print "A lot!"

end select

        wait 250 ms
loop
```

WAIT

Syntax:

Fixed Length Delay: **WAIT *time units***
Conditional Delay: **WAIT (*WIHLE / UNTIL*) *conditions***

Beschreibung:

Der WAIT Befehl hält das Programm für eine bestimmte ***(time units)*** Zeit an oder wenn die WHILE / UNIT Bedingung **war** ist.

When using the fixed-length delay, there is a variety of units that are available:

Unit	Length of unit	Delay range
us	1 microsecond	1 us - 255 us
10us	10 microseconds	10 us - 2.55 ms
ms	1 millisecond	1 ms - 255 ms
10ms	10 milliseconds	10 ms - 2.55 s
s	1 second	1 s - 255 s
m	1 minute	1 min - 255 min
h	1 hour	1 hour - 255 hours

The 10us and 10ms units exist as a work around for the 255 limit on variables.

Beispiel:

```
'This code will wait until a button is pressed, then it will flash
'a light every half a second.

WAIT UNTIL button pressed

FlashLight:

        SET green ON        'Turn on the light
        WAIT 50 10ms        'Wait for 50 x 10 (500) milliseconds
        SET green OFF       'Turn off the light
        WAIT 50 10ms        'Wait for 50 x 10 (500) milliseconds

GOTO FlashLight
```

LCD – Display

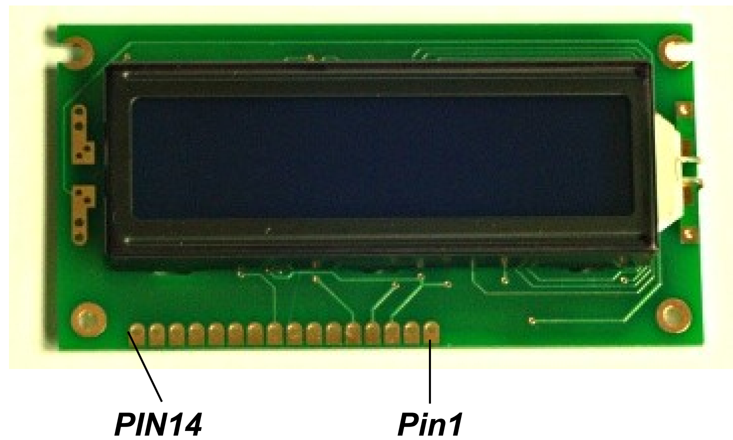
Wichtige Konstanten

Die Konstanten werden für die Kontrolleinstellungen innerhalb von GCBASIC benötigt. Zum Setzen, wird eine Zeile im Hauptprogramm mit ***#define*** der jeweilige Wert den Konstanten zugeordnet.
Einige Konstanten sind im 4 und 8 Bit – Mode andere für 4 Bit – Mode und andere für 8 Bit – Mode. Beim 2 Bit – Mode müssen nur drei Konstanten gesetzt werden, alle anderen sind zu ignorieren. Überprüfe die Spalte **Modes** zum festlegen der Konstanten.

Constant Name	Controls	Default Value	Modes
LCD_IO	The I/O mode. Can be 2, 4 or 8.	8	2, 4, 8
LCD_DB	The data pin used in 2-bit mode.	N/A - ***Must be set***	2
LCD_CB	The clock pin used in 2- bit mode.	N/A - ***Must be set***	2
LCD_RS	Specifies the output pin that is connected to Register Select on the LCD.	N/A - ***Must be set***	4, 8
LCD_RW	Specifies the output pin that is connected to Read/Write on the LCD. The R/W pin can be disabled*.	N/A - ***Must be set*** *(unless R/W is disabled)*	4, 8
LCD_Enable	Specifies the output pin that is connected to Read/Write on the LCD.	N/A - ***Must be set***	4, 8
LCD_DATA_PORT	Output port used to interface with LCD data bus	N/A - ***Must be set***	8 only
LCD_DB4	Output pin used to interface with bit 4 of the LCD data bus	N/A - ***Must be set***	4 only
LCD_DB5	Output pin used to interface with bit 5 of the LCD data bus	N/A - ***Must be set***	4 only
LCD_DB6	Output pin used to interface with bit 6 of the LCD data bus	N/A - ***Must be set***	4 only
LCD_DB7	Output pin used to interface with bit 7 of the LCD data bus	N/A - ***Must be set***	4 only

*Der R/W Pin kann deaktiviert werden durch setzen der LCD_NO_RW Konstanten. In diesem Fall wird R/W nicht mit dem Chip verbunden und die LCD_RW Konstante nicht gesetzt. Wenn der R/W Pin am LCD nicht benutzt wird diesen mit GND verbinden.

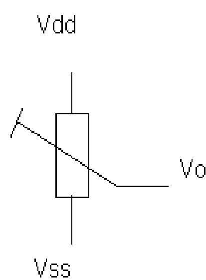
LCD 2x16 Characters (HD44780)



Pinbelegung

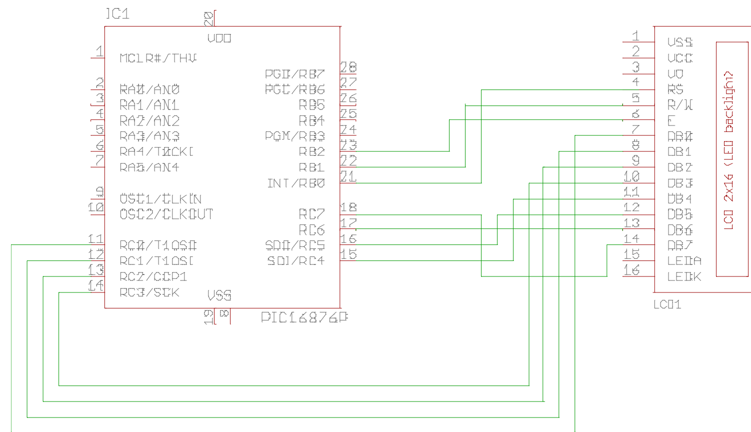
1	VSS (GND)
2	VDD (+5V)
3	Vo
4	RS
5	R/W
6	E
7	DB0
8	DB1
9	DB2
10	DB3
11	DB4
12	DB5
13	DB6
14	DB7

Anschlussbild LCD Kontrast

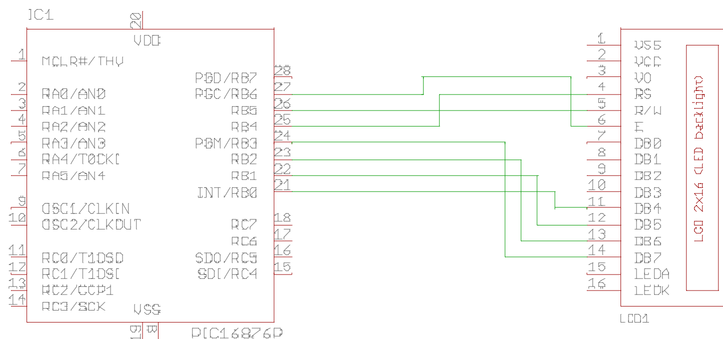


Beispiel 8 Bit – Mode (Pic 16f876)

(Syntax siehe z.B. Befehl PRINT)



Beispiel 4 Bit Mode



Beispiel Syntax:

```
'LCD connection settings 4 bit mode
#define LCD_IO 4
#define LCD_DB4         PORTB.0
#define LCD_DB5         PORTB.1
#define LCD_DB6         PORTB.2
#define LCD_DB7         PORTB.3
#define LCD_RS          PORTB.4
#define LCD_RW          PORTB.5
#define LCD_Enable      PORTB.6
```

PRINT

Syntax:

PPRINT *array()*

Beschreibung:

Der **PRINT** Befehl zeigt den Inhalt des ***array*** auf dem LCD und startet ab der momentanen Cursorposition. Das array sollte als String gesetzt sein bevor der Befehl aufgerufen wird.

Beispiel:

```
'A Hello World program for GCBASIC

'General hardware configuration
#chip 16F877A, 20

'LCD connection settings
#define LCD_IO 8
#define LCD_DATA_PORT PORTC
#define LCD_RS PORTD.0
#define LCD_RW PORTD.1
#define LCD_Enable PORTD.2

        PRINT "Hello World"
```

LOCATE

Syntax:

LOCATE ***line, column***

Beschreibung:

Der **LOCATE** Befehl wird verwendet um den Coursor im LCD in der Zeile ***(line)*** und der Spalte ***(column)*** zu platzieren.

Beispiel:

```
'A Hello World program for GCBASIC.
'Uses LOCATE to show "World" on the second line

'General hardware configuration
#chip 16F877A, 20

'LCD connection settings
#define LCD_IO 8
#define LCD_DATA_PORT PORTC
#define LCD_RS PORTD.0
#define LCD_RW PORTD.1
#define LCD_Enable PORTD.2

'Main routine
        PRINT "Hello"       'line 0 column 0
        LOCATE 1, 5         'line 1 column 5
        PRINT "World"
```

LCD – Display

H	e	l	l	o						0
					W	o	r	l	d	 1
0	1	2	3	4	5	6	7	8	9	

PUT

Syntax:

PUT ***line, column, character***

Beschreibung:

Der PUT Befehl schreibt den ASCII Code auf die durch line und column angegebene Stelle im LCD.

Beispiel:

```
'A scrolling star for GCBASIC.

'Misc Settings
#define ScrollDelay 250 ms

'General hardware configuration
#chip 16F877A, 20

'LCD connection settings
#define LCD_IO 8
#define LCD_DATA_PORT PORTC
#define LCD_RS PORTD.0
#define LCD_RW PORTD.1
#define LCD_Enable PORTD.2

'Main routine
FOR StarPos = 0 to 16
        if StarPos = 0 then
        PUT 0, 16, 32       'space
        PUT 0, 0, 42        'star (*)
end if
        if StarPos <> 0 then
        PUT 0, StarPos-1, 32
        PUT 0, StarPos, 42
end if
        Wait ScrollDelay
NEXT
```

ASCII – Tabelle

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177		DEL

GET

Syntax:

var = GET *line, column*

Beschreibung:

Die GET Funktion liest das ASCII Zeichen an der angegebenen Stelle des LCD und weist sie **var** zu.

GET ist im 2 Bit Mode nicht möglich.

CLS

Syntax:

CLS

Beschreibung:

Der CLS Befehl löscht den Inhalt des LCD und setzt den Courser in die linke obere Ecke des LCD.

Beispiel:

```
'A Flashing Hello World program for GCBASIC

'General hardware configuration
#chip 16F877A, 20

'LCD connection settings
#define LCD_IO 8
#define LCD_DATA_PORT PORTC
#define LCD_RS PORTD.0
#define LCD_RW PORTD.1
#define LCD_Enable PORTD.2

'Main routine
Main:
	PRINT "Hello World"
	Wait 1 sec
	CLS
	Wait 1 sec

goto Main
```

LCDCreateChar

Syntax:

LCDCreateChar *char, chardata()*

Beschreibung:

Der **LCDCreateChar** Befehl wird benutzt um einen benutzerdefinierten Character an das LCD zu senden.

Jeder Character des LCD besteht aus einer 8x5 Pixel – Matrix. Die Daten welche an das LCD gesendet werden bestehen aus einem 8 Elementen Datenfeld, wo jedes Element einer Zeile zugeordnet ist. Innerhalb jedes Elements bilden die 5 niedrigsten Bits die Daten für die Zeilenabhängigkeit. Wenn ein Bit gesetzt wird, wird ein Dot an die passende Stelle gezeichnet, wenn es gelöscht ist wird, wird kein Dot erscheinen.

Ein Datenfeld von mehr als 8 Elementen können benutzt werden, aber nur die ersten 8 können gelesen werden.

Char ist der ASCII Wert des Characters der gebildet wird. Der ASCII Code 0 bis 7 wird normalerweise für benutzerdefinierte Characters benutzt.

Chardata() ist ein Datenfeld welches die Daten für den Character beinhaltet.

Beispiel:

```
'This program draws a smiling face character

'General hardware configuration
#chip 16F877A, 20

'LCD connection settings
#define LCD_IO 8
#define LCD_DATA_PORT PORTC
#define LCD_RS PORTD.0
#define LCD_RW PORTD.1
#define LCD_Enable PORTD.2

'Create an array to store the character until it is copied
Dim TempArray(8)

'Set the array to hold the character
'Binary has been used to improve the readability of the code, but is not essential
TempArray(1) = b'00011011'
TempArray(2) = b'00011011'
TempArray(3) = b'00000000'
TempArray(4) = b'00000100'
TempArray(5) = b'00000000'
```

```
TempArray(6) = b'00010001'
TempArray(7) = b'00010001'
TempArray(8) = b'00001110'

'Copy the character from the array to the LCD
LCDCreateChar 0, TempArray()

'Draw the custom character
LCDWriteChar 0
```

LCDWriteChar

Syntax:

LCDWriteChar *char*

Beschreibung:

Der LCDWriteChar Befehl zeigt den spezifizierten Character an der angegebenen Position am LCD an.

Char ist der ASCII Wert des angezeigten Characters. Bei den meisten LCD's sind die Charters 0 bis 7 benutzerdefiniert und können mit dem LCDCreateCar Befehl gesetzt werden.

Beispiel:

```
'This program draws a smiling face character

'General hardware configuration
#chip 16F877A, 20

'LCD connection settings
#define LCD_IO 8
#define LCD_DATA_PORT PORTC
#define LCD_RS PORTD.0
#define LCD_RW PORTD.1
#define LCD_Enable PORTD.2

'Create an array to store the character until it is copied
Dim TempArray(8)

'Set the array to hold the character
TempArray(1) = b'00011011'
TempArray(2) = b'00011011'
TempArray(3) = b'00000000'
TempArray(4) = b'00000100'
TempArray(5) = b'00000000'
TempArray(6) = b'00010001'
TempArray(7) = b'00010001'
TempArray(8) = b'00001110'

'Copy the character from the array to the LCD
LCDCreateChar 0, TempArray()

'Draw the custom character
LCDWriteChar 0
```

LCDInt

Syntax:

LCDInt *value*

Beschreibung:

Der **LCDInt** Befehl zeigt den spezifizierten Wert im an der angegebenen Position am LCD an.

LCDInt ist begrenzt auf Werte zwischen 0 und 255 - für größere Werte muss der LCDWord Befehl benutzt werden.

Beispiel:

```
'A Light Meter program.

'General hardware configuration
#chip 16F877A, 20
#define LightSensor AN0

'LCD connection settings
#define LCD_IO 8
#define LCD_DATA_PORT PORTC
#define LCD_RS PORTD.0
#define LCD_RW PORTD.1
#define LCD_Enable PORTD.2

        CLS
PRINT "Light Meter:
        locate 1,2
PRINT "A GCBASIC Demo"
        wait 2 s

do while true
        CLS
 PRINT "Light Level: "
 LCDInt(ReadAD(LightSensor))
        wait 25 10ms
loop
```

LCDHex

Syntax:

LCDHex *value*

Beschreibung:

Der **LCDHex** Befehl zeigt den spezifizierten Wert im an der angegebenen Position am LCD an.

Beispiel:

```
'A program to count from 1 to FF on an LCD screen.

'General hardware configuration
#chip 16F877A, 20

'LCD connection settings
#define LCD_IO 8
#define LCD_DATA_PORT PORTC
#define LCD_RS PORTD.0
#define LCD_RW PORTD.1
#define LCD_Enable PORTD.2

For Counter = 1 to 255
	cls
	LCDHex Counter
	wait 25 10ms
next
```

LCDWord

Syntax:

LCDWord *value*

Beschreibung:

Der **LCDWord** Befehl zeigt den spezifizierten Wert im an der angegebenen Position am LCD an. LCDWord akzeptiert beide Variablen von 8 und 16 Bit verarbeiten und erlaubt Werte bis 65535.

LCDWord benötigt erheblich mehr Speicher und braucht mehr Zeit als der LCDInt Befehl. Wenn der größte Wert kleiner als 255 ist sollte der LCDInt Befehl bevorzugt werden.

Beispiel:

Siehe LCDInt

Keypad

Wichtige Konstanten

Diese Konstanten werden für die Steuereinstellungen des Lese-Code des Keypad im GCBASIC gebraucht. Zum setzen wird eine Zeile im Hauptprogramm platziert und mit #define ein Wert der jeweiligen Konstanten zugeordnet.

Constant Name	Controls	Default Value
KeypadPort	The port on the PIC chip that the keypad is connected to.	N/A **(must be set by user)**

To use these routines, the keypad must be connected as follows:

PIC Port pin	Keypad connector
0	Row 1
1	Row 2
2	Row 3
3	Row 4
4	Column 1
5	Column 2
6	Column 3
7	Column 4

Auch eine 3x3 (3x4) Keypad-Matrix kann mir dieser Routine gesteuert werden, wobei die nicht verwendeten Pins am PIC mit einem **pull up Widerstand** versehen werden.

KeypadData

Syntax:

***var* = KeypadData**

Beschreibung:

Die Funktion gibt einen Wert zurück in Abhängigkeit der gedrückten Taste. Beachte, dass wenn mehrere Tasten gedrückt werden wird lediglich ein Wert zurückgegeben.

Var kann eine der folgenden Werte haben.

Value	Constant Name	Key Pressed
0		0
1		1
2		2
3		3
4		4
5		5
6		6
7		7
8		8
9		9
10	KEY_A	A
11	KEY_B	B
12	KEY_C	C
13	KEY_D	D
14	KEY_STAR	Asterisk/Star (*)
15	KEY_HASH	Hash (#)
255	KEY_NONE	None

Beispiel:

```
'Program to show the value of the last pressed key on the LCD
#chip 18F4550, 20

'LCD connection settings
#define LCD_IO 4
#define LCD_DB4 PORTD.4
#define LCD_DB5 PORTD.5
#define LCD_DB6 PORTD.6
#define LCD_DB7 PORTD.7
#define LCD_RS PORTD.0
#define LCD_RW PORTD.1
#define LCD_Enable PORTD.2

'Keypad connection settings
#define KeypadPort PORTB

'Main loop
Do
'Get key
Temp = KeypadData

'If a key is pressed, then display if
If Temp <> KEY_NONE Then
        CLS
        LCDInt Temp
        Wait 100 ms
End If
Loop
```

KeypadRaw

Syntax:

***largevar* = KeypadRaw**

Beschreibung:

Die Funktion gibt einen 16 Bit Wert zurück in welchem jedes Bit einer entsprechenden Taste zugewiesen ist. Wenn die Taste gedrückt ist hat das Bit eine 1 und nicht gedrückt eine 0.

Die nachfolgende Tabelle zeigt die Zuordnung der Bits zu den jeweiligenTasten.

Bit	Key Position (row, col)	Common Key Symbol
15	1,1	1
14	1,2	2
13	1,3	3
12	1,4	A
11	2,1	4
10	2,2	5
9	2,3	6
8	2,4	B
7	3,1	7
6	3,2	8
5	3,3	9
4	3,4	C
3	4,1	*
2	4,2	0
1	4,3	#
0	4,4	D

Beispiel:

```
'Program to show the keypad status using LEDs
#chip 16F877A, 20

'Keypad connection settings
#define KeypadPort PORTB

'LEDs
#define LED1 PORTC
#define LED2 PORTD
dir LED1 out
dir LED2 out

'Declare a 16 bit variable for the key value
dim KeyStatus As Word

'Main loop
do
'Get key
KeyStatus = KeypadRaw
'Display
LED1 = KeyStatus_H 'High Byte
LED2 = KeyStatus 'Low Byte

loop
```

Wichtige Konstanten

Diese Konstanten werden für die Steuereinstellungen der Pulsweitenmodulations – Module des jeweiligen PIC Bausteins verwendet. Zum setzen wird eine Zeile im Hauptprogramm platziert und mit #define ein Wert der jeweiligen Konstanten zugeordnet.

Beachte, dass es zwei Sets von Konstanten gibt. Eine für Hardware PWM und eine für Software PWM. Hardware WM setzt ein CCP – Modul auf dem PIC Baustein voraus. Software PWM stellt keine weiteren Ansprüche an den PIC Chip.

Hardware PWM

Die Konstanten sind ausschließlich für PWMOn, HPWM und PWMOff und es werden keine anderen Konstanten benutzt.

Constant Name	Controls	Default Value
PWM_Freq	Specifies the output frequency of the PWM module in KHz.	38
PWM_Duty	Sets the duty cycle of the PWM module . Given as percentage.	50

Die Harware PWM ist nur erlaubt mit den „CCP1“ oder „CCP“ Pin. Es begrenzt sich auf die Hardware des PIC Mikrocontrollers.

Software PWM

Constant Name	Controls	Default Value
PWM_Delay	The PWM Period. The length of any delay used will be multiplied by 255. If no value is specified, no delays will be inserted into the PWM routine.	Not defined - no delay
PWM_Out*n*	The port physical port on the PIC that corresponds to channel *n*. *n* can represent 1, 2, 3 or 4.	Not Defined

More than 4 channels are possible, but for this the PWMOut routine in include\lowlevel\stdbasic.h must be altered.

PWMOff

Syntax:

PWMOff

Beschreibung:

Dieser Befehl deaktiviert die Ausgänge der PWM – Module des PIC Chip.

Beispiel:

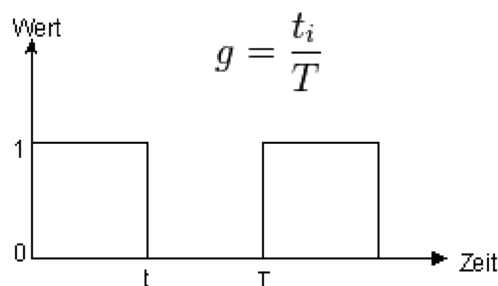
```
'This program will enable a 76 Khz PWM signal, with a duty cycle
'of 80%. It will emit the signal for 10 seconds, then stop.

#define PWM_Freq 76      'Set frequency in KHz
#define PWM_Duty 80      'Set duty cycle to 80 %

PWMOn                    'Turn on the PWM
        WAIT 10 s        'Wait 10 seconds
PWMOff                   'Turn off the PWM
```

Anmerkung:

Frequenz = 76 kHz
Tastverhältnis = 80%



$$g = \frac{t_i}{T}$$

mit

g: **Tastverhältnis**
ti: **Impulsdauer**
T: **Periodendauer**

PWMOn

Syntax:

PWMOn

Beschreibung:

Dieser Befehl aktiviert die Ausgänge der PWM – Module des PIC Chip.

Beispiel:

```
'This program will enable a 76 Khz PWM signal, with a duty cycle
'of 80%. It will emit the signal for 10 seconds, then stop.

#define PWM_Freq 76      'Set frequency in KHz
#define PWM_Duty 80      'Set duty cycle to 80 %

PWMOn                    'Turn on the PWM
        WAIT 10 s                'Wait 10 seconds
PWMOff                   'Turn off the PWM
```

PWMOut

Syntax:

PWMOut ***channel, duty cycle, cycles***

Beschreibung:

Der Befehl wird benutzt für eine Software PWM - Routine innerhalb von GCBASIC zur Erzeugung eines PWM – Signals am ausgewählten PORT des PIC. Diese Routine erfordert kein PWM – Modul auf dem Chip.

Channel setzt den Kanal welcher das PWM Signal erzeugt. Diesen muss man definiert haben bevor die Konstanten **SoftPWM1, SoftPWM2**, etc. gesetzt werden. Es sind maximal **4 Kanäle** erlaubt.

Duty cicle gibt das PWM Tastverhältnis an und geht von 0 – 255. 255 entspricht 100%, 127 <> 50%, 63 <> 25%, usw.

cycles wird benutzt um die Anzahl der PWM Pulse anzugeben die erzeugt werden. Dies wird benutzt bei einer spezifizierten Länge von Impulsen. Die folgende Formel beschreibt die Berechnung einer Periode.

TCYCLE = (28 + 10C)TOSC + 255PWM_Delay,

hierbei ist **C** die Nummer des benutzten Kanals und T_{OSC} ist die Länge der Zeit welche für die Ausführung eines Befehls des PIC benötigt wird (0,2µs bei 20 Mhz Chip, 1µs bei 4 MHz Chip). PWM_Delay ist eine Zeitspanne die mit der PWM_Delay Konstante angegeben wird.

Example:

```
'This program controls the brightness of an LED on PORTB.0
'using the software PWM routine and a potentiometer.

'Select chip model
#chip 16F877A, 20

'Set PWM Port
#define PWM_Out1 PORTB.0

'Set port directions
dir SoftPWM1 out    'PWM
dir PORTA.0 in      'Potentiometer

'Main routine
do
        PWMOut 1, ReadAD(AN0), 100   '100 cycles is a purely
                                     'arbitrary value here.
Loop
```

HPWM

Syntax:

HPWM ***channel, frequency, duty cycle***

Beschreibung:
Dieser Befehl ist zu Setzen des Hardware PWM – Moduls auf dem PIC Chip und zum generieren der PWM Signalform mit ***frequency*** und ***duty cycle***. Das PWM Signal wird solange erzeugt bis PWMOff gesetzt wird. Wenn man nur eine bestimmte Frequenz und Tastverhältnis benötigt, sollte stattdessen PWMOn und die Konstanten **PWM_Freq** und **PWM_Duty** verwendet werden.

Channel ist 1 oder 2 und abhängig von den Pins CCP1 bzw. CCP2. Chips mit nur einem CCP Port wird Pin CCP oder CCP1 immer verwendet und ***channel*** nicht berücksichtigt.

Frequency setzt die Frequenz am PWM Ausgang. Sie wird in kHz gemessen. Der maximal erlaubte Wert ist 255 kHz. Der untere Wert variiert in Abhängigkeit von der Taktfrequenz. 1 kHz ist das Minimum bei 16 Mhz Chips oder 2 kHz das Minimum bei 20 Mhz Chips. Wenn eine speziell Frequenz nicht erforderlich ist, sollte die PWM Frequenz ca. 1/500 der Taktfrequenz des PIC betragen. (40kHz<>20MHZ Chip, 16kHz<>8MHz Chip). Das ergibt das beste Ergebnis für das Tastverhältnis.

Duty cycle gibt das gewünschte Tastverhältnis des PWM Signals und reicht von 0 – 255 wobei 255 <> 100% Tastverhältnis.

Example:
```
'This program will alter the brightness of an LED using
'hardware PWM.

'Select chip model and speed
#chip 16F877A, 20

'Set the CCP1 pin to output mode
DIR PORTC.2 out

'Main code
do
'Turn up brightness over 2.5 seconds
	For Bright = 1 to 255
	HPWM 1, 40, Bright
	wait 10 ms
	next
'Turn down brightness over 2.5 seconds
	For Bright = 255 to 1
	HPWM 1, 40, Bright
	wait 10 ms
	next
loop
```

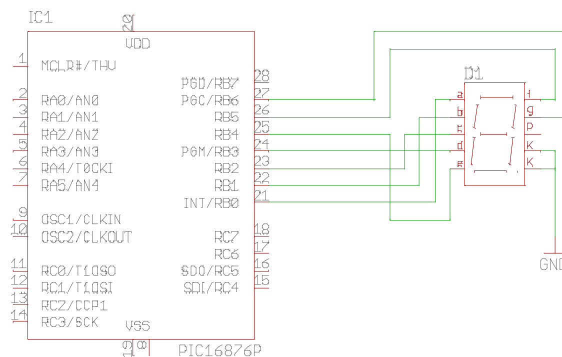
7-Segment Display

To use the **7-Segment display** routines supplied with GCBASIC, it is necessary to set these constants:

Constant Name	Controls	Default Value
DisplayCount	Specifies the number of 7-segment displays the program must control	1
DisplayPort*n*	Controls the output port used to control display *n*. *n* is A, B, C or D, corresponding to displays 1, 2, 3 and 4, respectively.	N/A - ***Must be set***
DispSelect*n*	The command used to select display *n*. Used to control addressing pins when several displays are multiplexed.	nop

To use these routines, the displays must be connected as follows:

PIC Port pin	Display Segment
0	A
1	B
2	C
3	D
4	E
5	F
6	G



DisplayValue

Syntax:

DisplayValue *(display, data)*

Explanation:

Der Befehl zeigt den jeweiligen Wert auf einem 7 Segment Display an. ***Desplay*** ist die Nummer des verwendeten Displays und ***data*** ist der angezeigte Wert zwischen 0 und 9.

Beispiel:

```
'This program will count from 0 to 9 on an LED display
'The display should be connected to PORTB

#define DisplayPortA PORTB

for Counter = 0 to 9
	DisplayValue 1, Counter
	Wait 1 sec
next
```

DisplayChar

Syntax:

DisplayChar *(display, character)*

Beschreibung:

Der Befehl zeigt den ASCII Code auf einem 7 Segment Display an. ***Desplay*** ist die Nummer des verwendeten Displays und ***character*** ist der angezeigte ASCII Character.

Beispiel:

```
'This program will show "Hello" on a LED display
'The display should be connected to PORTB

#define DisplayPortA PORTB

DIM Message(10)
Message() = "Hello "
for Counter = 0 to 6
DisplayChar 1, Message(Counter)
Wait 25 10ms
next
```

Serielle Schnittstelle (RS 232)

Bei der Seriellen Schnittstelle werden Daten seriell über die Leitungen TX und RX übertragen.

Signalerklärung der RS-232 Schnittstelle

RX
Empfangsdaten. Auf dieser Leitung werden die Datenbits vom Datenterminal (DTE) empfangen.

TX
Sendedaten. Auf dieser Leitung werden Daten vom Datenterminal (DTE) gesendet.

GND
Gehäusemasse (chassis ground). Das Datenterminal und das Modem müssen eine gemeinsame Masseverbindung haben um Masseschleifen etc zu verhindern.

DSR
Data Set Ready. Dieses Signal wird vom Modem ausgegeben und bedeutet, dass das Modem aktiv und betriebsbereit ist, um mit dem Datenterminal zu kommunizieren.

DTR
Data Terminal Ready. Dieses Signal wird vom Datenterminal an das Modem ausgegeben und bedeutet, dass das Datenterminal aktiv und betriebsbereit ist, um mit dem Modem zu kommunizieren.

CD
Data Carrier Detect oder *Carrier Detect*. Dieses Signal zeigt an, dass die Modems der beiden Seiten über die Telefonleitung miteinander verbunden sind und Daten über diese Verbindung austauschen können.

RTS
Request To Send. Dieses Signal wird vom Datenterminal ausgegeben und bedeutet, dass das Terminal Daten übertragen möchte.

CTS
Clear To Send. Ist das Antwortsignal des Modems an das Datenterminal auf ein RTS hin und zeigt an, daß das Modem bereit ist die Daten vom Terminal aufzunehmen und auf die Leitung umzusetzen.

SIG GND
Signalmasse (signal ground). Diese Masse dient als Referenzpotential für alle Signale. Je nach Gerät kann das ein von der Gehäusemasse getrenntes Potential sein, oder auch mit ihr verbunden sein.

RI
Ring Indicator. Ein Signal vom Modem zum Datenterminal, das anzeigt, dass der Telefonanschluß von einem externen Teilnehmer angewählt wurde d.h. daß das Telefon klingelt. Je nach Anwendung wird nach einer bestimmten Anzahl von Klingelimpulsen "abgehoben" (ist im Modem einstellbar).

Übertragungsverfahren

Wie sich erkennen lässt, sind die Datenleitungen invertiert, das heißt in diesem Fall, dass eine logische 0 durch eine Spannung von +3V bis +12V repräsentiert wird. Eine logische 1 hingegen durch eine Spannung von -3V bis -12V. Wenn keine Daten übertragen werden (Ruhezustand), dann liegen an der Datenleitung -3V bis -12V an. Jede Datenübertragung beginnt mit einem Startbit, welches dazu dient Sender und Empfänger zu synchronisieren. Der Pegel der Datenleitung wechselt also von logisch 1, dem Ruhezustand, auf logisch 0.

Anschließend werden fünf bis neun Bits übertragen, in der Regel jedoch, wie bereits erwähnt, acht. Die Bits werden "so wie sie sind" übermittelt, das heißt es werden keinerlei Synchronisierungsinformationen mitgeschickt.

Anschließend kann das Paritätsbit (Paritybit) folgen, welches dazu dient mögliche Übertragungsfehler zu entdecken. Man unterscheidet hierbei:

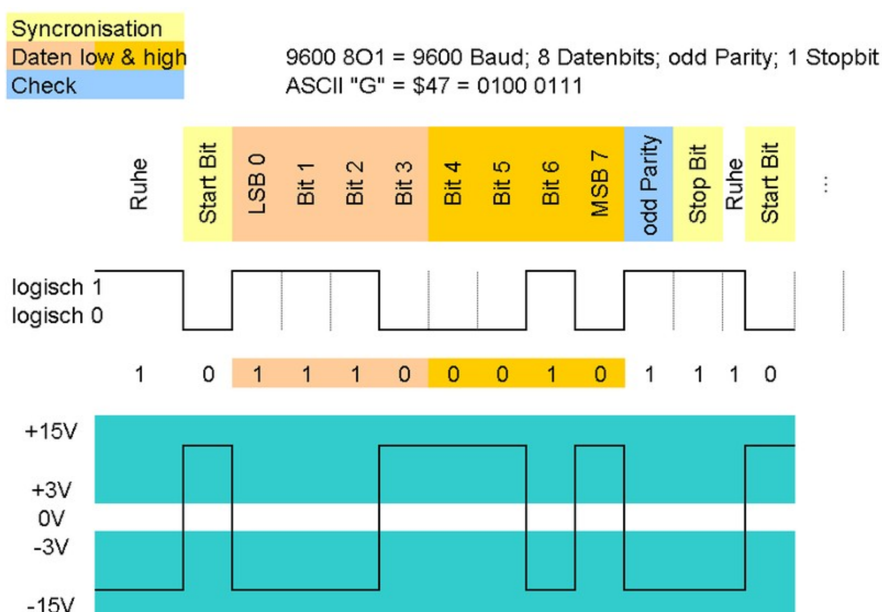
- Even-parität
- Odd-parität

Ist als Parität "even" gewählt, dann wird genau dann das Paritätsbit gesetzt, wenn die Anzahl der übertragenen Einsen in den Nutzdaten ungerade ist. Andernfalls ist es nicht gesetzt.

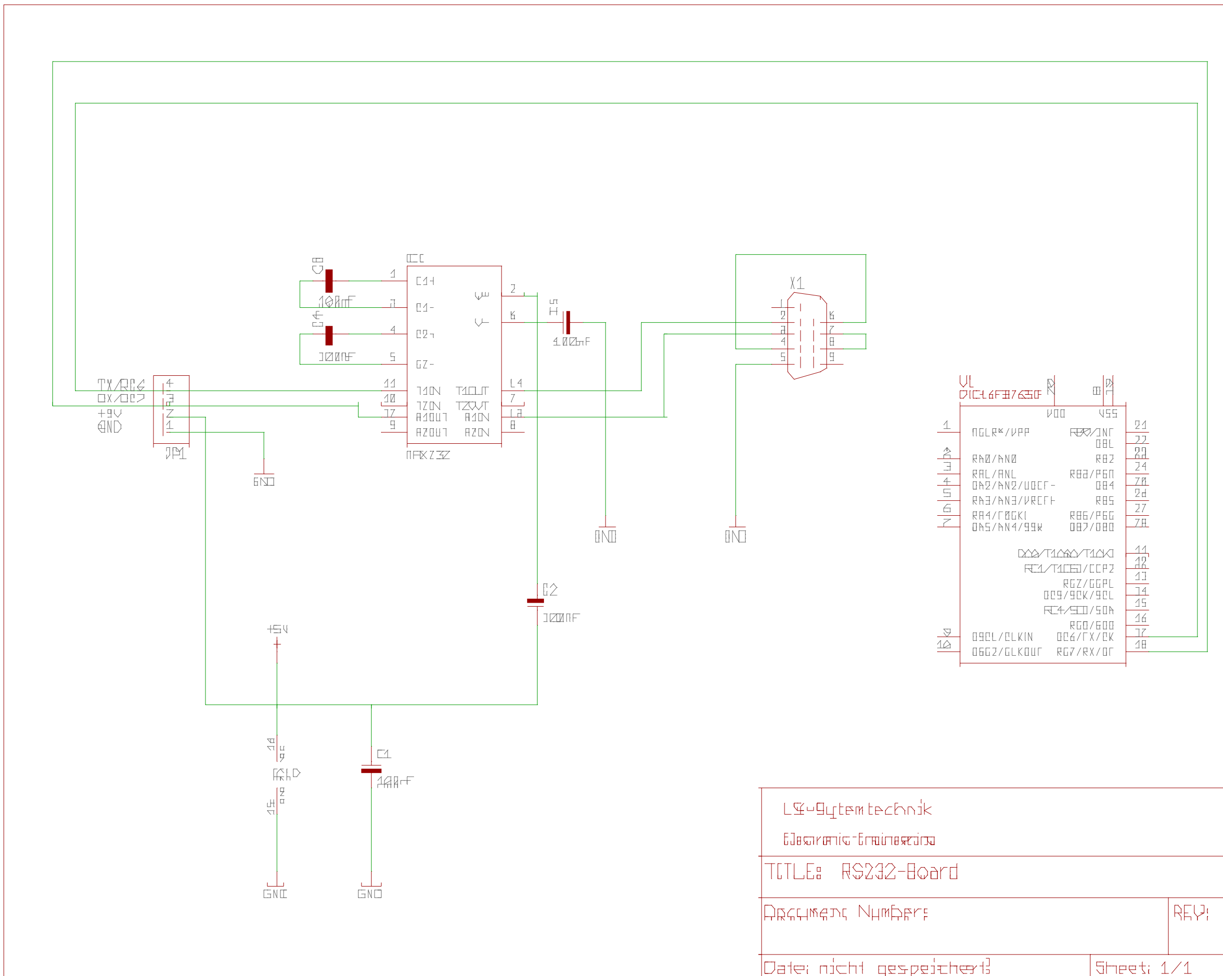
Wählt man jedoch "odd" als Parität, dann wird das Paritätsbit gesetzt, wenn die Anzahl der übertragenen Einsen in den Nutzdaten gerade ist.

Nach dem Paritätsbit folgen ein oder zwei Stoppbits, das bzw. die auch zur Synchronisierung dienen. Ein Stoppbit wird durch einen Pegel von -3 bis -12V dargestellt, also einer logischen 1. Nach dem (den) Stoppbit(s) folgt eine Ruhephase, deren Länge nicht bestimmt ist. Folglich darf nach einem Stoppbit sofort wieder das Startbit folgen, es darf aber auch unendlich lange gewartet werden, bis das nächste Startbit geschickt wird.

Timing Diagramm



Anschlussbeispiel mit PIC16f876 und MAX 232



Manche der PIC Bausteine haben eine eingebaute UART zur Seriellen Datenübertragung wie z.B der PIC 16f876.

Über den Pegelwandler **MAX 232** kann der PIC wie oben gezeigt an die RS 232 Schnittstelle angeschlossen werden.

Der 9 – Polige Stecker wird dann mit dem PC verbunden.

USART Routine für Hardware Kommunikation

Viele PIC's haben eine interne SR232 Hardware wie z.B. 16f876 usw. an Pin RC.6 / RC.7.
Mit o.A. Schaltung (MAX232) kann dann eine Kommunikation aufgebaut werden.
Nachfolgende Routine steuert die Kommunikation.

Hierbei wird der Wert von SPBRG_VAL wie folgt berechnent:

SPBRG_VAL = ((Fosc / 16) / Baudrate) - 1

```
' For Microchip PIC MCU's with hardware serial port

' Library for use with Great Cow Basic


'#define SPBRG_Val 51 '9600 Baud @8Mhz
#define SPBRG_Val 8 '57600 Baud @8Mhz

'Must be called once before any others
Sub InitUSART
SPBRG = SPBRG_Va
Set TXEN On 'enable transmission
Set BRGH On 'BRGH=1
Set SPEN On 'enable serial port
Set CREN On 'enable receive
End Sub

'Send a byte

Sub HSerSend(In SerData)
Do Until TXIF 'check if transmitter is busy
Loop
TXREG = SerData 'transmit data
End Sub

'Send a string

sub HserPrint(In PrintData$)
PrintLen = PrintData(0)
If PrintLen = 0 Then Exit Sub
For SysPrintTemp = 1 To PrintLen
HSerSend(PrintData(SysPrintTemp))
Next
End Sub

'Send value of byte as characters

sub HserPrintByte(In PrintValue)
ValueTemp = 0
If PrintValue >= 100 Then
ValueTemp = PrintValue / 100
HserSend(ValueTemp + 48)
PrintValue = PrintValue - ValueTemp * 100
End If
If ValueTemp > 0 Or PrintValue >= 10 Then
ValueTemp = PrintValue / 10
HserSend(ValueTemp + 48)
PrintValue = PrintValue - ValueTemp * 10
End If
```

```
HSerSend(PrintValue + 48)
End Sub

'Send value of byte as characters with CR and LF

Sub HserPrintByteCRLF(In PrintValue)
HserPrintByte(PrintValue)
HserSend(13)
HserSend(11)
End Sub

'Send CR and LF

Sub HserPrintCRLF
HserSend(13)
HserSend(11)
End Sub

'Receive a byte if avialable, otherwise return zero

Function HSerReceive
HSerReceive = 0
If RCIF On Then HSerReceive = RCREG
End Function
```

Bei Software Steuerung der Seriellen Schnittstelle

InitSer

Syntax:

InitSer *(channel, rate, start, data, stop, parity, invert)*

Beschreibung:

Dieser Befehl richtet die serielle Kommunikation mit den folgenden Parametern ein.

- ***Channel*** ist 1, 2 oder 3 und bezieht sich auf die I/O Ports welche für die Kommunikation benutzt werden.
- ***Rate*** ist die Bit – Rate, welche mit dem Buchstaben r angegeben wird und die gewählte Rate in bps. Akzeptable Einheiten sind r300, r600, r1200, r2400, r4800, r9600 und r19200.
- ***Start*** nennt die Nummer der Start Bits, welche gewöhnlich 1 ist. Der PC wartet auf das Start Bit bevor der Empfangsprozess beginnt, 128 zu start hinzufügen. (Beachte: Es währe wünschenswert die **WaitForStart** Konstanten hier zu benutzen)
- ***data*** teilt dem Programm mit wie viele Daten Bits gesendet oder empfangen werden. In den meisten Fällen sind dies 8 aber es können auch zwischen 1 und 8 inklusive sein.
- ***Stop*** ist die Nummer der Stop Bits. Wenn Start Bit 7 ein ist, dann wird diese Nummer ignoriert.
- ***Parity*** bezieht sich auf ein Fehlersystem Check, welcher bei einigen Bausteinen gebraucht wird. Sie können ungerade (müssen immer eine ungerade Nummer der high Bits sein). Gerade (müssen immer gerade Nummer der high Bits sein). Ohne (für Systeme welche keine parity benötigen)
- ***Invert*** kann entweder normal oder invertiert sei. Bei invert, wechseln die high Bits nach low und von low nach high.

Beispiel:

'Sample usage for communication with Lego RCX:

InitSer(1,r2400,1+WaitForStart,8,1,odd,invert)

Wichtige Konstanten

These constants are used to control settings for the **RS232 serial communication** routines. To set them, place a line in the main program file that uses #define to assign a value to the particular constant.

Constant Name/s	Controls	Default Value
Ser_Links	The number of serial links supported. Setting it to a lower number reduces the amount of code downloaded to the PIC.	3
SendALow, SendBLow, SendCLow	These are used to define the commands used to send a low (0) bit on serial channels A, B and C respectively.	nop **(must be set by user)**
SendAHigh, SendBHigh, SendCHigh	These are used to define the commands used to send a high (1) bit on serial channels A, B and C respectively.	nop **(must be set by user)**
RecALow, RecBLow, RecCLow	The condition that is true when a low bit is being received	Sys232Temp.0 OFF **(must be set by user)**
RecAHigh, RecBHigh, RecCHigh	The condition that is true when a high bit is being received	Sys232Temp.0 ON **(must be set by user)**

SerSend

Syntax:

SerSend *(channel, data)*

Beschreibung:

Dieser Befehl sendet ein Byte mit dem Inhalt von ***data*** an den RS232 Kanal bezogen auf ***channel*** gemäß den Bestimmungen der gesetzten Initialisierung durch **InitSer.**

Beispiel:

```
'This program will send a byte using PORTB.2, the value of which
'depends on whether a button is pressed.

#chip 16F819, 8
#config Osc = Int

#define SendAHigh Set PORTB.2 ON
#define SendALow Set PORTB.2 OFF
#define Button PORTA.0

Dir Button In
Dir PORTB.2 Out

InitSer 1, r9600, 1+WaitForStart, 8, 1, none, normal
Do
	If Button On Then Temp = 0
	If Button Off Then Temp = 100
	SerSend 1, Temp
	Wait 100 ms
Loop
```

SerReceive

Syntax:

SerReceive *(channel, output)*

Beschreibung:

Dieser Befehl liest ein Byte von dem RS232 Kanal bezogen auf ***channel*** gemäß den Bestimmungen der gesetzten Initialisierung durch **InitSer** und speichert das empfangene Byte in die Variable ***output***.

Beispiel:

```
'This program will read a byte from PORTB.2, and set the LED
'on if the byte is more than 50.

#chip 16F88, 8
#config Osc = Int

#define RecAHigh PORTB.2 ON
#define RecALow PORTB.2 OFF
#define LED PORTB.0

Dir PORTB.0 Out
Dir PORTB.2 In

InitSer 1, r9600, 1+WaitForStart, 8, 1, none, normal
Do
	SerReceive 1, Temp
	If Temp <= 50 Then Set LED Off
	If Temp > 50 Then Set LED On
Loop
```

Allgemeine Betrachtung zum Serial Peripheral Interface (SPI)

SPI wurde von Motorola (heute Freescale) entwickelt, jedoch nie standardisiert. Da SPI nie patentiert oder lizensiert wurde und einfach zu implementieren ist, hat es eine weite Verbreitung erlangt. SPI ist ein synchroner serieller Bus. Dieser ermöglicht die Anbindung von Peripheriekomponenten an einen Mikrocontroller. Auch die Verbindung mehrerer Mikrocontroller untereinander ist möglich. SPI erreicht hier sehr hohe Datenübertragungsraten, da das Taktsignal bis zu mehreren MHz betragen kann. Außerdem werden die Daten vollduplex in beide Richtungen gleichzeitig übertragen.

SPI ist als Master-Slave Bus ausgelegt, d. h. ein Master muss immer die Datenübertragung einleiten und den jeweiligen Slave selektieren. Der Master stellt auch das Taktsignal bereit. Die Daten werden auf zwei Datenleitungen übertragen. Diese werden mit **MISO** (Master-In Slave-Out, Dateneingang Master) und **MOSI** (Master-Out Slave-In, Datenausgang Master) bezeichnet. Jeder Slave besitzt ein Slave-Select–Signal oder Chip-Select–Signal (Low-Aktiv). Solange ein Slave nicht ausgewählt ist, sind Takt- und Datenleitunge im TriState-Mode. Synchron zum Taktsignal des Masters werden Informationen über die Datenleitungen ausgegeben. Master und Splave verhalten sich wie Schieberegister.

Je nach Aufgabenstellung weist ein SPI-Baustein eine unterschiedlich hohe Komplexität auf, die von einem einfachen Schieberegister bis hin zu einem eigenem Subsystem reicht. Hier zeichnet sich schon ein wichtiger Punkt ab, der bei der Konzeption eines SPI-Bussystems bzw. bei der Verarbeitung berücksichtigt werden muß: Die Länge der Schieberegister ist nicht fest definiert, sondern kann von Baustein zu Baustein verschieden sein. Normalerweise sind die Schieberegister acht Bit lang (oder ein ganzzahlig Vielfaches davon). Die empfangenen Daten liegen nach dem Datentransfer im gleichen Register wie die Sendedaten. Es existiert also nur ein Register für Sende-/Empfangsdaten.

Da es keine offizielle Spezifikation des SPI gibt, müssen gegebenenfalls die Datenblätter der verwendeten Bausteine zu Rate gezogen werden. Wichtig sind hierbei die erlaubte Taktfrequenz und die Art der gültigen Flanken. Bei welcher Taktflanke die Daten übernommen werden, ist nicht vorgeschrieben. In der Praxis haben sich vier Betriebsarten herausgebildet. Diese vier Betriebsmodi ergeben sich durch die Möglichkeit der Einstellungen von CPOL und CPHA.

SPI-Modus	CPOL	CPHA
0	0	0
1	0	1
2	1	0
3	1	1

Ist die Phase des Clocksignals gleich null, also CPHA=0, dann findet bei CPOL=0 die Datenübernahme bei steigender Taktflanke, bei CPOL=1 bei fallender Taktflanke statt. Ist CPHA=1, dann drehen sich die Polaritäten um. Bei CPOL=0 findet die Datenübernahme dann bei fallender Taktflanke und bei CPOL=1 bei steigender Flanke statt.

Bei den Mikrocontrollern von Frescale kann die Polarität und die Phase des Taktsignals eingestellt werden. Wird eine positive Polarität gewählt erfolgt die Übernahme bei steigender Taktflanke. Die Daten werden aber schon bei der fallenden Flanke auf die Datenleitung geschickt, um sicherzustellen, daß sie bei der Übernahme stabil vorliegen. Bei den meisten Peripheriebausteinen, die nur Slave sein können, kann man von dieser Konfiguration ausgehen.

SPI kennt verschiedene Modi, die über Konfigurationsregister eingestellt werden. Am häufigsten kommt der 3-Wire-Master-Slave-Modus vor. Hierbei werden nur die beiden Datenleitungen und die Taktleitung benötigt. Das Chip-Select- oder Slave-Select-Signal des Slaves liegt fest auf Masse. Damit ist der Slave immer angewählt und eskann kein weiterer Slave angesteuert werden.

Beim 4-Wire-Master-Slave- bzw. 4-Wire-Multi-Master-Modus existiet zusätzlich noch ein Slave-Select-/Chop-Select-Signal vom Master. Dieser wählt vor dem Datentransfer den Slave aus, mit dem er kommunizieren will. Der Multi-Master-Modus wird hauptsächlich verwendet, wenn zwei Mikrocontroller miteinander verbunden sind. Hier kann jeder Mikrocontroller einen Datentransfer einleiten. Während eines Datentransfers ist einer als Master und einer als Slave konfiguriert. Im Ruhezustand verhalten sich beide als Slave. Die Umschaltung erfolgt automatisch.

Die SPI-Peripheriebausteine lassen sich in folgende Kategorien unterteilen:

- Wandler (A/D- und D/A-Wandler)
- Speicher (EEPROM und FLASH)
- Real Time Clocks (RTC)
- Sonstige (Signalmixer, Potentiometer, LCD-Controller, UART, Verstärker, Sensoren etc.)

In den ersten drei Kategorien sind die meisten Bausteine zu finden. Die Auswahl bei den Analog/Digital- und Digital/Analog-Wandlern ist gross. Die Auflösung der Wandler reicht von 8 bis 24 Bit (am häufigsten 8, 10, 12 und 16 Bit), wobei die Taktfrequenz zwischen 30.000 und 600.000 Samples/s liegt. Die Kapazität der Speicherbausteine reicht von einigen Bits bis hin zu 64 KBit. Bei den Speichern kann eine Taktfrequenz von bis zu 3 MHz errreicht werden.

Durch den SPI-Bus ist ein Embedded System recht einfach erweiterbar. Das Embedded System selbst bildet dabei den SPI-Master. Die Programmierung einer SPI-Schnittstelle unter Linux ist sehr einfach, weil in der Regel nur parallele E/A-Bits benötigt werden. Somit sind nur die elementaren E/A-Funktionen notwendig, um aus einem Linux-Programm auf einen SPI-Slave zuzugreifen. Einziger Nachteil: die Leitung zwischen E/A-Port und Peripherie sollte höchstens einige Zentimeter betragen. Man kann aber tricksen und RS232- oder RS485-Pegelwandler einsetzen, um die Verbindung zu verlängern.

SPITransfer

Syntax:

SPITransfer *tx, rx*

Beschreibung:

Dieser Befehl ist gleichzeitig zum Senden und Empfangen eines Byte mit dem SPI Protokoll. Es verhält sich abhängig von der Ausführung als **Master** oder **Slave** verschieden. Bei der Master Operation wird **SPITransefer** eine Übertragung ausführen. Die Daten in ***tx*** werden an den Slave gesendet, während die im Slave gepufferten Byte in ***rx*** geschrieben werden. Im Slave Mode wird das Programm durch den SPITransfer Befehl solange angehalten, bis eine Übertragung durch den Master ausgelöst wurde. An diesem Punkt werden die Daten in tx gesendet, während bei der Übertragung vom Master in die rx Variable geschrieben wird.

Beispiel:

Dies sind zwei Beispielprogramme für diesen Befehl. Eines auf dem Slave des PIC und das andere auf dem Master. Ein Messwert wird von einem Sensor zum Slave geschickt und weiter zum Master gesendet welcher diesen dann am LCD anzeigt.

Slave Program:

```
#chip 16F88, 20
#config MCLR_OFF

'Set directions of SPI pins
dir PORTB.2 out
dir PORTB.1 in
dir PORTB.4 in
'Set direction of analogue pin
dir PORTA.0 in

'Set SPI mode to slave
SPIMode Slave

'Allow other PIC to initialise LCD
Wait 1 sec

'Main loop - takes a reading, and then waits to send it across.
do
'Note that rx is 0 - this is because no data is to be received.
SPITransfer ReadAD(AN0), 0
loop
```

Master Program:

```
'General hardware configuration
#chip 16F877A, 20

'LCD connection settings
#define LCD_IO 8
#define LCD_DATA_PORT PORTC
#define LCD_RS PORTD.0
#define LCD_RW PORTD.1
#define LCD_Enable PORTD.2

'Set SPI pin directions
dir PORTC.5 out
dir PORTC.4 in
dir PORTC.3 out

'Set SPI Mode to master, with fast clock
SPIMode MasterFast

'Main Loop
do
'Read a byte from the slave
'No data to send, so tx is 0
SPITransfer 0, Temp

'Display data
if Temp > 0 then
        CLS
        Print "Light: "
        LCDInt Temp
        Temp = 0
end if

'Wait to allow time for the LCD to show the given value
wait 100 ms
loop
```

Wichtige Konstanten

These constants are used to control settings for the **tone generation** routines. To set them, place a line in the main program file that uses #define to assign a value to the particular constant.

Constant Name	Controls	Default Value
SoundOut	The output pin to produce sound on	N/A - ***Must be set***

Tone

Syntax:

Tone (*Frequency, Duration)*

Beschreibung:

Dieser Befehl erzeugt einen bestimmten Tone von einer bestimmten Dauer. ***Frequency*** wird in Hz gemessen und ***Duration*** in Einheiten zu10ms.

Beachte, dass der Befehl nicht die exakte Frequenz wiedergibt. Es ist lediglich gedacht für Error Beeps oder monophone Signale und nicht für exakte Wiedergabe der Frequenzen hoher Qualität.

Beispiel:

```
'Sample program to produce a constant A note (440 Hz)
'on PORTB bit 1.

#chip 16F877A, 20
#mem 368

#define SoundOut PORTB.1
dir SoundOut OUT

do
	Tone 440, 1000
loop
```

ShortTone

Syntax:

ShortTone *(Frequency, Duration)*

Beschreibung:

Dieser Befehl erzeugt einen bestimmten Tone von einer bestimmten Dauer. ***Frequency*** wird in Einheiten zu 10 Hz gemessen und ***Duration*** in Einheiten zu1 ms.
Beachte, dass der Befehl nicht die exakte Frequenz wiedergibt. Es ist lediglich gedacht für Error Beeps oder monophone Signale und nicht für exakte Wiedergabe der Frequenzen hoher Qualität.

Beispiel:

```
'Sample program to produce a tone on PORTB bit 1,
'based on the reading of an LDR on AN0 (usually
'PORTA bit 0).

#chip 16F88, 20

#define SoundOut PORTB.1
dir SoundOut OUT

do
	ShortTone ReadAD(AN0), 100
loop
```

PIC TIMER

Allgemeines zu Timern

Alle Flash-PICs besitzen mindestens einen Timer. Größere PICs haben derer sogar drei. Ein Timer ist nichts anderes als eine normale Zählschaltung. Sein Eingang kann mit einem I/O-Pin oder mit dem internen PIC-Takt verbunden werden. Dann zählt er mit dem eingespeisten Takt. Ist der Zähler mit dem PIC-Takt verbunden wird er im Allgemeinen als Timer bezeichnet. Ist er mit einem externen I/O-Pin verbunden, bezeichnet man ihn auch als Counter.

Jeder Zähler läuft einmal über. Ein 8-Bit Timer kann von 0 bis 255 zählen, ein 16-Bit-Timer kommt bis 65535. Hat der Timer seinen höchsten Zählwert erreicht, und bekommt dann einen weiteren Zählimpuls, dann springt er wieder auf 0 und kann von vorn beginnen. Bei diesem Überlauf gibt der Timer ein Signal aus, das ein bestimmtes Bit in einem Register des PIC setzt. Wenn man dieses Bit abfragt, weiß man also, ob der Timer übergelaufen ist. Das kann man in einem Programm als Zeitbasis nutzen. Außerdem kann beim Überlauf auch ein Interrupt ausgelöst werden. Das erspart das dauernde Abfragen (pollen) des Überlaufbits.

Der momentane Zählerstand des Timers ist nicht geheim, er liegt immer in einem speziellen Register, von wo man ihn auslesen kann. Man kann dieses Register auch beschreiben, und dadurch den Timer auf einen bestimmten Wert setzen, von dem er dann weiterzählt.

PICs besitzen bis zu 3 Timer: (PIC18Fxxx haben bis zu 5 Timer)

- Timer0 - 8 Bit , Vorteiler
- Timer1 - 16 Bit, Vorteiler
- Timer2 - 8 Bit, Vorteiler&Nachteiler, Periodenregister
- Timer3 - 16 Bit, Vorteiler (ähnlich wie Timer1)
- Timer4 - 8 Bit, Vorteiler&Nachteiler, Periodenregister (ähnlich wie Timer2)

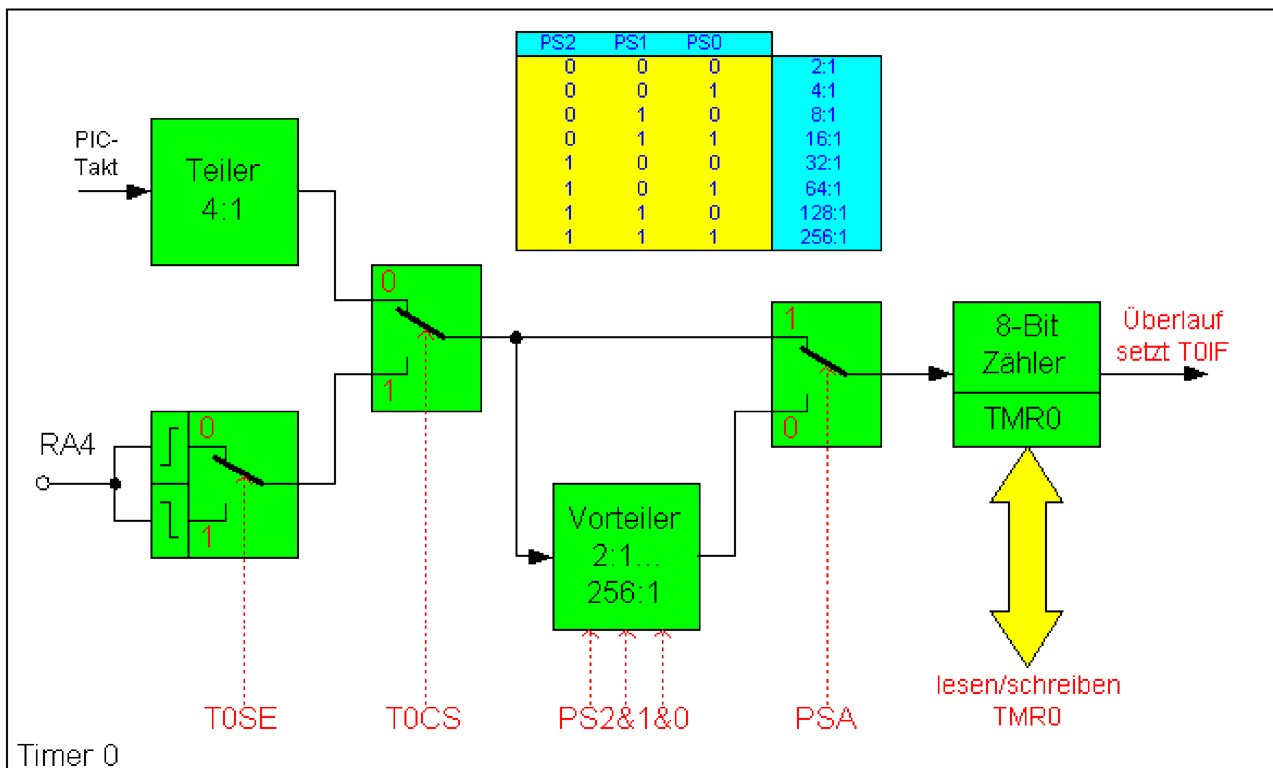
PIC-Typ	12F629/675	16F84(A)	16F628/627	16F7x	16F87x(A)
Timer0	X	X	X	X	X
Timer1	X		X	X	X
Timer2			X	X	X

Timer 0

Der Timer 0 gehört zur Grundausstattung aller Flash-PICs. Er ist ein 8-Bit-Timer, kann also nur bis 255 zählen.

Seine Zählimpulse erhält er entweder über das Pin **RA4**, oder er wird mit 1/4 des PIC-Taktes gespeist. Der ausgewählte Takt kann direkt auf den Zähler gegeben werden, oder man verwendet einen Vorteiler, dessen Teilverhältnis von 1:2 bis zu 1:256 in 8 Stufen eingestellt werden kann.
Der Zähler zählt vorwärts von 0 bis 255 und beginnt dann von vorn.
Der momentane Zählerstand ist im Register **TMR0** zugänglich.

Bei Überlauf wird das Bit **T0IF** im **INTCON** Register gesetzt.



T0CS (Timer0 clock source select)
Mit diese Bit wählt man die Taktquelle des Timer0 aus

- 0 - der Takt ist 1/4 des PIC-Taktes (bei einem 10 MHz-Quarz also 2,5 MHz)
- 1 - derTakt kommt vom Pin RA4

T0SE (Timer0 clock source edge select)
Falls man sich für RA4 als Taktquelle entschieden hat, kann man wählen, bei welcher Flanke an RA4 der Timer einen Schritt weiter zählt.

- 0 - Low-High-Flanke wird gezählt
- 1 - High-Low-Flanke wird gezählt

PSA (pre scaler assignment)
Mit diesem Bit kann man bei Bedarf den Vorteiler zuschalten

- 0 - der Vorteiler wird verwendet
- 1 - den Vorteiler nicht verwenden

PS2, PS1, PS0 (pre scaler rate select)
Falls man den Vorteiler verwendet, kann man hier sein Teilverhältnis einstellen

- PS2, PS1, PS0 = '000' - Teilverhältnis 2:1
- PS2, PS1, PS0 = '001' - Teilverhältnis 4:1
- PS2, PS1, PS0 = '010' - Teilverhältnis 8:1
- PS2, PS1, PS0 = '011' - Teilverhältnis 16:1
- PS2, PS1, PS0 = '100' - Teilverhältnis 32:1
- PS2, PS1, PS0 = '101' - Teilverhältnis 64:1
- PS2, PS1, PS0 = '110' - Teilverhältnis 128:1
- PS2, PS1, PS0 = '111' - Teilverhältnis 256:1

InitTimer0

Syntax:

InitTimer0 *source, prescaler*

Beschreibung:

Der Befehl InitTimer0 setzt den Timer 0 entsprechend den Einstellungen. Source kann der Osc oder Ext sein. Prescaler kann eine der folgenden Einstellungen haben.

PS0_1/2
PS0_1/4
PS0_1/8
PS0_1/16
PS0_1/32
PS0_1/64
PS0_1/128
PS0_1/256

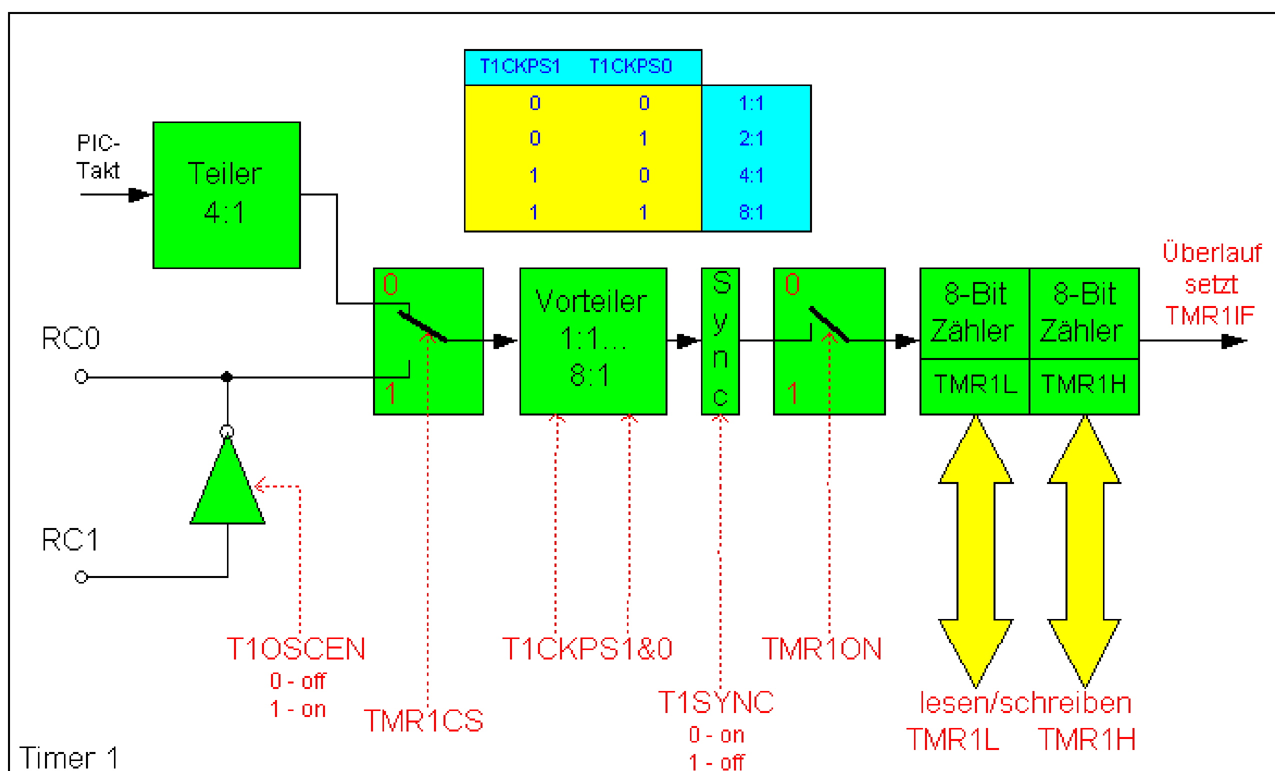
Beispiel:

See InitTimer1 for an example.

Timer1
Der Timer 1 ist ein 16-Bit Timer, der seine Zählimpulse entweder vom PIC-Takt oder vom I/O-Pin RC1 bzw. RC0 erhält.

Der ausgewählte Takt wird durch einen Vorteiler, einen Synchronisator (wenn ausgewählt) und eine Torstufe zum 16-Bit-Zähler geleitet.

Der Zähler zählt von 0 (0000h) bis 65535 (FFFFh). Dann läuft er über und beginnt bei 0 von vorn. Beim Überlauf wird das Bit **TMR1IF** gesetzt, das auch einen Interrupt auslösen kann (wenn aktiviert). Der Zählerstand kann über die beiden Register **TMR1L** (low-Teil) und **TMR1H** (High-Teil) ausgelesen und verändert werden. Das CCP-Modul benötigt Timer1 für die Capture- sowie für die Compare-Funktion. Wird eine dieser Funktionen im CCP-Modul aktiviert, steht der Timer1 also nicht mehr frei zur Verfügung.



TMR1CS (Timer1 clock source select)
Mit diese Bit wählt man die Taktquelle des Timer1 aus

- 0 - der Takt ist 1/4 des PIC-Taktes (bei einem 10 MHz-Quarz also 2,5 MHz)
- 1 - derTakt kommt vom Pin RC0 bzw. RC1

T1OSCEN (Timer1 oscillator enable control bit)
ist nur von Interesse, wenn **TMR1CS = 1**
Mit diese Bit aktiviert man den Verstärker zwischen RC1 und RC0

- 0 - RC0 ist der Takteingang des Timer1
- 1 - RC1 ist der Takteingang des Timer1, an RC0 (nun Output) liegt das invertierte RC1-Signal an.
  Wird ein Quarz (bis wenige 100 kHz) zwischen RC0 und RC1 angeschlossen (+ 2

Kondensatoren)
dann schwingt dieser und liefert den Takt für Timer1.

T1CKPS1, T1CKPS0 (input clock prescaler select select)
Hier stellt man das Teilverhältnis des Vorteilers ein

- T1CKPS1, T1CKPS0 = '00' - Teilverhältnis 1:1
- T1CKPS1, T1CKPS0 = '01' - Teilverhältnis 2:1
- T1CKPS1, T1CKPS0 = '10' - Teilverhältnis 4:1
- T1CKPS1, T1CKPS0 = '11' - Teilverhältnis 8:1

T1SYNC (Timer1 external clock input syncronisation control bit)
Mit diese Bit wählt man, ob der Takt vor dem Zähler mit dem Zyklustakt des PIC (1/4 PIC-Frequenz) syncronisiert wird.
Eine solche Synchronisation ist z.B. nötig, wenn der Timer1 für Capture oder Compare-Funktionen des CCP verwendet werden soll.
Der Syncronisator funktioniert nicht im Sleep-Mode. Wenn der Timer1 auch wärend Sleep laufen soll (als Uhr oder zum Wecken des PIC via Interrupt), dann muß der Syncronisator abgeschaltet werden.

- 0 - der Syncronisator ist aktiv
- 1 - der Syncronisator wird nicht benutzt

TMR1ON (Timer1 on bit)
Mit diese Bit öffnet und schließt man das Tor vor dem Zähler.

- 0 - keine Impulse gelangen zum Zähler, der Timer1 steht still
- 1 - Impulse werden zum Zähler geleitet, der Timer1 läuft

TMR1L und TMR1H
Der momentane 16-bittige Zählerstand des Timer1 steht in den beiden 8-Bit-Registern **TMR1L** (untere 8 Bit) und **TMR1H** (obere 8 Bit). Da diese beiden Register nur nacheinander ausgelesen werden können, kann es passieren, dass zwischen den beiden Leseoperationen ein Übertrag von den unteren 8-Bit zu den oberen 8-Bit erfolgt. In diesem Fall liest man ein falsches Ergebnis aus.

InitTimer1

Syntax:

InitTimer1 ***source, prescaler***

Beschreibung:

Der Befehl InitTimer0 setzt den Timer 0 entsprechend den Einstellungen. Source kann der Osc oder Ext sein. Prescaler kann eine der folgenden Einstellungen haben.

PS1_1/1
PS1_1/2
PS1_1/4
PS1_1/8

Beispiel:

```
'This example will measure that time that a switch stays on for
#chip 16F819, 20
#define Switch PORTA.0

dir Switch In
DataCount = 0
InitTimer Osc, PS1_1/8

Start:
        ClearTimer 1
        Wait until Switch On
        StartTimer 1
        Wait until Switch Off
        StopTimer 1

        EPWrite(DataCount, TMR1H)
        EPWrite(DataCount + 1, TMR1L)

Goto Start
```

StartTimer

Syntax:

StartTimer *TimerNo*

Beschreibung:

Der Befehl **StartTimer** startet den ausgewählten Timer. Beachte, dass der Befehl nicht den Timer 0 starten kann, da dieser immer arbeitet.

Beispiel:

```
'This example will measure that time that a switch stays on for
#chip 16F819, 20
#define Switch PORTA.0

dir Switch In
DataCount = 0
InitTimer Osc, PS1_1/8

Start:
	ClearTimer 1
	Wait until Switch On
	StartTimer 1
	Wait until Switch Off
	StopTimer 1

	EPWrite(DataCount, TMR1H)
	EPWrite(DataCount + 1, TMR1L)

Goto Start
```

ClearTimer

Syntax:

ClearTimer *TimerNo*

Beschreibung:

ClearTimer löscht den ausgewählten Timer.

Beispiel:

```
'This example will measure that time that a switch stays on for
#chip 16F819, 20
#define Switch PORTA.0

dir Switch In
DataCount = 0
InitTimer Osc, PS1_1/8

Start:
        ClearTimer 1
        Wait until Switch On
        StartTimer 1
        Wait until Switch Off
        StopTimer 1

        EPWrite(DataCount, TMR1H)
        EPWrite(DataCount + 1, TMR1L)

Goto Start
```

StopTimer

Syntax:

StopTimer *TimerNo*

Beschreibung:

Der Befehl **StopTimer** stoppt den ausgewählten Timer. Beachte, dass der Befehl nicht den Timer 0 starten kann, da dieser immer arbeitet.

Beispiel:

```
'This example will measure that time that a switch stays on for
#chip 16F819, 20
#define Switch PORTA.0

dir Switch In
DataCount = 0
InitTimer Osc, PS1_1/8

Start:
	ClearTimer 1
	Wait until Switch On
	StartTimer 1
	Wait until Switch Off
	StopTimer 1

	EPWrite(DataCount, TMR1H)
	EPWrite(DataCount + 1, TMR1L)

Goto Start
```

Variablen Operationen

Setting Variables

Syntax:

Variable = data

Beschreibung:

Der Variablen wird der Inhalt von ***data*** zugeordnet. Dies kann ein fester Wert sein (z.B. 157), eine andere Variable oder eine Summe.

Wenn ***data*** ein fester Wert ist muss es ein Interger zwischen 0 und 255 inklusive sein.

Wenn data eine Berechnung ist dann kommen folgende Operanden zur anwendung:

- + (add)
- - (subtract)
- * (multiply)
- / (divide)
- % (modulo)
- & (and)
- | (or)
- # (xor)
- ! (not)
- = (equal)
- <> (not equal)
- < (less than)
- > (greater than)
- <= (less than or equal)
- >= (more than or equal)

Die ersten 6 Operanden sind zur Überprüfung von Bedingungen. Sie geben den Wert FALSE (0) zurück wenn die Bedingung falsch ist oder TRUE (255) wenn die Bedingung wahr ist.

Die AND, OR, XOR und NAND Operanden arbeiten auf zweierlei als Bitweise und Logikoperand.

GCBASIC versteht Anweisungen von Operationen. Wenn vielfache Operanden vorhanden sind werden sie wie nachfolgend arbeiten.

1. Brackets (Klammern)
2. Multiply/Divide/Modulo
3. Add/Subtract
4. Conditional operators
5. And/Or/Xor/Not

Es gibt zwei Mode in welchen die Variabeln arbeiten, Byte-Mode und Word-Mode. GCBASIC benutzt automatisch verschiedene Mode für jede Berechnung abhänging vom gesetzten Typ der Variablen. Wenn Byte-Variable gesetzt ist wird Byte-Mode verwendet, ist Word-Variable gesetzt wird Word-Mode verwendet. Wenn Byte gesetzt ist aber die Berechnung beinhaltet einen Wert größer als 255, kann der Word-Mode durch hinzufügen von [WORD] am ende der Befehlszeile gesetzt werden.

Wenn man „LET“ am Anfang der Befehlszeile bevorzugt, kann man dies tun, hierbei wird die Ausführung des Programms nicht beeinflusst.

Beispiel:

```
'This program is to illustrate the setting of variables.

Chipmunk = 46       'Sets the variable Chipmunk to 46
Animal = Chipmunk   'Sets the variable Animal to the value of the variable Chipmunk
Bear = 2 + 3 * 5    'Sets the variable Bear to the result of 2 + 3 * 5, 17.
Sheep = (2 + 3) * 5 'Sets the variable Sheep to the result of (2 + 3) * 5, 25.
Animal = 2 * Bear   'Sets the variable Animal to twice the value of Bear.
LargeVar = 321      'LargeVar must be set as a word - see DIM.
Temp = LargeVar / 5 [WORD]     'Note the use of [WORD] to ensure that the calculation is
performed correctly
```

ROTATE

Syntax:

ROTATE *variable* {LEFT | RIGHT} [SIMPLE]

Beschreibung:

Der ROTATE Befehl rotiert ***variable*** in der angegebenen Richtung.

This table shows the operation of the ROTATE command

Command	***variable***	**STATUS.C**
Values at start:	b'00110011'	1
ROTATE variable LEFT	b'01100110'	0
ROTATE variable RIGHT	b'10011001'	1
ROTATE variable LEFT SIMPLE	b'01100110'	N/A
ROTATE variable RIGHT SIMPLE	b'10011001'	N/A

Beispiel:

```
'This program will rotate the variable Pattern two places to the
'right.

Pattern = 15
'Value of Pattern in binary: 00001111
ROTATE Pattern RIGHT SIMPLE
'Value of Pattern in binary: 10000111
ROTATE Pattern RIGHT SIMPLE
'Value of Pattern in binary: 11000011
```

SET

Syntax:

SET *variable.bit* {ON | OFF}

Beschreibung:

Die Aufgabe des **SET** Befehls besteht darin, einzelne Bits An und Aus zu schalten. Der SET Befehl wird meisten zum Steuern der Ausgangs Ports verwendet. Kann aber auch zum setzen von Variablen verwendet werden.

Meist wird auch beim steuern der Ausgangs Ports der SET Befehl im Zusammenhang mit Konstanten verwendet. Dies wird manchmal in Beispielen gezeigt wie „SET green ON“. Hierbei ist green eine Konstante die an anderer Stelle definiert wurde und sich auf einen Ausgangs Port bezieht.

Beispiel:

```
'Blink LED sample program for GCBASIC
'Controls an LED on PORTB bit 0.

'Set chip model and config options
#chip 16F84A, 20

'Set pin direction
dir PORTB.0 OUT

'Main routine
do
	set PORTB.0 ON
	Wait 1 sec
	set PORTB.0 OFF
	Wait 1 sec
loop
```

Felder Arrays mit dem Dim Befehl, Allgemeine Betrachtung.

Ein Array (Feld) kann auch als Tabelle betrachtet werden.

Feldindex	0	1	2
Wert	mehrere	Dimensionen	Ende

Ein Array kann aber auch mehrere Dimensionen haben. Nachfolgen ein Beispiel zur Deklaration.

```
Dim dimens(2) As String

        dimens(0) = "mehrere"
        dimens(1) = "Dimensionen"
        dimens(2) = "Ende"
```

Man muss sich Merken, dass alle Arrays bei Null beginnen!

Und nun zur Syntax:

Dim *Variablenname*(*Feldgröße*) As *Datentyp*

Kommen wir dann noch mal kurz zur Syntax des Datenzugriffs:

Variablenname(*Index*) = *Wert*

Man kann aber die größe und den beginn eines Arrays also eines Feldes bzw. Datenfeldes auch Explicit angeben, die geht mit der von zu Anweisung (To). Nachfolgend ein Beispiel:

```
Dim dimens(1 To 3) As String

dimens(1) = "mehrere"
dimens(2) = "Dimensionen"
dimens(3) = "Ende"
```

Es hat sich hierbei nun nur der beginn und das Ende des Feldes geändert die Größe von 3 Eingaben ist gleich geblieben.

DIM

Der DIM Befehl hat zwei Anwendungen, beide beinhalten umfangreiche Variablen. Er kann benutzt werden um Arrays (Felder) zu definieren und um Variablen größer als 1 Byte zu deklarieren.

Syntax:

For Variables > 1 byte:
Dim *variable* [*, variable2*[*, variable3*]] [As *type*] [Alias *othervar* [,*othervar2*]]

For Arrays:
Dim *array*(*size*)

Beschreibung:

Der **DIM** Befehl wird verwendet um GCBASIC ***variable*** größer 1 Byte mitzuteilen oder ALIAS (andere) Namen für Variablen zu erzeugen.

Der **DIM *array*** Befehl bestimmt **Array** Variablen. Bei dieser Verwendung kann ***size*** eine Konstante bis 80 sein.

Type bezeichnet den Typ der Variablen der bestimmten wird. Verschiedene Variablen Typen können Werte hierbei haben über verschiedene Bereiche und benutzt verschiedene RAM Beträge. In der Beschreibung (Syntax)Variablen werden diese beschrieben.

Wenn Mehrfachvariablen am Anfang der Zeile stehen wird GCBASIC diese entsprechend dem spezifizierten Typ am Ende der Zeile. Wenn sie nicht spezifiziert sind werden sie von GCBASIC als Byte behandelt.

Alias erzeugt eine Variable durch Gebrauch des gleichen Speicherbereichs als eine oder mehrerer anderer Variablen. Es wird hauptsächlich innerhalb von GCBASIC benutzt um Systemvariablen als WORD zu betrachten. Zum Beispiel wird dieser Befehl verwendet um eine WORD Variable zu erzeugen welche aus zwei Speicherbereichen besteht und in welche das Ergebnis der A/D Wandlung gespeichert wir.

```
Dim ADResult As Word Alias ADRESH, ADRESL
```

Beispie:

```
'This program will set up an array, and a word variable

dim DataList(10)
dim Reading as word

Reading = 21978
DataList(1) = 15
```

Diverse Befehle

DIR

Syntax:

DIR *port.bit* {IN | OUT} *(Individual Form)*
DIR *port* {IN | OUT | *DirectionByte*} *(Entire Port Form)*

Explanation:

Der **DIR** Befehl wird verwendet um die Richtung der Ports des PIC Chip zu setzen. Die ***individual Form*** setzt die Richtung eines Pin eines Ports, wobei die ***entire Port Form*** alle Pins des Ports setzt.

Die individual Form spezifiziert den Port und Bit (z.B. PORTB.4) dann ist die Richtung entweder IN oder OUT.

Die entire Form ist entsprechend der TRIS Instruktion bei manchen Pic Chips. Bei Verwendung gibt man den Namen des Port an (z.B. PORTA) dann wird ein Byte in die TRIS Variable geschrieben. *Dieser Befehl ist für jene welche sich in der internen Architektur des Pic Chip auskennen.*

Für die 10/12 Serie Chips wird die entire Port Form benutzt. Spezifiziert „GPIO“ oder „IO“ als Port.

Beispiel:

```
'This program sets PORTA bits 0 and 1 to in, and the rest to out.
'It also sets all of PORTB to output, except for B1.
'Individual form is used for PORTA:

DIR PORTA.0 IN
DIR PORTA.1 IN
DIR PORTA.2 OUT
DIR PORTA.3 OUT
DIR PORTA.4 OUT
DIR PORTA.5 OUT
DIR PORTA.6 OUT
DIR PORTA.7 OUT

'Entire port form used for B:
DIR PORTB b'00000010'

'Entire port form used for C:
DIR PORTC IN
```

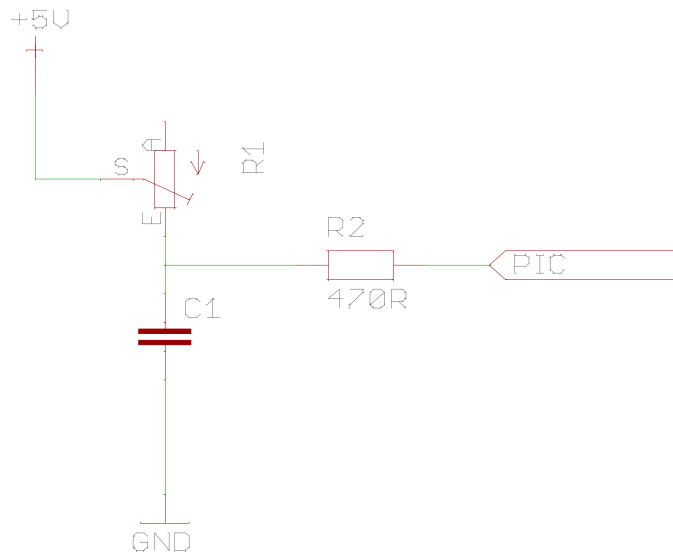
POT

Syntax:

POT ***pin, output***

Beschreibung:

POT macht es möglich einen analogen Widerstand mit einem digitalen Port zu messen bei Verwendung eines kleinen Kondensators. Nachfolgend hierzu die Schaltung.



Der Befehl wirkt unter Verwendung des PIC um entladen des Kondensators, dann wird die Zeit gemessen die der Kondensator wieder braucht um sich über den Widerstand R aufzuladen.

Der Wert des Kondensators ist hierbei abhängig von der Größe des variablen Widerstand R (τ = R x C). Die Aufladezeit beträgt ca. 2,5 ms bei größtem Wert von R. Bei einem typischen Wert für R von **50 kΩ** - Potentiometer oder LDR ist ein **50 nF** Kondensator erforderlich.

Der Befehl sollte mit bedacht eingesetzt werden. Er kann jederzeit eingesetzt werden und benötigt 20 Words des Programmspeichers des Chip, nach einer geschätzten Größe entspricht dies 15 mal der Größe des SET Befehls.

pin ist die Port Verbindung zur Schaltung. Die Zuordnung des Pin wird durch den POT Befehl erzeugt.

output ist der Name der Variablen welche den Wert enthält.

Beispiel:

```
'This program will beep whenever a shadow is detected
'A potentiometer is used to adjust the threshold

#chip 16F628A, 4
#config INTOSC_OSC_NOCLKOUT

#define Adjust PORTB.0
#define LDR PORTB.1
#define SoundOut PORTB.2

dir SoundOut out

Main:
        POT Adjust, Threshold
        POT LDR, LightLevel
        if LightLevel > Threshold then
        Tone 1000, 100
        end if
goto Main
```

See Also:
http://www.cvs1.uklinux.net/cgi-bin/calculators/time_const.cgi (Useful for calculating capacitor value. Not associated with GCBASIC.)

PULSEOUT

Syntax:

Der Befehl **PULSEOUT** setzt den ausgewählten ***pin*** auf HIGH, wartet den Betrag ***time units*** und setzt den Pin wieder auf LOW. Dies entspricht einem Monoflop. Dies geschieht auf die gleiche Weise wie bei SET Befehl und time ist das gleiche wie beim WAIT Befehl.

Beispiel:

```
'This program flashes an LED on GPIO.0 using PULSEOUT
#chip 12F629, 4
#config INTRC_OSC_NOCLKOUT

DIR GPIO.0 OUT

Main:

PULSEOUT GPIO.0, 1 sec 'Turn LED on for 1 sec
Wait 1 sec 'Wait 1 sec with LED off

goto Main
```

PEEK

Syntax:

OutputVariable **= PEEK (*location*)**

Beschreibung:

Der **PEEK** Befehl wird benutzt um Informationen vom on – chip RAM des PIC zu lesen.

location ist eine WORD Variable und kann einen Wert ***(value)*** annehmen bis 511, dies ist die Grenze der 16 xx PIC Chips..

Beispiel:

```
'This program will read and check a value from PORTA

Temp = PEEK(5)
IF Temp.2 ON THEN SET green ON
IF Temp.2 OFF THEN SET red ON
```

POKE

Syntax:

POKE (*location, Value*)

Beschreibung:

Der **POKE** Befehl wird benutzt um Informationen in den on – chip RAM des PIC zu schreiben.

location ist eine WORD Variable und kann einen Wert ***(value)*** annehmen bis 511, dies ist die Grenze der 16 xx PIC Chips.

Beispiel:

'This program will set all of the PORTB pins high
POKE (6, 255)

ReadTAble

Syntax:

ReadTable ***TableName, Item, Output***

Explanation:

Der **ReadTAble** Befehl wird dazu verwendet um Informationen von eine Lookup Tabelle zu lesen. ***TableName*** ist der Name der Tabelle von der gelesen wird, ***Item*** ist die Zeile die gelesen wird und ***Output*** ist die Variable in den der letzte Stand geschrieben wird.

Item ist 1 für die erste Zeile der Tabelle, 2 für die zweite usw. Item 0 zeigt die Größe der Tabelle. Man muss beachten, dass man sicherstellt das das Programm nicht über das Ende der Tabelle hinaus liest, da sonst seltsame Effekte passieren.

Beispiel:

```
'PIC Settings
#chip 16F88, 20
#config MCLR_OFF

'Hardware Settings
#define LED PORTB.0
dir LED out

'Main Routine
ReadTable TimesTwelve, 4, Temp
set LED off
if Temp = 48 then set LED on

'Lookup table named "TimesTwelve"
Table TimesTwelve
 12
 24
 36
 48
 60
 72
 84
 96
 108
 120
 132
 144
End Table
```

Compiler Directives

#chip

Syntax:

#chip *model, speed*

Explanation:

Mit der **#chip** Directive wird der Chip angegeben und die Geschwindigkeit (Osc.Frequ.) bestimmt die durch GCBASIC compiliert wird. ***Model*** ist der jeweilige PIC Chip (zB.16f876) ***speed*** ist die Geschwindigkeit des Chip in MHz und ist erforderlich für die Verzögerung (delay) und PWM Routinen.

#config

Syntax:

#config *option1, option2, ... , optionn*

Beschreibung:

Die #config Directive wird zur Spezifikation der configuration options des Chip verwendet. Dies wird genau am Anfang des Handbuches im Bereich Configuration erklärt.

See Also:
Configuration

#define

Syntax:

#define *Find Replace*

Beschreibung:

#define sucht durch das Programm nach ***Finde*** und ersetzt es durch den Wert für ***Replace.***

See Also:
Defines

#ifdef

Syntax:

#ifdef** **Constant | Constant Value | Var(VariableName)
...
#endif

Beschreibung:

Die **#ifdef** Directrive wird verwendet für ausgewählte Freigabeabschnitte des Code. Es gibt dabei drei Arten der Verwendung.

Der Vorteil bei der Verwendung von **#ifdef** liegt eigentlich darin eine gleichwertige Folge von IF Aussagen in der Summe des auf den Chip downgeloadeten Code zu erreichen. **#ifdef** steuert welcher Code compiliert und downgeloaded wird.

GCBASIC unterstützt die #ifndef Directrive, das ist das Gegenteil der #ifdef Directrive, es löscht den Code der #ifdef hinterlässt und umgekehrt.

Beachte: Der Code in den folgenden Beispielen wird nicht compiliert, da er ohne #chip Directrive und DIR Befehl geschrieben ist. Es ist nur als Beispiele gedacht.

Enabling code if a constant is defined

Syntax Example:

```
#define Blink1

#ifdef Blink1
        PulseOut PORTB.0, 1 sec
        Wait 1 sec
#endif
#ifdef Blink2
        PulseOut PORTB.1, 1 sec
        Wait 1 sec
#endif
```

Dieser Code pulst den PORTB.0 aber nicht PORTB.1. Dies ist weil Blink1 definiert wurde aber nicht Blink2. Wenn die Zeile

```
        #define Blink2
```

zusätzlich am Anfang des Programms steht, dann würden beide Pins pulsen. Der Wert der definierten Konstanten ist nicht wichtig und kann bei #define weitergehen.

Enabling code if a constant is defined and has a given value

Syntax Example:

```
#define PinsToFlash 2

#ifdef PinsToFlash 1,2,3
        PulseOut PORTB.0, 1 sec
#endif
#ifdef PinsToFlash 2,3
        PulseOut PORTB.1, 1 sec
#endif
#ifdef PinsToFlash 3
        PulseOut PORTB.2, 1 sec
#endif
```

Dieses Programm verwendet eine Konstante mit der Bezeichnung PinsToFlash welche steuert wie viele Lämpchen pulsen. PORTB.0 pulst wenn PinsToFlash gleich den Wert 2 oder 3 hat und PORTB2 flashed wenn PinstoFlash gleich 3 ist.

Enabling code if a system variable is defined

Syntax Example:

```
#ifdef NoVar(ANSEL)
        SET ADCON1.PCFG3 OFF
        SET ADCON1.PCFG2 ON
        SET ADCON1.PCFG1 ON
        SET ADCON1.PCFG0 OFF
#endif
#ifdef Var(ANSEL)
        ANSEL = 0
#endif
```

Der obige Codeabschnitt hat direkt in das ADCON1 Contol Register kopiert. Dieses Register stellt die AD Ports mit den PCFG (0 – 3) Bits als Analog (0) oder I/O (1) ein. Wenn ANSEL nicht als Systemvariable für einen bestimmten Chip deklariert ist, dann benutzt das Programm ADCON1 zum steuern der Port Modi. Wenn ANSEL definiert ist, dann ist der Chip neuer und seine Ports werden beim löschen von ANSEL als I/O gesetzt.

Enabling code if a system bit is defined

Darüber hinaus, ausgenommen mit Bit und NoBit beziehungsweise an der Stelle von Var und NOVar.

See Also:

Defines

#define

#include

Syntax:

#include *filename*

Beispiel:

#include teilt GCBASIC mit ein anderes File zu öffnen, liest alle Unterprogramme und Konstanten ein und kopiert diese dann in das aktuelle Programm.

Es gibt zwei Formen von include – absolut und relativ.

Absolut weist das File dem Pfad C:\Program Files\GCBASIC\include directory zu. Der Name des File wird angegeben zwischen < und > Symbolen. Zum Beispiel wird beim Einfügen (include) des File „srf04.h“ folgende Directive stehen

#include <srf04.h>

Relativ wird verwendet um Files in den gleichen Abschnitt des gegenwärtigen ausgeführten Programms einzulesen. Filenames sind eingeschlossen in Anführungszeichen wie folgt:

#include "mycode.h"

in dem mycode.h der Name des Files ist der eingelesen wird.

Es ist nicht notwendig das der eingefügte File Name mit .h endet. Der wichtige Teil ist, dass der Name der GCBASIC übergeben wird der exakte Name des File ist der eingefügt wird.

Das was mit #include eingefügt wird kann in assembler oder C geschrieben sein. Man muss bedenken, dass #include bei GCBASIC anders arbeitet als #include in anderen Programmiersprachen – Code ist nicht am Ort von #include eingefügt sondern vielmehr am Ende des aktuellen Programms.

#script

Syntax:

#script
[*scriptcommand1*]
[*scriptcommand2*]
...
[*scriptcommandn*]
#endscript

Beschreibung:

Der #script Block wird verwendet um kleine Codeabschnitt zu erstellen, welche mit GCBASIC während der Compilation laufen. Siehe weitere Erklärung im Script Artikel.

See Also:
Scripts

#startup

Syntax:

#startup *SubName*

Beispiel:

#startup wird verwendet um include Files beim automatischen einfügen in inizialisierte Routinen zu unterstützen.

Es gibt einige Begrenzugen bei dieser Directrive. Es kann nur einmal vorkommen bei einem File und keine Parameter können für das ausgeführte Unterprogramm spezifiziert werden.

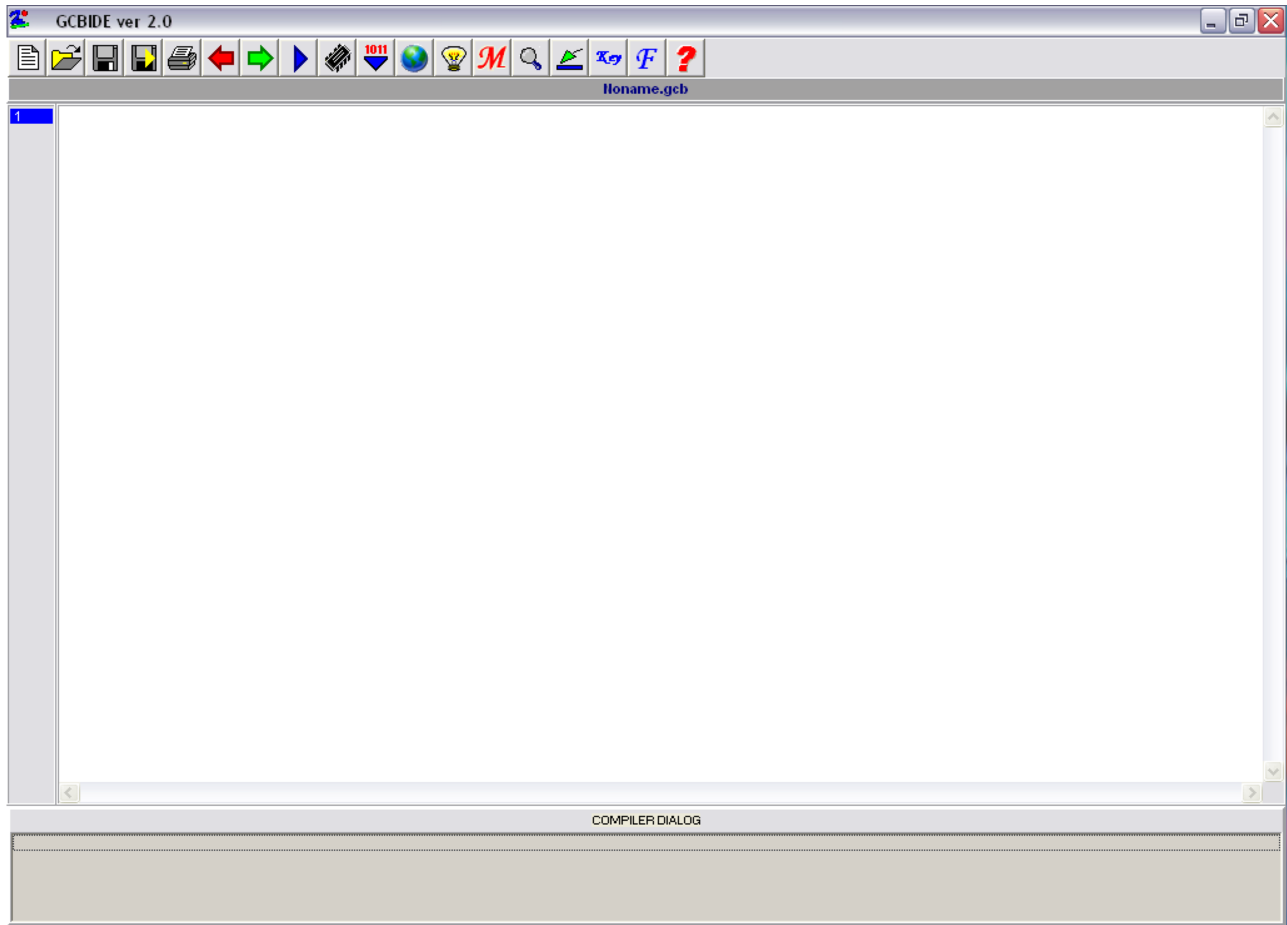
#mem

This directive is obsolete. GCBASIC can now determine the amount of memory on a chip automatically, and will ignore the #mem directive

GCBIDE

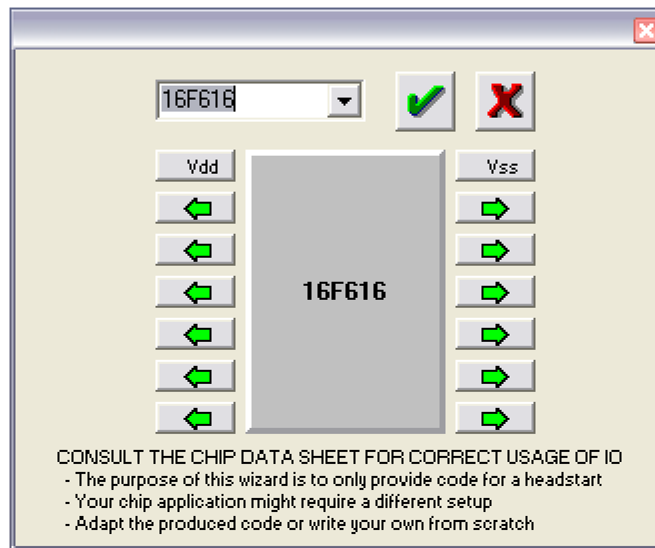
GCBIDE ist ein Editor zum vereinfachten schreiben der GCBASIC – Programme. Viele zusätzliche Features erleichtern die Arbeit.

Editorfenster

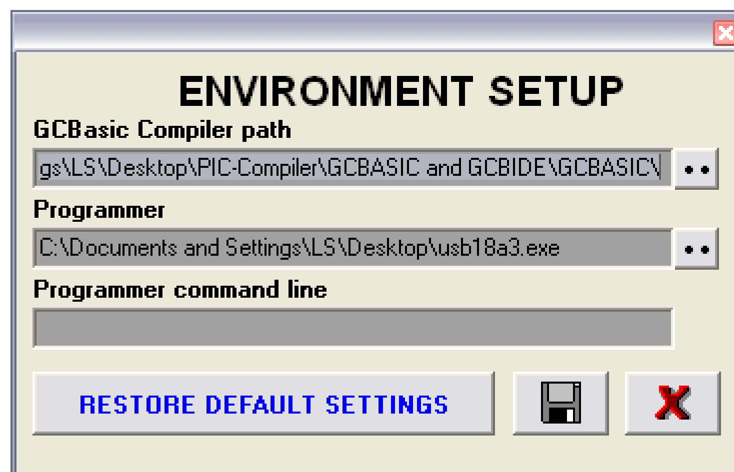


- **oben die Taskleiste mit den einzelnen Features**
- **mitte das Editorfenster**
- **unten das Compilerfenster**

Chip Auswahl



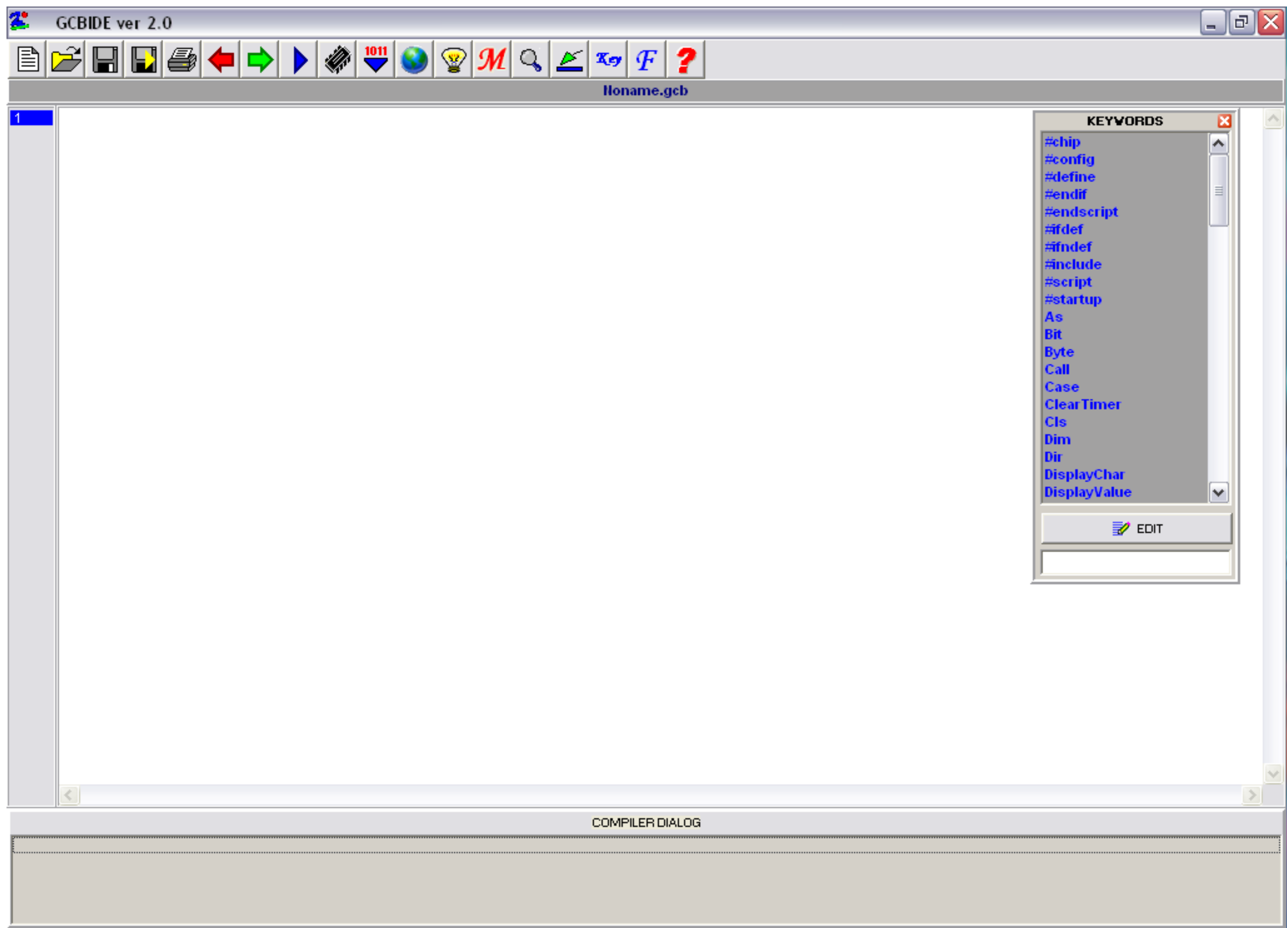
Umgebungseinstellung



Im Compiler path wird angegeben in welchem Pfad sich der Compiler befindet.

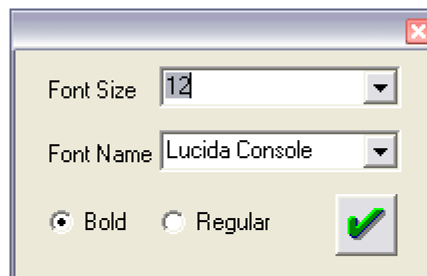
Programmer path ist gleich dem Ort des MC – Programmer – Software.

Keywords



Hier können die Befehle angeklickt werden, wobei sie dann im Programm gesetzt werden.

Font Settings



Hier wird die Buchstabengröße und Schrift festgelegt.