

Handbuch

iL_BAS16

BASIC Compiler für PIC Microcontroller

Compiler Version 6.0-4

compiliert: Januar 2010

gedruckt: 17.01.10

(c) Copyright
Ing.Büro Stefan Lehmann
Fürstenbergstraße 8a
D-77756 Hausach

Tel. ++49 (0)7831 452
Fax ++49 (0)7831 96 428
eMail: SL@iL-online.de
www.iL-online.de

PIC is a registered trademark of Microchip Technology Inc.

iL-TROLL is a registered trademark of Ing.Büro Lehmann

Allgemeines	
Einleitung	1
Änderungen in dieser Beschreibung	3
Kurzbeschreibung des Compilers	5
Neues in Version 6	6
Grundsätzlich gilt:	8
Installation für Windows	9
Installation für MPLAB	12
Projekt - Datei	20
Einsteigerprobleme	21
Editor	23
Debug (Entwanzen)	27
Produktinfo	28
iL-TROLL ®	29
 BASIC-Befehlsübersicht	
BASIC-Befehle	32
 Compilerschalter	
Compilerschalter	34
\$CCON und \$CCOFF	36
\$DEBUG	37
\$IF \$ELSE \$ENDIF	38
\$INCLUDE	39
\$LIST	40
\$LRANGE	41
\$LST2COD	43
\$NCALDEF	44
\$OBJ2HEX	45
\$OLDVAR	46
\$OSCCON	47
\$SELFPRG	48
\$TROFF	49
\$TRON	50
\$VCFG / \$VCFG n	51
\$WDTUSR	52
DEFINE Variable, Konstante	53
DEFINE Prozessor	55
DEFINE sonstige	57
DATE	58
TIME	59
XTAL	60
 Programmaufbau	
Assembler Code	61
Interrupts (allgemein)	62
Labels	65
Konstanten	66
Variablen	67
Rechenzeit optimiert programmieren	72
Tabelle für Variablenadressen im PIC	73
IF-Konstrukte	78
Komparator	79
Programm-Pages	80

8- und 16-Bit Arithmetik	
Mathematische Operatoren (8-/16-Bit)	81
Logische Operatoren (8-/16-Bit)	83
32-Bit Arithmetik	
32-Bit Arithmetik Einführung	84
Mathematische Operatoren (32-Bit)	85
Logische Operatoren (32-Bit)	86
BASIC-Befehlssatz	
ADDELAY	87
ADINP	88
ASM	91
BINTOASC	92
BINTOBCD	93
BINTODEC	94
BITPOS	95
CALVAL	96
CLOCK und CLOCK1	97
CLRWDT	99
CONF	100
CONFIG	101
CURSOFF	103
CURSON	104
DATA	105
DEC	106
DELAY	107
DOZE	108
DTMFOUT	109
EEDATA	111
END	112
ENDASM	113
ERR	114
FOR TO NEXT	115
FREQIN	116
GOSUB	117
GOTO	118
HIGH	119
I2CDELAY	120
I2CHARDS	121
I2CINIT	123
I2CRD	124
I2CRDB	125
I2CRDBN	126
I2CREAD	127
I2CSLAVE	128
I2CSP	129
I2CST	130
I2CWR	131
I2CWRB	132
I2CWRITE	133
IF...THEN...ELSE	134
INC	135
INKEY	136

INP	138
INPUT	139
INTERRUPT	140
INTEND	142
INTPROC	143
LCDCHAR	144
LCDCLEAR	145
LCDCTRST	146
LCDDELAY	147
LCDINIT	148
LCDTYPE n (,var)	150
LCDWRITE	152
LET	154
LOCATE	155
LOFREQ	156
LOOKDN	157
LOOKUP	158
LOW	159
ON GOSUB	160
ON GOTO	161
OUTP	162
OUTPUT	163
PEEK	164
POKE	165
PRINT (z.Z. nur	166
PULSIN	167
PULS_IN	168
PULSOUT	169
PWM	170
RANDOM	171
RCTIME	172
READDATA	173
READ	174
REM	175
RES	176
RESTORE	177
RETURN	178
REVERS	179
SELFPROG	180
SERIN	182
SEROUT	184
SET	186
SETBAUD	187
SLEEP	188
SOUND	189
SWAP	190
TXDELAY	191
TOGGLE	192
TRIS	193
VARPTR	194
WAIT	195
WRITE	196
Schnittstellen	
I2C-Schnittstelle	197

Assembler	
ASSEMBLER (allgemeines)	198
Assemblerdirektiven	199
Assembler Grundbefehlssatz	203
Hilfsprogramme	
BaudCalc	207
LCD Character Designer	208
7-Segment Pattern Designer	209
Anhang I	
Inhalt von DEFAULT.EQU	210
Anhang III	
Unterstützte Bausteine	211
Anhang IV	
FAQs und Bemerkungen	212

Einleitung

Diese Beschreibung deckt alle vier Compilerversionen (iL_BAS16TR, iL_BAS16SES, iL_BAS16STD, iL_BAS16SEP und iL_BAS16PRO) ab. Die Professional Version (iL_BAS16PRO) unterstützt fast die gesamte PIC-Familie 10F20x, 12Xxx und 16Xxx. Sie ist für den professionellen Einsatz gedacht. Die Standardversion bietet dem Hobbytät ein ideales Preis-/Leistungsverhältnis. Die Sonderversionen SES und SEP sind als preiswerter Einstieg gedacht. Aufrüsten auf die nächste Leistungsstufe kostet jeweils nur den Differenzbetrag (Ausnahme: ein Upgrade von Standard auf SEP ist nicht möglich). Welche Version welchen Prozessor unterstützt ist im Anhang "Unterstützte Prozessoren" aufgelistet.

Die Version iL_BAS16TR kennt einige BASIC-Befehle nicht. Dies ist bei den einzelnen Befehlen deutlich gekennzeichnet. iL_BAS16TR wurde eigens für die iL_TROLLs entwickelt. Es handelt sich dabei um vorprogrammierte Bausteine. Sie haben 2k Worte Programmspeicher und ca. 160 Byte für Daten. Es sind 2 8-Bit Ports (RB und RC) vorhanden. Ebenso verfügt er über 4 Analogeingänge (AN0 - AN3). Der eingebaute Kern erlaubt es diese Bausteine ohne Programmiergerät zu programmieren. Man muss diesen Baustein lediglich über einen Leitungstreiber (TTL <-> RS232) an die serielle Schnittstelle des PCs anschließen. iL_EDy stellt dann einen **DOWNLOAD**-Knopf zur Verfügung. Somit bieten diese iL_TROLLs den gleichen Programmierkomfort wie z.B. PICs mit eingebautem BASIC-Interpreter. Im Gegensatz zu diesen wird aber in die iL_TROLLs ein mittels Compiler übersetztes Programm geschrieben, so dass dessen Ausführungsgeschwindigkeit ein Vielfaches der Interpreter Bausteine ist.

Die Prozessorfamilie **PIC10F20x, PIC12C5xx, PIC12C6xx, PIC16C5x, 16C7x, 16C8x und 16F8x** von Microchip umfasst Bausteine, die durch ihr günstiges Preis-/Leistungsverhältnis in Applikationen vordringen, die bisher der festverdrahteten Logik vorbehalten waren. Speziell die seit Ende 2004 eingeführten PIC 10F20x im SOT23 Gehäuse dringen in einen Bereich ein, der den Microcontrollern bisher nicht zugänglich war. Daneben wird bei der Überarbeitung vieler Microcontrollerschaltungen der Controller durch einen PIC ersetzt, da der alte Baustein hoffnungslos überdimensioniert oder nicht mehr zeitgerecht ist. Seine einfache Handhabung und sein kompakter Befehlssatz machen diese Bausteinfamilie so erfolgreich und weit verbreitet. Das Ingenieurbüro Lehmann hat es sich zur Aufgabe gemacht, für diese Bausteine Entwicklungswerkzeuge anzubieten, die auch für den schmalen Geldbeutel erschwinglich sind. Dennoch stand bei deren Entwicklung die Professionalität im Vordergrund. Für eigene Applikationen werden diese Bausteine seit 1992 eingesetzt. Aus dieser langen Erfahrung heraus ist ein BASIC-Compiler entstanden, der den täglichen Anforderungen des Entwicklungsingenieurs standhält. Neue interessante Bausteine werden nach und nach in die Liste der unterstützten Controller aufgenommen. Kompetente Unterstützung bei Problemen runden das Leistungsangebot ab.

Der BASIC-Compiler iL_BAS16xxx ist für Bausteine entwickelt worden, deren Kennzeichen es ist, einen sehr kleinen Programmspeicher zu besitzen. Fragt man nach dem Sinn und Zweck eines Hochsprachencompilers für solche Microcontroller, so hört man oft das Argument, dass sehr viel Performance durch unnötigen Code verloren geht. Dieser Codeüberhang ist in der Regel bei allen Compilern anzutreffen. Eine gewisse Abhilfe schafft hier die **Codeoptimierung**, die zum einen den redundanten Code entfernt und zum anderen bereits bei der Compilierung unterschiedlichen und deshalb effizienteren Code erzeugt. Auch die Laufzeitbibliothek wird so optimiert, dass immer nur die notwendigen Module mit eingebunden werden. Diese Fähigkeit des Compilers, situationsabhängigen Code zu generieren, ist nicht einfach zu realisieren und erfordert viel Wissen um die internen Zusammenhänge im PIC und im Compilerbau.

iL_BAS16xxx ist ausschließlich für die PIC-Controller von Microchip entwickelt worden. Somit kann der generierte Code sehr spezifisch auf die Eigenheiten des PICs optimiert werden.

Mitgeliefert wird EDy, eine **Entwicklungsumgebung** für den PIC-BASIC-Compiler iL_BAS16 und dessen Zubehör. Dies sind u.a. der PIC-Assembler iL_ASS16, der PIC-Simulator iL_SIM16 und das PIC-Programmiergerät iL_PRG16. Falls die Professionalversion vorliegt, ist auch das Programmmodul iL_PAGE0.EXE vorhanden. Dieses Modul ist für die Berechnung der Sprünge und Unterprogrammaufrufe über Programmspeicherseiten hinweg zuständig. Der Editor umfasst insgesamt 8 Arbeitsblätter, wobei das erste und letzte Blatt eine Sonderstellung einnehmen. Wird der Compiler (auch Simulator, Programmiergerät usw.) gestartet, wird der Name des ersten Arbeitsblattes als Parameter an das aufgerufenen Programm übergeben. Eventuelle Fehlermeldungen erscheinen dann auf dem letzten Blatt. Die übrigen Arbeitsblätter dienen zum Anlegen von INCLUDE-Dateien oder auch für die Dokumentation.

Einleitung (cont.)

Seit Januar 2006 wird auch ein **Remote-Debugger** für die Bausteine 16F628, 16F88, 16F873 -16F877 und iL-TROLL mitgeliefert. Durch Setzen eines einfachen Compilerschalters wird ein Code generiert, der über eine bidirektionale 1-Draht-Verbindung mit dem PC kommunizieren kann. Mit den Befehlen EINZELSCHRITT, KONTINUIERLICH, RESET und BREAKPOINT lassen sich selbst schwierige Programme einfach testen. Die Generierung des Debugcodes lässt sich mittels \$TROFF und \$TRON steuern um so auch zeitkritische Routinen in Echtzeit ablaufen lassen zu können.

Der PIC-Basic-Compiler iL_BAS16 ist speziell auf die Belange der PIC10F20x, PIC12- und PIC16-Familie zugeschnitten. Er zeichnet sich durch einen sehr umfangreichen und leistungsfähigen Befehlssatz aus. Der erzeugte Code ist sehr kompakt und sehr schnell. Aus diesem Grund ist es fast nicht mehr nötig, bestimmte Programmsequenzen aus Gründen der Geschwindigkeitsoptimierung in Assembler zu schreiben.

Änderungen in dieser Beschreibung

Hier eine Liste der letzten Änderungen:
(bezogen auf das jeweilige **Kapitel**)

15.01.2009

Ergänzungen zu \$VCFG (Kapitel: Compilerschalter)

05.01.2009

Nutzbarer RAM-Speicher bei der 18er-Familie (Kapitel: Variablen)

ADCFG- vs. ANSEL-Tabelle bei ADINP (Kapitel: ADINP)

Komparatoren zusammen mit ADC (Kapitel: Komparatoren)

05.12.2009

Ergänzende Hinweise zum Variableneinsatz und den Bankgrenzen
(Kapitel: Variablen)

02.09.2009

Ergänzende Hinweise bei den ROT-Befehlen
(Kapitel: Mathematische Operatoren (8-/16-Bit))

21.08.2009

Kapitel CONF und CONFIG wurden ergänzt

17.04.2009

Compilerversion 6.0

Neues zu Version 6

22.01.2009

LCDCTRST neuer Befehls

CONFx neuer Befehl

CONFIG Ergänzungen

03.12.2008

LCDCHAR Neuer Befehl erlaubt das definieren von eigenen Zeichen für die Standard-LCDs

15.09.2008

PICKit2 als Programmiergerät direkt aus iL_EDy (**Editor**) aufrufbar.

14.07.2008

PRINT_8_16 (wegen Kompatibilität zu Versionen vor 2002).

NEU: **ROTR8** und **ROTL8**

28.02.2008

in der Beschreibung der **I2CINIT**- und **I2CSLAVE**-Funktion.

bei **DEFINE Prozessor** => CPD_ON und CPD_OFF

22.11.2007

Compilerschalter **\$VCFG** wurde um ein Argument erweitert. Dieses entspricht dem direkten Wert der VCFG-Bits im ADCON1 Register. Da es hier große Unterschiede bei den einzelnen PICs gibt, muss man in jeweiligen Datenblatt nachschlagen.

PRINT-Befehl funktioniert jetzt wieder. Die Funktion BIN2ASC wurde verbessert und dabei der richtige Einsprung für PRINT verschoben.

Änderungen in dieser Beschreibung (cont.)

15.08.2007

Beschreibung, wie die Option STACK bei den Interrupts verwendet wird, wurde im Kapitel **Interrupt (allgemein)** und der Befehlsbeschreibung **INTERRUPT**, verbessert.

07.03.2007

Editor können jetzt Bereiche mit der Funktion JALOUSIE auf- und zugeklappt werden.
name = AUTO(6) definiert ein 6 Byte großes Feld

01.02.2007

I2CSP

(neu)

Kurzbeschreibung des Compilers

Rechtliches:

Die Sharewareversion entspricht einer Vollversion mit der Begrenzung des Bausteins auf den PIC 16C83. Diese darf und soll im vollen Umfang kopiert und an andere Interessenten verteilt werden. Die Nutzung im kommerziellen Bereich beschränkt sich allerdings auf max. zwei Projekte. Somit kann diese Software ausgiebig geprüft werden.

Die Vollversion dieses Programms unterliegt dem Urheberrecht und darf deshalb nicht vervielfältigt werden. Kopien sind einzig und allein dem Lizenznehmer im Rahmen von Sicherungskopien erlaubt. Zuwiderhandlungen werden strafrechtlich verfolgt.

Der Compiler:

Der Compiler übersetzt ein BASIC-Programm in Maschinensprache. Dabei wird sofort versucht, das Ergebnis zu optimieren. Dazu werden redundante Befehle soweit wie möglich automatisch entfernt. Allerdings ist das Ergebnis u.U. manuell noch weiter zu verbessern, da die Überprüfung nur innerhalb einer Funktion geschieht. Ergibt sich redundanter Code zwischen zwei Funktionen, wird dieser nicht entfernt. Die Laufzeitroutinen werden nur dann in das Programm eingebunden, wenn auch wirklich ein entsprechender Aufruf erfolgt.

In der jetzigen Standardversion unterstützt das Programm nur Bausteine, deren Programmspeicher nicht größer als eine Seite ist. Bei den 12-Bit-PICs sind die 512 Programmschritte, bei den 14-Bit-PICs sind es 2048 Programmschritte. Die Berechnung von Programmverzweigungen über die Seitengrenzen hinaus wird nicht unterstützt. Eingebaute Hardwaregruppen wie beispielsweise der AD-Wandler, der UART oder das EEROM Daten-RAM werden selbstverständlich unterstützt. Ebenso kann bei diesen interruptfähigen Bausteinen eine Zeitbasis implementiert werden. Das Einklinken eigener Interruptserviceroutinen ist wie bei der Professionalversion möglich.

Die Professionalversion unterstützt die Bausteine 10F200 - 10F206, 12C508, 12C509, 16C53 - 16C58, 16C61 - 16C66, 16C71 - 16C74, 16C83 - 16C84, 16F83 - 16F84 sowie 16F87x. Weitere werden folgen. Da die Implementierung der pro Baustein unterschiedlichen Hardwareresourcen äußerst komplex ist, wird diese sukzessive realisiert. Hier möchte ich die Anwender dazu einladen mir Lösungsvorschläge mitzuteilen. Der zusätzliche RAM-Speicher des 16C57 kann als Datenarray genutzt werden (siehe Variablen).

Updates und Upgrades:

Wie jedes Softwareprodukt, werden auch die Funktionen des BASIC-Compilers iL_BAS16 laufend verbessert und ergänzt. Unser kostengünstiger UPDATE- bzw. UPGRADE-Service erlaubt es Ihnen immer, mit den neusten Programmversionen zu arbeiten.

Auf unserer Homepage finden Sie die neusten Hinweise auf Befehlserweiterungen, Verbesserungen usw. Der Update-Service läuft in der Regel über eMail. Alle Kunden können 6 Monate lang nach Kauf des Produktes ihr Produkt nach vorheriger Rücksprache updaten. Danach beträgt die Updategebühr zwischen 6 und 35 Euro pro Teil. Das Aufrüsten (Upgrade) von der Standardversion auf die Professionalversion kostet jeweils nur den Differenzbetrag.

Weitere Informationen finden Sie im Internet unter www.iL-online.de

Neues in Version 6

Die Version 6 unterstützt nun auch einige Microcontroller der 18er Familie.

Diese Familie unterscheidet sich von den 10Fxxx, 12Fxxx und 16Fxxx nicht nur in der Registerstruktur und der fehlenden Programmspeichersegmentierung sondern auch im Bereich der Konfigurationsworte. Damit hier wieder eine einheitliche Handhabung gewährleistet werden kann, mussten einige Änderungen in diesem Bereich durchgeführt werden.

DEFINE DEVICE darf nur noch der Bausteintyp stehen; weitere Argumente sind nicht erlaubt.

jedem CONFIG(*) darf ebenfalls nur ein Argument stehen.

ein CONFIG nach einem CONFx aufgeführt, hat dieses eine höhere Priorität. Damit kann man eine Standardkonfiguration mittels CONFx einstellen, um dann gezielt wichtige Konfigurationsbits umzudefinieren.

Programmspeicher ist aber noch auf 64k begrenzt.

neue Compilerversion benötigt auch die aktuellen Versionen des Compilers und ggf. des iL_PAGE-Moduls.

jetzt muss für Controller mit ANSEL-Register auch dieses definiert werden. Eine ADCFG-Definition erzeugt eine Fehlermeldung, umgekehrt ebenfalls.

Compilerschalter \$INITVAR werden alle definierten Variablen auf 0 initialisiert.

WICHTIG!

(*) bei den PIC18-Typen sind noch nicht alle CONFIG-Angaben implementiert. Deshalb unbedingt mit CONF arbeiten.

Im Detail:

In dieser Version wurden einige Änderungen durchgeführt die für die Implementierung weiterer PIC16-Typen aber vor allem der PIC18-Typen notwendig wurden. Die größte Änderung gab es im Bereich der Konfigurationsbeschreibung. In der DEVICE-Zeile darf nur noch der Prozessor und die ADCFG-Anweisung stehen. Die ADCFG-Angabe kann jedoch wie die anderen Angaben hinter der CONFIG-Anweisung stehen. Pro CONFIG darf nur eine Angabe stehen. Z.B: CONFIG WDT_ON. Bei manchen Configurationsangaben sind mehrere Bits möglich. So kann beispielsweise WRT auf mehrere Bits aufgeteilt sein. In diesem Fall wird die gewünschte Maske als Zahl angefügt. Soll WRT0 = 0 und WRT1 = 1 sein, so schreibt man: CONFIG WRT_ON 2. Der Compiler verschiebt die Maske so, dass sie an der richtigen Stelle zu liegen kommt. Damit diese Flexibilität erreicht werden kann und zukünftige neue Configurationsbits leichter implementiert werden können, werden diese Parameter in der iL_BAS16.PIC-Datei entsprechend definiert.

Assembler

Auch der Assembler und das Page0-Modul mussten entsprechend angepasst werden.

Der neue Compiler Ver. 6 funktioniert n u r mit dem neuen Assembler und Page0-Modul zusammen.

Programmiersoftware

Auch die Programmiersoftware für den PC musste leicht angepasst werden. Auch sie kann nicht mehr mit den OBJ- bzw. Hex-Dateien des alten Compilers / Assemblers arbeiten.

Neuerungen / Änderungen

Bei PICs mit ANSEL-Register muss auch dieses definiert werden. Ein ADCFG führt zu einer Fehlermeldung. Das Argument hinter ADCFG bzw. ANSEL ist nun durch ein Leerzeichen getrennt.

Z.B. CONFIG ANSEL 10 oder ADCFG 4,R

Falls es zwei ANSEL-Register gibt (ANSEL und ANSELH) müssen auch beide gesetzt werden und die Formatanweisung R bzw. L muss in beiden Fällen identisch sein.

Bei PICs mit internem Oszillator dessen Frequenzen umgeschaltet werden können, muss mittels \$OSCCON die gewünschte Frequenz eingestellt werden. Im Gegensatz zu LET OSCCON = xx sorgt \$OSCCON dafür, dass bereits in der Initialisierungsphase die Taktfrequenz umgeschaltet wird. Bei LET OSCCON erfolgt dies zu einem späteren Zeitpunkt. Dieser Umstand ist vor allem bei der Nutzung von SELFPROG und DEBUG von großer Bedeutung!

Neues in Version 6 (cont.)

A propos DEBUG und SELFPROG: Denken Sie immer daran, dass der MCLR-Pin aktiv ist (MCLR_Ext). Außerdem darf beim Debugging kein Programmcode durch ASM und ENDASM generiert sein. Diese Sequenz muss auskommentiert sein.

Bei den PIC18-Typen ist ein Zugriff auf die Adressen 00H bis 7FH eigentlich verboten und wird auch mit einem entsprechenden Fehler quittiert. Nun kann es manchmal aber doch interessant sein, auf Compilervariablen, die in diesem Bereich liegen, zuzugreifen. Das lässt sich über folgende Anweisung lösen:

```
DEFINE ERG_TEMP = ERG.
```

ERG kennt der Compiler aus internen EQU-Anweisungen.

Vermeiden Sie aber solche Tricks! Die Gefahr, dass es schief geht ist sehr groß.

Die Programmierung der PIC18Fxxx kann nicht mit dem iL_PRG16PRO oder iL_ISP_U erfolgen. Damit aber die erzeugten Dateien Microchip kompatibel werden müssten die Compilerschalter \$/M_OBJ und \$OBJ2HEX gesetzt sein. Der Compiler nimmt dies im Falle der PIC18Fxxx als Defaulteinstellung. Daher müssen diese Schalter nicht explizit gesetzt werden.

Üblicherweise werden, im Gegensatz zu C, im Standard-BASIC und auch in PASCAL die Variablen mit 0 initialisiert. iL_BAS16 erlaubt es nun beide Fälle auszuwählen. Früher wurden keine Variablen initialisiert, nun steht dafür der Compilerschalter \$INITVAR zur Verfügung. Ist er aktiv, werden alle Variablen (Byte, Word, Dblword und Bit) auf 0 gesetzt. Da dies natürlich Speicherplatz benötigt, ist es dennoch sinnvoll ohne Schalter zu arbeiten und ggf. im Programm diese zu initialisieren.

Grundsätzlich gilt:

WICHTIG!!!

Beim iL_BAS16 gelten folgende Einschränkungen:

Pro Zeile nur ein Befehl bzw. eine Anweisungen.

Ausnahme REM bzw (')

Der Gebrauch von Doppelpunkten ist nicht möglich.

Ausnahme LABEL (Sprungmarken).

Innerhalb eines Befehls oder einer Funktion dürfen
keine Ausdrücke stehen

(z.B. INC AA * 2)

sondern nur Variablen und Konstanten.

Array-Variablen können nur in einfachen Zuweisungen
verwendet werden.

In den Vergleichsoperationen von IF-Abfragen dürfen keine
arithmetischen und logischen Verknüpfungen stehen.

Diese sind vor der IF-Anweisung durchzuführen.

Bei Symbolen

und Labels (also Variablennamen, Sprungmarken, Konstanten-Namen) gilt :

>>>

müssen mit Buchstaben beginnen und mindestens 2 Zeichen lang sein

>>>

die ersten 16 Zeichen sind signifikant.

>>>

außer im ersten Zeichen sind auch Ziffern und folgende Zeichen erlaubt ...

>>>

Umlaute sind erlaubt

>>>

es wird nicht zwischen Groß- und Kleinschreibung unterschieden

Verändern Sie

nie

das OPTION-Register.

Zeitabhängige Befehle können bis zu 10% von Nennwert abweichen

Installation für Windows

iL_BAS16 wird auf einer CD oder per eMail ausgeliefert. Auf der CD liegen die Dateien ungepackt vor. Ausnahme sind die Microchip Datenblätter im Unterverzeichnis DOC. Im Fall des eMail-Versands sind alle Dateien gepackt und es fehlen die Datenblätter. Diese kann man problemlos aus dem Internet herunterladen. Da die Namen der Datenblätter letztendlich nur aus Zahlen bestehen, gibt es keinen logischen Zusammenhang zwischen Dateiname und dem beschriebenen PIC. Welche Datei vom Internet heruntergeladen werden muss steht in der Datei PIC_LIT.LST.

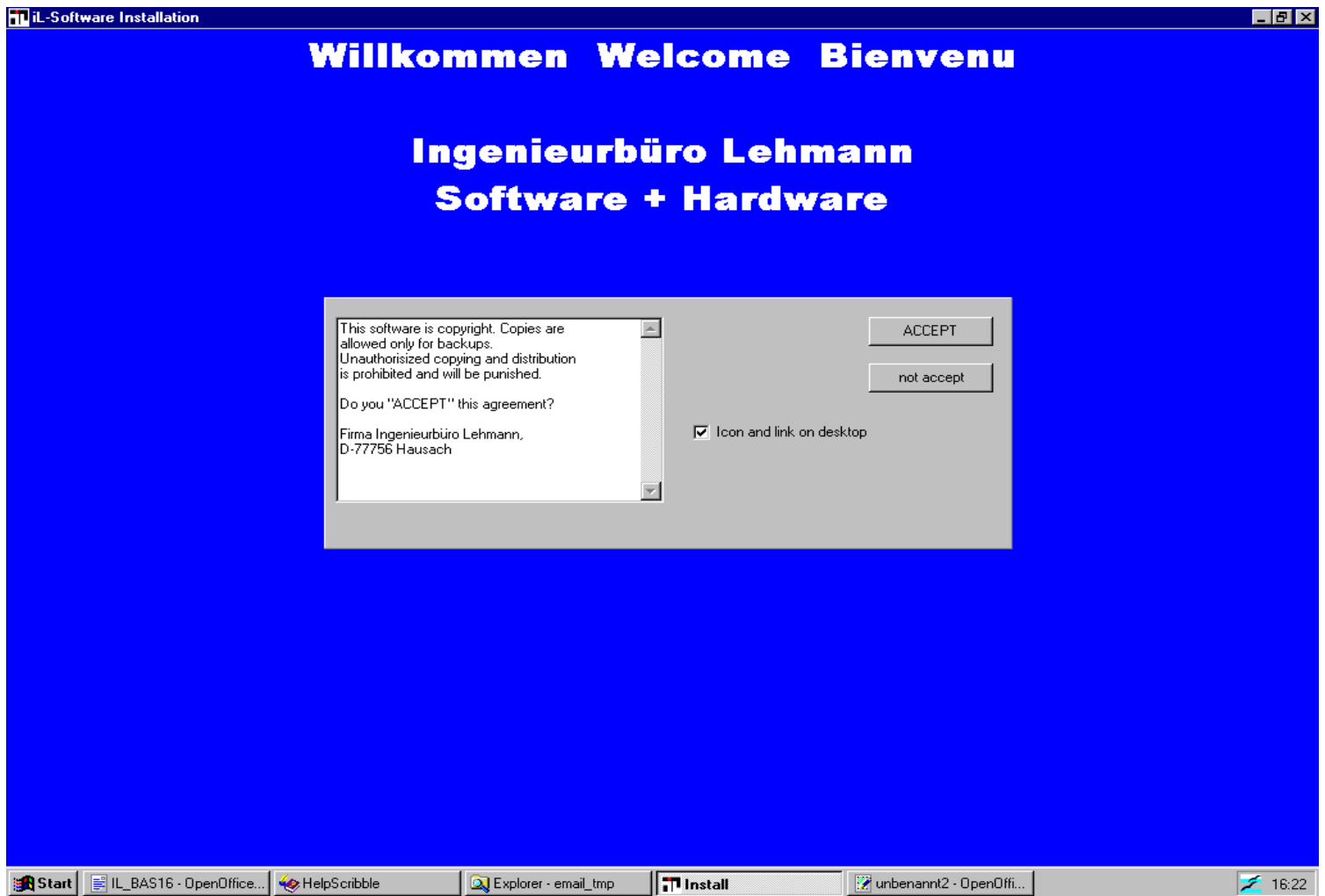
Liegt die Software im gepackten Format vor, muss dieses zuerst entpackt werden. Manche Entkomprimierer unterstützen den Start von gepackten Dateien. Sollte Sie diese Möglichkeit nicht besitzen, entpacken Sie in ein neues, temporäres Unterverzeichnis. Die CDs haben keine Autostartfunktion. Um die Installation zu starten, öffnen Sie den Explorer und klicken Sie auf INSTALL.EXE.



Es erscheint folgendes Fenster. Wählen Sie hier die Sprache aus, mit der der Compiler und der Editor arbeiten sollen. Bei der Installation werden dann, alle Programmmodule für diese Sprache konfiguriert. Sollten Sie widererwartend die Sprache wechseln wollen, müssen Sie eine Neuinstallation durchführen.

Danach erscheinen die nachfolgenden Fenster.

Installation für Windows (cont.)



Zum Abschluss werden alle Dateien und Unterverzeichnisse in das ausgewählte Verzeichnis kopiert. Falls Sie die "automatische Verknüpfung auf dem Desktop" aktiviert hatten, wird diese nun erstellt. Außer dem Unterverzeichnis und der Verknüpfung werden keinerlei weitere Änderungen oder Einträge vorgenommen. Auch die Konfigurationsdateien bleiben lokal in diesem Verzeichnis. Damit lässt sich der Compiler durch einfaches Löschen des Verzeichnisses und der Verknüpfung aus dem Rechner entfernen.

Es erscheint folgendes Fenster.

Installation für Windows (cont.)



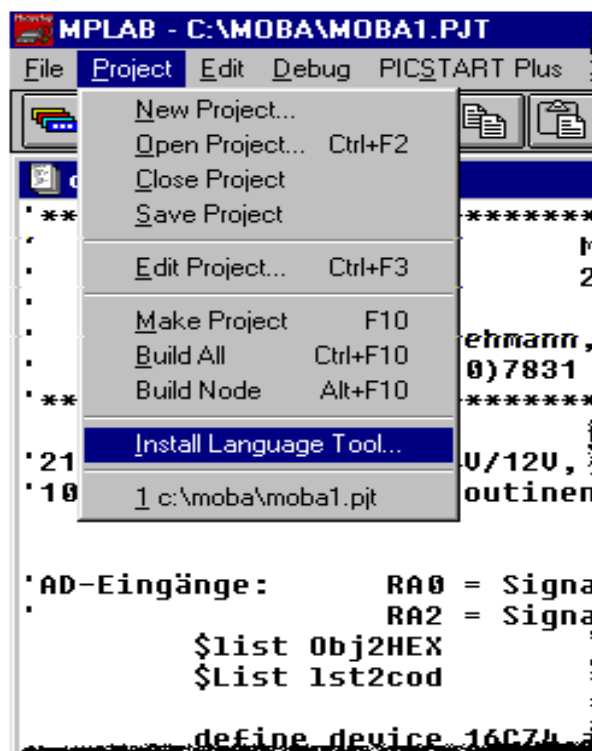
Die Programm-IDE starten Sie entweder durch klicken auf die Verknüpfung auf dem Desktop oder der Datei iL_EDy.EXE aus dem Explorer heraus. Die Beschreibung des Editor finden Sie in einem eigenen Kapitel.

Installation für MPLAB

iL_BAS16 kann nun sehr einfach in die Entwicklungsumgebung MPLAB (ab Version 5.2 bis max. 5.6*) von MICROCHIP integriert werden.

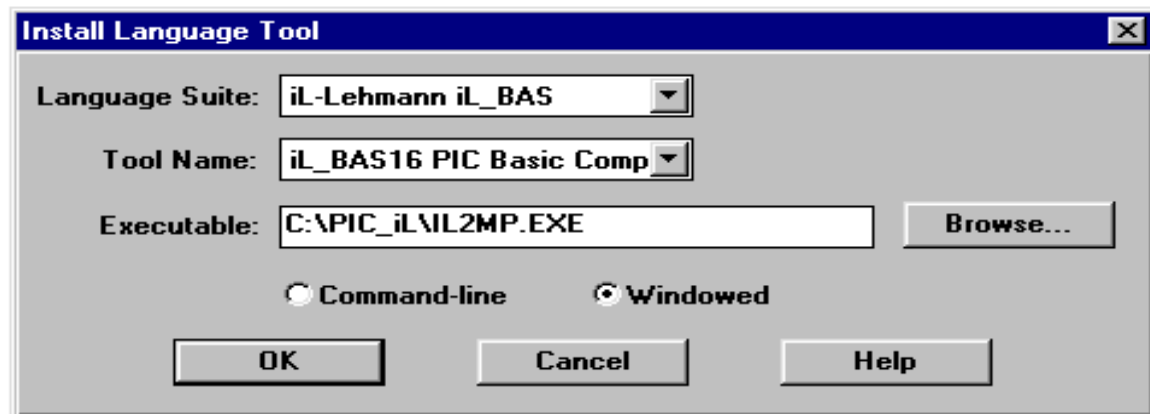
Führen Sie dazu die nachfolgenden Schritte durch:

- Installieren Sie MPLAB ab der Version 5.2 (max. Version 5.6), falls nicht bereits geschehen.
- Installieren Sie die Windows-Version von iL_BAS16.
- Kopieren Sie manuell die Dateien iL_BASIC.MTC und TLiLPic.INI aus dem Verzeichnis des Compilers in das Verzeichnis von MPLAB.
- Starten Sie MPLAB.
- Falls Sie noch kein Projekt angelegt haben, legen Sie eines jetzt an.
- Wählen Sie unter **Project** den Punkt **Install Language Tool...**



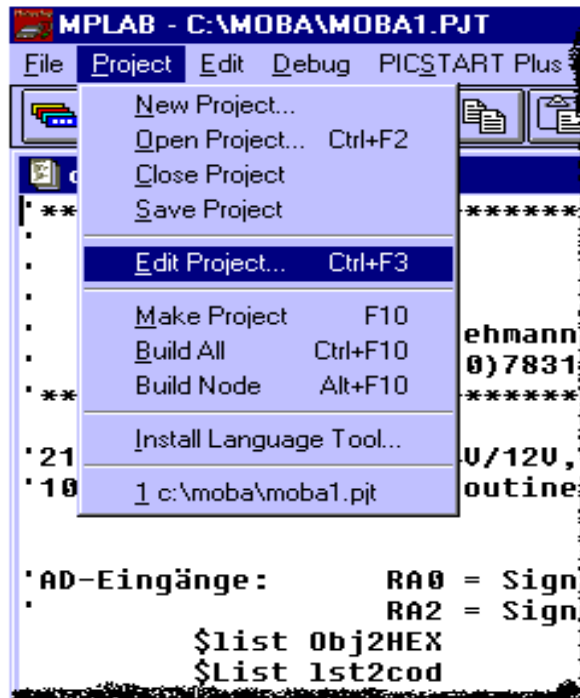
- Wählen Sie unter **Language Suite** den Eintrag **iL-Lehmann iL_BAS** aus
- Der richtige **Tool Name** erscheint selbständig.
- Bei **Executable** tragen Sie die Datei **iL2MP.EXE** incl. dem Pfadnamen, in den der Compiler installiert wurde, ein.
- Aktivieren Sie **Windowed** (ganz wichtig!).
- Bestätigen Sie die Eingaben mit **OK**.

Installation für MPLAB (cont.)



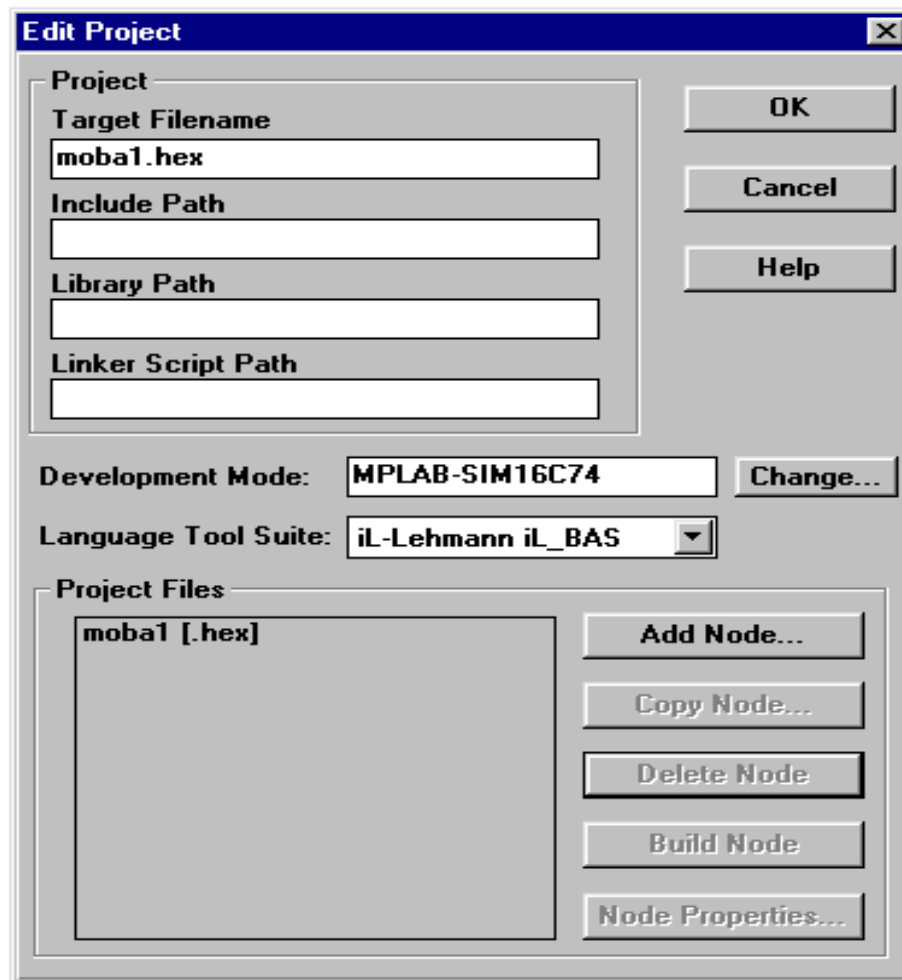
- Wählen Sie unter **Project** den Punkt **Edit Project...**

Installation für MPLAB (cont.)



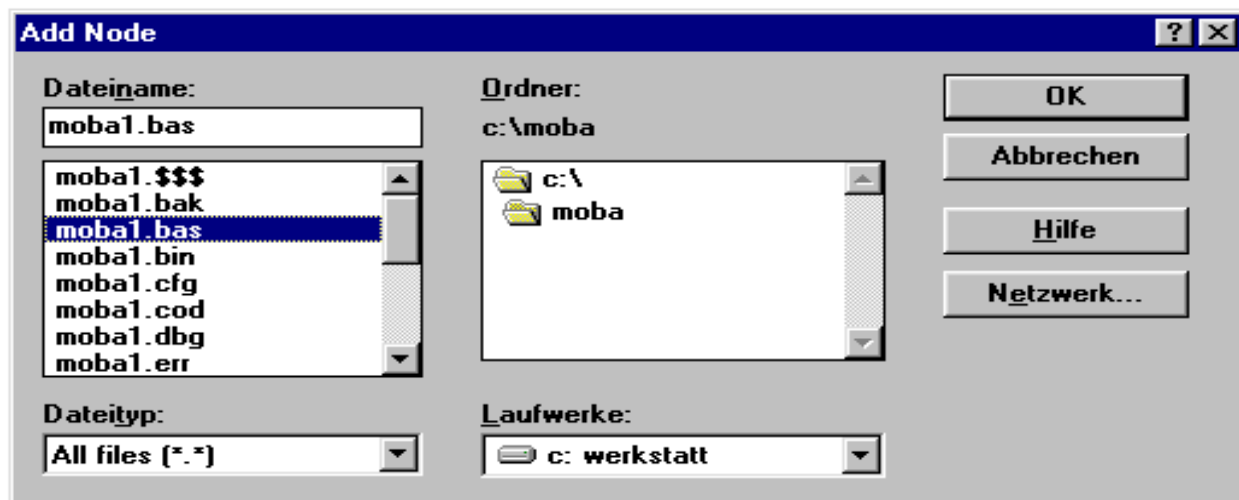
- Wenn Sie ein Projekt bereits angelegt haben, erscheint im Fenster **Target Filename** die entsprechende Hex-Datei.
- Die Felder **Include Path** , **Library Path** und **Linker Script Path** bleiben leer.
- Bei wählen Sie aus, ob Sie nur mit Simulator oder auch mit dem Emulator arbeiten wollen.
- In der **Language Tool Suite** wählen Sie wieder **iL-Lehmann iL_BAS** aus.
- Im Fenster **Project Files** steht die unter Target Filename eingetragene Datei. Klicken Sie auf **Add Node...**

Installation für MPLAB (cont.)



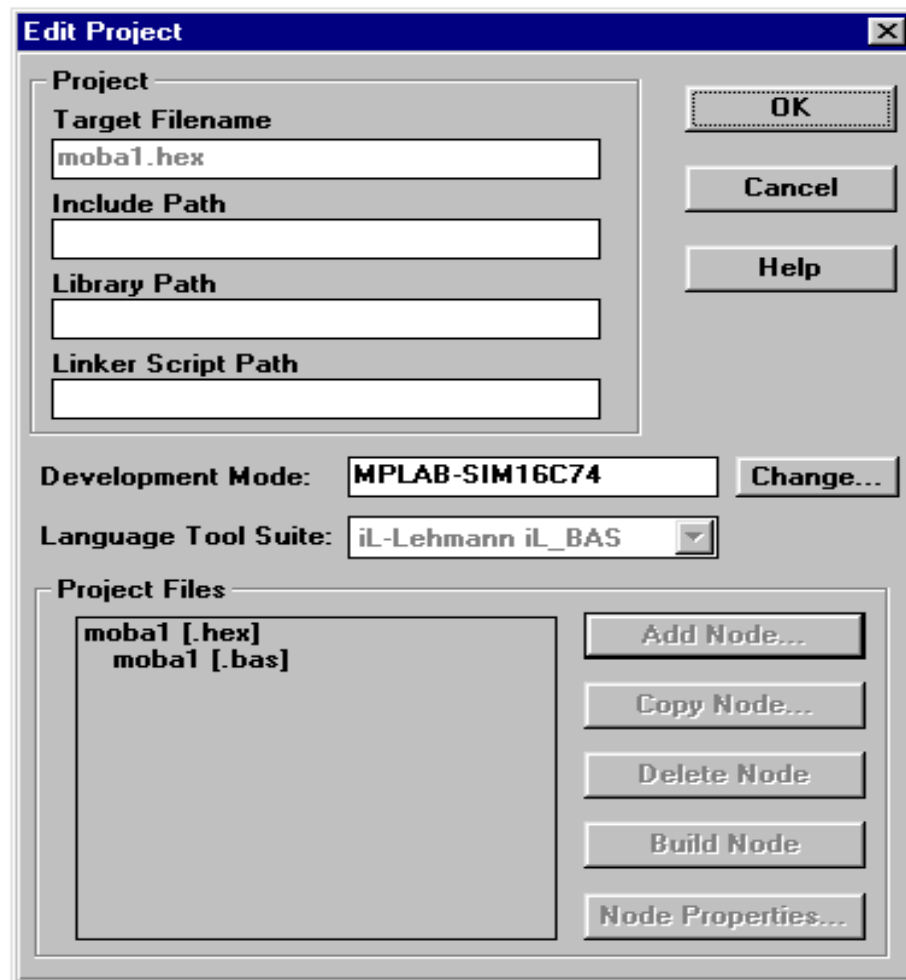
- Im Feld **Filename** geben Sie den unter Target Filename eingebene Datei mit der Extension **BAS** ein.

Installation für MPLAB (cont.)



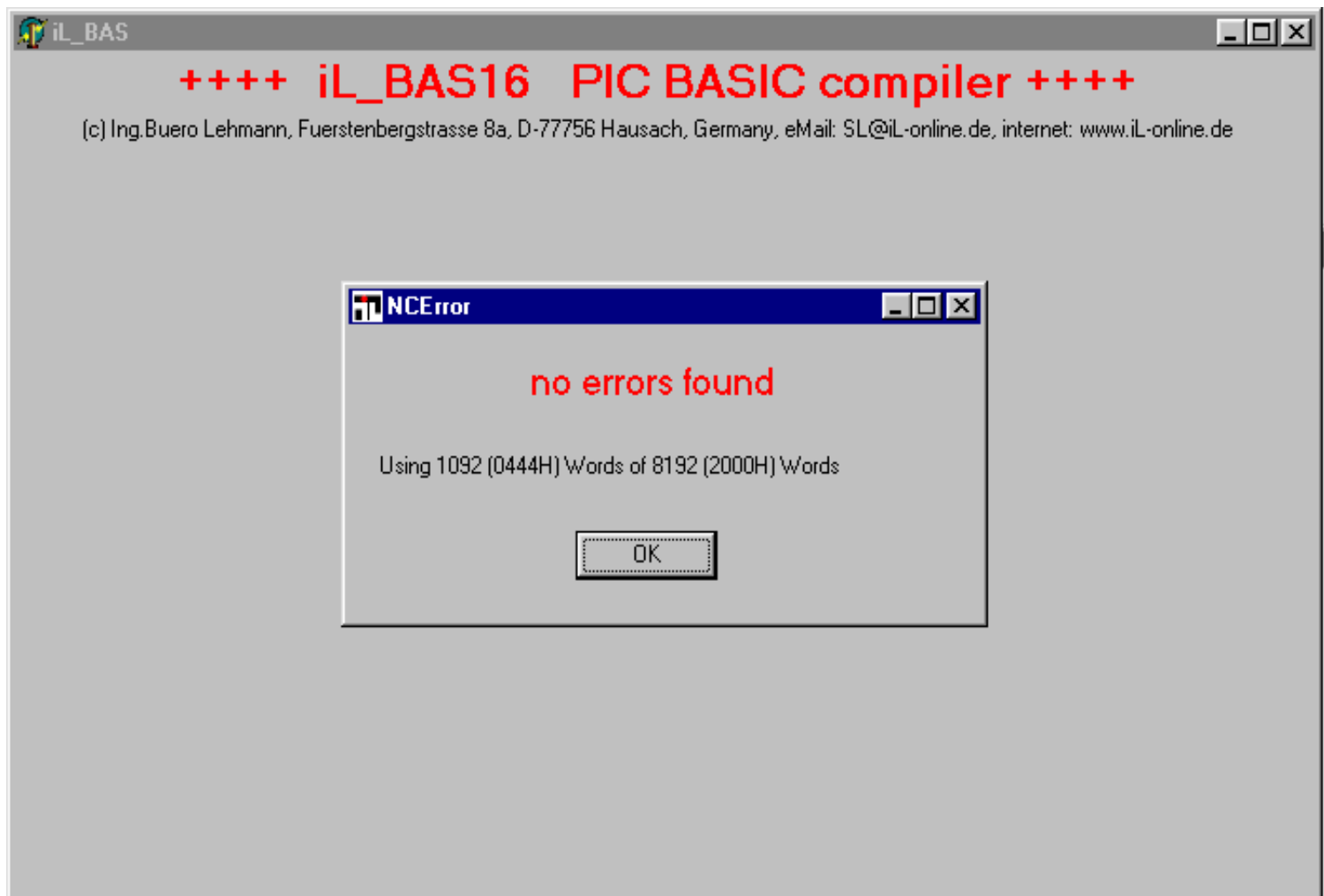
- Diese Eingabe erscheint im Fenster Project Files

Installation für MPLAB (cont.)



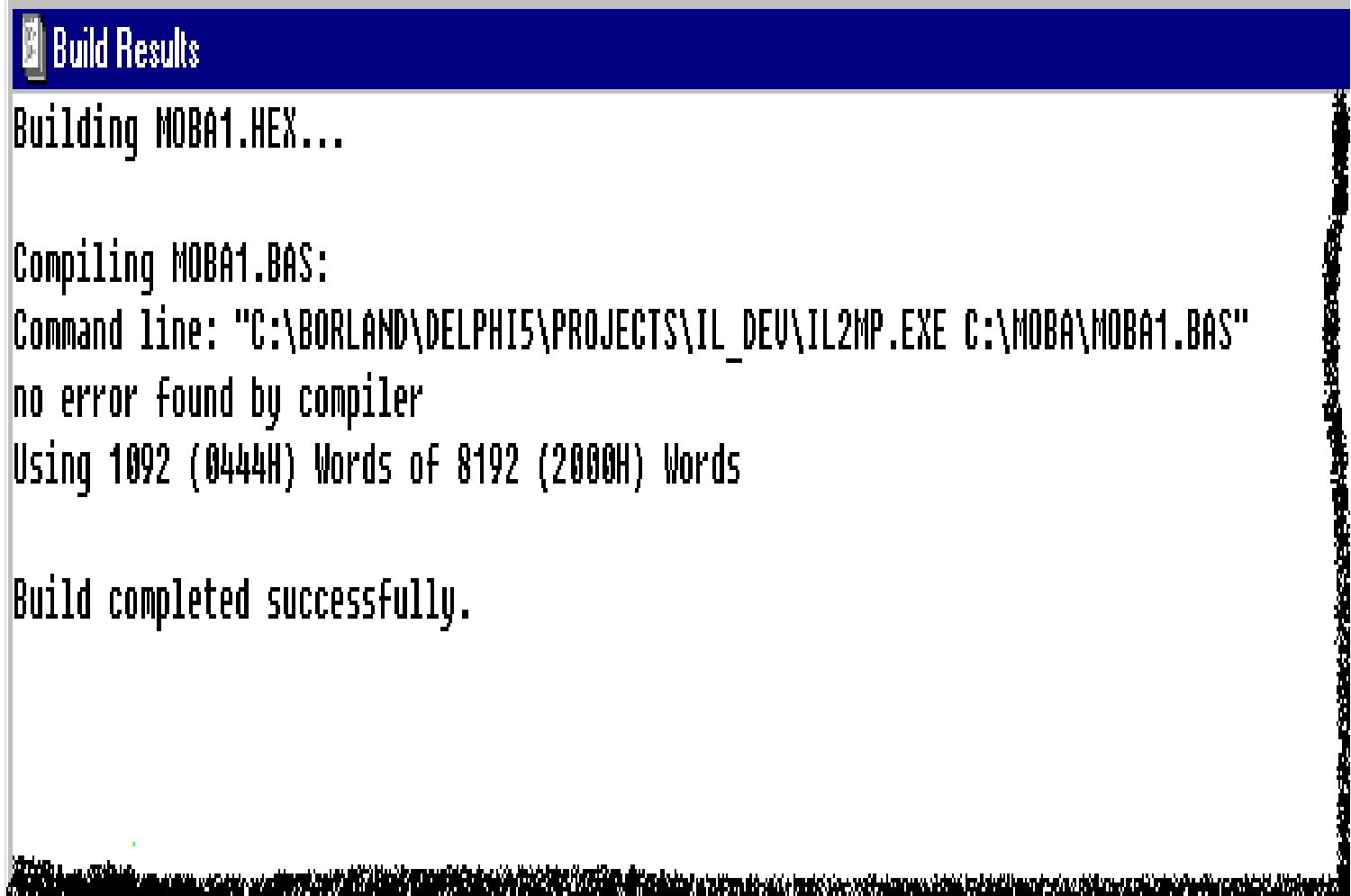
- Bestätigen Sie mit **OK**. Damit ist die Integration von iL_BAS16 in MPLAB abgeschlossen.
- Geben Sie in ihrem **BASIC-Quelltext** den Schalter **\$LST2COD** und **\$OBJ2HEX** an. Somit wird ein COD-File erzeugt, das MPLAB für das symbolische Debugging benötigt.
- Wenn Sie unter Project die Punkte Make Project, Build All oder Build Node aktivieren, wird der Compiler automatisch gestartet.

Installation für MPLAB (cont.)



Bei erfolgreicher Compilierung erscheint folgendes MPLAB Systemfester.

Installation für MPLAB (cont.)



```
Build Results
Building MOBA1.HEX...

Compiling MOBA1.BAS:
Command line: "C:\BORLAND\DELPHI5\PROJECTS\IL_DEV\IL2MP.EXE C:\MOBA\MOBA1.BAS"
no error found by compiler
Using 1092 (0444H) Words of 8192 (2000H) Words

Build completed successfully.
```

Ansonsten heißt es Fehlersuchen!!!

Wichtiger Hinweis!

*) Mit der Einführung der Version 6 bei MPLAB erfolgte auch der Umstieg vom COD-Format auf das COFF-Format. Diese Formate sind für die Steuerung und Darstellung des Quelltextes im MPLAB-Fenster verantwortlich. Da iL-BAS16 nur das COD-Format unterstützt, ist eine Zusammenarbeit mit MPLAB-Versionen ab 6.0 nicht möglich.

Projekt - Datei

Ein Projekt besteht aus ein oder mehreren (bis 8) Dateien. Beim Öffnen eines neuen Projekts wird eine neue Projektdatei im Verzeichnis des Editors iL_EDy angelegt. Sie beinhaltet neben den 8 Dateinamen auch deren Attribute.

Hinweis!!!

Zur Zeit sollte der Projektname nur aus max. 8 Zeichen (DOS-Konvention) bestehen, da der Simulator noch ein DOS-Programm ist und Datei- und Verzeichnisnamen mit mehr als 8 Zeichen nicht handhaben kann.

Der neue Projektname wird gleichzeitig als neuer Dateiname für die erste Mappe vorgeschlagen. Die Extension die vorgeschlagen wird, lautet BAS. Der Name der ersten Mappe wird dem Compiler und Assembler ohne die Extension als Parameter übergeben. Diese Programme ergänzen nun wiederum den Namen mit BAS (Compiler iL_BAS15), SRC (Assembler iL_ASS16, Pagemodul iL_PAGE0), LST (Simulator iL_SIM16) und OBJ (Programmiergerät iL_PRG16).

Einsteigerprobleme

Lesen Sie auch die FAQs und Grundsätzlich gilt:

Anschlusspins:

Durch die Vielseitigkeit der PIC-Bausteine sind viele Pins doppelt und dreifach belegt, manche sogar vierfach. Beim Versorgen der Bausteine mit Spannung sind diese Pins natürlich vordefiniert. Dabei gilt, dass alle I/O-Pins immer als Eingang arbeiten. Ist dieser Pin auch als Analogeingang verwendbar, arbeitet er als solcher und nicht als Digital I/O. Grundsätzlich sollten alle Pins die als Analogeingang fungieren auch im Tris-Register auch auf Input stehen.

Quarz-Oszillatoren:

Hier sollten Sie bei Quarzen die über 1/4 der maximalen Betriebsfrequenz des PICs liegen den Typ HS anwählen.

Beispiel: PIC mit 4 MHz Betriebsfrequenz und einem Quarz von 1 MHz bekommt den Typ XT eingestellt. Wird aber ein 3,2MHz Quarz verwendet, sollten man HS auswählen.

Bedenken Sie, dass die Oszillatorschaltung äußerst hochohmig ist und sich eine feuchte Umwelt bereits negativ auswirken kann. In solchen Fällen ist es ratsam einen externen Quarzoszillator mit TTL-Ausgang zu verwenden. Wählen Sie als Oszillatortyp entweder XT oder HS.

Interner RC-Oszillator:

RC-Oszillatoren sind relativ ungenau. Deren Frequenz schwankt stark in Abhängigkeit von Temperatur und Betriebsspannung. Der Hersteller optimiert diese Oszillatoren i.d.R. für 20°C Umgebungstemperatur und 5V Betriebsspannung. Dabei werden zwei unterschiedliche Vorgehensweisen realisiert.

Zum einen erfolgt der Frequenzabgleich während des Fertigungsprozesses über eine Lasertrimmung des Widerstandes vom RC-Glied. Bei diesen Bausteinen kann man die Betriebsfrequenz nur mittels des _OSCTUNE-Registers innerhalb gewisser Grenzen variieren. Steht in diesem Register der Wert 0, schwingt der Oszillator auf der werkseitig abgeglichenen Nominalfrequenz. Diese Vorgehensweise gilt z.B. für 16F818 u.ä.

Der zweite Weg den der Hersteller eingeschlagen hat ist die Verwendung des _OSCCAL-Registers und eines Befehls an der höchsten Adresse im Programmspeicher. Nach dem Fertigungsprozess werden vom Hersteller alle Bausteine abgeglichen. Das funktioniert so: Es wird ein Wert für das OSCCON-Register gesucht, bei dem die Oszillatorfrequenz der Nominalfrequenz (i.d.R. 4MHz) am nächsten kommt. Dieser Wert wird dann an der höchsten Speicheradresse abgespeichert. Um diesen Wert aus dem Programmspeicher lesen zu können, muss es in einen RETLW-Befehl gepackt werden. Auf diese Weise braucht man nur noch zum Programmstart einen Unterprogrammaufruf zu dieser höchsten Programmspeicheradresse auszuführen und den im W-Register zurückgegebenen Wert in das OSCCAL-Register zu schreiben. Dieser Vorgang wird vom Compiler allerdings automatisch durchgeführt. Das große Problem ist bei dieser Art des Oszillatorabgleichs, dass beim Löschen des Bausteins dieser RETLW-Befehl und damit der fabrikseitig ermittelte Abgleichwert verloren geht.

Deshalb sollten Sie bei solchen Bausteinen unbedingt vor deren erstmaliger Verwendung den Inhalt der höchsten Programmspeicheradresse mittels Programmiergerät auslesen und notieren. Ich persönlich habe mir angewöhnt diesen Wert (34xx oder 08xx) auf dem Baustein einzugravieren (Anreissnadel). Gilt z.B. für 12F675, 16F676 u.ä.

Inzwischen gibt es Bausteine, bei denen mehrere Frequenzen beim internen RC-Oszillator ausgewählt werden können (z.B. 16F88, 16F76x u.ä.). Beim Einschalten schwingt der interne RC-Oszillator dann auf der niedrigsten Frequenz von ca. 31kHz. Diese muss während des Programmlaufs verändert werden. Auf 4 MHz wird mit LET OSCCON = \$60 umgeschaltet.

\$00 = 31,25 kHz	\$10 = 125 kHz	\$20 = 250 kHz	\$30 = 500 kHz
\$40 = 1 MHz	\$50 = 2 MHz	\$60 = 4 MHz	\$70 = 8 MHz

MCLR (RESET)

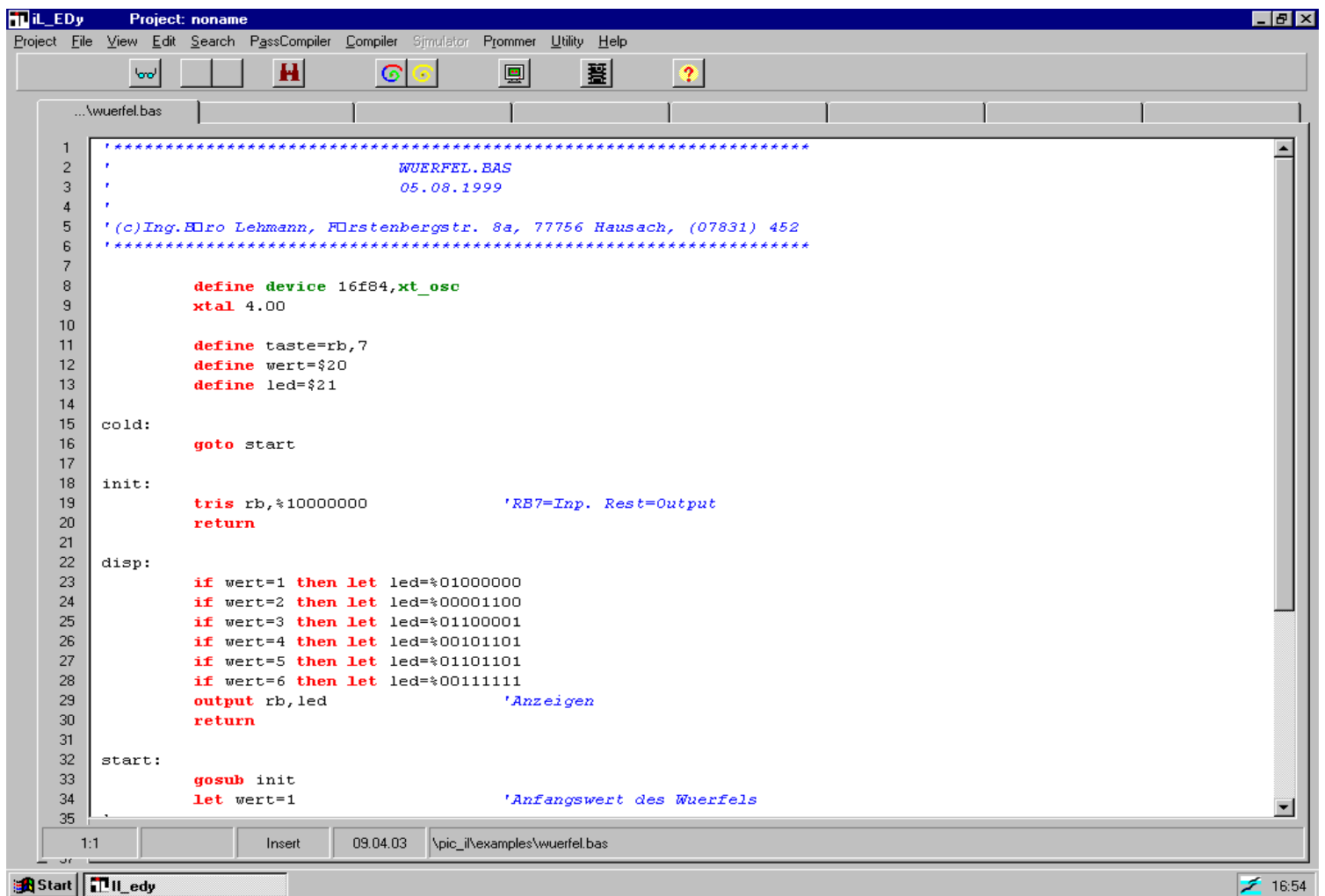
Bei manchen Bausteinen kann der MCLR (RESET)-Pin als normaler I/O-Pin verwendet werden. Dann wird die Resetlogik intern mit der Betriebsspannung verknüpft. Hier muss man größte Vorsicht walten lassen, damit ein Aufhängen des Bausteins nie erfolgen kann. Insbesondere sollte man die Brown-Out-Detect Funktion einschalten. Brown-Out-Detect ist bei den einzelnen PIC sehr unterschiedlich implementiert. Lesen Sie deshalb unbedingt das entsprechende Datenbuch von Microchip.

Einsteigerprobleme (cont.)

ADCFG und CMCFG

Viele PIC-Microcontroller haben AD-Wandler oder Komparatoren. Diese müssen in der DEFINE DEVICE-Zeile entsprechend eingestellt werden. Die Bausteine mit AD-Wandlern und Komparatoren werden ebenfalls immer zahlreicher. Bei diesen (z.B. 12F675) muss sowohl ADFG als auch CMCFG gesetzt werden. Damit beim 12F675 alles als Ausgang arbeitet, muss ADFG7 und CMCFG7 gewählt werden. Dabei sind auch noch T0CS, MCLR usw. zu beachten.

Editor



The screenshot shows the iL_EDy editor window with the following content:

```
Project: noname
Project File View Edit Search PassCompiler Compiler Simulator Prommer Utility Help

...wuerfel.bas

1  '*****
2  '                                WUERFEL.BAS
3  '                                05.08.1999
4  '
5  '(c)Ing. Elro Lehmann, Flirstenbergstr. 8a, 77756 Hausach, (07831) 452
6  '*****
7
8  define device 16f84,xt_osc
9  xtal 4.00
10
11 define taste=rb,7
12 define wert=$20
13 define led=$21
14
15 cold:
16     goto start
17
18 init:
19     tris rb,%10000000          'RB7=Inp. Rest=Output
20     return
21
22 disp:
23     if wert=1 then let led=%01000000
24     if wert=2 then let led=%00001100
25     if wert=3 then let led=%01100001
26     if wert=4 then let led=%00101101
27     if wert=5 then let led=%01101101
28     if wert=6 then let led=%00111111
29     output rb,led             'Anzeigen
30     return
31
32 start:
33     gosub init
34     let wert=1                'Anfangswert des Wuerfels
35
```

Beim Arbeiten mit dem Editor iL_EDy ist es ratsam, zuerst den Namen des Projektes festzulegen. Dann können die einzelnen Programmmodule erstellt werden. Diese Module können in einem beliebigen Verzeichnis liegen. Das Sichern der einzelnen Module erfolgt unter dem Menüpunkt DATEI.

Die Hauptdatei muss sich im ersten Fenster befinden. Es wird deren Name als Parameter an den Compiler, Assembler, Simulator und Programmer weitergereicht. In den übrigen Fenstern befinden sich dann zum Beispiel die Includedateien oder die zum Projekt gehörende Dokumentation. In das letzte Fenster (8) kann die Fehlerdatei geladen werden.

Die Projektverwaltung, d.h. die jeweilige Projektdatei, befindet sich immer im Verzeichnis des iL_Edy-Programms. Beim Anlegen eines neuen Projektes vergibt der Editor iL_EDY automatisch einen Namen für das erste Modul. Dieser Name besteht aus dem Projektnamen und der Extension '.BAS'. Dieser Name und das gewünschte Verzeichnis können jedoch problemlos verändert werden.

Wichtige Menüpunkte können direkt durch Klick auf Schaltflächen aktiviert werden. Unter der Funktion "PassCompiler" lassen sich die einzelnen Übersetzungsmodule nacheinander manuell aktivieren. Dabei muss die Reihenfolge iL_BAS16 (Compiler) - iL_PAGE0 (nur bei PRO und SEP aktiv) - und iL_ASS16 (Assembler) konsequent eingehalten werden. Dagegen übersetzt die Funktion "Compiler" das gesamte Projekt auf einen "Rutsch". Diese Funktion sollte man bevorzugt verwenden.

ACHTUNG!!!

Vor dem Aufruf des Compilers, Simulators usw. werden alle Dateien (Module) automatisch gesichert. Ebenso wird die Projektverwaltung aktualisiert.

Ab September 2008 kann auch **PICKit2** direkt aus dem iL_EDy aufgerufen werden. Dazu muss ins Verzeichnis von iL_EDy die Dateien PK2CMD.EXE und PK2DEVICEFILE.DAT kopiert werden. Zu finden sind die beiden Dateien unter www.microchip.com/pickit2.

Ebenso ist der aktuelle Assembler (ab 6.6-20) notwendig. Die Spannungsversorgung der Zielhardware über PICKit2 (USB) kann ein- und ausgeschaltet werden.

Editor (cont.)

Beim Compilieren müssen die beiden Compilerschalter \$LIST /M_OBJ und \$LIST OBJ2HEX aktiv sein.

Seit 07.03.2007 beherrscht iL_EDy auch das verbergen von Textpassagen. Dazu wird der gewünschte Bereich markiert und die rechte Maustaste gedrückt. Dort wählt man die Funktion JALOUSIE EIN. Der Block wird links durch einen roten vertikalen Balken markiert.

The screenshot shows the iL_EDy editor window with the menu bar: **Projekt**, **Datei**, **Ansicht**, **Bearbeiten**, **Suchen**, **PassCompiler**, **Comp**. Below the menu is a toolbar with icons for **LET**, **UNDO**, and others. The file list shows `... \07-03-07.bas` and `... \07-03-07.lst`. The main editor area displays a code file with line numbers 1 to 20 on the left. A red vertical bar marks a block of code from line 9 to line 17. The code content is as follows:

```

1  ' *****
2  '
3  '
4  '
5  '
6  '
7  '
8  '
9  '07.03.2007 Programmer einlesen
10
11 '01.06.2006 Sensor und Potis Einlese
12 '31.05.2006 Grundgerüst
13
14 '-----
15 'AD-Eingänge:          RA0 = Analog Sign
16 '                      RA2 = Analog Sign
17 '-----
18
19 $ccoff
20 define device 16F88,irc_c
    $WDTUSR
  
```

Klickt man auf das -, wird der Block geschlossen. Sichtbar bleibt nur die erste Zeile. Um bei diesem Block die Markierung wieder aufzuheben, positioniert man den Cursor auf den gewünschten Balken, klickt auf die rechte Maustaste und wählt JALOUSIE AUS an. Der vertikale Balken verschwindet. Wird ein übergeordneter Bereich gelöscht, verschwindet der untergeordnete Bereich ebenso.

Editor (cont.)

```

1  ' *****
2  '
3  '
4  '
5  '
6  ' (c) T. Sch
7  ' Kapellens
8  ' *****
9  ' 07.03.2007 Programmer einlesen
15 'AD-Eingänge:      RA0 = Analog Sign
16 '                  RA2 = Analog Sign
17 ' -----
18                $ccoff
19                define device 16F88,irc_c
20                $WDTUSR
21                xtal 8.00
22

```

Bis zu zwei Ebenen lassen sich schachteln.

Seit Januar 2004 sind die nachfolgenden Editbefehle über Tastenkombinationen ausführbar. Sie entsprechen eine Subset der alten Wordstar-Befehlen.

CTRL-Q und CTRL-K Funktionen

CTRL-K 1..9 Buchmarke setzen/löschen

CTRL-K B Blockbeginn markieren

CTRL-K C Block kopieren

CTRL-K K Blockende markieren

CTRL-K S aktuelle Datei speichern

CTRL-K T Wort unterm Cursor markieren

CTRL-K V Block verschieben

CTRL-K Y Block löschen

CTRL-Q 1..9 gehe zur Buchmarke

CTRL-Q A Suche und Ersetzen

CTRL-Q F Suchen

CTRL-Q L Rückgängig machen

CTRL-L Weitersuchen

CTRL-N Zeile einfügen

CTRL-U Suchen bzw. Ersetzen abbrechen

Editor (cont.)

CTRL-Y Zeile löschen

Debug (Entwanzen)

(Zur Zeit nur für die Bausteine 16F628, 16F88 und 16F87x sowie für den iL-TROLL verfügbar)

Zusammen mit dem iL-TROLL und der zugehörigen Hardware (am Besten mit der Platine iL-TROLL-TB1) ist ein komfortables Austesten des BASIC-Programms möglich. Dazu muss der Compilerschalter \$DEBUG gesetzt sein. Er weist den Compiler an, vor jedem BASIC-Befehl eine kurze Sequenz (5 Bytes) Debugcode zu generieren.

Bei den andern oben genannten Bausteinen muss 1 Pin für die Kommunikation zur Verfügung gestellt werden. Ggf. können über diesen Pin auch die Funktionen des SELFPROG-Befehls angesprochen werden. Der Debugkernel ist ein Teil vom SPBIOS der Selfprog-Funktion. Das notwendige Programmierinterface wird im Kapitel iL-TROLL beschrieben. In einer verkleinerten Version passt es in einen Sub-D-Stecker (als AS106a bzw. iL_SPGAP lieferbar).

Nach dem Download des Programm in den iL-TROLL oder einen anderen PIC-Baustein (siehe oben), startet es normalerweise sofort. Beim gesetzten \$DEBUG-Schalter wartet es jedoch bei der ersten logischen BASIC-Anweisung, also dem Befehl der als erstes ausgeführt wird. Es wartet bis vom PC eine gültige Anweisung über die Downloadverbindung kommt. Dazu muss auf dem PC das Programm iL_DEBUG.EXE gestartet sein. Dieses benutzt die gleiche serielle Schnittstelle wie das Downloadprogramm (iL_DNLT.EXE).

Mit \$TRON und \$TROFF kann man die Generierung des Debugcodes in weiten Grenzen steuern. So kann man nicht nur Speicherplatz sparen, sondern auch die zeitkritischen Routinen in Echtzeit ausführen lassen. Kommunikation über serielle Schnittstelle mittels SERIN / SEROUT sind somit kein Problem.

Zum Programmtest stehen die Funktionen RESET (der Hardware), EINZELSCHRITT, KONTINUIERLICH, TRACE und BREAKPOINT (Haltepunkt) zur Verfügung. Während des EINZELSCHRITT und KONTINUIERLICH wird im Programmfenster die aktuelle Zeile angezeigt. Da der Debugcode vor der jeweiligen BASIC-Anweisung steht, sieht man also immer die Zeile markiert, die als nächstes abgearbeitet wird.

Im Variablenfenster werden alle im BASIC-Programm definierten Variablen und deren aktuelle Inhalte angezeigt. Das Ändern einzelner Variableninhalte ist ebenfalls möglich.

Weitere Informationen und die Beschreibung der Bedienung des Debuggers iL-DEBUG entnehmen Sie der Online-Hilfe des Programms.

Achtung!

Sobald \$DEBUG aktiv ist, startet das BASIC-Programm nicht automatisch, sondern benötigt dazu ein laufendes Debug-Programm auf dem PC sowie eine funktionierende Verbindung.

Soll auch die Funktion SELFPROG verwendet werden, muss dies entsprechend eingestellt werden. Lesen Sie dazu \$SELFPRG und SELFPROG

Produktinfo

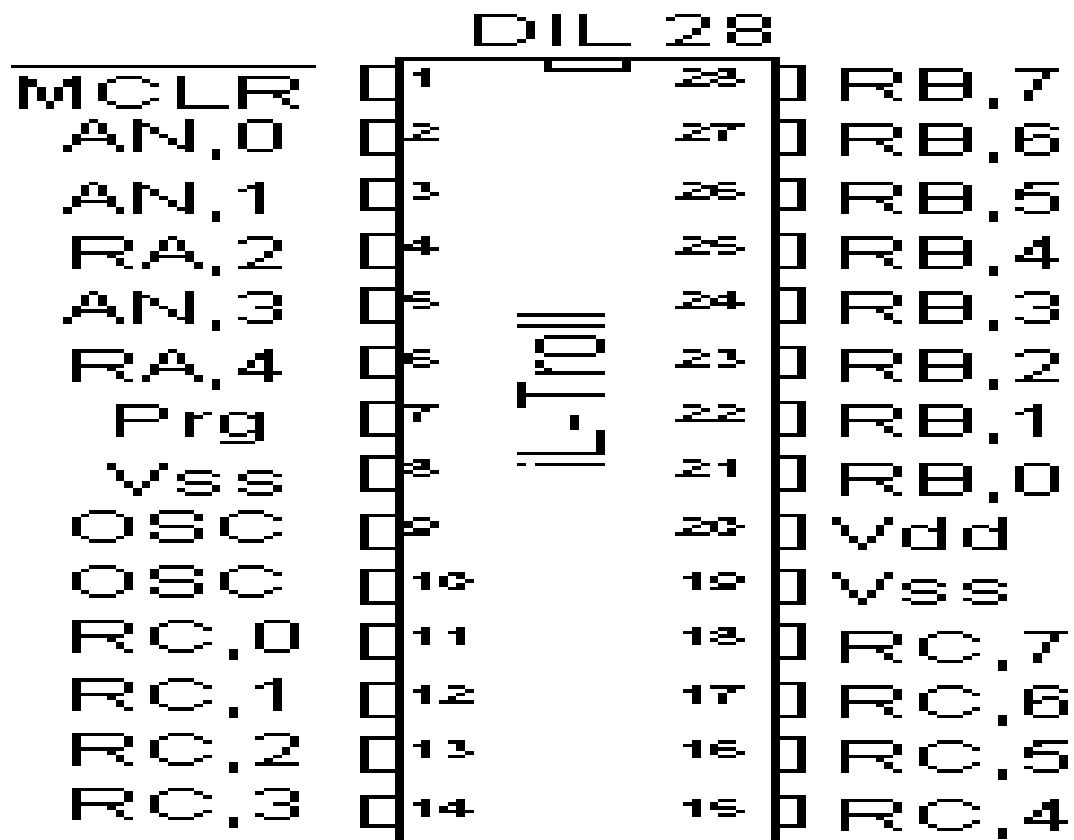
Es sind vier Versionen erhältlich:

- iL_BAS16STD Standardversion
- iL_BAS16PRO Professionalversion
- iL_BAS16SES Spezial Standard (nur 16F84 und 16F628)
- iL_BAS16SEP Spezial Professional (nur 12F629, 16F84, 16F627, 16F628 und 16F877)
- iL_BAS16TRO Spezialversion für die vorprogrammierten Bausteine
(der verwendete Baustein heißt iL_TROLL)

iL-TROLL ®

der freundliche Professor

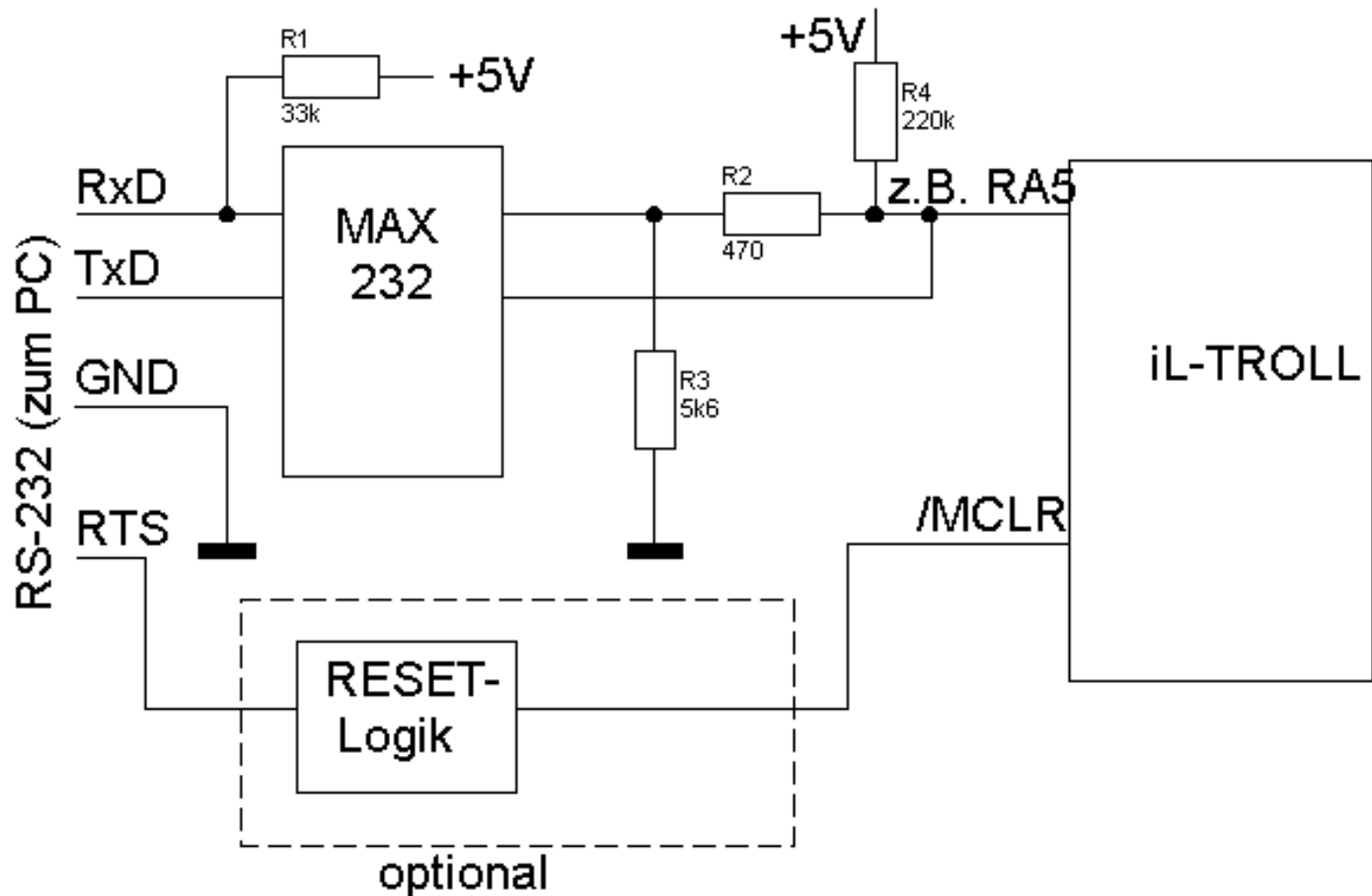
iL-TROLL sind 28polige vorprogrammierte Bausteine. Sie umfassen ca. 1,8k Programmspeicher und ca. 150 Bytes für Daten. Es sind zwei komplette 8-Bit Ports (RB und RC) sowie 4 Analogeingänge vorhanden. Das ganze findet in einem 28 poligen Gehäuse Platz.



Um ein Programm in solch einen iL-TROLL zu laden benötigt man kein Programmiergerät. Lediglich der PC muss eine serielle Schnittstelle besitzen. Falls nur noch USB-Anschlüsse vorhanden sind, lässt sich ein USB-RS232 Konverter dazwischen schalten. Da iL-TROLL nur TTL-Pegel (0 - 5V) kennt, muss ein TTL-RS232-Pegelwandler angeschlossen werden. Für die bidirektionale Verbindung zum PC benötigt man auf der iL-TROLL Seite lediglich einen (1) Pin (hier RA,5). Über diesen Pin wird auch der Remotedebugger des iL-TROLLs gesteuert.

Schaltbild:

iL-TROLL ® (cont.)



Die Programmentwicklung erfolgt mittels dem Editor iL_EDy. Hier erstellen Sie ihr BASIC-Programm, compilieren es und laden es in den iL-TROLL. Dazu wird zuerst der iL-TROLL mit der seriellen Schnittstelle verbunden und der RESET-Knopf gedrückt. der iL-TROLL geht dann in sein vorprogrammiertes Betriebssystem und wartet dort auf die Daten vom PC. Sobald die Programmierung beendet ist, wird das neugeladene Programm gestartet.

(Der RESET des iL-TROLL wird auch über die Programmiersoftware gesteuert).

Der Befehlsumfang des iL-TROLLs ist ein wenig eingeschränkt.

Es fehlen folgende Befehle:

- ASM
- ENDASM
- PEEK
- POKE
- TRIS RA (fest vorgegeben)
- XTAL (fest auf 4.0 MHz)

Weiterhin wird keine SRC- und LST-Datei erzeugt. Das Programm wird nicht in einer OBJ-, HEX- oder Binärdatei abgespeichert, sondern in einem Sonderformat das für die DOWNLOAD-Software notwendig ist. Diese DOWNLOAD-Software wird auch über eine Schaltfläche in der IDE iL_EDy aufgerufen.

Es wird nur noch die Variablendeklaration mit AUTO (siehe DEFINE Variable, Konstante) unterstützt.

Nur zur Information:

iL-TROLL stellt folgende Speicherplätze dem Programmierer zu Verfügung:

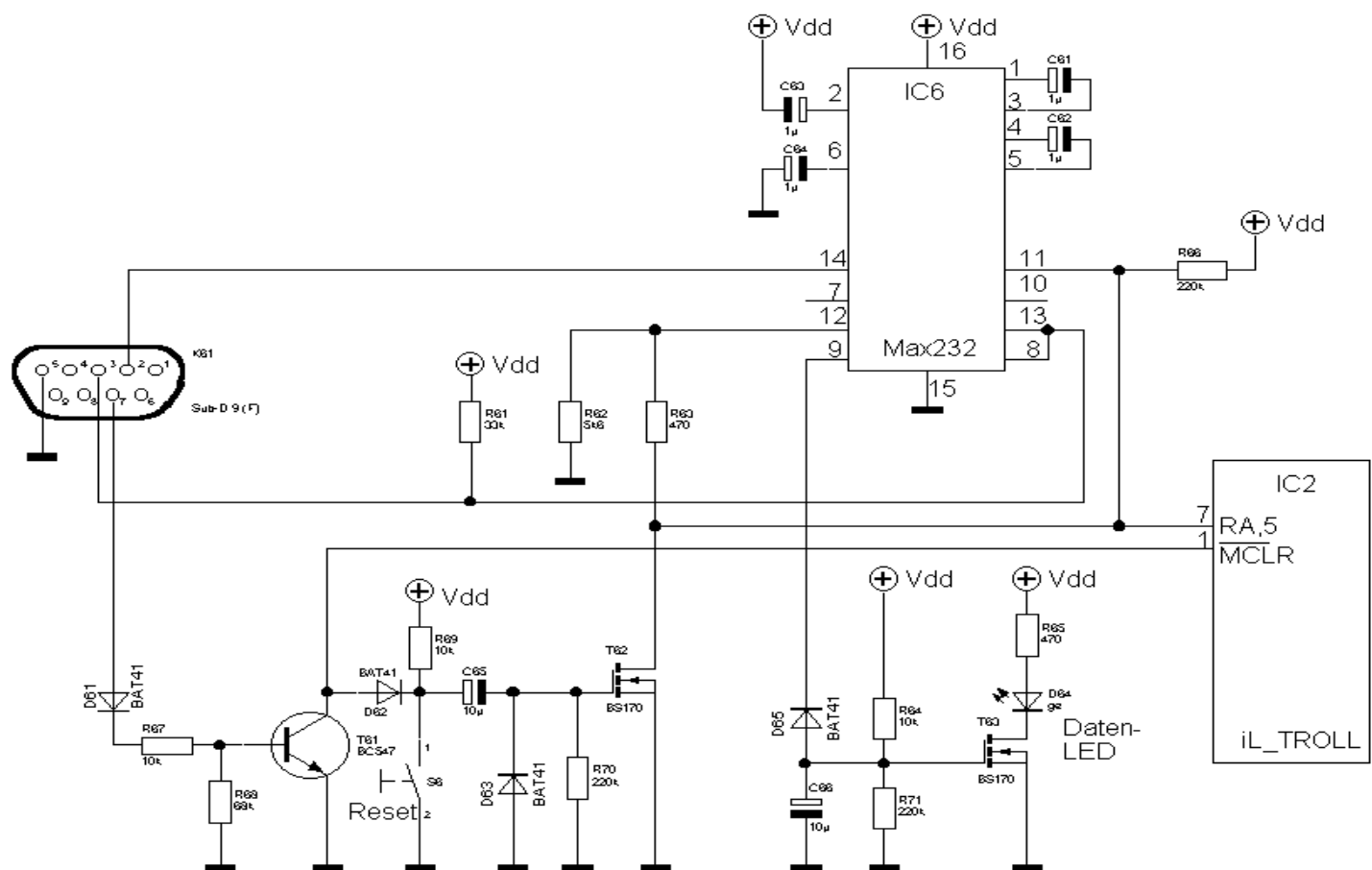
32 - 108 entspricht 20H - 6CH

iL-TROLL ® (cont.)

160 -236 entspricht A0H -ECH

Bank 2 und 3 sind nicht implementiert!

Nachfolgendes Bild zeigt das vollständige Programmierinterface. Natürlich muss die Reset-Logik so aufgebaut werden, dass zwischen einem manuellen Reset und einem Reset vom PC (RTS) unterschieden werden kann. T62, D63, C65 und R270 können dann entfallen. Der Funktionsblock aus T63, D64, D65, C66, R64, R65 und R71 dienen als optische Kontrolle. LED brennt, wenn die Verbindung zum PC vorhanden ist. Bei der Datenübertragung blinkt sie. Auch dieser Funktionsblock kann weggelassen werden.



Programmierinterface mit Komfort-Reset

BASIC-Befehle

Grundsätzlich gilt:

ADDELAY zusätzliche Verzögerung beim Umschalten der AD-Kanäle (nur 16C7x und 16F87x)
ADINP Analoge Werte über AD-Wandler einlesen (nur 16C7x und 16F87x)
ASMBeginn einer Assemblersequenz
BINTOASC (auch CONASC) Konvertiert einen Zahlenwert in einen ASCII-String
BINTOBCD Konvertiert einen Zahlenwert in eine BCD-Zahl
BINTODEC (auch CONDEC) Konvertiert einen Zahlenwert in einen Dezimalstring
BITPOS Ermittelt die Wertigkeit einer Bitposition
CALVAL Definiert einen Kalibrationswert für den RC-Oszillator (nur PICs mit aktivem int. RC-Osz.)
CLOCK und CLOCK1 Definiert eine interruptgesteuerte Echtzeituhr (nicht für 10F2xx, 12C5xx, 12E5xx und 16C5x)
CLRWDT Setzt den Watchdog zurück
CONF Konfigurationswort direkt beschreiben
CONFIG definiert die Bits im Konfigurationswort
CURSOFF Schaltet den Cursor auf der LCD aus
CURSON Schaltet den Cursor auf der LCD ein
DATA Definiert Konstantenfelder (nicht für 10F2xx, 12C5xx, 12E5xx und 16C5x)
DEC Erniedrigt den Variableninhalt um eins
DELAY Verzögert den Programmablauf um $x * 100\mu s$
DOZE Kurzschlaf mit Stromsparfunktion
DTMFOUTDTMF-Signal erzeugen
EEDATA Vordefinieren der Werte für das interne EEPROM
END Programmende
ENDASM Ende der Assemblersequenz
ERR Überträgt den Fehlercode in eine Variable
FOR-TO-NEXT Schleifenkonstrukt
FREQIN Misst eine Frequenz in Hz
GOSUB Unterprogrammaufruf
GOTO Unbedingte Programmverzweigung
HIGH Setzt Pin auf Ausgang und High-Pegel
I2CDELAY Verlangsamt den I2C-Bustakt
I2CHARDSAktiviert das interne I2C-Modul als I2C-Slave (nur wenn Hardwaremodul implementiert)
I2CINIT Initialisiert den I2C-Bus (Master oder Slave)
I2CRD Adressiert einen Baustein und liest Werte heraus
I2CRDBLiest Werte aus Baustein und meldet ACK (ohne Adressierung)
I2CRDBN Liest ein Byte aus Baustein und meldet NACK (ohne Adressierung)
I2CREADKomplett abgeschlossener Lesezugriff auf I2C-Baustein
I2CSLAVEKomplett abgeschlossene Kommunikation mit einem Master
I2CSP Generiert eine I2C-Stopbedingung
I2CST Generiert eine I2C-Startbedingung
I2CWR Adressiert einen Baustein und schreibt Werte hinein
I2CWRB Schreibt Werte in den Baustein (ohne Adressierung)
I2CWRITEKomplett abgeschlossener Schreibzugriff auf einen I2C-Baustein
IF-THEN-ELSE Bedingte Programmausführung
INC Erhöht den Variableninhalt um eins
INKEY Liest eine Matrixtastatur ein
INP Definiert einen Pin als Eingang
INPUT Liest einen Port komplett ein
INTERRUPT Initialisiert einen Interrupt-Handler
INTEND Beendet die Interruptroutine
INTPROC Beginn der Interruptroutine

BASIC-Befehle (cont.)

LCDCHARDefinieren von bis zu 8 eigenen Zeichen in einer LCD
LDCLEARLöscht das LCD
LCDCTRST Kontrasteinstellung bei LCD-Controllern ST7036
LCDDELAYVerlangsamt die Datenübertragung ans LCD
LCDINITInitialisiert das LCD
LCDTYPELegt die Art der LCD-Beschaltung fest
LCDWRITESchreibt auf LCD
LETZuweisung
LOCATE Positioniert den Cursor auf der LCD
LOFREQEzeugt eine niedrige Frequenz
LOOKDNErmittelt die Position in einer Vergleichsliste
LOOKUPLiest ein Listenelement von einer gegebenen Position
LOWDefiniert einen Pin als Ausgang und gibt einen LOW-Pegel aus
ON-GOSUBVariablenabhängiger Anspruch verschiedener Unterprogramme
ON-GOTOVariablenabhängige Programmverzweigung
OUTPDefiniert einen Pin als Ausgang
OUTPUTSchreibt einen Wert komplett auf den Port
PEEKLiest den Inhalt einer Datenspeicheradresse
POKESchreibt an eine Datenspeicheradresse
PRINTSerielle Datenausgabe für Debug-Zwecke (nur 16F627/628, 16F818/819 und 16F87x)
PULSINMisst die Dauer eines Impulses (relativ)
PULS_INMisst die Dauer eines Impulses (in us)
PULSOUTGibt einen Impuls aus
PWMERzeugt ein PWM-Paket
RANDOMErzeugt eine Pseudozufallszahl
RCTIMEMisst die Ent-/Ladezeit einer RC-Kombination
READDATAliest Daten aus DATA-Feld
READLiest Daten aus dem internen Daten-EEPROM (nur falls im PIC vorhanden)
REMLeitet einen Kommentar ein
RESSetzt den Wert eines Bits auf low
RESTORESetzt den Lesezeiger für DATA-Felder
RETURNBeendet Unterprogramm
REVERSDefiniert einen Pin als Ausgang und invertiert dessen Pegel
SELFPROG Programmiert den Programmspeicher während des Betriebs
SERINLiest Daten seriell ein
SEROUTGibt Daten seriell aus
SETSetzt den Wert eines Bits auf high
SLEEPVersetzt den PIC in den Stromsparmodus
SOUNDERzeugt eine Frequenz
SWAPVertauscht unteres und oberes Halbbyte in einer Variablen
TXDDelayFügt eine zusätzliche Pause zwischen den Bytes ein
TOGGLEInvertiert den Logikpegel eines Pins
TRISDefiniert welche Pins als Aus- bzw Eingang arbeiten
VARPTR Gibt die Adresse einer Variablen zurück
WAITVerzögert den Programmablauf um x ms
WRITESchreibt Daten in das interne Daten-EEPROM (nur falls im PIC vorhanden)

HINWEIS:

Zum Teil unterscheiden sich die PIC-Microcontroller untereinander sehr stark. Deshalb ist es ratsam, im original Microchip Datenblatt das Kapitel über die entsprechende Hardwareimplementierung zu lesen. Beispiel: ADCFG <-> ANSEL, Speichereinteilung, Oszillator, MCLR, Comparatoren, etc.

Compilerschalter

Compilerschalter oder Compilerdirektiven dienen dazu, bestimmte Vorgaben für den Compilerlauf festzulegen. Dadurch ist es möglich, bestimmte Anpassungen zu machen. Beispielsweise definiert der Schalter XTAL auf welche Quarzfrequenz sich das erstellte Programm beziehen soll, um bei Verzögerungsschleifen u.ä. den exakten Zeitbezug herzustellen.

\$CCON und \$CCOFF

aktivieren und deaktivieren der Bereichsprüfung bei 8-Bit Addition und Subtraktion

\$DEBUG

generiert zusätzlich Debugcode vor jeder BASIC-Anweisung

\$IF \$ELSE \$ENDIF

ermöglicht die bedingte Compilierung

\$INCLUDE

einbinden von Includedateien

\$LIST

steuert den Aufbau der OBJ-, BIN- und LST-Dateien

\$LRANGE

definiert den Adressraum an der Seitengrenze für iL_PAGE0

\$LST2COD

erzeugt ein COD-File, das für das symbolische Debugging von u.a. MPLAB benötigt wird.

\$NCALDEF

unterdrückt die Ausgabe des Defaultwertes für den Calibrationswert

\$OBJ2HEX

erzeugt neben der OBJ-Datei auch die HEX-Datei.

\$OLDVAR

Compiler arbeitet mit der alten Variablendeklaration (nicht mehr verwenden)

\$OSCCON

schaltet den internen RC-Oszillator auf eine andere Frequenz. Nicht alle PICs haben den entsprechenden Oszillator.

\$SELFPRG

installiert ein zusätzliches kleines Betriebssystem, das es erlaubt den Programmcode während des Programmlaufes zu verändern. Ideal für Programm-Updates mittels Telefon o.ä. Teile von diesem Kernel werden auch für das Debugging verwendet.

\$TROFF

ab diesem Schalter wird den nachfolgenden BASIC-Befehlen kein zusätzlicher Debugcode vorangestellt.

\$TRON

ab diesem Schalter wird dem nachfolgenden BASIC-Befehlen zusätzlicher Code für den Debugger vorangestellt.

\$VCFG

Bei manchen PIC-Bausteinen mit AD-Wandler kann man die Referenz einem Pin oder Vcc zuordnen

Compilerschalter (cont.)

\$WDTUSR

Erlaubt dem Programmierer den CLRWDT-Befehl gezielt an bestimmten Stellen im Programm zu setzen.

DEFINE Variable, Konstante

definieren von Symbolen u.a.

DEFINE Prozessor

DEFINE sonstiges

DATE

übernimmt das aktuelle Datum in die SRC- und LST-Datei

TIME

übernimmt die aktuelle Zeit in die SRC- und LST-Datei

XTAL

Quarzfrequenz mit der die Zielhardware arbeitet

\$CCON und \$CCOFF

Mit diesem Compilerschalter legt man fest, ob bei der 8-Bit-Addition bzw. Subtraktion die Überlaufprüfung aktiviert oder deaktiviert ist. (2 Direktive zum Ein- bzw. Ausschalten)

Defaultstellung: ON. Bei allen Additionen und Subtraktionen wird ein entstehender Bereichsüberlauf (Carry) in das Bit 7 der ERR-Variable übernommen.

In OFF-Stellung kann unter Verzicht auf die Überlaufprüfung Programmspeicher eingespart werden, die Laufzeit wird ebenfalls beschleunigt. Oft ist es bei einfachen Additionen und Subtraktionen unwichtig, ob ein Bereichsüberlauf (Carry) entsteht.

Man kann \$CCON und \$CCOFF beliebig oft im BASIC-Programm verwenden. Somit ist ein gezieltes Aktivieren an bestimmten Programmstellen möglich, während der Rest des Programms entsprechend kompakter generiert wird. Der Code, den der Compiler nach dem Auftreten des Schalters \$CCOFF generiert, führt diesen Übertrag dann nicht aus. Ab dem Zeitpunkt, wenn \$CCON angetroffen wird, wird das Carrybit wieder in die ERR-Variable kopiert.

8-Bit Multiplikations- und Divisionsbefehle, sowie alle 16- und 32-Bit Rechenbefehle werden nicht von \$CCON und \$CCOFF beeinflusst. Hier wird immer ein Übertrag verarbeitet.

\$DEBUG

(zur Zeit nur für 16F628, 16F88 und 16F87x sowie den iL-TROLL)

Dieser Schalter aktiviert den zusätzlichen Debugcode. Dieser steht vor jedem BASIC-Befehl. Die Generierung kann durch \$TRON und \$TROFF gesteuert werden.

Es werden beim 16F628 und iL-TROLL lediglich 5 zusätzliche Assembleranweisungen pro BASIC-Anweisung mit in das Programm eingebunden. Bei den anderen PIC-Bausteinen sind es 7 bzw. 9 zusätzliche Maschinenbefehle, abhängig wieviel Programmspeicher (4k, 8k) implementiert sind. Kommt der Microcontroller an diese Stelle, verzweigt er in das kleine zusätzliche Betriebssystem (SELFPROG). Dort wird entschieden, ob sofort die nächste Anweisung ausgeführt werden soll. Das ist der Fall, wenn man einen Haltepunkt (Breakpoint) gesetzt hat und das Programm bis zu diesem Haltepunkt durchlaufen lässt.

In EINZELSCHRITT bzw. TRACEMODE werden nach dem Verzeigen zum Kernel zuerst die Inhalte der Variablen zum PC übertragen. Anschließend wird auf eine gültige Anweisung, z.B. EINZELSCHRITT gewartet, also in Interaktion mit dem Programmierer getreten.

Beim iL-TROLL darf dieser Compilerschalter keine Argumente führen. Bei den anderen PICs muss man angeben, welcher Selfprog-Modus (hier aber immer 0), welcher Port und Pin verwendet werden soll.

z.B.

DEBUG 3,ra,5

(in Verbindung mit der DEBUG-Funktion sollte nur der Modus 3 verwendet werden)

Soll auch die Selfprog-Funktion verwendet werden, reicht es aus den Selfprog-Compilerschalter mit den entsprechenden Argumenten zu setzen. In diesem Fall darf der \$DEBUG-Schalter keine Parameter enthalten, da diese schon bei \$SELFPRG definiert sind, sonst wird eine Fehlermeldung ausgegeben.

Beispiele:

\$DEBUG 'beim iL-TROLL

\$DEBUG modus,port,pin 'wenn keine SELFPRG definiert ist

\$SELFPRG modus,port,pin 'es soll sowohl SELFPROG als auch
\$DEBUG 'verwendet werden

\$IF \$ELSE \$ENDIF

Die Compilerschalter `$IF cond`, `$ELSE` und `$ENDIF` erlaubt eine bedingte Compilierung. Damit haben Sie die Möglichkeit, durch Setzen der Bedingung `cond` den Programmcode zwischen `$IF` und `$ELSE` bzw. `$ENDIF` übersetzen zu lassen. Ist die Bedingung nicht gesetzt, wird der Programmcode zwischen `$ELSE` und `$ENDIF` übersetzt. `cond` ist eine mittels `DEFINE` definierte Konstante. Ist der Wert 1 oder ein anderer Wert $\neq 0$, ist die Bedingung wahr und der `$IF`-Block wird ausgeführt. Die Bedingung wird unwahr, wenn der Wert auf 0 steht. In diesem Fall wird der `$ELSE`-Block, falls vorhanden, abgearbeitet.

Beispiel 1:

Die Zeile zwischen `$IF` und `$ENDIF` soll ausgeführt werden.

```
define schalter = 1 as const
```

```
$IF schalter
```

```
    LET ....
```

```
$ENDIF
```

Beispiel 2:

Die Zeile zwischen `$ELSE` und `$ENDIF` soll ausgeführt werden.

```
define schalter = 0 as const
```

```
$IF schalter
```

```
    LET wert = 10
```

```
$ELSE
```

```
    LET wert = 20
```

```
$ENDIF
```

Achtung!

Es sind keine Verschachtelungen möglich.

\$INCLUDE

\$INCLUDE erlaubt das Einbinden einer Datei während des Compiliervorgangs. Damit können große Projekte in handhabbare Dateigrößen unterteilt werden.

Viele größere Projekte werden unübersichtlich, wenn Funktionen weit verstreut über das gesamte Projekt stehen.

Außerdem sind viele Funktionen bereits getestet oder sollen aus einem anderen Projekt übernommen werden. Werden diese Funktionsgruppen z.B. Variablendeklaration, Ein- Ausgaberroutinen usw. in einzelne Module zusammengefasst, wobei jedes Modul als eigenständige Datei vorliegt, werden auch die größten Projekte übersichtlich und leicht zu handhaben.

Fehlt die Dateiextension, wird INC benutzt. Fehlt auch die Pfad- und Laufwerksangabe, so werden diejenigen Bezeichnungen des Hauptfiles übernommen.

Die DEVICE-Anweisung mit der Angabe des Prozessortyps **muss** vor dem ersten \$INCLUDE-stehen, ansonsten erfolgt die Fehlermeldung "Unbekannte CPU".

\$LIST

\$LIST dient eigentlich zur Steuerung des Assemblers, der das vom Compiler übersetzte Programm in ein OBJ-File übersetzt. Manchmal ist es notwendig, Einfluss auf den Assembler zu nehmen, um beispielsweise die OBJ-Datei so zu gestalten, dass das PICSTART-plus Programmiergerät von MICROCHIP diese Datei richtig lesen kann (iL_PRG16 verwendet ein etwas erweitertes Format).

\$LIST /M_OBJ erzeugt ein für den PICSTART-plus geeignetes File.

\$LIST INHX8 erzeugt eine OBJ-Datei im Intel-Hex8-Format (default).

\$LIST INHX16 erzeugt eine OBJ-Datei im Intel-Hex16-Format.

\$LIST C=xxx legt die Anzahl der Zeichen pro Zeile im LST-File fest.

\$LIST BIN erzeugt zusätzlich eine Binärdatei (wird von manchen Programmiergeräten benötigt).

\$LIST BINX erzeugt ein Binärfile mit vertauschten High- und Lowbyte.

\$LIST OBJ2HEX das Hexfile bekommt anstatt OBJ die Dateiendung HEX

\$LRANGE

\$LRANGE dient dazu, bei PICs mit Programmspeicher, die größer sind als eine Seite, (Siehe "Programm-Pages") das Compilermodul iL_PAGE0 zu veranlassen, dass die relevanten Bereiche um die Seitengrenzen individuell einstellbar sind. Es legt somit fest, wie groß der Bereich um diese Seitengrenzen sein soll, bei der ein GOTO in ein LJMP bzw. ein CALL in ein LCALL umgesetzt wird. Diese individuelle Anpassung ist nötig geworden, da es hin und wieder Konstellationen im Compiler gab, bei denen ein Programmabsturz erfolgte, weil iL_PAGE0 eine Umsetzung in einen LJMP oder LCALL nicht durchgeführte, weil die Grenzen zu eng waren (16 vor und 16 nach der Seitengrenze). Wenn ein seltsames Programmverhalten auftritt, nachdem das Programm größer als eine Seite wird, ist es ratsam, wenn man diesen Schalter verwendet und den Wert z.B. auf 32 oder 48 setzt.

\$LRANGE 48

Wichtig!!!

Natürlich sind die Möglichkeiten von \$LRANGE auch begrenzt. Je nach Programmaufbau kann es sein, dass der Bereich sehr groß gewählt werden muss, z.B. dann, wenn der Einsprungpunkt eines großen Unterprogramm noch auf der gleichen Seite liegt, das RETURN jedoch bereits in der nächsten. \$LRANGE muss also nur wegen eines ungeschickt im Speicher liegenden Unterprogramms sehr groß gewählt werden. Das führt letztendlich zur Verschwendung von Ressourcen (hier: des Speicherplatzes). Ein Ausweg aus diesem Dilemma bietet eine Sprungleiste am Programmanfang. Der Trick besteht darin, dass man das eigentliche Unterprogramm nur zwei Zeilen lang macht und dadurch ein Überschreiten der Speichergrenzen innerhalb des Unterprogramms verhindert. Ein Beispiel verdeutlicht die Wirkungsweise:

'Soweit wie möglich am Programmanfang:

'Nach den DEFINE-Anweisungen, aber vor

'den ersten BASIC-Befehlen

```
                GOTO MAIN 'Nachfolgende UPs überspringen
UP1             GOTO UP1X
UP1Z           RETURN
UP2            GOTO UP2X
UP2Z           RETURN
UP3            GOTO UP3X
UP3Z           RETURN
               ....
UP1X            tu irgendwas
               GOTO UP1Z
               ....
UP2X            tu irgendwas
               GOTO UP2Z
               ....
UP3X            tu irgendwas
               GOTO UP3Z
               ....
               ....
               GOSUB UP1
               ....
               GOSUB UP3
               ....
               GOSUB UP2
```

Nun ist es egal, von wo die Unterprogramme aufgerufen werden, und wo deren Ein- bzw. Aussprung liegen, denn ein GOTO bzw. LJMP setzt jetzt die entsprechenden Seitenbits, im Unterschied zu RETURN, immer richtig. Jetzt reicht es in der Regel, ein \$LRANGE16 einzustellen. Dies ist der Defaultwert und kann weggelassen werden.

\$LRANGE (cont.)

siehe auch Programm-Pages

\$LST2COD

(nicht für iL_TROLL)

Der Compilerschalter LST2COD erzeugt ein COD-File. Diese Datei beinhaltet neben dem Programmcode auch die Symboltabellen und die Informationen für den Debugger bzw. Emulator. Wenn Sie mit MPLAB den BASIC-Quelltext testen wollen, benötigen Sie diese Datei.

Desweiteren muss der Compiler eine HEX-Datei erzeugen. Der dafür notwendige Compilerschalter ist \$OBJ2HEX.

Damit der Compiler unter MPLAB richtig arbeitet, muss dieser sorgfältig in die IDE von MPLAB integriert werden. Lesen Sie dazu das Kapitel [Integration in MPLAB](#).

\$NCALDEF

Normalerweise erzeugt der Compiler für Bausteine mit eingeschaltetem INTERNEN RC-OSZILLATOR einen Defaultwert (80H), der dann in den Baustein programmiert wird, wenn es sich um einen JW-Typen handelt, dessen höchste Speicherstelle gelöscht ist. Da aber manche Programmiergeräte diese Information falsch interpretieren und damit einen falschen Wert in das OSCCAL-Register bekommen, kann dieser Automatismus durch \$NCALDEF (No CALibration DEFault) unterdrückt werden.

siehe auch CALVAL

\$OBJ2HEX

Manche Programmiergeräte oder Emulatoren (PICSTART bzw. MPLAB) benötigen eine HEX-Datei. Standardmäßig erzeugt der Compiler iL_BAS16 nur eine OBJ-Datei. Der Compilerschalter \$OBJ2HEX erzwingt die Generierung einer solchen HEX-Datei.

Wenn Sie mit MPLAB arbeiten wollen, müssen Sie noch weitere Punkte beachten. Diese sind im Kapitel [Integration in MPLAB](#) beschrieben.

\$OLDVAR

\$OLDVAR erlaubt die Anwendung der alten Form der Variablendeklaration für die Prozessoren, die in der Compilerversion 1 bis 4 definiert waren. Diese Abwärtskompatibilität erspart das Umschreiben bestehender Programme. \$OLDVAR muss ganz am Programmanfang stehen.

neue Variablendeklaration
alte Variablendeklaration siehe Handbuch

Wichtig!!!

Bitte verwenden Sie \$OLDVAR nur in Projekten die mit den Compilerversionen 2, 3 und 4 generiert wurden.

\$OLDVAR arbeitet nur mit den Prozessortypen, die zu jenem Zeitpunkt überhaupt definiert waren. Prozessoren die ab der Version 5 neu hinzu kamen, funktionieren deshalb nicht.

Ab Version 6 wird dieser Compilerschalter nicht mehr unterstützt.

\$OSCCON

Bei manchen PIC-Bausteinen ist ein RC-Oszillator implementiert, dessen Taktgeschwindigkeit in großen Bereichen veränderbar ist. Mittels des OSCCON-Registers wird diese Taktfrequenz eingestellt. Beim Power-Up des Bausteins wird dieser Taktgenerator auf die niedrigste Frequenz gestellt. Dies ist in der Regel ca. 31,25 kHz. Um den Baustein höher zu takten, muss das OSCCON-Register entsprechend modifiziert werden.

\$OSCCON kann mit und ohne Parameter benutzt werden. Wird kein Parameter verwendet, dann wird automatisch auf die Frequenz 4 MHz eingestellt. Soll eine andere Frequenz verwendet werden, muss dieser Compilerschalter mit einem der nachfolgenden Parametern aufgerufen werden.

Parameter => Frequenz

\$00 => 31,25 kHz

\$10 => 125 kHz

\$20 => 250 kHz

\$30 => 500 kHz

\$40 => 1 MHz

\$50 => 2 MHz

\$60 => 4 MHz

\$70 => 8 MHz

Es sind noch andere Werte möglich. Lesen Sie dazu im entsprechenden Datenblatt die Bedeutung der übrigen Bits nach. Der Parameter wird vom Compiler direkt in das OSCCON-Register geschrieben.

Hinweis:

Im Programm kann dieser Compilerschalter nur einmal vorkommen. Will man während des Programmlaufes die Taktfrequenz ändern, um beispielsweise Strom zu sparen, dann schreiben Sie den gewünschten Wert direkt nach OSCCAL. Die BASIC-Anweisung dazu lautet:

LET OSCCAL = \$10 'PIC läuft nun mit 125 kHz

\$SELFPRG

(nur 16F818/16F189, 16F87x, 16F87/16F88 und 18Fxxx)

\$SELFPRG *mode,port,pin* installiert das SP_BIOS in den obersten Adressbereich des PICs. *mode* kann die Werte 0, bis 4 annehmen. *port* und *pin* können beliebig gewählt werden, müssen jedoch dieser Funktion exklusiv zur Verfügung stehen. Bei *mode* 0 und 2 wird zuerst das SP_BIOS einfach ignoriert. Der Programmierer kann an jeder beliebigen Stelle des Programms (aber nicht unbedingt in einer Interruptroutine) mit der Anweisung SELFPROG ins SP_BIOS springen. Allerdings wird sofort versucht die Daten von der Schnittstelle zu lesen. Nach der Programmierung verzweigt *mode* 0 zur Adresse 0 und *mode* 2 zum Befehl hinter SELFPROG.

Bei *mode* 1 verzweigt das Programm sofort nach dem Einschalten oder einem Reset ins SP_BIOS und prüft dort den Kommunikationspin ab. Ist dieser LOW (Pull-Down-Widerstand) wird sofort zum eigentlichen Programm zurückgesprungen. Bei einem High-Pegel wartet das SP_BIOS auf die Datenübertragung. Sobald die Programmierung beendet ist, wird das Anwenderprogramm gestartet.

mode 3 und *mode* 4 legen das SP_BIOS in den unteren Speicherbereich.

Mode => Speicher => Einsprung => Rücksprung
0 => oben => SELFPROG => Adresse 0
1 => oben => nach Power-On => Anwenderprogramm
2 => oben => SELFPROG => hinter SELFPROG
3 => unten => nach Power-On => Anwenderprogramm
4 => unten => SELFPROG => hinter SELFPROG

Modus 3 ist die, für die meisten Anwendungsfälle sinnvollste Variante. Die übrigen Modi verlangen eine sehr detaillierte Kenntnis der Interna.

\$SELFPRG 3,port,pin

Bei der Familie der PIC18 sind die Speicherbereiche mit unterschiedlichen Schutzmaske vor dem Überschreiben schützbar. Es empfiehlt sich zumindest am Anfang die Schutzmaske nach folgendem Schema zu verwenden:

conf1 = \$22	'nach eigenen Vorstellungen
conf2 = \$0f	' "
conf3 = \$00	
conf5 = \$01	'01
conf6 = %10000001	
conf8 = \$0F	'0F normaler Bereich 0200-ende nicht geschützt
conf9 = \$C0	'\$80 = Daten-EEPROM ungeschützt; 0000-01FF geschützt
conf10 = \$0f	'Programmspeicher nicht schreibgeschützt
conf11 = \$E0	'0000-01FF schreibgeschützt
conf12 = \$0f	'0200-ende nicht geschützt vor TABLE-READ
conf13 = \$40	'0000-01FF geschützt vor TABLE-READ

Weitere Informationen bei SELFPROG.

\$TROFF

(zur Zeit nur für 16F628, 16F88 und 16F87x sowie den iL-TROLL)

Wird mittels dem Compilerschalter \$DEBUG die Generierung von Debugcode an Anfang einer jeden BASIC-Anweisung aktiviert, wird normalerweise dies für alle BASIC-Anweisungen durchgeführt. Das kostet aber nicht nur zusätzlichen Programmspeicher, sondern reduziert auch die echte Ablaufgeschwindigkeit des Programms. Das ist insbesondere bei zeitkritischen Routinen nachteilig. Durch \$TROFF kann die Generierung des Debugcodes unterdrückt werden. Alle BASIC-Anweisungen, die diesem Schalter folgen, haben keinen zusätzlichen Code und laufen deshalb auch wieder mit der Originalgeschwindigkeit ab. Mit \$TRON kann diese Unterdrückung wieder aufgehoben werden.

\$TRON

(zur Zeit nur für 16F628, 16F88 und 16F87x sowie den iL-TROLL)

Wird mittels dem Compilerschalter \$DEBUG die Generierung von Debugcode an Anfang einer jeden BASIC-Anweisung aktiviert, wird normalerweise dies für alle BASIC-Anweisungen durchgeführt. Das kostet aber nicht nur zusätzlichen Programmspeicher, sondern reduziert auch die echte Ablaufgeschwindigkeit des Programms. Das ist insbesondere bei zeitkritischen Routinen nachteilig. Durch \$TROFF kann die Generierung des Debugcodes unterdrückt werden. Alle BASIC-Anweisungen, die diesem Schalter folgen, haben keinen zusätzlichen Code und laufen deshalb auch wieder mit der Originalgeschwindigkeit ab. Mit \$TRON kann diese Unterdrückung wieder aufgehoben werden.

\$VCFG / \$VCFG n

Bei PIC-Bausteinen mit einem ANSEL-Register kann man die Referenzspannung für den AD-Wandler dem dafür vorgesehenen Pin oder Vcc zuordnen. Da dieser Schalter mit dem Registerbit VCFG verknüpft ist, lässt sich die Referenz sogar während des Programmablaufs ändern.

Normalerweise hat \$VCFG keinen Parameter. Als Defaultwert ist Vref auf Vcc eingestellt. Sobald dieser Schalter gesetzt ist (existiert), wird Vref auf den entsprechenden Pin gelegt.

Bei manchen PICs gibt es mehrere Varianten Vref und -Vref zu definieren. In diesem Fall bekommt \$VCFG noch einen Parameter nachgestellt. In diesem Parameter steht die entsprechende Bitmaske für das Register in welchem VREF definiert wird.

Beispiel:

beim 12F683

\$VCFG 'Vref an RA1 (Analogeingang 1)

beim 16F88

\$VCFG \$00 => VRef+ = Vdd; VRef- = Vss (= default)

\$VCFG \$10 => VRef+ = Vdd; VRef- = RA2

\$VCFG \$20 => VRef+ = RA3; VRef- = Vss

\$VCFG \$30 => VRef+ = RA3; VRef- = RA2

\$WDTUSR

Sobald in der DEFINE DEVICE Zeile der Watchdog-Timer mit dem Schalter WDT_ON aktiviert ist, generiert der Compiler nach jedem BASIC-Befehl ein CLRWDT. Einzige Ausnahme dabei ist eine Bit-Abfrage mit der Verzweigung auf sich selbst (z.B. WARTE: IF RA,0=0 THEN GOTO WARTE). Hier kann der Watchdog aktiv werden, was auch beabsichtigt ist. Soll auch hier ein CLRWDT vom Compiler eingefügt werden, muss eine Dummyzeile eingefügt werden.

Es gibt aber auch sehr kritische Anwendungen, wo man ganz gezielt nur wenige CLRWDT-Befehle wünscht. Hier kann diese automatische Generierung von CLRWDT-Befehlen unterdrückt werden. Dabei bleibt in der DEFINE DEVICE Zeile das WDT_ON stehen, damit das Programmiergerät iL_PRG16PRO diese Einstellung auch übernimmt. Allerdings muss man nun an geeigneter Stelle den CLRWDT-Befehl selbst platzieren. CLRWDT ist als BASIC-Befehl verfügbar, so dass, im Gegensatz zu früheren Versionen (vor 5.5-09), keine Assemblerzeile eingebaut werden muss.

DEFINE Variable, Konstante

DEFINE definiert Variablenamen, Konstantenamen und legt den Prozessortyp und dessen Konfigurationsparameter fest.

Variablen werden definiert, indem man dem Symbol, bestehend aus mind. zwei Zeichen, wobei das erste Zeichen ein Buchstabe sein muss, die absolute Speicheradresse angibt, an dem der Compiler die Variablenzuweisungen ablegen soll.

z.B.

```
DEFINE counter = $30 AS BYTE
```

damit bekommt die Speicherzelle 30H den Symbolnamen *counter* zugewiesen (Angabe BYTE ist optional).

```
DEFINE summe = 49 AS WORD
```

damit bekommt die Speicherzelle 31H(=49d) den Symbolnamen *summe* zugewiesen. Da es sich um eine Wortvariable handelt, wird in 31H das Lowbyte und in 32H das Highbyte abgespeichert.

```
DEFINE wert = $40 AS DBLWORD
```

damit werden die Speicherplätze 40H bis 43H als 32-Bit Variable mit dem Namen *wert* definiert. (Siehe auch DEFINE sonstige).

Um die Zuordnung der Variablen zu den freien Speicherstellen zu erleichtern wurde der **AUTO**-Typ eingeführt. In diesem Fall übernimmt der Compiler die Zuordnung der Speicheradressen.

Beispiel:

```
DEFINE counter = AUTO AS BYTE
```

```
DEFINE summe = AUTO AS WORD
```

```
DEFINE wert = AUTO AS DBLWORD
```

Ist *counter* die erste Definition, sucht der Compiler für den jeweiligen PIC-Typ die erste freie Speicherstelle und weist sie *counter* zu. *summe* bekommt die nächsten zwei Speicherplätze und auf diese folgen dann die 4 Bytes des Doppelwortes *wert*

Um getrennt auf das obere Byte der Variable *summe* zugreifen zu können, kann man folgendes definieren.

```
DEFINE summe_hbyte = summe + 1
```

Eine Definition von Byte-Arrays ist auch mit der Option AUTO möglich. Dazu gibt man hinter AUTO die gewünschte Arraygröße an.

Beispiel:

```
DEFINE feld = AUTO(10)
```

oder

```
DEFINE feld = AUTO(10) AS BYTE
```

Konstanten werden definiert, indem man dem Symbol, bestehend aus mind. zwei Zeichen, wobei das erste Zeichen ein Buchstabe sein muss, den gewünschten Zahlenwert zuweist. Damit der Compiler zwischen Variablen und Konstanten unterscheiden kann, folgt als Kennung AS CONST.

Z.B.

```
DEFINE grenzwert = 100 AS CONST
```

damit wird dem Symbol *grenzwert* der Zahlenwert 100 zugewiesen. *grenzwert* wird als Bytekonstante behandelt.

```
DEFINE maximal = $4000 AS CONST
```

damit wird dem Symbol *maximal* der Zahlenwert 4000H zugewiesen. *maximal* wird hier als Wordkonstante behandelt, weil der Wert größer 255 ist.

DEFINE Variable, Konstante (cont.)

Definition von IO-Pins oder einzelnen Bits.

Auch die IO-Pins lassen sich per DEFINE Symbolen zuordnen. Z.B. kann man dem Symbol TASTE den Pin RB2 zuordnen. Die notwendige Anweisung lautet: `DEFINE TASTE = RB,2`.

Um in einer Variablen ein einzelnes Bit zu definieren schreibt man: `DEFINE EIN_AUS = variablenname,bitnr`. Damit lassen sich 8 Bitvariablen pro Byte definieren.

Bitvariablen können nur innerhalb Bytevariablen definiert werden (nicht 16- oder 32-Bit-Variablen).

Hinweis:

Variablendefinitionen sind beim iL-TROLL nur noch mittels AUTO möglich.

DEFINE Prozessor

Der Prozessortyp wird durch folgende Anweisung festgelegt:

bis einschließlich Version 5 gilt:

```
DEFINE DEVICE cpu_type, watchdog, code_protection, oscillator-type, ad_converter, misc
```

ab Version 6 ist hier nur noch *cpu_type* erlaubt. Restliche Angaben erfolgen unter CONFIG bzw CONF.

```
DEFINE DEVICE cpu_type
```

mit:

cpu_type (hängt von der Compilerversion ab und wird laufend ergänzt;)

10F200, 10F202, 10F204, 10F2006, 12C508, 12F508, 12C509, 12F509, 12E518, 12E519, 12F629, 12C671, 12C672, 12E673, 12E674, 12F675, 16C505, 16C53, 16C54, 16F54, 16C55, 16C554, 16C556, 16C558, 16C56, 16C57, 16F57, 16C58, 16F59, 16C61, 16C62, 16C62A, 16C620, 16C621, 16C622, 16E623, 16E624, 16E625, 16F627, 16F627A, 16F628, 16F628A, 16F630, 16F648A, 16C63, 16C64, 16C64A, 16C65, 16C65A, 16C66, 16C66A, 16C67, 16C67A, 16F676, 16C71, 16C710, 16C711, 16C715, 16C72, 16C73, 16C73A, 16F737, 16C74, 16C74A, 16F747, 16C76, 16F767, 16C77, 16F777, 16F818, 16F819, 16C83, 16F83, 16C84, 16F84, 16F870, 16F871, 16F872, 16F873, 16F873A, 16F874, 16F874A, 16F876, 16F876A, 16F877, 16F877A, 16F87, 16F88, 18F242, 18F252, 18F442, 18F452, 18F2420, 18F2520, 18F4420, 18F4520, 18F2525, 18F4525

watchdog:

WDT_ON (*) = Watchdog ist eingeschaltet

WDT_OFF (*) = Watchdog ist ausgeschaltet (default)

code_protection:

PROTECT_ON (*) = Code ist auslesegeschützt

PROTECT_OFF (*) = Code kann mittels Programmiergerät ausgelesen werden (default)

JW (überschreibt ein PROTECT_ON; JW-Bausteine sollten niemals lesegeschützt werden!)

data ee memory code protection: (Daten-EEPROM falls vorhanden)

CPD_ON = Daten sind auslesegeschützt

CPD_OFF = Daten sind mittels Programmiergerät lesbar

oscillator_type: (*)

RC_OSC = externes RC-Glied bestimmt die Arbeitsfrequenz

LP_OSC = niedrig Energiemodus für 32 kHz Quarze

XT_OSC = Quarzbetrieb, auch bei externem Quarzoszillator einzustellen (default)

HS_OSC = Quarzbetrieb, für Frequenzen ≥ 4 MHz

IRC_OSC = interner RC-Oszillator (nicht alle PICs)

ERC_OSC = bei einigen PICs ist dies die Einstellungen für externen RC-Oszillator, bei anderen PICs für externen Quarzoszillator. (Datenblatt)

ad_converter (nur PICs mit AD-Converter z.B. 16C7x und 16F87x):

ADCFG0, ADCFG1, ADCFG2,...

Einstellung, welcher Pin als Analog- und welcher als Digital-IO-Pin arbeiten soll.

(Wichtig!!! Bei den Bausteinen 12F629, 12F675, 16F676 u.a. hat ADCFG eine andere Zuordnungsvorschrift.)

Bei iL_TROLL wird immer ADCFG4 automatisch eingestellt!

siehe auch ADINP

Komparator (nur PICs mit Komparator z.B. 16C62x und 16E62x):

CMCFG0, CMCFG1, CMCFG2,...

DEFINE Prozessor (cont.)

Legt fest, wie die PINs zusammen mit den Komparatoren arbeiten sollen, bzw. welche als Digital-IO fungieren.

VRCFG = Definiert den Wert, der in das VRCON-Register geladen wird. Hier wird u.a. die Referenzspannung eingestellt. Diese Einstellung kann natürlich auch während des Programmablaufes durch Beschreiben des VRCON-Register geändert werden.

siehe auch KOMPARATOR

misc (werden nicht von jedem PIC unterstützt):

PWRTE_OFF (*) = Der Power-Up-Timer (Startverzögerung, damit der Quarz sauber anschwingen kann) ist abgeschaltet.

PWRTE_ON (*) = Power-Up-Timer ist eingeschaltet (72 ms nach dem Einschalten beginnt die CPU zu arbeiten).

MCLR_EXT (*) = Die Master-Reset-Logik ist mit dem MCLR-Pin verbunden. Legt man diesen Pin auf Masse, wird ein Reset ausgelöst.

MCLR_INT (*) = Der MCLR-Pin arbeitet als Digital-IO-Pin. Die Resetlogik ist intern mit 5V verbunden.

LVP_ON (*) = RB3 hat die Funktion PGM. Wenn Low, geht der PIC in den Programmiermodus.

LVP_OFF (*) = RB3 hat normale IO-Funktion. Der Programmiermodus wird nur durch 13V am MCLR Pin aktiviert.

RBPU_ON = Die internen sehr hochohmigen Pullup-Widerstände sind aktiv (nur wenn Pin als Eingang definiert).

RBPU_OFF = Die internen Pullup-Widerstände sind abgeschaltet.

GPWU_ON = Im Sleepmodus wird bei einer Pegeländerung an einem Pin ein Reset ausgelöst (wake up).

GPWU_OFF = obige Funktion ist abgeschaltet.

GPPU_ON = Wie RBPU_ON.

GPPU_OFF = Wie RBPU_OFF

BODEN_ON (*) = Der Brown-out Detektor (Unterspannungserkennung) ist eingeschaltet.

BODEN_OFF (*) = Die Unterspannungserkennung ist abgeschaltet.

OSC2_IO = Der OSC2-Pin arbeitet als Digitaler-IO-Pin.

T0CS_INT = Der Takt für den Timer0 wird vom internen Befehlstakt (Quarzfrequenz/4) abgeleitet.

T0SE0 = Timer0 wird bei steigender Flanke inkrementiert.

T0SE1 = Timer0 wird bei fallender Flanke inkrementiert.

PSA0 = Der Vorteiler (Optionregister) wird Timer0 zugeordnet.

PSA1 = Der Vorteiler (Optionregister) wird dem Watchdog zugeordnet.

PSV0...PSV7 = Vorteilerwert

0 = 1:2 (TMR0) bzw. 1:1 (WDT)

1 = 1:4 (TMR0) bzw. 1:2 (WDT)

2 = 1:8 (TMR0) bzw. 1:4 (WDT)

3 = 1:16 (TMR0) bzw. 1:8 (WDT)

4 = 1:32 (TMR0) bzw. 1:16 (WDT)

5 = 1:64 (TMR0) bzw. 1:32 (WDT)

6 = 1:128 (TMR0) bzw. 1:64 (WDT)

7 = 1:256 (TMR0) bzw. 1:128 (WDT)

(*)

Unter Umständen kann die Liste der Parameter für diesen Befehl sehr umfangreich werden. Man kann aber die Angaben für das Konfigurationswort auch über CONFIG definieren.

DEFINE sonstige

Weitere Funktionen von DEFINE

DEFINE STACK = *adr*

hier wird ein 16-Byte großer Speicherbereich zum Ablegen von Registerinhalten beim Auftreten eines Interrupts definiert (Stack). *adr* ist die untere Startadresse dieses Bereiches der unbedingt in der Page0 liegen muss.

DEFINE SERIN = SOFT

DEFINE SEROUT = SOFT

Zwingt den Compiler, die Softwarelösung von SERIN bzw. SEROUT einzubinden, obwohl dieser Baustein an diesen Pins die entsprechende Hardware implementiert hat.

DEFINE KEYS *port, tabelle*

initialisiert für den Befehl INKEY eine Tastaturmatrix am ausgewählten Port. Optional kann in einer Tabelle festgelegt werden, welchem Scancode welcher Tastenwert zugewiesen werden soll. Normalerweise sind die Scancodes von 1 bis 16 definiert. Der Scancode 0 wird dann übergeben, wenn keine Taste gedrückt wurde. Soll beispielsweise der Scancode 1 das ASCII-Zeichen "0", Scancode 2 des Zeichen "1" usw. ergeben, so schreibt man:

DEFINE KEYS RB, 0,"0","1","2","3","4","5","6","7","8","9","A","B","C","D","E","F"

DEFINE ARITH32 = *adresse*

Legt den Speicherbereich fest, wo die 32-Bit Operationen ihre Zwischen- und Endergebnisse berechnen. Der Bereich umfasst 16 Byte und kann u.U. auch anderweitig benutzt werden. VORSICHT!!!

DATE

Der Compiler ersetzt dieses Schlüsselwort in der neu erzeugten SRC-Datei durch das aktuelle Systemdatum. Es erscheint somit nach dem Assemblieren auch in der LST-Datei.

TIME

Der Compiler ersetzt dieses Schlüsselwort in der neu erzeugten SRC-Datei durch die aktuelle Systemzeit. Sie erscheint somit nach dem Assemblieren auch in der LST-Datei.

XTAL

(nicht für iL_TROLL)

Das Schlüsselwort XTAL legt die Frequenz der Applikation fest. Die Angabe erfolgt in MHz. Aus dieser Angabe berechnet der Compiler die Zählwerte für Verzögerungsschleifen in Routinen wie z.B. die serielle Empfangs- bzw. Sendefunktionen.

Z.B.

XTAL 4.0

die Applikation wird mit 4MHz betrieben.

XTAL 0.032

die Applikation wird mit 32kHz betrieben.

iL_TROLL arbeitet grundsätzlich mit 4.0 MHz!

CLK1XTAL

Soll statt dem TMR0 der TMR1 als Zeitbasis für den CLOCK-Befehl dienen (CLOCK1), dann kann dieser Timer TMR1 statt mit dem internen Befehlstakt auch durch einen zweiten vom Hauptquarz unabhängigen Oszillator betrieben werden. Damit der Compiler die Teilerfaktoren richtig einstellen kann, muss man die Quarzfrequenz dieses zweiten Oszillators mit CLK1XTAL definieren.

Assembler Code

Generell ist es möglich, Programmabschnitte in Assemblercode zu schreiben und einzugliedern (etwa mit den Basic-Befehlen ASM und ENDASM). Obwohl in der Regel nicht notwendig, kann es insbesondere dann sinnvoll sein, wenn man sozusagen Fein-Tuning betreiben will. Greift man dabei auf die Hardware-Ressourcen (Timer, deren Prescaler, Interrupts, Schnittstellen, etc) zu, sind Konflikte nicht auszuschließen, da auch der Compiler auf einige solcher Hardware-Ressourcen zugreift. Zum Beispiel bei Verwendung der entsprechenden Befehle (CLOCK, I2CHARDS, SERIN etc) oder durch die stets im Compiler eingebundenen Voreinstellungen auf die Special-Functions-Register des PIC.

Um fehlerverursachende Kollisionen zu vermeiden, kann man mehrere (alternative) Strategien empfehlen:

1) Den Assembler-geschriebenen Code auf diejenigen Hardware-Ressourcen einschränken, auf die iL_BAS16 nicht zugreift. Je nach PIC-Variante gibt es verschiedene Anzahl an Spezialfunktionen (z.B. bis zu vier Timer). Es können also je nach PIC-Typ freie Ressourcen übrig bleiben, die von Basic nicht benutzt werden und daher frei verwendbar sind.

2) Den Zugriff des Compilers einschränken, sodass er auf diejenigen Ressourcen nicht zugreift, die man in Assembler selber programmiert. Hierzu erstens aus der Default-Liste alle Zeilen löschen, die die betreffenden Voreinstellungen (der Spezialfunktionen-Register) enthalten.

Dann gelten die PIC-eigenen Default-Einstellungen. Zweitens dürfen dann keine der betroffenen BASIC-Befehle verwendet werden. Welche Befehle betroffen sind, ist ersichtlich in "Übersicht der Basic-Befehle" (Spalte: verwendete Ressourcen)

Dieser zweite Schritt ist allerdings nicht zu empfehlen, da hierzu exakte Kenntnisse über die interne Funktionsweise des Compilers notwendig sind.

3) In Assembler-Programmteilen zunächst alle betroffenen Register wie benötigt selber einstellen, vor Beenden dieses Programmteils, spätestens aber vor Gebrauch betreffender BASIC-Befehle, alle veränderten Voreinstellungen wie laut Liste zurücksetzen. Zu beachten ist aber, dass einige Befehle durchweg agieren (z.B. Timer - Interrupts).

Lesen Sie dazu auch das Kapitel ASSEMBLER

Interrupts (allgemein)

Die Nutzung der Interrupts bei den Bausteinen 12C67x, 16C6x, 16C7x und 16X8x ist für BASIC-Programmierer nun wesentlich einfacher. Die neudefinierten Schlüsselworte INTERRUPT, INTPROC und INTEND sorgen für einfaches Einbinden der eigenen Interruptroutinen in die Kette der compilereigenen Interruptbehandlung. Das Schlüsselwort INTERRUPT teilt dem Compiler mit, dass der Anwender selbst eine Interruptroutine erstellen möchte. Diese Interruptserviceroutine ISR wird dann vom Compiler in die vorhandene ISR (CLOCK) eingeklinkt. Diese CLOCK-ISR ist extrem kurz gehalten und wird bei einem TMR0-Interrupt ausgeführt. Die eigentliche Anwender-ISR wird durch die Schlüsselworte INTPROC und INTEND eindeutig definiert (ähnlich wie ASM und ENDASM). Die ISR kann sowohl in BASIC als auch in Maschinensprache geschrieben sein. Trifft der Compiler auf das Schlüsselwort INTERRUPT, stellt er zwei zusätzliche Register zur Verfügung. Das ist zum einen das INTCON0 (=0BH) und das ADCON0 bzw EECON0 als INTCON1 (=88H). Eine eigene Definition für diese Register ist dann nicht mehr notwendig. Diese Steuerregister müssen in der ISR ausgewertet werden. Ebenso muss das interruptauslösende Flag per Software zurückgesetzt werden.

Wenn Sie die Interrupts verwenden wollen, müssen Sie neben der eigentlichen Interruptroutine noch die Interrupts freischalten. Dies erfolgt z.B. mit SET INTE oder SET RBIE. Verwenden Sie jedoch BASIC-Befehle, die die Interrupts nutzen, wie z.B. CLOCK oder interruptgesteuertes SERIN, dann übernimmt die Freischaltung dieser speziellen Interrupts der Compiler selbst.

Die interruptfähigen PICs kennen verschiedene Interruptquellen. Allerdings kann keine Prioritätskette definiert werden; d.h. alle Interrupts haben die gleiche Priorität. Welcher PIC welche Interrupts ermöglicht entnehmen Sie bitte den einzelnen Datenblättern des PIC-Herstellers. Es gibt u.a. folgende Interruptquellen:

INT	flankensensitiv an RB0
RTCC	beim Wechsel von FF auf 0 des RTCC-Registers
RB	Ein Pegelwechsel an Pin RB4 bis RB7 (log. ODER)
EEPROM	Sobald ein Schreibzyklus abgeschlossen ist, wird dieser Interrupt ausgelöst
ADC	Am Ende einer AD-Wandlung wird ein Interrupt ausgelöst.

Nachfolgend sind einige Interrupts und deren Funktionsweisen kurz erklärt. Weitergehende Informationen finden Sie in den Datenblättern der einzelnen PIC-Bausteine.

INT-Interrupt

Der externe Interrupteingang RB0/INT ist flankensensitiv. Über die aktive Flanke entscheidet das INTEDG-Bit im Option-Register. Ist dieses Bit 0, löst eine fallende Flanke an diesem Pin den Interrupt aus, bei einer 1 ist es die steigende Flanke. Erscheint die aktive Flanke an diesem Pin, wird zuerst das INTF-Bit im INTCON0 Register gesetzt. Ein Interrupt wird jedoch erst dann ausgelöst, wenn auch das INTE-Bit gesetzt ist und zusätzlich noch die Interrupts global freigegeben sind (GIE). In der Interruptroutine muss das INTF-Bit wieder per Software zurück gesetzt werden. Dieser Interrupt kann auch den SLEEP-Modus des Prozessors beenden, falls das INTE-Bit auf 1 gesetzt ist. Der Status des GIE-Bits entscheidet dann, ob der Prozessor neu gestartet wird (GIE=0, RESET) oder ob in die ISR verzweigt wird.

RTCC-Interrupt

Ein Überlauf des RTCC-Registers von FF auf 0 setzt das RTIF-Bit auf 1. Auch hier wird nur dann ein Interrupt ausgelöst, wenn die korrespondierenden Bits RTIE und GIE auf 1 gesetzt sind.

RB-Interrupt

Ein Pegelwechsel an den Eingängen RB4 bis RB7 setzt das RBIF-Flag. Ein Interrupt wird nur ausgelöst, wenn GIE und RBIE auf 1 gesetzt sind.

ADC-Interrupt (nur 16C7x und 16F87x)

Der AD-Wandler setzt am Ende einer Datenwandlung das ADIF-Bit. Ein Interrupt wird nur dann ausgelöst, wenn neben dem GIE- auch das RBIE-Bit gesetzt ist.

Interrupts (allgemein) (cont.)

EEPROM-Interrupt (nur 16X8x und 16F87x)

Wenn der Schreibzyklus für eine EEPROM-Datenzelle abgeschlossen ist (ca. 10ms), wird das EEIF-Bit gesetzt. Ist das GIE- und das EEIE-Bit auf 1 gesetzt, wird ein Interrupt ausgelöst.

Was passiert bei einem Interrupt?

Bei einem Interrupt wird zuerst der Programmzähler auf dem Stack abgelegt. Dann verzweigt das Programm zur Adresse 0004H. Dort muss an Hand der gesetzten Bits im INTCON0 bzw. INTCON1 ermittelt werden, welche Quelle den Interrupt ausgelöst hat. Dann verzweigt man zur entsprechenden Interruptbehandlungsroutine. Da der PIC keine PUSH und POP Befehle kennt, muss man die Register, die vom ISR verwendet werden, u.U. sichern. Um das W- und Statusregister zu retten, kann man folgende Programmsequenz verwenden:

push	movwf	temp_w	;w in eine RAM-Zelle
	swapf	status,w	;Statusregister ins W
	movwf	temp_s	;und in eine RAM-Zelle
pop	swapf	temp_s,w	;Statusregister wieder herstellen
	movwf	status	
	swapf	temp_w	;W-Register
	swapf	temp_w,w	

Diese Routinen werden allerdings vom BASIC-Compiler automatisch eingebunden, sobald der CLOCK-, I2CHARDS- oder der interruptgesteuerte SERIN-Befehl im Programm vorkommt.

Das INTCON0 Register

Im Interrupt-Control-Register werden die einzelnen Interruptquellen gesperrt oder freigegeben und beim Auftreten eines Interrupts das entsprechende Zustandsbit gesetzt. Alle Interrupts können einzeln maskiert werden. Zusätzlich ist noch ein Flag vorhanden, mit dem alle Interrupts global gesperrt und freigegeben werden können.

Der Inhalt der INTCON-Register kann von Baustein zu Baustein unterschiedlich sein. Unter Umständen muss auch das Register PIE berücksichtigt werden! Im Zweifelsfall muss im entsprechenden Datenblatt des Herstellers nachgesehen werden.

Bit 7	GIE	Global Interrupt Enable, 0=disable, 1=enable
16C7x		
Bit 6	ADIE	AD-Wandlung fertig, 0=sperren des AD_Interrupts, 1=Freigabe der AD-Interrupts
16C8x		
Bit 6	EEIE	EEPROM Schreiben fertig, 0=sperren, 1=freigegeben
Bit 5	RTIE	0=keine Interrupts vom RTCC möglich, 1=RTCC Überlauf löst Interrupt aus
Bit 4	INTE	0= RB0 löst keine Interrupts aus, 1=Flankenwechsel an RB0 löst Interrupt aus
Bit 3	RBIE	0=RB4-RB7 keine Interrupts, 1= Wechsel an RB4 bis RB7 löst Interrupt aus
Bit 2	RTIF	1 wenn RTCC-Überlauf, muss per Software zurückgesetzt werden
Bit 1	INT	Flankenwechsel an RB0 hat Interrupt ausgelöst
Bit 0	RB	Wechsel an RB4 bis RB7 hat Interrupt ausgelöst

Das INTCON1-Register

16C7x ADCON0 (08H)

Bit 7	ADCS1
Bit 6	ADCS0
Bit 5	res.
Bit 4	CHS1
Bit 3	CHS0

Interrupts (allgemein) (cont.)

Bit 2	GO/DONE	
Bit 1	ADIF	Gesetzt, wenn AD-Wandler fertig ist, wird per Software zurückgesetzt
Bit 0	ADON	

16X8x EECON1 (88H)

Bit 7	res	
Bit 6	res	
Bit 5	res	
Bit 4	EEIF	gesetzt, wenn Schreibzyklus beendet ist, zurücksetzen per Software
Bit 3	WRERR	
Bit 2	WREN	
Bit 1	WR	
Bit 0	RD	

WICHTIG!!!

Beim Arbeiten mit Interrupts gibt es immer wieder Probleme, die deshalb entstehen, weil die PIC-Controller über keine PUSH und POP Befehle verfügen und der Compiler selbst noch bestimmte Register benötigt, um die einzelnen BASIC Anweisungen verarbeiten zu können. Bearbeitet beispielsweise der PIC gerade die BASIC Anweisung $LET\ ZZ = TT * VV$, so werden dazu die Hilfsregister ARG0 bis ARG5 mit den entsprechenden Variableninhalten beschrieben. Dann verzweigt das Programm in die Laufzeitbibliothek, wo sich u.a. die 16 Bit Multiplikation befindet. Es sind also viele Programmschritte notwendig, um diese einzelne BASIC Anweisung auszuführen.

Kommt nun aber bei dieser Abarbeitung ein Interrupt, wird das laufende Programm (hier mitten in der Multiplikation) unterbrochen und der Interrupt bedient. Kommen in der benutzerdefinierten Interrupt-Serviceroutine komplexere Befehle vor, werden diese ebenfalls mit Hilfe der Hilfsregister ARG0 bis ARG5 bearbeitet. Da sich zu diesem Zeitpunkt aber noch wichtige Informationen von der Multiplikation darin befinden, werden diese gnadenlos überschrieben, was zu fehlerhaften Ergebnissen führt. Es werden zwar immer das STATUS- und das W-Register gerettet, der Rest jedoch nicht.

Bei den PICs mit mehr Datenspeicher übernimmt der Compiler die Arbeit und rettet auch die Hilfsregister. Bei den kleineren PICs ist leider jedes Byte extrem wichtig, so dass hier eine andere Lösung des Problems gesucht werden müsste.

Die eine Lösung heißt GIED und wird beim Schlüsselwort INTERRUPT als Zusatz mit angegeben. Diese Anweisung zwingt den Compiler die Interrupts vor jeder komplexeren BASIC-Anweisung global zu sperren und am Ende der Anweisung wieder freizugeben. Damit wird verhindert, dass ein Interrupt die Abarbeitung einer BASIC-Anweisung unterbricht und so Fehler erzeugt. Nachteil dieser Methode ist allerdings die nicht konstante Latenzzeit der eintreffenden Interrupts. D.h.: kommen in einem zeitlich festen Raster Interrupts herein, werden diese durch die unterschiedlich lange Abarbeitungszeit der einzelnen BASIC-Anweisungen eben nicht im festen Zeitraster abgearbeitet. Das hat bei einer interruptgesteuerten Ausgabe zur Folge, dass das Ausgangssignal einen Jitter bekommt.

Sollte dieses Problem auftauchen, muss das ganze Programm unter Berücksichtigung der Interrupts in Maschinensprache geschrieben werden oder es muss ein PIC-Baustein eingesetzt werden, der mehr Daten-RAM zur Verfügung stellt, damit der Compiler auch die ARG0 bis ARG5 Hilfsregisterinhalte retten kann.

Die andere Lösung des Interruptproblems ist ein STACK. INTERRUPT hl, STACK = adr legt bei den "größeren PICs" mit mehr Datenspeicher die Startadresse für einen 16 Byte großen Speicherbereich fest, in den, im Fall eines Interrupts, die Hilfsregister vom Compiler gerettet werden. Ein GIED wird dadurch überflüssig.

Labels

Label müssen mit einem Buchstaben beginnen. Innerhalb des Labels darf jede Kombination aus Buchstaben, Zahlen und Unterstrichen benutzt werden. Ausgenommen sind alle reservierten Worte, wie Befehls- und Variablennamen. Nur die ersten 16 Zeichen sind signifikant.

Richtig:

START
schleife
Schleife_1

Falsch:

99 (1. Zeichen ist kein Buchstabe)
_schleife (")
HIGH (reserviertes Wort)

Adress-Label sind Sprungziele und müssen bei ihrer Definition durch einen Doppelpunkt abgeschlossen werden. Jeder Aufruf innerhalb des Programms erfolgt dann ohne den Doppelpunkt.

Beispiel:

```
START:
    FOR a1 = 0 TO 100
    . . .
    NEXT a1
    GOTO start
```

Label, die als Bezeichner einer Variablen oder Konstanten dienen, werden ohne Doppelpunkt bezeichnet. Sie müssen vor ihrer ersten Verwendung mit der `DEFINE` Direktive definiert werden.

Beispiel:

```
    DEFINE start=1
    DEFINE ende=100
    DEFINE zaehler=a

TEST:
    FOR zaehler=start to ende
    . . .
    NEXT zaehler
```

Label dürfen innerhalb eines Programmtextes nicht erneut definiert werden, der Compiler meldet dies als Fehler.

Beispiel:

```
    DEFINE start=1
    . . .
    . . .
    DEFINE start=100    erzeugt Fehlermeldung!!!!
```

Konstanten

Konstanten müssen mit einem Buchstaben beginnen. Innerhalb der Konstanten darf jede Kombination aus Buchstaben, Zahlen und Unterstrichen benutzt werden. Ausgenommen sind alle reservierten Worte, wie Befehls- und Variablennamen. Konstanten haben maximal 16 signifikante Zeichen.

Konstante Werte können auf vier Arten deklariert werden:

dezimal , hex, binär, ASCII

Hex-Zahlen werden durch ein vorgestelltes Dollar-Zeichen (\$) gekennzeichnet, Binär-Zahlen durch ein Prozent-Zeichen (%) und ASCII-Werte werden in Anführungszeichen eingeschlossen ("). Wird keine spezielle Kennung angegeben, nimmt der Compiler einen Dezimalwert an.

z.B.:

DEFINE start = 50	REM Dezimal
DEFINE start = \$FF	REM Hex (Dezimal 255)
DEFINE start = %00001111	REM Binär (Dezimal 15)
DEFINE start = "A"	REM ASCII ("A" = Dezimal 65)

Konstanten mit einem Label als Bezeichner werden mit der DEFINE-Direktive erzeugt. Dabei sind einige Besonderheiten zu beachten.

Der zulässige Wertebereich für Konstanten beträgt 0 ... 255, 0 ... 65535 oder 0 ... 4294967296. Der Versuch, einer Konstanten einen negativen Wert zuzuweisen, wird vom Compiler mit einer Fehlermeldung quittiert.

Richtig:

```
DEFINE start = 1 AS CONST
```

Falsch:

```
DEFINE start = -1 AS CONST
```

Mathematische Ausdrücke auf der rechten Seite des Gleichheitszeichens sind nicht erlaubt, daher können leider keine verketteten Konstanten definiert werden, die oft zur Wartbarkeit eines Programms beitragen.

ABER ACHTUNG

Bei der FOR-NEXT-Schleife sind allerdings Ausdrücke wie der nachfolgende erlaubt.

```
FOR a = start TO start+99
```

Variablen

Ab der Compilerversion 5.5 ist jetzt für die Version `il_BAS16PRO` die 32-Bit Arithmetik implementiert. Dazu musste der Variablentyp `DBLWORD` eingeführt werden. Diese Variablen benötigen 4 Speicherplätze. Zusätzlich benötigt der Compiler RAM-Speicher für die Berechnungen. Da dieser Speicher ausschließlich für die 32-Bit Operationen benötigt wird, wurde eine Methode implementiert, die es erlaubt, in der übrigen Zeit diesen RAM-Speicher anderweitig zu nutzen. Damit sind aber natürlich große Risiken verbunden. Eine 32-Bit Operation überschreibt diese Daten ohne wenn und aber. Falls diese Daten zu einem späteren Zeitpunkt noch gebraucht werden, dürfen sie nicht in diesem Bereich abgelegt werden. Der zusätzliche RAM-Speicher heißt `ACCU32AB` und wird mit `DEFINE` festgelegt. Er umfasst 16 Speicherplätze. Code für die 32-Bit-Arithmetik aber nur dann generiert, wenn auch tatsächlich Rechenoperationen mit 32-Bit-Variablen durchgeführt werden.

Für Umsteiger aus früheren Versionen

Ab der Compilerversion 5 wurde eine neue Variablendeklaration eingeführt. Dies wurde notwendig, weil die neuste Generation von PICs (16F87x) Datenbereiche besitzt, die über 4 Bänke verteilt sind. Das ergibt sich eine riesige Anzahl von Variablen, die mit der bisherigen Systematik nicht sinnvoll zu bewältigen ist.

Haben Sie bereits mit Vorgängerversionen dieses Compilers gearbeitet, so können Sie ihre alten Programme auch mit diesem neuen Compiler verarbeiten. Es ist allerdings notwendig, den Compiler anzuweisen, mit der alten Variablendeklaration zu arbeiten. Dies erfolgt mittels Compilerschalter `$OLDVAR`. Allerdings können so nur die ersten beiden Bänke für die Daten erreicht werden (genaue Beschreibung dieser alten Variablendeklaration). Die Datei `il_BAS16.EQU` beinhaltet weiterhin die Variablen nach der alten Fassung, die neue Art der Variablendeklaration zeigt sich in der Datei `DEFAULT.EQU`. `$OLDVAR` wird ab Version 6 nicht mehr unterstützt.

Die neuen Variablen bestehen grundsätzlich aus mindestens zwei Zeichen, wobei das erste Zeichen ein Buchstabe sein muss. Die maximale Zeichenzahl pro Variable (und auch Labels) darf 16 nicht überschreiten. Innerhalb des Variablennamens sind sowohl Umlaute als auch der Unterstrich erlaubt. Vermeiden Sie das Dollar-, Prozent- und Rautesymbol (#). Alle wichtigen Variablendefinitionen, die vom Compiler selbst, aber auch vom Assembler verwendet werden, stehen in der Datei `DEFAULT.EQU`. Sie haben sowohl vom BASIC aus, als auch über Assemblerbefehle Zugriff auf diese Variablen. Ihre eigenen Variablen definieren Sie wie bisher mittels `DEFINE`-Anweisung am Programmanfang oder aber in einer eigenen Includedatei. Für die absolute Speicherzelle, in der die Variable abgelegt wird, wird nicht mehr eine Pseudovariablen (`A,B,...AA, AB, usw.`) verwendet sondern man gibt die absolute Speicheradresse an.

Um dem Einsteiger diese Zuordnung von Variablen zur absoluten Speicheradresse abzunehmen, gibt es das Schlüsselwort **AUTO**. Damit überlässt man dem Compiler diese Zuordnung. Siehe dazu auch `DEFINE Variable, Konstante`

z.B.

```
DEFINE grenze = 10 AS CONST  
das Symbol grenze hat den Wert 10
```

```
DEFINE counter = $40  
DEFINE counter = $40 AS BYTE  
Damit bekommt die 8-Bit Variable counter den Speicherplatz $40 (64d) zugewiesen.
```

Um die 16-Bit Variable *ergebnis* zu deklarieren schreiben Sie:

```
DEFINE variable = $50 AS WORD  
DEFINE variable = 80 AS WORD
```

32-Bit Variablen werden mit (nur Professionalversion)

```
DEFINE variable = $50 AS DBLWORD  
festgelegt.
```

Die 32-Bit-Variablen in der Professionalversion (ab 5.5) werden gemäß dem obigen Vorgehen so definiert:

Variablen (cont.)

```
DEFINE    ZW_ERG = $60 AS DBLWORD
oder      DEFINE    ZW_ERG = 96  AS DBLWORD.
```

Zusätzlich benötigt man hier die Anweisung:

```
DEFINE ARITH32 = adresse
```

Dies legt den Speicherbereich fest, wo die 32-Bit Operationen ihre Zwischen- und Endergebnisse berechnen. Der Bereich umfasst 16 Bytes und kann u.U. auch anderweitig benutzt werden. VORSICHT!!!

Eine 32-Bit-Variable belegt somit 4 Speicherplätze.

Die Angabe AS BYTE ist optional und darf auch weggelassen werden. Aus Gründen der Übersichtlichkeit empfiehlt es sich jedoch, 0 das bisschen Mehr an Schreibaufwand auf sich zu nehmen.

Das Basic des BASIC-Compilers kann nur mit ganzen Zahlen umgehen, daher sind auch die Variablen auf Integer-Typen beschränkt. Alle Variablen sind grundsätzlich vorzeichenlos. Die 8-Bit-Variablen (Byte-Variable) haben einen Wertebereich von 0-255, die 16-Bit-Variablen (Word-Variable) haben einen Wertebereich von 0 - 65535.

WICHTIG!!!

Achten Sie darauf, dass die letzte Speicherzelle einer Seite keine WORD-Variable beinhaltet, da sonst deren höherwertiges Byte auf der neuen Seite liegen würde, diese aber an dieser Stelle keinen freien Speicher zur Verfügung stellt. Achten Sie darauf, welche Speicherzellen vom Compiler fest belegt werden und welche man benützen darf.

Ab der Version vom 05.12.2009 überwacht der Compiler beim automatischen Zuweisen der Variablenadresse (AUTO) diese Bereichsgrenzen. Falls es eine solche Überschneidung gibt, meldet der Compiler einen Fehler und zeigt dabei an, um wieviel Bytes diese Grenze überschritten wurde. So kann man durch Verschieben der Variablendeklaration das Problem behoben werden. Auf eine automatische Umverteilung bzw. Neuverteilung der Variablendeklaration wurde bewusst verzichtet, um die äußerst effiziente Nutzung der direkten Variablenzugriffe nicht zu verhindern. Bei Arrays wird diese Überwachung ebenfalls durchgeführt.

Dies alles gilt auch für die 18er Bausteine. Obwohl dort der Speicher in einem Block mit fortlaufenden Adressen strukturiert ist, ist er letztendlich doch segmentiert. Diese Segmentierung kann nur bei der indirekten Adressierung umgangen werden. Da aber der Compiler den Code sehr stark optimiert, ist es möglich, dass nicht immer die indirekte Adressierung genutzt wird. Vorallem dann, wenn der Index keine Zahl sondern eine Konstante ist. Dann kann nämlich bereits der Compiler die absolute Adresse berechnen und dadurch den erzeugten Code sehr kompakt gestalten. Aber diese absolute Adresse umfasst lediglich 8 Bit. Die Auswahl des gewünschten Blocks erfolgt im BSR-Register.

Für die Anwendervariablen wird bei der 18er-Familie i.d.R. der RAM-Bereich von 100H bis ans Ende des implementierten Speichers, maximal EFFFH (beim 18F25J11 etc jedoch nur bis DFFFH), verwendet. Meist kann auch noch der Bereich von 80H bis FFH verwendet werden, da die Compilervariablen z.Z. noch im Bereich 00H bis 7FH Platz finden. Auch STACK und ARITH32 liegen hier. Sollte dieser Bereich nicht mehr ausreichen, werden zukünftige Erweiterungen der Compilervariablen in den Bereich 80H bis FFH belegen.

ARRAYs

Bei den Bausteinen PIC16C6x, PIC16C7x, PIC16C8x und PIC16F8x kann der Datenspeicher als Array oder eine Anzahl von Arrays definiert werden. Dazu wird der Variablen einfach ein Index angehängt. Dieser Index dient als Offset zur Berechnung der absoluten Speicheradresse. Dazu wird dieser Offset einfach zur Adresse der Bezugsvariablen addiert.

Bsp.

Variablen (cont.)

```
DEFINE FELD = $50 AS BYTE
```

```
...
```

```
...
```

```
LET FELD(2)=10          'Der Wert 10 kommt nach $52 (=$50+2)
```

Beim PIC 16C56 kann nur der Datenspeicher im Bereich 30H bis 3FH für Arrays benutzt werden. Beim 16C57 und 16C58 sind es die Speicherbereiche 30H - 3FH, 50H - 5FH und 70H - 7FH. Bei diesen Bausteinen ist es immer ein Feld mit 16 Elementen bzw. ein großes Feld mit 48 Elementen.

Es gilt, dass mit Feldvariablen keine Berechnungen und logischen Verknüpfungen durchgeführt werden können. Nur einfache Zuweisungen mittels LET sind erlaubt!

Eine Definition von Byte-Arrays ist auch mit der Option AUTO möglich. Dazu gibt man hinter AUTO die gewünschte Arraygröße an.

Beispiel:

```
DEFINE feld = AUTO(10)
```

oder

```
DEFINE feld = AUTO(10) AS BYTE
```

Auf diese Weise beginnt die nächste mit AUTO definierte Variable um diese Byteanzahl später.

INTERNE COMPILER VARIABLEN

Vom Compiler werden intern noch folgende 16-Bit-Variablen verwendet. Auch auf diese Variable können Sie natürlich zugreifen. Beispielsweise kann man so auf den Divisionsrest zugreifen (ARG1):

ARG0 o. ERG niederw. 16-Bit Ergebnis

ARG1 höherw. 16-Bit Ergebnis

ARG2 16-Bit Argument

ARG3 16-Bit Argument

ARG4 16-Bit Argument

ARG5 8-Bit Argument

ERR 8-Bit Fehlervariable

Die vordefinierten Variablen ERG, ARG1, ARG2, ARG3, ARG4 und ARG5 werden vom Compiler für Berechnungen und als Schleifenzähler verwendet. Wo alle internen Variablen im Speicher des PICs zu liegen kommen, hängt vom jeweiligen Baustein ab und ist in der Tabelle für Variablenadressen in der Spalte "Compiler intern" definiert.

Die ERR-Variable

Eine ERR-Variable gibt Auskunft über interne Vorgänge. Sie ist bitorientiert mit folgenden Funktionen:

Bit 7 Überlauf bei Arithmetikroutinen (wird vor jeder Operation gelöscht)

Bit 6 werden vom Compiler zum Merken der Rückkehradresse aus der RUN-

Bit 5 TIME-Bibliothek beim 16C54 und 16C55 verwendet, da diese nur einen

Bit 4 zweistufigen Stack besitzen, der hier nicht ausreichend ist.

Bit 3 Timeout in SERIN oder I2C aufgetreten.

Bit 2 Zählt die Schachteltiefe der GOSUB-Routinen, da diese Überprüfung zum

Bit 1 Zeitpunkt der Compilierung nicht durchgeführt werden kann.

Bit 0 000 = kein UP, 001 = 1 UP, 101 = 5 UP = nicht erlaubt.

32-Bit Arithmetik

Variablen (cont.)

Bei der 32-Bit-Arithmetik (nur PRO-Version) werden zusätzliche "interne" Compilervariablen benötigt, die mit DEFINE ARITH32 festgelegt werden.

Die PORT- und TRIS-Register

Beim 10F2xx, 12C5xx und 16C5x müssen außerdem die TRIS-Informationen im Arbeitsspeicher bereitgehalten werden, da hier die TRIS-Register, im Gegensatz zu dem 16C6x, 16C7x und 16X8x nicht ausgelesen werden können.

RATRIS
RBTRIS
RCTRIS

Dies alles zeigt, warum bei den Bausteinen 10F2xx, 12C5xx und 16C5x nur acht 8-Bit- bzw. vier 16-Bit Variablen zur Verfügung stehen.

Die Variablen RA, RB, RC, RD, RE bezeichnen die entsprechenden Ports. Neben dem Port muss bei vielen Befehlen auch der entsprechende Pin angegeben werden. Dies geschieht entweder durch ein Komma oder einen Punkt. Dabei hat die Angabe nach dem Komma den Vorteil, dass es sich dann auch um eine Variable handeln darf.

z.B.

LET	a1 = 7	
HIGH	RA,a1	REM setzt das höchstwertige Bit auf 1 (Bit 7 = 1)
LOW	RB.3	

Bei einer Wertzuweisung an eine Variable muss auf den zulässigen Wertebereich geachtet werden. Der Compiler nimmt keine Überprüfung vor. Er kürzt das Ergebnis entsprechend des Variablentyps.

z.B.

LET	a1 = 266	REM Zuweisung an eine 8-Bit Variable
PRINT	"Variable a1 = ",a1	REM Ausgabe auf Anzeige

Der Wert von Variable a1 -> 10 (a1 = 266 - 256 = 10)

Die Variablendefinitionen der DEFAULT.EQU Datei finden Sie im Anhang I

Die Bausteine mit mehr Speicherplatz bieten eine elegante Lösung des Reentrance-Problems beim Auftreten eines Interrupts. Während bei den "kleinen" Bausteinen nur das W- und Statusregister gerettet werden, können hier alle Variablen die die Laufzeitbibliothek verwendet zwischengespeichert werden (Ersatz für PUSH und POP). Dazu wird ein 16 Byte großer Speicherbereich auf der 1 RAM Seite (Adresse <80H) mittels DEFINE-Anweisung festgelegt.

Auf den nachfolgenden Seiten finden Sie die Speicherbereiche, die für die Anwendervariablen genutzt werden können. Bei einigen PIC-Typen sind sehr wenig Variablen vorhanden. In solchen Fällen können Sie z.B. zwei weitere Speicherplätze nutzen, wenn Sie die Befehle DATA, READDATA und RESTORE nicht verwenden, da diese Befehle einen 16-Bit Zeiger benötigen. An welcher Stelle dieser Zeiger liegt, entnehmen Sie den folgenden Tabellen bzw. DEFAULT.EQU

Kompatibilität der Variablengrößen

Variablen (cont.)

Wenn Variable_c und Variable_a verschiedene Typen sind, sind Zuweisungen zulässig:

```
LET Variable_C = Variable_A
```

Um dabei Fehler zu vermeiden, muss Variable_A vom Typ mindestens die gleiche Größe haben wie Variable_C, oder aber sichergestellt sein, dass der Zahlenwert von Variable_A klein genug ist, dass er in die Variable_C passt. Andernfalls werden die höheren Bits unterschlagen und gleichzeitig in der Error-Variablen signalisiert, (falls diese Option nicht durch Compilerschalter \$CCOFF abgewählt wurde).

Gleiches gilt bei allen Funktionen bezüglich ihrer Ergebnisse, etwa

```
LET Variable_C = Summand_a + Summand_b
```

Bei Multiplikation ist zu beachten, dass das Ergebnis normalerweise die nächstgrößere Variablengröße erfordert gegenüber der Größe der Faktoren.

In die Argumente (Eingangswerte) einer Funktion kann man verschiedene Variablentypen beliebig gemischt schreiben.

Zu beachten ist nur, dass dann automatisch die rechenzeitaufwändigere Variante der Funktion durchgeführt wird, die dem größeren Typ der beiden Eingangsvariablen entspricht.

Die Variablen werden nach folgendem Schema im Speicher abgelegt:

```
DEFINE var8 = $40 as byte => in 40H liegt der Inhalt der Variablen var8
```

```
DEFINE var16=$40 as word => in 40H liegt das Low-Byte und in 41H das High-Byte der 16-Bit-Variablen
```

```
DEFINE var32=$40 as dblword => in 40H liegt das Low-Byte, in 41H das nächste, in 42H das übernächste und in 43H das höchstwertige Byte der Doppelwort-Variablen.
```


Rechenzeit optimiert programmieren

Rechenzeitoptimiert programmieren

1.

Für PICs mit mehreren Arbeitsspeicher-Seiten (Banks) gilt: Variablen, die häufig gebraucht werden, werden vom Compiler am schnellsten verwaltet, wenn sie in Bank 0 liegen. Bank 1 und 2 sind i.d.R. schneller als Bank 3.

2.

Programmabschnitte, z.B. oft verwendete Routinen, werden am schnellsten bearbeitet, wenn sie in Seite 0 stehen (dort stehen auch die compilerinternen Funktionsroutinen). Aus diesem Grund sollten auch die Unterprogramme dort liegen.

3.

Arbeiten Sie wenn irgendwie möglich ausschließlich mit Byte-Variablen. Hier wird bei den logischen und arithmetischen Befehlen i.d.R. ein sehr kompakter Code generiert. Durch geschicktes Aufteilen der Rechenschritten können 16-Bit Operationen meistens umgangen werden.

4.

Verwenden Sie auch SET, RES und TOGGLE um einzelne Bits zu manipulieren. Der PIC kennt solche kompakten Bitbefehle und der Compiler generiert diese dann auch.

5.

Prüfen Sie bei IF-Abfragen auf 0 (= 0) bzw. ungleich 0 (<> 0). Bei der Verwendung von Byte-Variablen wird dann ein kompakterer Code generiert, als bei den Abfragen <, >, <= oder >=. Auch 16-Bit bzw. 32-Bit Vergleiche benötigen die Subtraktionsroutinen in der Laufzeitbibliothek. Dadurch wird viel Code und Zeit benötigt.

6.

Wenn Sie bei Byte Addition bzw. Subtraktion keine Bereichsprüfung benötigen, schalten Sie mit \$CCOFF die Codegenerierung entsprechend ein.

7.

Texte im LCDWRITE sollten vermieden werden. Legen Sie stattdessen die Texte in DATA-Feldern ab und lesen diese Zeichen für Zeichen aus. Eine fortlaufende Beschreibung der LCD ist durch das Setzen der Cursorposition auf 0,0 möglich.

8.

Nutzen Sie die Befehle INC var und DEC var statt LET var = var +1 bzw. LET var = var -1. Allerdings erfolgt dann keine Bereichsprüfung (ERR-Variable).

Tabelle für Variablenadressen im PIC

Diese Tabelle diene letztendlich nur dazu dem Benutzer zu zeigen, welche Speicherstelle für seine Variablen zur Verfügung stehen. Bei der automatischen Zuweisung der Adressen übernimmt der Compiler diese Arbeit. Dennoch ist es in manchen Fällen immer noch interessant, die Variablen absolut zu zuweisen. Da bei neueren PIC der Speicher sehr sehr groß werden kann, wird diese Tabelle nicht mehr fortgeführt. In Zweifelsfällen muss auf das Datenblatt und die Datei DEFAULT.EQU zurückgegriffen werden.

Diese Datei zeigt die Verwendung des Datenspeichers der einzelnen PICs sowie die neue Variablenverwaltung des Compilers iL_BAS16STD und iL_BAS16PRO).

Die in der Spalte "ANWENDER" aufgelisteten Adressen sind für den freien Variablengebrauch bestimmt.

(Arrays siehe unten)

Für A- oder B-Type gilt i.d.R. die gleiche Speichereinteilung. Im Zweifelsfall bitte im Microchip-Datenblatt die beiden Versionen vergleichen!

PIC	ROM	PIC	Compiler	ANWENDER	ANWENDER	Anz.	Bemerkung
	intern	intern	(Hexwert)	(Dezimal)	Byte		

iL_TROLL	1792	000-01F	06D-07F	> 020-06C	32-108	< = 76	
		080-09F	0ED-0FF	> 0A0-0EC	160-236	< = 76	

10F200	256	000-007	013-01F	> 010-012	16- 18	< = 3	
10F202	512	000-007	008-016	> 017-01F	23- 31	< = 8	
10F204	256	000-007	013-01F	> 010-012	16- 18	< = 3	
10F206	512	000-007	008-016	> 017-01F	23- 31	< = 8	

12C508	512	000-006	008-016	> 017-01F	23- 31	< = 8	
12E508	512	000-006	008-016	> 017-01F	23- 31	< = 8	
12F508	512	000-006	008-016	> 017-01F	23- 31	< = 8	
12C509	1024	000-006	008-016	> 017-01F	23- 31	< = 8	030-03F für Arrays
12E509	1024	000-006	008-016	> 017-01F	23- 31	< = 8	030-03F für Arrays
12F509	1024	000-006	008-016	> 017-01F	23- 31	< = 8	030-03F für Arrays

12F629	1024	000-01F	04E-05F	> 020-04D	32- 77	< = 45	
		080-09F					

12C671	1024	000-01F	020-022	> 023-06F	35-111	< = 76	
		080-09F	070-07F	> 0A2-0BF	162-191	< = 29	
		0A0-0A1					

12C672	2048	000-01F	020-022	> 023-06F	35-111	< = 76	
		080-09F	070-07F	> 0A2-0BF	162-191	< = 29	
		0A0-0A1					

12E673	1024	000-01F	020-022	> 023-06F	35-111	< = 76	
		080-09F	070-07F	> 0A2-0BF	162-191	< = 29	
		0A0-0A1					

12E674	2048	000-01F	020-022	> 023-06F	35-111	< = 76	
		080-09F	070-07F	> 0A2-0BF	162-191	< = 29	
		0A0-0A1					

12F675	1024	000-01F	04E-05F	> 020-04D	32- 77	< = 45	
		080-09F					

16C505	1024	000-007	008-017	> 018-01F	24- 31	< = 8	030-03F für Arrays
--------	------	---------	---------	-----------	--------	-------	--------------------

Tabelle für Variablenadressen im PIC (cont.)

050-05F für Arrays

070-07F für Arrays

16C53	384	000-007	008-017	>	018-01F	24- 31	< = 8
16C54	512	000-007	008-017	>	018-01F	24- 31	< = 8
16F54	512	000-007	008-017	>	018-01F	24- 31	< = 8
16C55	512	000-007	008-017	>	018-01F	24- 31	< = 8
16C56	1024	000-007	008-017	>	018-01F	24- 31	< = 8
16C57	2048	000-007	008-017	>	018-01F	24- 31	< = 8
							030-03F für Arrays
							050-05F für Arrays
							070-07F für Arrays
16F57	2048	000-007	008-017	>	018-01F	24- 31	< = 8
							030-03F für Arrays
							050-05F für Arrays
							070-07F für Arrays
16C58	2048	000-007	008-017	>	018-01F	24- 31	< = 8
							030-03F für Arrays
							050-05F für Arrays
							070-07F für Arrays
16F59	2048	000-007	008-017	>	018-01F	24- 31	< = 8
							030-03F für Arrays
							050-05F für Arrays
							070-07F für Arrays
16C554	512	000-01F	020-031	>	032-06F	50-111	< = 61
							080-09F
16C556	1024	000-01F	020-031	>	032-06F	50-111	< = 61
							080-09F
16C558	2048	000-01F	020-031	>	032-07F	50-127	< = 77
							080-09F 0A0-0A0 > 0A1-0BF 161-191 < = 31
16C61	1024	000-00B	00C-01E	>	01F-02F	31- 47	< = 17
16C62	2048	000-01F	020-032	>	033-07F	51-127	< = 76
							080-09F 0A0-0A0 > 0A1-0BF 161-191 < = 31
16C620	512	000-01F	020-032	>	033-06F	51-111	< = 60
							080-09F
16C621	1024	000-01F	020-032	>	033-06F	51-111	< = 60
							080-09F
16C622	2048	000-01F	020-032	>	033-07F	51-127	< = 76
							080-09F 0A0-0A0 > 0A1-0BF 161-191 < = 31
16E623	512	000-01F	020-022	>	023-06F	35-111	< = 76
							080-09F 070-07F
16E624	1024	000-01F	020-022	>	023-06F	35-111	< = 76
							080-09F 070-07F
16E625	2048	000-01F	020-022	>	023-06F	35-111	< = 76
							080-09F 070-07F > 0A0-0BF 161-191 < = 32
16F627	1024	000-01F	06D-07F	>	020-06C	32-108	< = 76
							080-09F 0ED-0FF > 0A0-0EC 160-236 < = 76
							100-11F 170-17F > 120-14F 288-335 < = 47
							180-1FF
16F628	2048	000-01F	06D-07F	>	020-06C	32-108	< = 76

Tabelle für Variablenadressen im PIC (cont.)

080-09F 0ED-0FF > **0A0-0EC** 160-236 < = 76
 100-11F 170-17F > **120-14F** 288-335 < = 47
 180-1FF

16C63 4096 000-01F 04E-05F > **020-04D** 32- 77 < = 45
 080-09F 0A0-0A0 > **0A1-0FF** 161-255 < = 94

16C630 1024 000-01F 020-032 > **033-05F** 51-127 < = 76

16C64 2048 000-017 020-032 > **033-07F** 51-127 < = 76
 080-094 0A0-0BF > **0A0-0BF** 161-191 < = 32

16C65 4096 000-01D 020-032 > **033-07F** 51-127 < = 76
 080-09D 0A0-0FF > **0A0-0FF** 161-255 < = 94

16C66 8192 000-01F 020-032 > **033-07F** 51-127 < = 76
 080-09F 0A0-0A0 > **0A1-0FF** 161-255 < = 94
 100-11F 120-120
 180-19F 1A0-1A0

16C67 8192 000-01F 020-032 > **033-07F** 51-127 < = 76
 080-09F 0A0-0A0 > **0A1-0FF** 161-255 < = 94
 100-11F 120-120
 180-19F 1A0-1A0

16C71 1024 000-00B 00C-01E > **01F-02F** 31- 47 < = 17
 080-09F

16C72 2048 000-01F 020-031 > **032-07F** 50-127 < = 77
 080-09F 0A0-0A0 > **0A1-0BF** 161-191 < = 31

16C73 4096 000-01F 020-031 > **032-07F** 50-127 < = 77
 080-09F 0A0-0A0 > **0A1-0FF** 161-255 < = 94

16C74 4096 000-01F 020-031 > **032-07F** 50-127 < = 77
 080-09F 0A0-0A0 > **0A1-0FF** 161-255 < = 94

16C710 512 000-00B 00C-01E > **01F-02F** 31- 47 < = 17
 080-08B

16C711 1024 000-00B 00C-01E > **01F-04F** 31- 79 < = 49
 080-08B

16C715 2048 000-01F 06D-07F > **020-06C** 32-108 < = 76
 080-09F

16F737 2048 000-017 06D-0FF > **020-06C** 32-108 < = 76
 080-094 0A0-0BF > **0A0-0EC** 161-236 < = 76
 100-11F 16D-17F > **120-16C** 288-364 < = 76
 180-19F 1ED-1FF > **1A0-1EC** 416-492 < = 76

16F747 2048 000-017 06D-0FF > **020-06C** 32-108 < = 76
 080-094 0A0-0BF > **0A0-0EC** 161-236 < = 76
 100-11F 16D-17F > **120-16C** 288-364 < = 76
 180-19F 1ED-1FF > **1A0-1EC** 416-492 < = 76

16F767 2048 000-017 06D-0FF > **020-06C** 32-108 < = 76
 080-094 0A0-0BF > **0A0-0EC** 161-236 < = 76

Tabelle für Variablenadressen im PIC (cont.)

100-11F 16D-17F > **120-16C** 288-364 < = 76
 180-19F 1ED-1FF > **1A0-1EC** 416-492 < = 76

16F777 2048 000-017 06D-0FF > **020-06C** 32-108 < = 76
 080-094 0A0-0BF > **0A0-0EC** 161-236 < = 76
 100-11F 16D-17F > **120-16C** 288-364 < = 76
 180-19F 1ED-1FF > **1A0-1EC** 416-492 < = 76

16C83 512 000-00B 00C-01E > **01F-02F** 31- 47 < = 17
 080-08B

16C84 1024 000-00B 00C-01E > **01F-02F** 31- 47 < = 17
 080-08B

16F83 512 000-00B 00C-01E > **01F-02F** 31- 47 < = 17
 080-08B

16F84 1024 000-00B 00C-01E > **01F-04F** 31- 79 < = 49
 080-09B

16F818 1024 000-01F 06D-07F > **020-06C** 32-108 < = 76
 080-09F > **0A0-0BF** 160-191 < = 32
 100-10F
 180-18F

16F819 1024 000-01F 06D-07F > **020-06C** 32-108 < = 76
 080-09F 0ED-0FF > **0A0-0EC** 160-236 < = 76
 100-10F 16D-17F > **120-16C** 288-364 < = 76
 180-18F 1ED-1FF

16F870 2048 000-01F 06D-07F > **020-06C** 32-108 < = 76
 080-09F
 100-11F
 180-19F

16F871 4096 000-01F 06D-07F > **020-06C** 32-108 < = 76
 080-09F
 100-11F
 180-19F

16F872 4096 000-01F 06D-07F > **020-06C** 32-108 < = 76
 080-09F
 100-11F
 180-19F

16F873 4096 000-01F 06D-07F > **020-06C** 32-108 < = 76
 080-09F 0ED-0FF > **0A0-0EC** 160-236 < = 76
 100-11F
 180-19F

16F874 4096 000-01F 06D-07F > **020-06C** 32-108 < = 76
 080-09F 0ED-0FF > **0A0-0EC** 160-236 < = 76
 100-11F
 180-19F

16F876 8192 000-01F 06D-07F > **020-06C** 32-108 < = 76
 080-09F 0ED-0FF > **0A0-0EC** 160-236 < = 76
 100-11F 16D-17F > **120-16C** 288-364 < = 76
 180-19F 1ED-1FF > **1A0-1EC** 416-492 < = 76

Tabelle für Variablenadressen im PIC (cont.)

16F877 8192 000-01F 06D-07F >**020-06C** **32-108**< = 76
080-09F 0ED-0FF > **0A0-0EC** **160-236**< = 76
100-11F 16D-17F > **120-16C** **288-364**< = 76
180-19F 1ED-1FF > **1A0-1EC** **416-492**< = 76

WICHTIG!!!

Achten Sie darauf, dass die letzte Speicherzelle einer Seite keine WORD-oder Dblword-Variable beinhaltet, da sonst deren höherwertiges Byte auf der neuen Seite liegen würde, diese aber an dieser Stelle keinen freien Speicher zur Verfügung stellt.

ARRAYs

Bei den Bausteinen PIC16C6x, PIC16C7x, PIC16C8x und PIC16F8x kann der Datenspeicher als Array oder eine Anzahl von Arrays definiert werden. Dazu wird der Variablen einfach ein Index angehängt. Dieser Index dient als Offset zur Berechnung der absoluten Speicheradresse. Dazu wird dieser Offset einfach zur Adresse der Bezugsvariablen addiert.

Einschränkungen bei Verwendung dieser Arrays sind entsprechend dem der Arrays für den PIC16C57.

Bsp.

```
DEFINE FELD = $50 AS BYTE
...
...
LET FELD(2)=10 'Der Wert 10 kommt nach $52 (=$50+2)
```

Die Bausteine mit mehr Speicherplatz bieten eine elegante Lösung des Reentrance-Problems beim Auftreten eines Interrupts. Während bei den "kleinen" Bausteinen nur das W- und Statusregister gerettet werden, können hier alle Variablen, die die Laufzeitbibliothek verwendet, zwischengespeichert werden (Ersatz für PUSH und POP). Dazu wird ein 16 Byte großer Speicherbereich auf der 1 RAM Seite (Adresse <80H) mittels DEFINE-Anweisung festgelegt.

Die Zuordnung des Variablennames und der absoluten Adresse des Fileregisters ist in der Datei DEFAULT.EQU definiert. Diese Datei wird sowohl vom Compiler und Assembler als auch vom Simulator genutzt. Ein Verändern der Einträge kann äußerst unangenehme Folgen haben.

IF-Konstrukte

Für Vergleiche von Variablen und/oder Konstanten wird die IF-THEN-ELSE Anweisung verwendet.

```
IF var_a = 1 THEN LET var_b = var_a * 2 ELSE GOTO WEITER
```

IF-Konstrukte dürfen nicht verschachtelt werden. Verboten ist:

(falsch) IF var_a = 1 THEN IF...

Auch können in IF-Anweisungen keine logischen Verknüpfungen verwendet werden. Verboten ist:

(falsch) IF a1 and 1 = 1 THEN ...

Da im obigen Fall nur ein Bit überprüft wird, sollte man hier besser die sehr effiziente Formulierung

```
IF a1,0 = 1 THEN ...
```

verwenden. Ansonsten muss man die logische Verknüpfung in eine eigene Zeile schreiben.

siehe auch IF-THEN-ELSE

Wichtig!

Bei Bitabfragen kann und darf auch deshalb nur auf 0 oder 1 (immer als Konstante) abgefragt werden.

Komparator

Manche PIC-Bausteine besitzen am Port RA Komparatoreingänge die unterschiedlich konfiguriert werden können. Die Konfiguration erfolgt wie beim AD-Wandler in der DEFINE DEVICE Zeile. Die Schlüsselworte dafür lauten: CMCFGx und VRCFGx.

Mit diesen Schaltern werden die Komparatoren bzw. die Spannungsreferenzen eingestellt. Für den 16F628 ergeben sich folgende Zusammenhänge (*):

CMCFG0 Komparator Reset

CMCFG1 3 Inp mux	C1OUT = RA2-RA0 (od. RA2-RA3) C2OUT = RA2-RA1
CMCFG2 4 Inp mux	C1OUT = Vref-RA0 (Vref-RA3) C2OUT = Vref-RA1 (Vref-RA2)
CMCFG3 2 gem.Vref	C1OUT = RA2-RA0 C2OUT = RA2-RA1 RA3=digital IO
CMCFG4 2 Komp.	C1OUT = RA3-RA0 C2OUT = RA2-RA1
CMCFG5 1 Komp.	C2OUT = RA2-RA1 RA0 und RA3 sind digital IO
CMCFG6 2 gem.Vref	C1OUT(RA3)=RA2-RA0 C2OUT(RA4)=RA2-RA1
CMCFG7 aus	RA0..RA3 sind digital IO

Bei CMCFG1 und CMCFG2 entscheidet das CIS-Bit im CMCON-Register ob RA0 oder RA3 bzw. RA1 oder RA2 auf den Komparator geführt wird. Dieses Bit kann nicht in der DEFINE-Anweisung gesetzt werden, sondern muss während des Programmlaufes mittels SET- bzw. RES-Befehl nach Bedarf geändert werden.

Beim VRCFG muss das Konfigurationsbyte direkt, d.h. ohne Leerzeichen, dem Schlüsselwort VRCFG folgen. Dazu muss aus der Tabelle der entsprechende Wert berechnet werden. Die Angabe erfolgt als Dezimalwert. Die entsprechende Zuordnung lautet:

Bit 7	Wert = 0 -> Vref aus Wert = 128 -> Vref ein
Bit 6	Wert = 0 -> Vref keine Verbindung zu RA2 Wert = 64 -> Vref mit RA2 verbunden
Bit 5	Wert = 0 -> Vref großer Spannungsbereich Wert = 32 -> Vref kleiner Spannungsbereich
Bit 4-	
Bit 3	bit Bit 0 Spannungsbereich errechnet sich wie folgt: wenn Bit 5 = 1 -> $V_{ref} = (x/24) * V_{dd}$ mit $x = 0$ bis 15 wenn Bit 5 = 0 -> $V_{ref} = 0.25 * V_{dd} + (x/32) * V_{dd}$

Hinweis:

Andere PIC-Controller haben eine andere Zuordnungstabelle. Bitte lesen Sie dazu das entsprechende Kapitel im jeweiligen Datenblatt.

Falls in einem Baustein sowohl Analogwandler als auch Komparatoren implementiert sind, müssen diese u.U. explizit abgeschaltet werden, damit der Pin ordnungsgemäß arbeitet (im Zweifelsfall: Datenblatt).

Programm-Pages

iL_PAGE0 ist ein Programmmodul, das nur beim iL_BAS16PRO und iL_BAS16SEP mitgeliefert wird. Ebenso kann es aus den übrigen iL_BAS16-Compilerversionen nicht aktiviert werden. Es dient dazu, bei PIC-Bausteinen mit Programmspeichergrößen von mehr als einer Seite die Verzweigungen (Sprung und Unterprogrammaufrufe) herauszusuchen, deren Zieladresse in einer anderen Seite liegt. Die PIC-Struktur verlangt in solchen Fällen, dass vor Ausführung des Sprunges das PCLATH-Register bzw. die Page-Preselect-Bits entsprechend der Seitenadressen gesetzt werden. Diese in Assemblerprogrammen mühselige und fehlerträchtige Arbeit übernimmt dieses Programmmodul.

Die Arbeitsweise ist mit der eines Assemblers vergleichbar. In einem iterativen Algorithmus werden diese Verzeigungen gesucht und durch Sonderbefehle ersetzt. Diese Sonderbefehle zwingen den Assembler dazu, neben dem eigentlichen Sprungbefehl automatisch die zusätzlich notwendigen Befehle für die Seitenumschaltung einzufügen.

Die Möglichkeiten von iL_PAGE0 sind allerdings auch begrenzt. Man kann natürlich mit \$LRANGE 2048 iL_PAGE0 anweisen, alle im Programm vorkommenden CALLs und GOTOs in deren Weitbereichsäquivalent umzusetzen, was aber eine enorme Verschwendung der Speicherressourcen wäre. Wie man als BASIC-Programmierer mit einwenig Programmierdisziplin dem Modul iL_PAGE0 "entgegenkommt", wird in einem Beispiel unter \$LRANGE beschrieben.

Mathematische Operatoren (8-/16-Bit)

Als mathematische Operatoren unterstützt der Compiler die vier Grundrechenarten.

Mögliche Operatoren sind:

Rechenfunktionen:

Addition + $a + b$
Subtraktion - $a - b$
Multiplikation * $a * b$
Division / a / b
Modulo mod $a \bmod b$ (modulo, Divisionsrest)

Schiebe- und Rotationsbefehle:

shl var shl 11110011 ? 11100110
Inhalt von var nach links schieben (mit 0 füllen), was herausfällt kommt ins Carry
shr var shr 11110011 ? 01111001
Inhalt von var nach rechts schieben (mit 0 füllen), was herausfällt kommt ins Carry

rotr var rotr 11110011 ? 11100111
Inhalt von var nach links rotieren, Bit 7 ins Carry
rotr var rotr 11110011 ? 11111001
Inhalt von var nach rechts rotieren, Bit 0 ins Carry

rotl8 var rotl 11110011 ? 11100111
Inhalt von var nach links rotieren, Bit 7 nach Bit 0
rotr8 var rotr 11110011 ? 11111001
Inhalt von var nach rechts rotieren, Bit 0 nach Bit 7
(Zahlen sind im Binärformat)

Der Unterschied zwischen ROTL und ROTL8 bzw. ROTR und ROTR8:

ROTR8 bzw. ROTL8 rotieren das Bitmuster in der 8-Bit-Variable in sich, d.h. das Carrybit wird vor dem eigentlichen Rotationsbefehl entsprechend gesetzt bzw. zurückgesetzt. Es kommt somit kein neues Bit hinzu oder wird gelöscht. Nach 8-maligem rotieren steht der ursprüngliche Wert wieder in der Variablen.

Nicht so bei ROTL bzw. ROTR. Hier wird der gerade im Carrybit stehende Wert in die Variable übernommen. Man muss also sehr genau darauf achten, was im Carry stehen kann. Damit wieder der ursprüngliche Wert in der Variablen steht, muss man diese Befehle 9 mal direkt hintereinander aufrufen. Dieser Befehl sollten nur ganz erfahrene Programmierer verwenden.

ACHTUNG!

Werden 16- oder 32-Bit-Variablen verwendet, so wird das Carryflag immer vorher entsprechend gesetzt. Somit sind hier 16 bzw. 32 Rotationen notwendig, damit der ursprüngliche Wert wieder in der Variablen steht.

Vergleichsoperatoren:

Gleichheit $a = b$
ungleich $a <> b$
kleiner als $a < b$
größer als $a > b$
kleiner oder gleich $a \leq b$
größer oder gleich $a \geq b$

Berechnung für Multiplikation, Division und Modulo erfolgt immer mit 16 Bit-Genauigkeit. Das Ergebnis wird entsprechend dem Zielvariablentyp übernommen. Bei Addition und Subtraktion wird ein optimierter Code generiert.

Mathematische Operatoren (8-/16-Bit) (cont.)

z.B.

var_a = 200	REM 8-Bit Variable
var_b = 200	REM 8-Bit Variable
var_c = 0	REM 8-Bit Variable
var_u = 0	REM 16-Bit Variable
var_c = var_a * var_b	REM Ergebnis auch 8-Bit (u = 64)
var_u = var_a * var_b	REM Ergebnis 16-Bit (u = 40000)

Der Compiler rundet alle internen Zwischenergebnisse auf 16-Bit (bei 8-Bit Multiplikation und -Division).

z.B.

var_a = 200	REM 8-Bit Variable
var_b = 200	REM 8-Bit Variable
var_u = 0	REM 8-Bit Variable
var_u = var_a * var_b / var_b	REM Ergebnis (u = 200) Richtig
var_s = var_a * var_b * var_a / var_b / var_b	REM Ergebnis (u = 0) Falsch

32-Bit Operationen werden unabhängig von der 8- und 16-Bit Arithmetik ausgeführt. Es sind auch unabhängige Routinen in der Laufzeitbibliothek dafür zuständig.

Bei der Formulierung mathematischer Ausdrücke sind die Besonderheiten des iL_BAS16-Compilers zu beachten:
Der Compiler verarbeitet, aufgrund des sehr kleinen Speichers des Microcontrollers, mathematische Ausdrücke streng von links nach rechts, ohne Rücksicht auf übliche Konventionen, wie Punkt- vor Strichrechnung.

z.B. var_a = 3 + 4 * 5
Das Ergebnis ist 3 + 4 * 5 = 35 anstatt 3 + (4 * 5) = 23

Logische Operatoren (8-/16-Bit)

Als logische Operatoren unterstützt der Compiler folgende Funktionen:

logisch	UND	and	a and b
logisch	ODER	or	a or b
logisch	EXOR	xor	a xor b

z.B.

LET var_a = 5 and 3	REM 101 and 11 = 001 = 1
LET var_a = 5 or %1001	REM 101 or 1001 = 1101 = 13
LET var_a = 5 xor \$A	REM 101 xor 1111 = 1010 = 10

LET var_a XOR var_b

Wahrheitstabelle:

	AND	OR	XOR
0			
0			
1			
1			

Auch bei der Kombination logischer Operatoren ist auf die Verarbeitungsreihenfolge des Compilers zu achten.

32-Bit Arithmetik Einführung

(nur Professionalversion)

Die 32-Bit Arithmetik der Professionalversion bietet das Arbeiten mit Zahlen von 0 bis 4294967296, also Zahlen bis ca. 4,3 Mrd. Es sind vorzeichenlose Zahlen und belegen im Speicher 4 Bytes. Um sie zu definieren verwendet man das Schlüsselwort DBLWORD. Da die 32-Bit Arithmetik sehr speicherintensiv ist, wurde ein Weg gewählt, um die schon z.T. sehr geringen Speicherressourcen so gut wie möglich zu schonen. Die 8- bzw. 16-Bit Arithmetik verwendet zum Rechnen einen Speicherbereich, der ausschließlich dem Compiler zur Verfügung steht. Diese Variablen sind als ARG, ARG1, ARG2, ARG3, ARG4 und ARG5 vordefiniert und dürfen nicht an einen anderen Speicherort verschoben werden. Die 32-Bit Arithmetik benötigt in der Runtime Library zusätzlichen Speicher für die Rechenroutinen. Dieser Speicher muss in BANK 0 liegen. Die Startadresse dieses Speicherbereiches wird mit der DEFINE ARITH32 Anweisung festgelegt. Der Bereich umfasst 16 Bytes, die an einem zusammenhängenden Stück vorliegen müssen. Somit ist äußerste Sorgfalt in der Nähe der Bankgrenzen angesagt. Es greifen jedoch nur die 32-Bit Arithmetikroutinen auf diesen Speicherbereich zu. Somit ist es möglich, diesen Speicherbereich auch anderweitig zu benutzen. Diese Doppelnutzung birgt aber große Gefahren, so dass sich nur "Profis" daran wagen sollten.

Um eine DBLWORD-Variable auf 2 Wort- bzw. 4 Bytevariablen aufzuteilen werden einfach zusätzliche Variablen definiert, die auf den entsprechenden Speicherplätzen der DBLWORD-Variablen liegen.

Nachfolgendes Beispiel zeigt einen Mischbetrieb aus Byte-, Word- und Dblword-Variablen.

```
define device 16f877
define a1=$20 as word
define a2=$22 as dblword
define a2l=$22 as word      'Zugriff auf das untere Wort von a2
define a2h=$24 as word      'Zugriff auf das obere Wort von a2
define a2ll=$22 as byte     'Zugriff auf das unterste Byte (Byte 0) von a2
define a2lh=$23 as byte     'Zugriff auf Byte 1 von a2
define a2hl=$24 as byte     'Zugriff auf Byte 2 von a2
define a2hh=$25 as byte     'Zugriff auf oberstes Byte (3) von a2
define a3=$120              'Variablen in BANK 2
define a4=$1A0              'Variablen in BANK 3
define xx=$30
define arith32=$40          'definiert die Rechenreg.
```

start:

```
let a2=1234567
let a2 = a2 * 3
let a2 = a2 and $00ffff00
inc a2l          'erhöht das untere Wort von a2, kein Übertrag
let xx=a4 'Variable aus Bank 4 nach Bank 1
high rb,1
wait 2000
low rb,1
wait 2000
goto start
```

Mathematische Operatoren (32-Bit)

Als mathematische Operatoren unterstützt der Compiler die vier Grundrechenarten.

Mögliche Operatoren sind:

Addition + $a + b$

Subtraktion - $a - b$

Multiplikation * $a * b$

Division / a / b

Vergleichsoperatoren:

Gleichheit $a = 5$

ungleich $<>$ $a <> b$

kleiner als $a < b$

größer als $a > b$

kleiner oder gleich $<=$ $a <= b$

größer oder gleich $>=$ $a >= b$

32-Bit Operationen werden unabhängig von der 8- und 16-Bit Arithmetik ausgeführt. Es sind dafür auch unabhängige Routinen in der Laufzeitbibliothek zuständig.

Logische Operatoren (32-Bit)

Als logische Operatoren unterstützt der Compiler folgende Funktionen:

logisch UND	and	a and b
logisch ODER	or	a or b
logisch EXOR	xor	a xor b

z.B.

LET var_a = 5 and 3	REM 101 and 11 = 001 = 1
LET var_a = 5 or %1001	REM 101 or 1001 = 1101 = 13
LET var_a = 5 xor \$A	REM 101 xor 1111 = 1010 = 10
LET var_a XOR var_b	

Wahrheitstabelle:

	AND	OR	XOR
0			
0			
1			
1			

ADDELAY

Syntax: ADDELAY const

Funktion: Fügt eine Zeitverzögerung zwischen Einschalten des AD-Wandlers bzw. dem Kanalwechsel und dem eigentlichen Wandlungsbeginn.

Es bedeuten:

const (1 ... 255) Zeitverzögerung mit $t = (16/fq) * const$

Beschreibung: Bei den Bausteinen mit 10-Bit ADWandlern z.B. 16F87x wurde der Haltekondensator des AD-Wandlers auf 120pF erhöht. Damit kann es passieren, dass nach dem Umschalten auf einen anderen Kanal die Zeit zum Laden bzw. Entladen des Kondensators auf den neuen Spannungswert zu kurz ist. Das zeigt sich z.B. dann, wenn der eine Kanal Spannungen gegen Null, der andere Spannungen gegen 5V messen soll. In diesem Fall kann es vorkommen, dass diese Werte überhaupt nicht stimmen, ja sich die einzelnen Spannungen quasi gegenseitig beeinflussen. Hier greift nun ADDELAY ein. Damit wird eine bestimmte Zeitverzögerung zwischen Auswahl des neuen Kanals und dem Beginn der Wandlung definiert. Dadurch bekommt der Haltekondensator genügend Zeit, sich auf die neue Spannung auf- bzw. abzuladen.

ADINP

Syntax: ADINP *adr, var*

Funktion: Wandelt einen analogen Pegel am Analogeingang in einen digitalen Wert um.

Es bedeutet:

adr (0 ... 3) gibt I/O Pin des 16C71 an, darf Variable oder Konstante sein,
var (8-Bit) speichert das Ergebnis der Analog-Digitalwandlung

Beschreibung: Der ADINP-Befehl bietet die Möglichkeit, einen analogen Wert am Analogeingang 0-3 einzulesen und das Ergebnis als digitalen Wert in der Variablen *var* zu speichern. Dazu konfiguriert der Befehl ADINP zuerst gemäß der zuletzt gültigen DEVICE-Anweisung das AD-Control-Register, um dann den AD-Wandler einzuschalten, den AD-Kanal auszuwählen und schließlich den Wandlungsvorgang zu starten. Als Zeitbasis wird grundsätzlich der interne RC-Oszillator verwendet. Damit beträgt die Wandlungszeit für einen 8-Bit-Wert ca. 20 us bis 60 us (typ. 40 us). Bei Applikationen mit hoher Präzision ist zu beachten, dass diese Wandlungszeiten von der Betriebsspannung, der Temperatur und den Fertigungstoleranzen abhängig sind, ansonsten können sie vernachlässigt werden. Nach Ende der Wandlungszeit (der Prozessor wartet so lange), wird das 8-Bit-Ergebnis in die Variable *var* geschrieben. Die übrigen Bausteine der 16C7x-Reihe haben u.U. mehr Analogeingänge.

Beispiel:

```
define device 16C71,wdt_off,adcfg0
xtal 4.19
adinp 0,a
```

Bemerkung:

Bei PICs mit 10- oder 12-Bit AD-Wandlern ist i.d.R. ein ADDELAY notwendig.

Befehl nur für 12C67x, 16C7x und 16F87x. Dazu muss ADCFGx in der DEVICE-Zeile definiert werden. Der PIC16C71 besitzt vier Analogeingänge, die per Multiplexer auf einen 8-Bit AD-Wandler geführt werden. Diese Analogeingänge sind an den I/O-Pins RA0 bis RA3 realisiert. Verschiedene Konfigurationseinstellungen erlauben folgende Zuordnung der Pin-Funktionen:

16C71, 16C710, 16C711: ADCFGx legt die Nutzung des PORT A fest. (Eingang)

	RA0	RA1	RA2	RA3	Ref.
ADCFG0	analog	analog	analog	analog	Vref=Vdd
ADCFG1	analog	analog	analog	ref.inp	RA3
ADCFG2	analog	analog	digital	digital	Vref=Vdd
ADCFG3	digital	digital	digital	digital	nc

16C72..77: ADCFGx legt die Nutzung von Port A und Port E fest.

	RA0	RA1	RA2	RA3	RA5	RE0	RE1
RE2	Ref.						
ADCFG0	Vdd						
ADCFG1	Vref	RA3					
ADCFG2	DI	Vdd					
ADCFG3	Vref	RA3					
ADCFG4	Vdd						
ADCFG5	Vref	RA3					
ADCFG6	nc						
ADCFG7	nc						

RE0,RE1,RE2 sind nur bei 40pol. Bausteinen vorhanden

ADINP (cont.)

Bei den Bausteinen 16F873, 16F874, 16F876 und 16F877 liefert der AD-Wandler einen 10-Bit Wert. Dieser kann entweder als 8-Bit Wert in eine 8-Bit-Variable geladen werden oder als 10-Bit-Wert in eine 16-Bit-Variable. Dabei lässt sich bestimmen, ob der Wert links- oder rechtsbündig übernommen werden soll. Dazu wird der Modus an den ADCFGx angehängt (z.B. ADCFG0,L). Wenn der Wert in eine 8-Bit-Variable übernommen werden soll, ist es sinnvoll, den Mode "linksbündig" zu wählen. Auf diese Weise fallen die niederwertigsten 2 Bits weg. Soll der Wert in eine 16-Bit-Variable geschrieben werden, sollte man den Modus "rechtsbündig" einstellen (siehe auch das dazugehörige Datenblatt von Microchip).

16F87x	RA0	RA1	RA2	RA3	RA5	RE0	RE1
RE2	Ref.						
ADCFG0	Vdd						
ADCFG1	Vref+	RA3					
ADCFG2	Vdd						
ADCFG3	Vref+	RA3					
ADCFG4	Vdd						
ADCFG5	Vref+	RA3					
ADCFG6	nc						
ADCFG7	nc						
ADCFG8	Vref-	Vref+	RA3,RA2				
ADCFG9	Vdd						
ADCFG10	Vref+	RA3					
ADCFG11	Vref-	Vref+	RA3,RA2				
ADCFG12	Vref-	Vref+	RA3,RA2				
ADCFG13	Vref-	Vref+	RA3,RA2				
ADCFG14	Vdd						
ADCFG15	Vref-	Vref+	RA3,RA2				

RE0,RE1,RE2 sind nur bei 40pol. Bausteinen vorhanden

12F675, 12F683, 16F676, 16F88x u.a.

Hier wird über eine Bitmaske (ANSEL-Register) zwischen Analog- und Digitaleingang entschieden.

ACHTUNG!!! Dass der Pin als Analogeingang arbeiten kann, muss das entsprechende Bit im TRIS-Register auf 1 (INPUT) gesetzt sein. Ab Version 6 muss in diesen Fällen statt ADCFGx ein ANSEL x stehen. Damit ergibt sich folgender Zusammenhang:

AN0	AN1	AN2	AN3
RA0	RA1	RA2	RA4

ANSEL 0
ANSEL 1
ANSEL 2
ANSEL 3
ANSEL 4
ANSEL 5
ANSEL 6
ANSEL 7
ANSEL 8
ANSEL 9
ANSEL 10
ANSEL 11
ANSEL 12

ADINP (cont.)

ANSEL 13

ANSEL 14

ANSEL 15

Hier kann nur AN1 als Referenzeingang freigegeben werden. Dies erfolgt durch den Compilerschalter \$VCFG.

Es handelt sich hierbei um einen 10Bit-Wandler. Deshalb muss das Darstellungsformat des Ergebnisses berücksichtigt werden (siehe 16F87x).

Wichtig!

Normalerweise bleibt das Analog-Digital-Umsetzer-Modul während des Programmlaufs immer eingeschaltet. Soll allerdings dieses Modul aus Stromspargründen u.ä. nur dann aktiv werden, wenn eine Wandlung durchgeführt werden soll, kann man durch anhängen eines '*'-Zeichens den Compiler anweisen, den Code so zu erzeugen, dass das AD-Modul automatisch ein und ausgeschaltet wird.

Beispiel:	ADCFG3*	
	ADCFG3*,R	'nur 16F87x
	ANSEL 3*	'z.B. 12F683

Tipp!

Bei 10-Bit-Wandlern wird bei Verwendung von 8-Bit-Variablen das Format linksbündig verwendet, bei 16-Bit-Variablen dagegen rechtsbündig.

8-Bit -> ,L

16-Bit -> ,R

ASM

(nicht für iL_TROLL)

Syntax: ASM

Funktion: Zwischen dem Befehl ASM und ENDASM steht Assembler-Code

Beschreibung: alles nachfolgende bis zum Befehl ENDASM wird direkt eins-zu-eins in den Zielcode übertragen. Dabei ist die übliche Notation des Assemblers zu berücksichtigen. Die erste Spalte ist reserviert für Labels usw. Der ASM Befehl muss allein in einer Zeile stehen, auch ein REM ist nicht erlaubt. Eventuelle Befehle in der gleichen Zeile werden also nicht berücksichtigt. Die Assemblerbefehle werden im Kapitel ASSEMBLER beschrieben.

Beispiel:

```

ASM
*****
;
;*      Analogsignal wird über Port A0 des PIC16C71 eingelesen
;*      und digital über Port B ausgegeben
*****
;
      ORG          03FFh          ;Endadresse
      GOTO         START          ;zum Programm
      ORG          0004h          ;Interruptroutine
IRQ    MOVWF        TEMP          ;W-Reg retten; TEMP mit DEFINE def.
      MOVF         ADRES, 1       ;A/D-C auslesen
      MOVWF        RB            ;und auf Port B
      BSF          ADCON0, 2      ;erneut starten
      MOVF         TEMP, 1       ;W-Reg zurück
      RETFIE       ;und zum Hauptprog.
START  MOVLW        00h          ;Port B löschen
      MOVWF        RB
      BSF          STATUS, 5      ;zu Seite 1
      MOVWF        TRISB         ;TRIS-Reg und
      MOVLW        02h
      MOVWF        ADCON1        ;A/D-CON1 init
      BCF          STATUS, 5
      MOVLW        C1h
      MOVWF        ADCON0        ;A/D-CON0 init
      MOVLW        C0h
      MOVWF        INTCON        ;A/D-Int ermöglichen
SAMPLE BSF          ADCON0, 2    ;erste Wandlung starten

      PROG        ;hier steht nun ein eigenes Programm
      ENDASM

```

Bemerkung: ASM und ENDASM sind Befehle und müssen deshalb mind. ein führendes Leerzeichen besitzen. Bei Sprüngen über die Seitengrenzen hinweg liegt das Setzen und Löschen der Page-Preselect-Bits in ihrer Verantwortung.

Allgemein ist zu beachten, welche der Hardware-Ressourcen durch den Compiler benutzt bzw. voreingestellt werden.

BINTOASC

(auch CONASC)

BINTOASC wandelt eine 8- oder 16-Bit Zahl in einen ASCII-String um. Bei einem 8-Bit Wert benötigt man einen Puffer mit 3 Bytes, bei 16-Bit Zahlen muss der Puffer 5 Bytes groß sein.

BINTOASC *var,pufferanfang*

Die Zahl 123 wird umgerechnet. Im Puffer steht dann die Zeichenfolge 51, 50, 49 (33H, 32H, 31H).

Beispiel:

```
DEFINE a1 = $30 as byte
```

```
DEFINE buffer = $31 as byte
```

```
...
```

```
LET a1 = 87
```

```
BINTOASC a1,buffer          'buffer contains 55 56 32 (" 87")
```

*) führende Nullen werden unterdrückt (=> Blank, Leerzeichen)

BINTOBCD

(auch CONBCD)

BINTOBCD wandelt eine 8-, 16- oder 32-Bit Zahl in eine BCD-gepackte Zahl. Es wird ein Puffer von 5 Bytes benötigt, auch wenn nur eine Bytevariable umgesetzt werden soll!

`BINTOBCD var,pufferanfang`

Die Zahl 123 wird umgerechnet. Im Puffer steht dann die Zeichenfolge 23H 01H 00H 00H 00H.

BINTODEC

(auch CONDEC oder CONDEZ)

BINTODEC wandet eine 8- oder 16-Bit Zahl in eine 3- bzw. 5-stellige Dezimalzahl um. Für die Wandlung benötigt man einen Puffer von 3 bzw 5 Bytes Länge.

BINTODEC var,pufferanfang

Beispiel:

```
DEFINE a1 = $30 as byte
DEFINE buffer = $31 as byte
...
LET a1 = 87
CONASC a1,buffer          'buffer contains " " 8 7
```

Die Zahl 12345 wird im 5-Byte-Puffer als 5,4,3,2,1 abgelegt.

*) führende Nullen werden unterdrückt (=> Blank, Leerzeichen)

BITPOS

Syntax: BITPOS *var1,var2*

Funktion: Umwandlung einer Zahl zwischen 0 und 7 bzw. 0 und 15 in die entsprechende Bit-Position.

Es bedeutet:

var1 Ergebnisvariable 8-/16-Bit

var2 Parametervariable 8-/16-Bit

Beschreibung: Beim Maskieren von einzelnen Bits steht man oft vor dem Problem, dass die Position des entsprechenden Bits in einer Variablen (z.B. Zählschleife) steht. Beispielsweise soll beim ersten Durchlauf das Bit 0, beim zweiten Durchlauf das Bit 1 usw. geprüft werden. BITPOS wandelt nun diese Zahl um. Dabei wird aus:

0	%00000001	4	%00010000
1	%00000010	5	%00100000
2	%00000100	6	%01000000
3	%00001000	7	%10000000

Beispiel:

```
LET var_A=5
BITPOS var_B,var_A REM in var_B steht anschließend
REM %00100000 = 32
```

Bemerkung: BITPOS kann nur auf 8-und 16-Bit-Variablen angewendet werden, eignet sich aber auch zur Berechnung von 2^n mit $n=0..7$ bzw. $n=0..15$. Es werden nur die untersten 3 bzw. 4 Bits der Parametervariablen berücksichtigt. Damit hat ein Wert 9 in der Variablen *var_A* (obiges Beispiel) des Ergebnis %00000010 zur Folge!

CALVAL

(nur PICs mit aktivem internen RC-Oszillator)

Syntax: CALVAL *wert*

Funktion: Bei Bausteinen mit aktivem internen RC-Oszillator wird der Calibrationswert (Abgleichwert) von der letzten Speicherstelle ins W-Register und von dort ins OSCCAL-Register geladen. Bei OTP-Bausteinen gibt es hier keinerlei Probleme. Anderst sieht es aus, wenn JW-Bausteine (UV löschbare) verwendet werden. Bei diesen geht der Calibrationswert beim Löschen des Bausteins verloren. Deshalb ist es wichtig, diesen Baustein vor der erstmaligen Benutzung unbedingt auszulesen und den Wert der letzten Speicherstelle zu notieren. Damit damit auch bei JW-Typen das Timing immer stimmt, muss dieser Wert ins OSCCAL-Register geladen werden. Der Compiler nimmt normalerweise einen Defaultwert (80H) an. Dieser kann aber eine starke Änderung der Betriebsfrequenz bewirken, so dass der genaue Wert für den jeweiligen Baustein individuell eingestellt werden muss.

CALVAL \$C0

sorgt dafür, dass C0H ins OSCCAL-Register geschrieben wird.

CLOCK und CLOCK1

(nicht bei 10F2xx, 12C50x und 16C5x)

Syntax: `CLOCK var(,intervall)`

Es bedeutet:

var 16 Bit Ergebnisvariable, diese wird alle Sekunde um eins erhöht.

intervall Konstante zw. 10 und 1000, bzw. 2000 bei CLOCK1. Falls *intervall* angegeben wird, lässt sich dieses Inkrementierungsintervall verändern. Die interne Zeitbasis (Timer-Interrupt-Intervall) bleibt allerdings mit 10ms immer konstant.

Funktion: Beim 12C67x, 16C6x, 16C7x und 16X8x kann mittels dieser Funktion eine interruptgesteuerte Zeitbasis initialisiert werden. Dabei wird alle 1/100s die Speicherzelle *TIMERX* inkrementiert und beim Erreichen von Hundert zusätzlich der Inhalt von *var*. Bei Quarzfrequenzen unter 100kHz ist die Zählgrenze von *TIMERX* 128. Aber immer wird *var* pro Sekunde um eins erhöht.

Beschreibung: Der Befehl `CLOCK var` dient zum Initialisieren dieser Zeitbasis. Im weiteren Programmverlauf ergibt sich der Automatismus, dass die Speicherzelle *var* jede Sekunde um eins erhöht wird, wobei es keine Rolle spielt, wie Sie die Variable nutzen.

Beispiel:

```
CLOCK zeit    REM Initialisierung der Zeitbasis
START:
            IF zeit = 60 then GOTO minute    REM 1 Minute ist vergangen
            GOTO start    REM
```

Der Befehl wurde dahingehend erweitert, dass neben der Standardzeitbasis von 10ms auch ein anderer Wert festgelegt werden kann. Dadurch sind auch 'krumme' Zeiten realisierbar. Dazu wird optional hinter die Angabe der Variablen ein Wert zwischen 10 und 2000 geschrieben. Bei manchen Wertangaben ergibt sich allerdings u.U. ein größerer Fehler, der durch die festen Vorteilerfaktoren im Option-Register entsteht.

Beispiel:

```
CLOCK zeit,100 'Uhr läuft 10x so schnell
```

CLOCK verwendet den Timer TMR0 und den Vorteiler.

CLOCK1

Manchmal benötigt man den Vorteiler auch für den Watchdog, wobei der Vorteilerwert durch den `CLOCK`-Befehl bestimmt wird. Sobald `CLOCK` aktiv ist, besitzt der Watchdog keinen Vorteiler mehr und die Periodendauer beträgt nur noch ca. 18ms. Dadurch ist die Periodendauer des Watchdogs i.d.R. viel zu kurz und es wird laufend ein `RESET` generiert. Dies passiert vor allem dann, wenn man den Schalter `$WDTUSR` gesetzt hat. Abhilfe schafft hier dann nur eine sorgfältige Verteilung der `CLRWDT`-Befehle oder man wählt einen anderen Timer für den `CLOCK`-Befehl. Viele PICs haben neben dem `TMR0`-Register auch einen weiteren Timer. Dieser `TMR1` ist ein 16-Bit Timer und kann alternativ für den `CLOCK`-Befehl herangezogen werden. Mit `CLOCK1` wird `TMR1` als Zeitbasis verwendet und somit der Vorteiler für den Watchdog freigehalten, da `TMR1` einen eigenen Vorteiler besitzt.

`TMR1` unterstützt teilweise auch einen zweiten, vom normalen Quarz unabhängigen, Oszillator. Der Quarz wird an `RB6` und `RB7` angeschlossen. Dieser Timer kann auch während einer Sleepphase des PICs laufen. Damit die Zeitbasis auch mit diesem zweiten zusätzlichen Quarz laufen kann, muss man dessen Frequenz mit dem BASIC-Schlüsselwort `CLK1XTAL wert` dem Compiler mitteilen. *wert* wird in MHz angegeben (siehe auch `XTAL`).

CLOCK und CLOCK1 (cont.)

Beispiel:

CLOCK1 zeit,100

'Uhr1 läuft 10x so schnell

CLRWDT

Syntax: CLRWDT

Funktion: Falls der Watchdog aktiviert ist, wird er mit diesem Befehl zurückgesetzt.

Beschreibung: Sobald in der DEFINE DEVICE Zeile der Watchdog-Timer mit dem Schalter WDT_ON aktiviert ist, generiert der Compiler nach jedem BASIC-Befehl ein CLRWDT. Einzige Ausnahme dabei ist eine Bit-Abfrage mit der Verzweigung auf sich selbst (z.B. WARTE: IF RA,0=0 THEN GOTO WARTE). Hier kann der Watchdog aktiv werden, was auch beabsichtigt ist. Soll auch hier ein CLRWDT vom Compiler eingefügt werden, muss eine Dummyzeile eingefügt werden.

Es gibt aber auch sehr kritische Anwendungen, wo man ganz gezielt nur wenige CLRWDT-Befehle wünscht. Um das zu erreichen, muss man diese automatische Generierung von CLRWDT-Befehlen unterdrücken. Dabei bleibt in der DEFINE DEVICE Zeile das WDT_ON stehen, damit das Programmiergerät iL_PRG16PRO diese Einstellung auch übernimmt. Der Compilerschalter \$WDTUSR deaktiviert die automatische Generierung. Nun muss allerdings an geeigneter Stelle der CLRWDT-Befehl platziert werden.

Beispiel:

```
                DEFINE DEVICE 16F..., wdt_on, ....
$WDTUSR                REM keine autom. Generierung
START:
                CLRWDT                REM muss genügend oft aufgerufen werden
```

CONF

Syntax: CONF_x n

Funktion: Definiert Einstellungen im Konfigurationswort direkt als Zahlenwert.

Es bedeutet:

x Zahl zwischen 0 und 13 (0 bis 3 für 16Xxx, 0 bis 13 für 18Xxx)
n 8-Bit-Konstante

Beschreibung: Neben dem Befehl CONFIG kann man das Konfigurationswort direkt als Zahlenwert angeben. Da das Argument aber nur 8-Bit sein darf, bei den 16er Typen ein teilweise ein 14-Bit Wort erwartet wird, muss es für diese PIC Typen aus zwei Werten zusammengesetzt werden.

Configurationsword 1 = CONF1 + CONF0

Configurationsword 2 = CONF3 + CONF2

Das zweite Konfigurationswort ist nicht bei allen PICs vorhanden. CONF0 und CONF2 sind jeweils die niederwertigsten Bits (0-7), CONF1 und CONF3 repräsentieren die Bits 8 bis 13.

Die Bedeutung der einzelnen Bits im Konfigurationswort sind dem Datenblatt zu entnehmen. Die Positionen der einzelnen Funktionen innerhalb des Wortes können u.U. sehr stark variieren.

Bei den 18-er Typen sind die Configbytes immer 8 Bit. Welche Konfigurationsregister mit welchen Bits vorhanden sind, entnehmen Sie bitte dem jeweiligen Datenblatt.

Hinweis:

Wenn CONF_x und CONFIG gemischt verwendet werden sollen, so ist es ratsam, zuerst die CONF_x-Definitionen vorzunehmen. Danach die einzelnen CONFIG Anweisungen. Immer die zuletzt aufgeführte Angabe gilt. Da aber CONF_x sehr viele Parameter beeinflussen kann, sollte die obige Reihenfolge eingehalten werden.

Achtung:

Anhand von CONF_x erkennt der Compiler nicht, ob z.B. der Watchdog ein- oder ausgeschaltet ist (vgl. CLRWDT, \$WDTUSR etc.).

CONFIG

Syntax: CONFIG [wdt_off/wdt_on, protect_off/protect_on, mclr_int/mclr_ext, lvp_off/lvp_on,...]

Funktion: Definiert Einstellungen im Konfigurationswort.

Beschreibung: Inzwischen wird die Argumentenliste bei der DEVICE-Zeile sehr lang. Die Vorgaben für das Konfigurationswort des PIC lassen sich nun auch über CONFIG definieren.

Ab Compilerversion 6.0 darf pro CONFIG-Anweisung immer nur ein Argument folgen (außer Unterargumente).

Es dürfen mehrere CONFIG-Anweisungen (z.B. pro Zeile je ein Argument) verwendet werden, was die Übersichtlichkeit deutlich erhöht.

Anweisungen für den Compiler, die z.T. auch in der DEVICE-Zeile stehen, z.B. ADCFGx können weiterhin dort definiert werden. Am Besten man definiert die Bits für das Konfigurationswort ausschließlich mit CONFIG, die übrigen in der DEVICE-Zeile.

Hinweis:

In den Datenblättern werden für die gleiche Funktion u.U. verschiedene Bezeichnungen verwendet. Die nachfolgende Liste zeigt die Bezeichner wie sie der Compiler kennt.

Aktuell gibt es folgende Parameter:

LP_OSC	low power Oszillator
XT_OSC	Quarzoszillator (sicherer ist aber HS_OSC)
HS_OSC	high speed Quarzoszillator
RC_OSC	externer RC-Oszillator
IRCC_OSC	interner RC-Oszillator mit CLKOUT an OSC2
IRCP_OSC	interner RC-Oszillator mit IO-Pin an OSC2
ERCC_OSC	externer RC-Oszillator mit CLKOUT an OSC2
ERCP_OSC	externer RC-Oszillator mit CLKOUT an OSC2

bei der PIC18F Familie sind dies:

LP_OSC	low power Oszillator
XT_OSC	Quarzoszillator (sicherer ist aber HS_OSC)
HS_OSC	high speed Quarzoszillator
RCC_OSC	externer RC-Oszillator mit CLOCKOUT an OSC2
ECC_OSC	externer Takt mit CLOCKOUT an OSC2
ECP_OSC	externer Takt mit IO-Funktion an OSC2
HSPLL_OSC	Quarzoszillator mit interner PLL
IRCC_OSC	interner RC-Oszillator mit CLKOUT an OSC2
IRCP_OSC	interner RC-Oszillator mit IO-Pin an OSC2
RCP_OSC	externer RC-Oszillator mit IO-Pin an OSC2

WDTEN_ON	Watchdog ein (auch WDT_ON)
WDTEN_OFF	Watchdog aus (auch WDT_OFF)

PWRTEN_ON	Power-Up-Timer ein (auch PWRTEN_ON)
PWRTEN_OFF	Power-Up-Timer aus (auch PWRTEN_OFF)

MCLRE_ON	MCLR-Pin arbeitet als Reseteingang (auch MCLR_EXT)
MCLRE_OFF	MCLR-Pin arbeitet als I/O-Pin (MCLR_INT)

CONFIG (cont.)

BOREN_ON Brown-Out-Erkennung eingeschaltet
BOREN_OFF Brown-Out-Erkennung ausgeschaltet
(wird manchmal auch als BODEN bezeichnet)

BORSEN_ON Brown-out-Reset Software Enable bit ein
BORSEN_OFF Brown-out-Reset Software Enable bit aus

CP_ON Programmspeicher auslesegeschützt (auch PROTECT_ON)
CP_OFF Programmspeicher kann ausgelesen werden (auch PROTECT_OFF)
(hier gibt es auch Einstellungen um den Speicher partiell zu schützen; diese Einstellungen können nur mit CONFx realisiert werden)

CPD_ON Daten-EEPROM lesegeschützt
CPD_OFF Daten-EEPROM nicht lesegeschützt

WRT_ON Flash Programmspeicher schreibgeschützt
WRT_OFF Flash Programmspeicher kann beschrieben werden

LVP_ON Low-Voltage-Programming möglich (ein I/O-Pin geht dabei verloren)
LVP_OFF Low-Voltage-Programming nicht möglich

Nachfolgende Einstellungen können nur direkt über CONFx vorgenommen werden

BORVx Brown-Out-Spannungseinstellung
CCPMX CCP2-Pinzuordnung
IESO Internal External Switchover
FCMEN Fail-Safe Clock Monitor
etc.

Beispiel:

```
conf0 = %11111111
conf1 = %11011111
conf2 = %11111111
Conf3 = %11111111
config anse1 11,r 'nachfolgende Angaben überschreiben
config ircp_osc 'die unter CONFx gemachten
config wdt_on
config pwrten_on
config mclr_int
config protect_off
```

CURSOFF

Syntax: CURSOFF

Funktion: Schaltet den Cursor auf einer LCD aus.

Beschreibung: Dieser Befehl schaltet auf einer LCD-Anzeige den Cursor aus.

Beispiel:

START:

LCDINIT rb,2,40

...

CURON REM Cursor ein, Bedienereingabe anzeigen

LOOP:

INKEY var_a

IF var_a = 0 THEN GOTO LOOP

LCDWRITE 0,0,var_a REM Eingabe auf LCD schreiben

CURSOFF REM Cursor wieder ausschalten

CURSON

Syntax: CURSON

Funktion: Schaltet den Cursor auf einer LCD ein.

Beschreibung: Dieser Befehl schaltet auf einer LCD-Anzeige den Cursor ein.

Beispiel:

START:

```
LCDINIT rb,2,40
```

```
...
```

```
CURON          REM Cursor ein, Bedienereingabe anzeigen
```

LOOP:

```
INKEY var_a
```

```
IF var_a = 0 THEN GOTO LOOP
```

```
LCDWRITE 0,0,var_a  REM Eingabe auf LCD schreiben
```

```
CURSOFF        REM Cursor wieder ausschalten
```

DATA

(nicht für 10F2xx, 12C5xx und 16C5x)

Syntax: DATA *const1,const2,const3,...*

Funktion: Definiert ein Konstantenfeld mit bis zu 2048 Elementen (hängt vom verfügbaren Speicherplatz ab).

Beschreibung: Oft werden in einem Programm Konstanten benötigt, die während des Programmlaufs in unterschiedlicher Reihenfolge den Variablen zugewiesen werden müssen. Bisher war dies nur mittels LOOKUP- und LOOKDWN-Befehl möglich. Diese sind aber nur beschränkt einsetzbar, da deren Anzahl auf etwa 100 Elemente beschränkt ist. DATA beseitigt diese Schranke und legt die Konstanten, im Gegensatz zu LOOKUP und LOOKDWN an das Programmende. Ein Mechanismus sorgt auch beim Wechsel auf eine andere Programmspeicherseite für die korrekte Funktion. Die Elemente dieser Datentabelle können nur gelesen werden, da sie fester Bestandteil des Programmcodes sind. Der Zugriff erfolgt mit dem Schlüsselwort READDATA. Dabei wird ein interner Zeiger mitgeführt, der nach jedem Aufruf von READDATA inkrementiert wird. Für die Überwachung des Zugriffs auf das letzte Element muss der Programmierer sorgen. Dieser Zeiger lässt sich durch den Befehl RESTORE manipulieren, so dass nicht nur sequentielle Zugriffe auf dieses Datenfeld möglich sind.

Beispiel:

START:

```
        RESTORE      REM Lesezeiger auf Anfang
        READDATA var_A      REM liest 1. Wert, hier 65
        RESTORE label1      REM setzt Lesezeiger
        READDATA var_A      REM liest 5. Wert, hier 12
        READDATA var_B,var_C,var_D  REM liest die nächsten 3 Werte
        GOTO start
        DATA 65,66,67,"F"
label1:  DATA 12,45,32,17,25
```

DEC

Syntax: DEC *var*.

Funktion: Vermindert den Inhalt der Variablen *var* um 1.

Beschreibung: Der Befehl DEC *var* erniedrigt den Inhalt der Variablen *var* um 1. Dieser Befehl wird codeoptimiert abgespeichert, im Gegensatz zu LET *var=var-1*. Während letzterer den Gültigkeitsbereich des Ergebnisses überprüft, nimmt der DEC-Befehl darauf keine Rücksicht.

Beispiel:

START:

```
LET  var_a=10  REM  Variable var_a auf 10 setzen
DEC  var_a      REM  in var_a steht jetzt 9
```

DELAY

Syntax: DELAY *var*

Es bedeutet:

var 8- oder 16-Bit Variable oder Konstante

Funktion: Führt eine Warteschleife aus. Die Zeit wird hier in 100us angegeben.

Beschreibung: Der Befehl DELAY unterbricht das laufende Programm für die angegebene Zeit, ohne in den Sleep-Modus zu gehen. Der Stromverbrauch wird bei diesem Befehl nicht reduziert.

Beispiel:

START:

DELAY 5	REM Verzögerung von 500us
GOTO start	REM

Bemerkung: Wartezeiten liegen zwischen 100us und 6,5s (der Maximalwert hängt vom Quarz ab -> Meldung: maximale Zeitauflösung nicht erreichbar, ist beim Einsatz von Variablen wichtig). DELAY wird rein softwaremäßig realisiert.

DOZE

Syntax: DOZE *dauer*

Funktion: Die CPU geht für eine definierte kurze Zeit in den Sleep-Mode

Es bedeutet:

dauer (0 ... 7) Zeitdauer der Sleep-Phase (2 *dauer* x 18 ms), kann Variable oder Konstante sein.

Beschreibung: Der Befehl DOZE veranlasst die CPU, für eine definierte Zeit in den Sleep-Mode zu schalten. In dieser Betriebsart sinkt die Stromaufnahme der CPU auf etwa 20 uA (ohne Pin-Belastung). Die Variable oder Konstante *dauer* kann die Werte 0 bis 7 annehmen. Größere Zahlen werden entsprechend maskiert.

Damit stehen folgende Zeiten zur Verfügung:

DOZE 0 18 ms

DOZE 1 36 ms

DOZE 2 72 ms

DOZE 3 144 ms

DOZE 4 288 ms

DOZE 5 576 ms

DOZE 6 1.152 ms

DOZE 7 2.304 ms ca. 2,3 s

Beispiel:

LET zeit = 7 REM Initialisierung der Variablen *zeit* (2,3 s)

START:

DOZE zeit REM Schlafe 2,3 s

GOTO start REM und schlafe 2,3 s, und schlafe 2,3 s und

DOZE verwendet den Watchdog-Timer (muss daher aktiviert sein) und den Vorteiler. Falls gleichzeitig CLOCK aktiv ist, wird der Vorteiler zum Watchdog umgeschaltet.

DTMFOUT

Syntax: DTMFOUT *port,pin_hf,pin_lf,dauer,pause,key1,key2, ...,keyn*

Funktion: Erzeugt DTMF Frequenzen (Telefon)

Es bedeutet:

port/O Port, kann eine Variable oder Konstante sein

pin_hf/O Pin, an dem die Spaltenfrequenz ausgegeben wird, kann eine Variable oder Konstante sein

pin_lf/O Pin, an dem die Zeilenfrequenz ausgegeben wird, kann eine Variable oder Konstante sein

*dauer*ca. Zeitdauer des Tones in msec (muss Konstante sein)

*pause*ca. Zeitdauer der Pause zw. den Tönen in msec (muss Konstante sein)

key, keyn(0 ... 15)Taste(n) wie beim Telefon, kann eine Variable oder Konstante sein

Beschreibung: Der DTMFOUT Befehl dient zur Ausgabe von DTMF-Tönen an zwei I/O-Pins der CPU. Die Pins müssen am gleichen Port liegen. Diese Funktion entspricht dem Wählen auf einer Telefon-Tastatur. Der Ausgangspin muss entsprechend der unten dargestellten Skizze beschaltet sein. Der DTMF-Ton, der generiert wird, besteht aus der Kombination einer Reihen- und einer Spaltenfrequenz.

Dazu wird *port,pin_hf und pin_lf* als Ausgang konfiguriert und für ca. *dauer* Millisekunden eine Frequenz, die der Taste *key* entspricht, gesendet. Dann folgt eine Pause von *pause* Millisekunden.

Beispiel:

START:

1DTMFOUT ra.0,1,200,50,10,7,8,3,1,4,5,2

REM erzeugt Tonfolge der

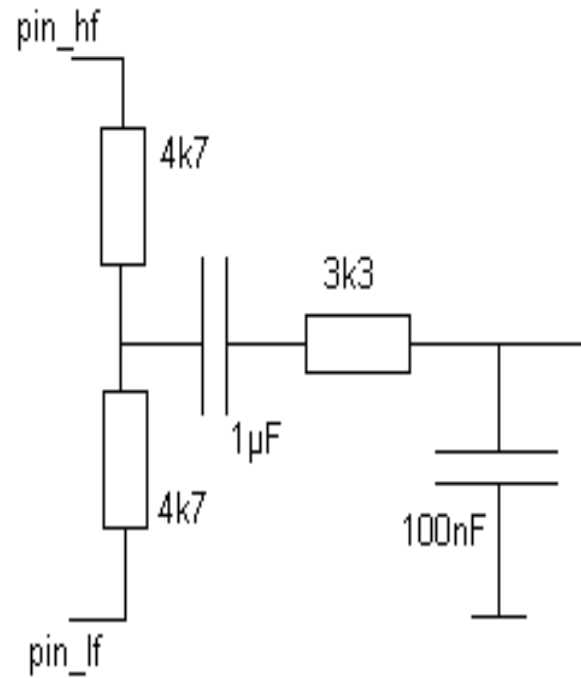
REM Tele.nr 07831 452

REM am Port ra.0 und 1

GOTO START

DTMFOUT (cont.)

	1209	1336	1477	1633
697	1	2	3	A
770	4	5	6	B
852	7	8	9	C
941	*	0	#	D



Zeilen- und Spaltenfrequenz in Hz

Beschaltung der Ausgangspins

Bemerkung: Die Ausgabe ist für eine Taktfrequenz der CPU von 12 Mhz optimiert, dennoch kann bei bestimmten Quarzfrequenzen der Fehler ebenfalls so klein bleiben, dass dieser Befehl richtig funktioniert (z.B. bei 4.19MHz).

Dezimale Werte für key 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
 entspricht Telefon-Taste 1 2 3 4 5 6 7 8 9 * 0 # A B C D

EEDATA

Syntax: EEDATA(adr) wert1, wert2, wert3 [...] (max. 8 Byte-Werte)

Funktion: Vorgegebene Konstanten werden in das EEPROM-Datenfeld des PICs programmiert.

Beschreibung: Daten für das interne EEPROM eines PICs (nicht jeder verfügt über ein solches) können entweder durch den Befehl `WRITE` während des Programmlaufes oder bereits beim Programmieren des Bausteins hinterlegt werden. Damit können konstante Werte, die bereits vor dem Programmstart festliegen, in das Data-EEPROM gebracht werden z.B. Parameter, Passwörter u.ä.. Bei den meisten Programmiergeräten muss der Datenbereich explizit programmiert werden, andere programmieren nach dem Codebereich auch den Datenbereich.

Beispiel:

Bemerkung: Nicht jeder PIC-Mikrocontroller besitzt ein Daten-EEPROM!

END

Syntax: END

Funktion: Markiert das Programmende und schaltet die CPU in den Sleep-Mode.

Beschreibung: Die Programmausführung wird beendet, die zuletzt aktiven Ausgangszustände bleiben erhalten und der Prozessor wird in den Sleepmodus versetzt. Dieser Zustand kann nur durch einen externen Reset beendet werden.

Beispiel:

```
        LET var_a=0    REM Initialisierung der Variablen var_a
START:
        ....
        GOTO START
WEITER:
        END            REM Halte Programm an, gehe in den SLEEP Mode
```

Bemerkung: Der Strombedarf der CPU reduziert sich im Sleep-Mode auf etwa 20 uA; dazu addieren sich sämtliche Ströme, die aus den Portleitungen gezogen werden. Wenn die *Ports,Pins* nach dem END Befehl als Ausgang-High oder Ausgang-Low geschaltet sind, tritt folgendes auf: Alle 2,3 Sekunden wird der Strom an den *Ports,Pins* kurzzeitig für 18 Millisekunden unterbrochen.

ENDASM

(nicht für iL_TROLL)

Syntax: ENDASM

Funktion: Zwischen dem Befehl ASM und ENDASM steht Assembler-Code

Beschreibung: Alles nachfolgende bis zum Befehl ENDASM wird direkt eins-zu-eins in den Zielcode übertragen. Dabei ist die übliche Notation des Assemblers zu berücksichtigen. Die erste Spalte ist reserviert für Labels usw. Der ENDASM Befehl muss alleine in einer Zeile stehen, auch ein REM ist nicht erlaubt. Eventuelle Befehle in der gleichen Zeile werden also nicht berücksichtigt. Nach ENDASM wird wieder der BASIC-Befehlssatz übersetzt.

Beispiel:

```

ASM
*****
;
;*      Analogsignal wird über Port A0 des PIC16C71 eingelesen
;*      und digital über Port B ausgegeben
*****
;

      ORG      03FFh      ;Endadresse
      GOTO     START     ;zum Programm
      ORG      0004h      ;Interruptroutine
IRQ    MOVWF    TEMP      ;W-Reg retten; TEMP mit DEFINE def.
      MOVF     ADRES, 1   ;A/D-C auslesen
      MOVWF    RB         ;und auf Port B
      BSF      ADCON0, 2   ;erneut starten
      MOVF     TEMP, 1     ;W-Reg zurück
      RETFIE    ;und zum Hauptprog.
START  MOVLW    00h        ;Port B löschen
      MOVWF    RB
      BSF      STATUS, 5   ;zu Seite 1
      MOVWF    TRISB      ;TRIS-Reg und
      MOVLW    02h
      MOVWF    ADCON1     ;A/D-CON1 init
      BCF      STATUS, 5
      MOVLW    C1h
      MOVWF    ADCON0     ;A/D-CON0 init
      MOVLW    C0h
      MOVWF    INTCON     ;A/D-Int ermöglichen
SAMPLE BSF      ADCON0, 2  ;erste Wandlung starten

      PROG      ;hier steht nun ein eigenes Programm
      ENDASM

```

Bemerkung: ASM und ENDASM sind Befehle und müssen deshalb mind. ein führendes Leerzeichen besitzen. Falls Programmspeicherseitengrenzen überschritten werden könnten, verwenden Sie die Befehle LJUMP und LCALL.

ERR

Syntax: ERR *var*

Funktion: übergibt der Variablen *var* den zuletzt gesetzten Inhalt der Fehlervariablen.

Es bedeutet:

var (8-Bit) Variable mit dem zuletzt gesetzten Fehlercode

Beschreibung: Beim Aufruf einer Arithmetikfunktion, wird ERR auf 0 gesetzt. Wenn bei Rechenoperationen das Ergebnis < 0 oder > 65535 ist, tritt ein Überlauf ein. *var* hat dann den Wert 1.

Eine ERR-Variable gibt Auskunft über interne Vorgänge. Sie ist bitorientiert mit folgenden Funktionen:

Bit 7 Überlauf bei Arithmetikroutinen (wird vor jeder Operation gelöscht)

Bit 6 werden vom Compiler zum Merken der Rückkehradresse aus der RUN-

Bit 5 TIME-Bibliothek beim 16C54 und 16C55 verwendet, da diese nur einen

Bit 4 zweistufigen Stack besitzen, der hier nicht ausreichend ist.

Bit 3 Timeout in SERIN, PULSIN oder I2C-Kommunikationsfehler aufgetreten.

Bit 2 Zählt die Schachteltiefe der GOSUB-Routinen, da diese Überprüfung zum

Bit 1 Zeitpunkt der Compilierung nicht durchgeführt werden kann.

Bit 0 000 = kein UP, 001 = 1 UP, 101 = 5 UP = nicht erlaubt.

Beispiel: ERR *wert* 'übergibt der Variablen *wert* den Fehlercode (8-Bit)

FOR TO NEXT

Syntax: FOR *zählvariable* = *start* TO *end* NEXT *zählvariable*

Funktion: Führt eine Programmschleife durch. Schachteltiefen bis 16 Ebenen sind erlaubt.

Es bedeutet:

zählvariable (8- oder 16-Bit) dient als Schleifenzähler

start (8- oder 16-Bit) Startwert der Zählvariablen, kann eine Variable oder Konstante sein

end (8- oder 16-Bit) Endwert der Zählvariablen, kann eine Variable oder Konstante sein

Beschreibung: FOR-NEXT-Schleifen eröffnen Wiederholungsschleifen, in deren Verlauf eine Folge von Programmanweisungen mehrmals durchgeführt werden. Zu Beginn der Schleifenausführung wird die angegebene *zählvariable* auf den Anfangswert *start* gesetzt. Die Programmausführung läuft nun bis zur *Next* Anweisung, an der die *zählvariable* inkrementiert wird. Die Schleife wird so lange wiederholt, bis der Endwert *end* überschritten wird. Das Inkrement beträgt +1. Die Schachteltiefe beträgt max. 16 Ebenen. Zur Bestimmung der Start- bzw. Endwerte sind hier Ausdrücke erlaubt. Es gelten dabei die gleichen Vorgaben wie bei LET.

Beispiel:

```
START:      FOR var_a = 0 TO 5
             GOSUB lese      REM Abfrage externes Signal
             NEXT var_a
             END

LESE:       IF ra.0 = 1 THEN GOTO merke
             PULSOUT ra.1,5      REM kein Signal, Impuls ausgeben RETURN
MERKE:      LET var_b = 1      REM Signal erkannt, merken
             RETURN
```

Bemerkung: *zählvariable*, *start* - und *end*- Variable sind 8- oder 16-Bit-Variablen (0-255 bzw. 0-65535). Die Schleife wird unabhängig von den Werten *start* und *end* mindestens einmal durchlaufen. Die Schleife wird nur beendet, wenn *zählvariable* = *endvariable*

Erlaubt ist auch:

FOR var_a = var/const TO var_b +/- const

Sollte das Fehlen eines NEXT entdeckt werden, zeigt dies der Compiler erst am Ende des Compilervorganges an. Die mitgelieferte Zeilennummer stimmt deshalb nicht.

FREQIN

Syntax: FREQIN port,pin,torzeit,var

Funktion: FREQIN misst eine Frequenz an einem bestimmten Pin. Dabei kann man die Torzeit mit angeben.

Es bedeutet:

port	(8-Bit)	Name des Ports
pin	(8-Bit)	Nummer des Eingangspins
torzeit	(16-Bit)	Konstante (max. 10000)
var	(16-Bit)	Variable zur Aufnahme des Ergebnisses

Beschreibung: Die Torzeit muss eine Konstante sein. Die Werte liegen zwischen 1000 und 10. Der Wert 1000 liefert die gemessene Frequenz mit einer Auflösung von 1Hz. Die tatsächliche Messzeit beträgt dabei 500ms, da sowohl die steigende als auch die fallende Flanke des Messsignals gezählt werden. Bei einer Torzeit von 10 beträgt die Auflösung nur noch 100Hz, die tatsächliche Messzeit liegt bei 5ms.

Beispiel:

```
START:      FREQIN RB,1,1000,var_s    REM zählt die Impulse an RB,1 für
              REM die Zeit von 0,5s. Da beide
              REM Flanken gezählt werden, ent-
              REM spricht dies der Frequenz.
```

Achtung!

Dieser Befehl ist für eine Taktfrequenz bis 9.9 MHz Quarz verwendbar. Bei höheren Frequenzen erzeugt der Compiler eine entsprechende Fehlermeldung. Man kann durch Verkleinerung der Torzeit dennoch bei höheren Quarzfrequenzen messen. Dabei ist das Ergebnis natürlich entsprechend der neuen Torzeit zu korregieren.

Der Wert $TORZEIT * QUARZFREQUENZ$ darf 9900 nicht überschreiten!

GOSUB

Syntax: GOSUB *adr*

Funktion: Unterprogrammaufruf (Schachteltiefe beim 16C71 und 16C84 max. vier Ebenen, beim 16C5x und 12C5x dürfen GOSUBs nicht geschachtelt werden).

Es bedeutet:

adr Label als Sprungziel

Beschreibung: Sprung zu einem Unterprogramm. Der Befehl GOSUB speichert die Adresse des nächsten Befehls auf dem Stack und springt dann zu der im Befehl GOSUB angegebenen *adr*, um dort die Programmausführung fortzusetzen. Das aufgerufene Unterprogramm muss durch den Befehl RETURN beendet werden, welches eine Fortsetzung an der zuvor auf dem Stack abgelegten Adresse bewirkt.

Beispiel:

```
START:    FOR var_a = 0 TO 5
          GOSUB lese      REM Sprung zum Unterprogramm
          NEXT var_a
          END
          REM hier steht das Unterprogramm
          REM mit einer Abfrage eines externen
          REM Signals und einer PULS-Ausgabe
LESE:     IF ra.0 = 1 THEN GOTO merke
          PULSOUT ra.1,5
          RETURN
          REM Signal erkannt, merken
MERKE:    LET var_b = 1
          RETURN
```

GOTO

Syntax: GOTO *adr*

Funktion: Sprung zu einer Programmarke. Unbedingter Sprung

Es bedeutet:

adr Label als Sprungziel

Beschreibung: der Compiler iL_BAS16 kennt keine Zeilennummer. Sprungziele werden durch eine *adr* (Label) gekennzeichnet. Diese *adr* muss durch einen Doppelpunkt abgeschlossen werden. Der Befehl GOTO setzt die Programmausführung an der durch den nachfolgenden Label bestimmten *adr* fort. Dieses Schlüsselwort ist obligatorisch und darf deshalb nicht weggelassen werden.

Beispiel:

```
START:      INPUT ra,var_a
            IF var_a = 1 THEN GOTO weiter
            ...
            GOTO start
WEITER:     ...
```

HIGH

Syntax: HIGH *port,pin*

Funktion: Konfiguriert den entsprechenden *Port,Pin* als Ausgang (verändert also das TRIS-Register) und setzt ihn auf High-Signal (1)

Es bedeutet:

port/O Port, kann eine Variable (8 Bit) oder Konstante sein

pin/O Pin, kann eine Variable (8 Bit) oder Konstante sein

Beschreibung: Durch den Befehl HIGH wird der angegebene Pin auf High-Signal (1) gesetzt. Ist dieser Pin zum Zeitpunkt der Befehlsausführung als Eingang programmiert, wird er automatisch auf Ausgang umprogrammiert. Dieser Status bleibt auch nach der Ausführung des Befehls erhalten. d.h. das TRIS-Register wird verändert.

Beispiel:

```
                LET var_a = 1    REM Variable a = Pin 1
START:          INPUT ra,var_a  REM Pin 1 ist Eingang
                IF var_a = 0 THEN GOTO weiter
                HIGH ra,var_a    REM Pin 1 ist Ausgang und High
WEITER:         ...
```

Bemerkung: HIGH sollte nicht auf normale Variablen angewendet werden. Dazu steht der Befehl SET zur Verfügung.

I2CDELAY

Syntax: I2CDELAY *wert*

Funktion: Verringert die Clockfrequenz auf dem I2C-Bus und verlängert die Zeit zwischen den beiden fallenden und steigenden Flanken in der Start- bzw. Stop-Bedingung.

Es bedeutet:

wert nur Konstante, Zahl zwischen 1 und 99 (default = 2)

Beschreibung: Führt der I2C-Bus über lange Leitungen, ist es oftmals von Vorteil, wenn die Übertragungsgeschwindigkeit vermindert werden kann. Ebenso ist dies sinnvoll, wenn langsame Slavebausteine benutzt werden, die die Normgeschwindigkeit von 100kHz nicht erreichen, z.B. PIC als Slavebaustein.

Beispiel 1:

```
        DEFINE sda=2
        DEFINE scl=3
        I2CINIT rb,sda,scl REM Controller arbeitet als Master
        I2CDELAY 4

START:
    ....

    ....

WEITER:  ...
```

Beispiel 2:

```
        DEFINE sda=2
        DEFINE scl=3
        I2CINIT rb,sda,scl,SLAVE,ADR=170,BUFFER=$1c REM arbeitet als Slave

START:
    ....

    ....

WEITER:  ...
```

Diese Funktion ist nur beim Master sinnvoll, da nur dieser den Bustakt bestimmt. Wird das I2C-Hardwaremodul als Master verwendet, kann I2CDELAY nicht verwendet werden.

I2CHARDS

Syntax: I2CHARDS ADR=wert, RXP=var1, RXB=adr1, TXP=var2, TXB=adr2

Funktion: Nutzt die interne Hardware des PICs für die Funktion des I2C-Slaves aus.

Es bedeuten:

ADR=wert	legt die Geräteadresse fest, unter der der Baustein über den I2C-Bus angesprochen wird; muss eine Konstante sein.
RXP=var1	Zeiger für den Empfangspuffer, muss eine Variable sein.
RXB=adr1	Legt den Anfang des Empfangspuffers fest, ist eine Adresse oder eine Variable ab der die empfangenen Zeichen nacheinander abgelegt werden
TXP=var1	Zeiger für Sendepuffer, muss eine Variable sein.
TXB=adr2	Legt den Anfang des Sendepuffers fest (darf auch mit RXB identisch sein)

Beschreibung: Nutzt die I2C-Hardware Ressourcen des PICs für die SLAVE-Funktion. Hier handelt es sich noch um eine vorläufige Version. Die Kommunikation mit dem MASTER erfolgt vollständig im Hintergrund und interruptgesteuert. Das Schlüsselwort lautet I2CHARDS und hat folgende Parameter: ADR=wert, RXP=var, RXB=adr, TXP=var, TXB=adr. Die Angabe ADR ist die von diesem SLAVE verwendete Deviceadresse, also die Adresse, unter der der MASTER diesen Baustein bedienen kann. Hier muss die Adresse für den Schreibzugriff (LSB=0) verwendet werden. RXP und TXP sind Zeigervariablen, die auf den Empfangs- bzw. Sendepuffer zeigen. Diese Werte werden beim Empfang einer gültigen Deviceadresse automatisch auf den Pufferanfang gelegt. Dieser Puffer ist in den Angaben RXB und TXB definiert. Die Variablen für die Zeiger (RXP, TXP) müssen in der ersten RAM-Seite bis 7FH liegen. Die Puffer können auch im Bereich bis FFH liegen. Der Zugriff auf die Zeigervariablen erlaubt es, während des Programmlaufes zu erkennen, ob neue Daten vorliegen. Nachdem diese abgearbeitet sind, können sie auch vom Programm aus auf den jeweiligen Pufferanfang gelegt werden. Der Slave lässt sich über die Funktion "Interrupt sperren/freigeben" in gewissen Grenzen steuern. Dazu kann sowohl der globale Interrupt (GIE) als auch der I2C-Interrupt verwendet werden (PEIE). Allerdings muss man beachten, dass, falls die Interrupts gesperrt sind, das entsprechende Flag SSPIF im Register 0CH gesetzt ist. D.h., wenn der Interrupt wieder freigegeben wird, verzweigt der Prozessor einmal in das Interrupt Serviceprogramm (ISR), falls ein gültiger Slavezugriff vorgelegen hat. Aus diesem Grund empfiehlt es sich, vor der Freigabe diese Bit prophylaktisch zu löschen.

Beispiel:

```

xtal 4.194304
define device 16c74,wdt_off,xt_OSC
define rxzeiger=$50 as byte      'Empfangszeiger
define txzeiger=$52 as byte      'Sendezeiger
define rxbuffer=$60              'Startadresse für Empfangspuffer
define txbuffer=$68              'Startadresse für Sendepuffer
define stack=$40 'Stackbereich beim Interrupt
define syn_asy=$60 as word
define st_fadl=$62 as word

i2chards adr=100,rxp=rxzeiger,rxb=rxbuffer,txp=txzeiger,txb=txbuffer

cold:
    set intcon,6      'PEIE freigeben (auch SET PEIE)
    set intcon,7      'GIE freigeben
    asm
    bcf 0ch,3         'SSPIF-Flag löschen
    endasm
    res sspif         'alternativ zu bcf 0ch,3

loop:
    goto loop         'wartet endlos, da Interruptbetrieb
    let st_fadl=5

```

I2CHARDS (cont.)

```
let syn_asy=6  
end
```

Bemerkung: In I2CHARDS wird der Zeiger auf FFH begrenzt. Falls weitere Bytes kommen oder gelesen werden sollen, erfolgt immer ein Zugriff auf die Adresse FFH (Bank1).

I2CINIT

Syntax: (MASTER) 2CINIT *port,sdapin,sclpin*

Syntax: (SLAVE) 2CINIT *port,sdapin,sclpin,SLAVE,ADR=adr,BUFFER=adr (,WAIT=port,pin)*

Funktion: Konfiguriert die entsprechenden *Port* und *Pins* als Busleitungen für I2C-Bausteine

Es bedeutet

port I/O-Port, muss eine 8 Bit Konstante sein

sdapin I/O-Pin, wird als Datenleitung für den I2C-Bus definiert (8 Bit Konstante)

sclpin I/O-Pin, wird als Clockleitung für den I2C-Bus definiert (8 Bit Konstante)

Um die Slavefunktionen zu implementieren, sind weitere Initialisierungsangaben notwendig.

SLAVE der Controller führt die Slavefunktionen aus.

ADR=adr Bausteinadresse, kann Variable (8 Bit) oder Konstante sein

BUFFER=adr definiert die Startadresse des Sende- und Empfangspuffer

Beschreibung: Dieser Befehl initialisiert die entsprechenden Bits im TRIS-Register und legt die Ruhepegel dieser Leitungen an. SDAPIN und SCLPIN müssen am gleichen Port liegen. Dabei wird unterschieden, ob eine Master- oder Slavefunktion implementiert wird.

Beispiel1:

```
DEFINE sda=2 as const
DEFINE scl=3 as const
I2CINIT rb,sda,scl REM Controller arbeitet als Master
```

START:

....

....

WEITER:

...

Beispiel2:

```
DEFINE sda=2 as const
DEFINE scl=3 as const
I2CINIT rb,sda,scl,SLAVE,ADR=170,BUFFER=$1c REM arbeitet als Slave
```

START:

....

....

WEITER:

...

Hinweis zu Beispiel 2

Die Bausteinadresse lautet 170 für Schreiben und 171 für Lesen . Der I2C-Kommunikationspuffer umfasst 4 Bytes. Der Master darf also maximal 4 Bytes in diesen Baustein schreiben (siehe auch I2CSLAVE).

Bemerkung: Die entsprechenden Einträge im TRIS-Register dürfen während des Programmablaufs nicht verändert werden. Der Befehl I2CINIT ist ein bitorientiertes Kommando.

Der Schalter WAIT ist zur Zeit nicht implementiert. Damit ist aber die Taktgeschwindigkeit bei 4 Mhz auf etwa 30kHz beschränkt. Bei 20 Mhz wird somit die maximale Standardübertragungsgeschwindigkeit erreicht.

Laut I2C-Spezifikation müssen an den beiden Busleitungen SCL und SDA Pull-Up-Widerstände angeschlossen sein, da die Software dem Slavebaustein die Möglichkeit gibt, WAIT-Zyklen einzufügen. Dazu wird SCL als Input geschaltet.

Sobald Sie die Slavefunktion aktivieren, werden die Schlüsselworte SLAVE, ADR und BUFFER gültig und dürfen deshalb nicht mehr mit DEFINE definiert werden. Sonst wird ein Fehler bei I2CINIT angezeigt.

I2CRD

Syntax: I2CREAD *adr1*, *var*

Funktion: Liest aus einem Baustein mit der I2C-Adresse *adr1* den Wert und übergibt ihn der Variablen *var*.
Lese dabei, ohne dass eine Start- und Stop-Condition erzeugt wird.

Es bedeutet:

adr1 I2C-Bausteinadresse

var 8 Bit Variable, gelesener Wert wird hier abgelegt.

Beschreibung: Liest aus einem I2C-Baustein einen Wert und übergibt ihn der Variablen *var*. Normalerweise verfügen Speicherbausteine über einen internen Autoincrement-Mechanismus. Soll jedoch ein Wert von einer anderen als der nächsten Adresse gelesen werden, muss der Adresszähler im Baustein mittels I2CWR neu gesetzt werden.

Beispiel:

```
I2CINIT ra,2,3    REM I2C init.  
START:  I2CSP      'STOP condition  
        I2CST      'START condition  
I2CWR 160,2      REM Sekunde beim RTC PCF 8583P  
I2CRD 160,var_a   REM Sekunde lesen  
I2CRD 160,var_b   REM Minuten lesen  
I2CRD 160,var_c   REM Stunden lesen  
I2CSP           'STOP condition erzeugen
```

I2CRDB

Syntax: I2CRDB var [,var, var,...]

Funktion: Einzelnes Byte lesen. Liest aus einem zuvor adressierten Baustein einen Wert und übergibt ihn der Variablen *var* und quittiert das ACK-Bit entsprechend.

Es wird keine Start- und Stop-Condition erzeugt.

Es bedeutet:

var 8 Bit Variable, gelesener Wert wird hier abgelegt.

Beschreibung: Liest aus einem bereits mit I2CRD adressierten I2C-Baustein einen Wert und übergibt ihn der Variablen *var*. Normalerweise verfügen Speicherbausteine über einen internen Autoincrement-Mechanismus. Soll jedoch ein Wert von einer anderen als der nächsten Adresse gelesen werden, muss der Adresszähler im Baustein mittels I2CWR neu gesetzt werden.

Beispiel:

```
I2CINIT ra,2,3    REM I2C init.
START:  I2CSP      'STOP condition
        I2CST      'START condition
        I2CWR 160,2  REM Baustein adressieren und int. Zeiger positionieren
        I2CRDB var_a  REM Sekunde lesen
        I2CRDB var_b  REM Minuten lesen
        I2CRDB var_c  REM Stunden lesen
        I2CSP        REM Stopcondition erzeugen
```

Hinweis:

Manche Slavebausteine können eine STOP-Bedingung während einer Übertragung nicht erkennen und erwarten eine Beendigung mittels ACK-Bits. In solchen Fällen muss das letzte Byte über I2CRDBN eingelesen werden.

I2CRDBN

Syntax: I2CRDBN *var*

Funktion: Einzelnes Byte lesen. Liest aus einem zuvor adressierten Baustein einen Wert, übergibt ihn der Variablen *var* aber quittiert das ACK-Bit nicht. Dadurch wird die Übertragung beendet.

Es wird keine Start- und Stop-Condition erzeugt.

Es bedeutet:

var 8 Bit Variable, gelesener Wert wird hier abgelegt.

Beschreibung: Liest aus einem bereits mit I2CRD adressierten I2C-Baustein einen Wert und übergibt ihn der Variablen *var*. Will man eine variable Anzahl von Bytes aus einem Baustein lesen, bietet es sich an, das Ganze in einem Schleifenkonstrukt zu implementieren. Zur Beendigung der Übertragung reicht es oft aus, einfach eine STOP-Bedingung zu erzeugen. Manche Slavebausteine können dies jedoch nicht und erwarten ein Nichtquittieren des ACK-Bits. Der I2CRDB-Befehl quittiert jedoch immer, außer im Fehlerfall. Benötigt der Slavebaustein diese Art der Beendigung, muss das letzte Byte mit I2CRDBN eingelesen werden.

Beispiel:

```
I2CINIT ra,2,3      'I2C init.
START:  I2CSP        'STOP condition
        I2CST        'START condition
        I2CWR 160,2   'Baustein adressieren und int. Zeiger positionieren
        FOR ii=1 to anzahl
          IF ii = anzahl THEN I2CRDBN var_a ELSE I2CRDB var_a
          LET puffer(ii)=var_a
        NEXT ii
        I2CSP        'Stopcondition erzeugen
```

Wenn das letzte Byte gelesen werden soll ist die Bedingung *ii=anzahl* erfüllt.

I2CREAD

Syntax: I2CREAD *adr1*, *var*

Funktion: Liest aus einem Baustein mit der I2C-Adresse *adr1* den Wert und übergibt ihn der Variablen *var*

Es bedeutet:

adr1 I2C-Bausteinadresse

var 8 Bit Variable, gelesener Wert wird hier abgelegt.

Beschreibung: Liest aus einem I2C-Baustein einen Wert und übergibt ihn der Variablen *var*. Normalerweise verfügen Speicherbausteine über einen internen Autoincrement-Mechanismus. Soll jedoch ein Wert von einer anderen als der nächsten Adresse gelesen werden, muss der Adresszähler im Baustein mittels I2CWRITE neu gesetzt werden.

Beispiel:

```
I2CINIT ra,2,3    REM I2C init.
```

```
START:           I2CWRITE 160,2 REM Sekunde beim RTC PCF 8583P
                  I2CREAD 160,var_a      REM Sekunde lesen
                  I2CREAD 160,var_b      REM Minuten lesen
                  I2CREAD 160,var_c      REM Stunden lesen
```

```
....
```

```
....
```

```
WEITER:         ...
```

Tritt ein Kommunikationsfehler auf, wird das Bit 3 in der ERR_-Variablen gesetzt.

I2CSLAVE

Syntax: I2CSLAVE

Funktion: Der Controller arbeitet als I2CSLAVE-Baustein. Wenn dieser Befehl ausgeführt wird, erfolgt ein Sprung in die Laufzeitbibliothek. Hier wartet das Programm auf eine I2C-Adresse. Ist diese gleich der in I2CINIT definierten Adresse, wird in Abhängigkeit des Schreib-/Lesebits der Pufferinhalt an den Master übertragen bzw. von diesem empfangen. Dabei wird aber keine Puffergrenze überwacht. Dies liegt in der Verantwortung des Masters, da dieser die Clocksignale erzeugt und auch das Ende der Kommunikation festlegt. Das Konzept des Puffers wurde deshalb gewählt, um die Slavefunktionen möglichst flexibel zu gestalten. Sobald die Kommunikation beendet ist, erfolgt die Rückkehr zum BASIC-Programm. Hier wird dann der Inhalt des Puffers analysiert und weiterverarbeitet.

War die I2C-Adresse nicht gleich der eingestellten Adresse, wird ebenfalls zum BASIC-Programm zurück gesprungen, allerdings ist in diesem Fall das Bit 3 der ERR-Variablen (TIMEOUT) gesetzt. Was gegebenenfalls hier ausgewertet werden kann.

Da die Funktion z.Z. eine reine Softwarelösung ist, ist die Übertragungsgeschwindigkeit relativ niedrig. Bei 4 Mhz beträgt sie ca. 35kHz, beim ca. 12Mhz somit die Normgeschwindigkeit.

Beschreibung: Kommuniziert mit dem Master über den Kommunikationspuffer. Die Datenrichtung legt der Master mit dem WRITE-Bit in der Adresse fest.

Beispiel:

```
REM Übernimmt 4 Bytes vom I2C-Master und schreibt die Werte in die LCD
    DEFINE ERR=ERR_
    DEFINE buffer = $20 as byte
    DEFINE var_a = $20 as byte
    DEFINE var_b = $21 as byte
    DEFINE var_c = $22 as byte
    DEFINE var_d = $23 as byte
    I2CINIT ra,2,3,SLAVE,ADR=170,BUFFER=buffer    REM I2C init.
    LCDINIT rb,1,2
    LCDCLEAR

START:    IC2SLAVE
          IF ERR,3=1 THEN GOTO START    REM falsche Adresse
          LCDWRITE 1,1,var_a,var_b,var_c,var_d    REM diese Variablen sind
          REM der Puffer

WEITER:    ...
```

Bemerkung:

Sobald Sie die Slavefunktion aktivieren, werden die Schlüsselworte SLAVE, ADR und BUFFER gültig und dürfen deshalb nicht mehr mit DEFINE definiert werden. Sonst wird ein Fehler bei I2CINIT angezeigt.

I2CSP

Syntax: I2CSP

Funktion: Erzeugt eine STOP-Bedingung

Beschreibung: Bei den Befehlen I2CRD und I2CWT werden weder START- noch STOP-Bedingungen generiert. Diese muss man an den entsprechenden Stellen mit I2CST und I2CSP erzeugen.

WARUM?

Es gibt I2C-Bausteine, bei denen mit den Befehlen I2CREAD bzw. I2CWRITE nicht gearbeitet werden kann, da Sie ein etwas "modifiziertes" Protokoll besitzen. Auch um die Funktion REPEATED START zu implementieren, muss man diese Einzelbefehle verwenden.

Beispiel:

```
I2CINIT ra,2,3    REM I2C init.
```

```
START:  I2CSP      'STOP condition
        I2CST      'START condition
        I2CWR 160,2    REM Sekunde beim RTC PCF 8583P
        I2CRD 160,var_a  REM Sekunde lesen
        I2CRD 160,var_b  REM Minuten lesen
        I2CRD 160,var_c  REM Stunden lesen
        I2CSP      'STOP condition
```

I2CST

Syntax: I2CSP

Funktion: Erzeugt eine STOP-Bedingung

Beschreibung: Bei den Befehlen I2CRD und I2CWT werden weder START- noch STOP-Bedingungen generiert. Diese muss man an den entsprechenden Stellen mit I2CST und I2CSP erzeugen.

WARUM?

Es gibt I2C-Bausteine, bei denen mit den Befehlen I2CREAD bzw. I2CWRITE nicht gearbeitet werden kann, da Sie ein etwas "modifiziertes" Protokoll besitzen. Auch um die Funktion REPEATED START zu implementieren, muss man diese Einzelbefehle verwenden.

Beispiel:

```
I2CINIT ra,2,3    REM I2C init.  
START:  I2CSP      'STOP condition  
        I2CST      'START condition  
I2CWR 160,2      REM Sekunde beim RTC PCF 8583P  
I2CRD 160,var_a  REM Sekunde lesen  
I2CRD 160,var_b  REM Minuten lesen  
I2CRD 160,var_c  REM Stunden lesen  
I2CSP          'STOP condition
```

I2CWR

Syntax: I2CREAD *adr1*, *var*

Funktion: Schreibt in einen I2C-Baustein mit der I2C-Adresse *adr1* an die Speicheradresse *adr2* den Inhalt der Variablen oder die Konstante ein.

Lese dabei, ohne dass eine Start- und Stop-Condition erzeugt wird.

Es bedeutet:

adr1 I2C-Bausteinadresse

adr2 Adresse der Speicherzelle innerhalb des Bausteins

var/const Daten, die unter der Adresse *adr2* abgespeichert werden sollen.

Beschreibung: Der Inhalt der Variablen *var* oder die Konstante *const* wird in der Speicherzelle mit der Adresse *adr2* abgespeichert. Die Auswahl des Bausteins selbst erfolgt am I2C-Bus durch dessen Bausteinadresse *adr1*. Obwohl die meisten Bausteine über eine Autoincrementfunktion verfügen, d.h. der interne Adresszähler wird bei jedem Lese- bzw. Schreibzugriff automatisch um eins erhöht, muss bei diesem Befehl die Adressangabe *adr2* unbedingt angegeben werden.

Beispiel:

```
I2CINIT ra,2,3    REM I2C init.  
START: I2CSP      'STOP condition  
I2CST           'START condition  
I2CWR 160,2      REM Sekunde beim RTC PCF 8583P  
I2CRD 160,var_a  REM Sekunde lesen  
I2CRD 160,var_b  REM Minuten lesen  
I2CRD 160,var_c  REM Stunden lesen  
I2CSP           'STOP condition
```

I2CWRB

Syntax: I2CWRB *var/const*

Funktion: Schreibt in einen bereits adressierten I2C-Baustein den Inhalt der Variablen oder die Konstante ein.

Es bedeutet:

var/const Daten, die unter der nächstverfügbaren Adresse abgespeichert werden sollen.

Beschreibung: Der Inhalt der Variablen *var* oder die Konstante *const* wird in der nächstmöglichen Speicherzelle . Die Auswahl des Bausteins selbst erfolgt am I2C-Bus durch einen vorherigen Befehl. Die meisten Bausteine verfügen über eine Autoincrementfunktion.

Beispiel:

```
I2CINIT ra,2,3      REM I2C init.
```

START: I2CSP

```
  I2CST
```

```
  I2CWR 160,2          REM Adressiere Baustein und setze int. Zeiger
```

```
  I2CWRB var_a        REM Sekunde schreiben
```

```
  I2CWRB var_b        REM Minuten schreiben
```

```
  I2CWRB var_c        REM Stunden schreiben
```

```
  I2CSP
```

I2CWRITE

Syntax: I2CWRITE *adr1*, *adr2*, *var/const*

Funktion: Schreibt in einen I2C-Baustein mit der I2C-Adresse *adr1* an die Speicheradresse *adr2* den Inhalt der Variablen oder die Konstante ein.

Es bedeutet:

adr1 I2C-Bausteinadresse

adr2 Adresse der Speicherzelle innerhalb des Bausteins

var/const Daten, die unter der Adresse *adr2* abgespeichert werden sollen.

Beschreibung: Der Inhalt der Variablen *var* oder die Konstante *const* wird in der Speicherzelle mit der Adresse *adr2* abgespeichert. Die Auswahl des Bausteins selbst erfolgt am I2C-Bus durch dessen Bausteinadresse *adr1*. Obwohl die meisten Bausteine über eine Autoincrementfunktion verfügen, d.h. der interne Adresszähler wird bei jedem Lese- bzw. Schreibzugriff automatisch um eins erhöht, muss bei diesem Befehl die Adressangabe *adr2* unbedingt angegeben werden.

Beispiel:

I2CINIT ra,2,3 REM I2C init.

START: I2CWRITE 160,2 REM Sekunde beim RTC PCF 8583P
I2CREAD 160,var_a REM Sekunde lesen
I2CREAD 160,var_b REM Minuten lesen
I2CREAD 160,var_c REM Stunden lesen

....

....

WEITER: ...

Tritt ein Kommunikationsfehler auf, wird das Bit 3 in der ERR_-Variablen gesetzt.

IF...THEN...ELSE

Syntax:

```
IF var = vergleichsausdruck THEN adr  
IF var = vergleichsausdruck THEN adr1 ELSE adr2  
IF var,bit = 0 THEN adr1 ELSE adr2  
IF var,bit = 1 THEN adr1 ELSE adr2  
IF port,bit = 0 THEN adr1 ELSE adr2  
IF port,bit = 1 THEN adr1 ELSE adr2
```

Funktion: Bedingter Sprung, springe zur *adr* , wenn Bedingung wahr ist.

Es bedeutet:

port,bit(8-Bit) Port und Bitnummer 0 - 7
var,bit (8/16-Bit) Variable und Bitnummer 0 - 15
var(8/16-Bit) Variable mit dem zu vergleichenden Wert
vergleichsausdruck(8/16-Bit) Vergleichswert, kann eine Variable oder Konstante sein
*adr*Adresse (Label), zu der gesprungen wird

Beschreibung: IF vergleicht die angegebene Variable *var* nach einer bestimmten Funktion mit einem oder mehreren *vergleichsausdrücken* und springt, wenn der Vergleich positiv (wahr) ausfällt, auf die durch einen Label definierte *adr*.

Als Vergleichsoperatoren sind zulässig:

=	Gleichheit
<>	ungleich
<	kleiner als
>	größer als
<=	kleiner oder gleich
>=	größer oder gleich

Achtung: Bei Bitvergleichen ist nur der Operator '=' zulässig!!!
und die Bitnummer ist immer eine Konstante 0 oder 1.

Beispiel:

```
IF var_a <= 5 THEN LET var_b = 0 ELSE GOTO schleife  
IF RA,1=0 THEN GOTO schleife
```

Verboten ist:

```
IF var_a <= 5 THEN LET var_b = 0 ELSE var_b = 1: GOTO schleife
```

Bemerkung: Folgt auf THEN bzw. ELSE eine Anweisung, muss diese mit LET beginnen. Ebenso ist das GOTO oder GOSUB bei einer Verzweigung unbedingt notwendig. Auch hier gilt der Grundsatz: nur eine Anweisung bzw. Befehl für THEN bzw. ELSE .

Für Mehrfachverzweigungen steht der Befehl ON *var* GOTO bzw. ON *var* GOSUB zu Verfügung.

INC

Syntax: INC *var*.

Funktion: Erhöht den Inhalt der Variablen *var* um 1.

Beschreibung: Der Befehl DEC *var* erhöht den Inhalt der Variablen *var* um 1. Dieser Befehl wird codeoptimiert abgespeichert, im Gegensatz zu LET *var=var+1*. Während letzterer den Gültigkeitsbereich des Ergebnisses überprüft, nimmt der INC-Befehl darauf keine Rücksicht.

Beispiel:

START:

```
LET  var_a=10 REM Variable var_a auf 10 setzen
INC  var_a      REM in var_a steht jetzt 11
```

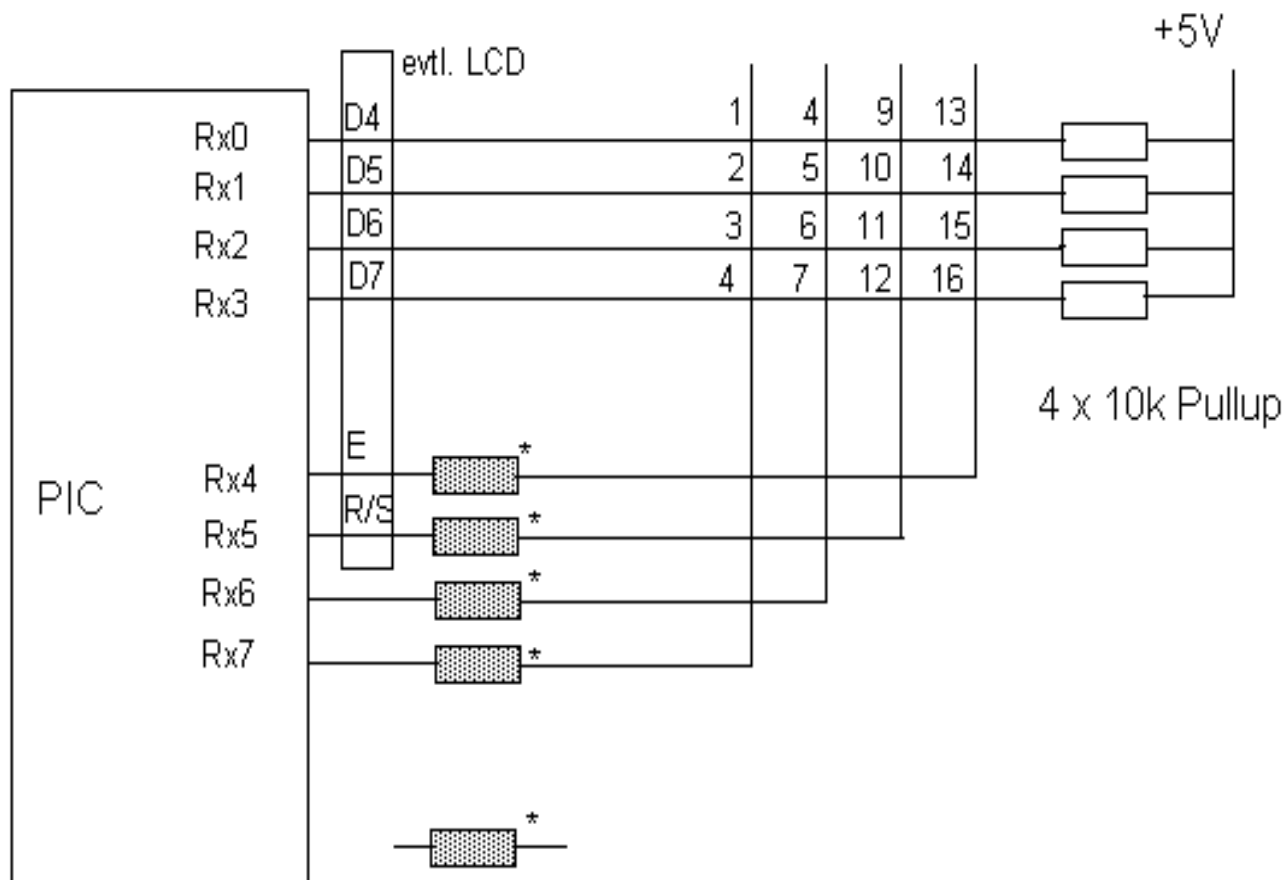

INKEY

Syntax: INKEY *var*

Funktion: Liest eine Tastenmatrix und weist ihr einen Wert zu (8-Bit Variable)

Beschreibung: Eine 4 x 4 Tastaturmatrix wird abgetastet und die gedrückte Taste der Variablen *var* übergeben. Diese Matrix wird nach einem festen Plan an den Port RB oder RC angeschlossen. Die Verbindungen sind so gewählt, dass an den gleichen Leitungen auch eine LCD angeschlossen werden kann. Allerdings reduziert sich die Matrix dann auf 3 x 4, da die E-Leitung des LCD-Controllers nicht für die Tastaturmatrix mitverwendet werden darf. Außerdem sind die Ausgangsleitungen mit Widerständen zu entkoppeln (siehe Skizze).

Der Befehl INKEY initialisiert die Portleitungen entsprechend seiner Funktion um. Am Ende werden die TRIS-Register wieder auf den ursprünglichen Wert gesetzt. Eine Restaurierung der Ausgangspegel erfolgt dagegen nicht.



* Entkopplungswiderstände falls LCD am gleichen Port arbeitet

Mit DEFINE KEYS wird der verwendete Port und eine Zeichenzuordnungstabelle definiert.

Beispiel:

```
DEFINE KEYS rb,0,"0","1","2","3","4","5","6","7","8","9",{+
    "A","B","C","D","E","F"
```

START:

```
INKEY var_a    REM Tastatur einlesen
IF var_a=0 THEN GOTO start    REM keine Taste gedrückt?
```

WEITER:

INKEY (cont.)

...

Bemerkung: siehe auch DEFINE KEYS

Bitte achten Sie darauf, dass INKEY nicht zu oft aufgerufen wird. Fügen Sie gegebenenfalls ein WAIT 10 ein.

INP

Syntax: INP *port,pin*

Funktion: Konfiguriert Pin als Eingang

Es bedeuten:

port I/O Port, kann eine Variable (8 Bit) oder Konstante sein

pin I/O Pin, kann eine Variable (8 Bit) oder Konstante sein

Beschreibung: Der INP Befehl programmiert das Datenrichtungsregister so um, dass der bezeichnete Pin zum Eingang wird. Dies schaltet den entsprechenden Ausgabetreiber ab und erlaubt dem Programm, jeden Status zu lesen, der an diesem Pin anliegt.

Beispiel:

```
INP ra.0      REM Pin 0 als Eingang
START:
  IF ra,0 = 1 THEN GOTO weiter    REM Falls Pin 0 = High,
  REM springe nach weiter
  GOTO start    REM Springe nach START
WEITER:
  ...
```

INPUT

Syntax: INPUT *port, var*

Funktion: Eingabe, 8-Bit Zugriff auf Port

Es bedeutet:

port/O Port, kann eine Variable (8 Bit) oder Konstante sein

var (8-Bit), eingelesener Wert, kann eine Variable oder Konstante sein

Beschreibung: Die Portzustände (alle 8 Bit) werden in die Variable *var* eingelesen. Das TRIS-Register wird nicht beeinflusst.

Beispiel:

```
                INP ra.0          REM Eingang Pin 0
START:
                INPUT  ra,var_a
                LET  var_a = var_a and 1          RM nur Bit 0 interessant
                IF var_a = 1 THEN GOTO weiter    REM Falls Pin 0 = High,
                REM  springe nach weiter
                GOTO start          REM Springe nach START
WEITER:
                ...
```

INTERRUPT

Syntax:

```
INTERRUPT label,pegel (,gied)  
INTERRUPT label,pegel,STACK = adr
```

Es bedeutet:

label Sprungadresse an der das Interruptunterprogramm (ISR) beginnt
pegel relevante Flanke im Fall eines RB0-Interrupts (manipuliert OPTION-Register), HL oder LH
gied schaltet vor jedem BASIC-Befehl die Interrupts aus, danach wieder ein
STACK definiert ein 16 Byte großes Datenfeld, wo die Inhalte der internen Variablen zwischengespeichert werden.

Funktion: Teilt dem Compiler mit, dass eine benutzerdefinierte Interruptroutine in die Interruptkette eingehängt werden soll.

Beschreibung: Die benutzerdefinierte Interruptroutine mit dem Einsprunghilfslabel *label* wird in die interne Interruptkette eingefügt. Die Angabe *pegel* legt die Flanke fest, bei der ein Interrupt erzeugt werden soll (Pin RB0). Die optionale Angabe *gied* umgeht das Problem des Reentrance. Das Reentrance Problem wird immer dann akut, wenn der Interrupt zu einem Zeitpunkt auftritt, wo das Programm gerade einen komplexen BASIC-Befehl abarbeitet und dazu interne Register verwendet. Trifft nun ein Interrupt ein, wird das laufende Programm (BASIC-Befehl) unterbrochen und die Interruptserviceroutine aufgerufen. Werden dann in dieser ISR ebenfalls BASIC-Befehle verwendet, die auf die internen Register zugreifen, ist der Konflikt natürlich vorprogrammiert, da am Ende der ISR die alten Inhalte dieser Register nicht mehr vorliegen und aufgrund des fehlenden Stacks auch nicht wiederhergestellt werden können. Die optionale Angabe *gied* sperrt nun vor jedem BASIC-Funktionsaufruf alle Interrupts und gibt sie am Ende wieder frei. Damit kann während der Abarbeitung eines BASIC-Befehls kein Interrupt behandelt werden. Falls in dieser Zeit aber ein Interrupt auftritt, wird dieser im INTCON-Register gespeichert und nach der Interruptfreigabe bearbeitet. Ein Überschreiben der gerade benutzten Register durch die ISR entfällt somit.

Beispiel 1:

```
define device 16c84  
xtal 4.19  
INTERRUPT intserv,hl,gied      REM intserv = Name des ISR  
REM hl = Fallende Flanke an RB0  
  
START:  
    SET  INTE      REM RB0-Interrupt freigeben  
    ....  
  
    SET  GIE      REM auch global freigeben  
  
LOOP:  
    ....  
  
    GOTO loop      REM Springe nach LOOP  
  
INTPROC  
INTSERV:  
    IF INTF=0 THEN GOTO INTSERVX  REM war nicht dieser Interr.  
    ...  
    ...  
    ...  
    RES  INTF      REM Interrupt-Flag zurücksetzen  
INTSERVX:  
    INTEND
```

GIED benötigt u.U. viel Speicherplatz, da ja vor und nach jedem BASIC-Befehl das GIE-Bit auf 0 bzw. auf 1 gesetzt wird. Eleganter ist es, alle Inhalte der internen Variablen, im Fall dass ein Interrupt auftritt, in einem eigenen Speicherbereich zwischenzuspeichern. Dazu wird statt GIED das Schlüsselwort STACK mit nachgestellter Startadresse verwendet. Die Startadresse ist die erste Speicherstelle eines 16 Byte großen Datenfeldes in der die Inhalte von ARG, ARG1 etc. abgelegt werden. Dieser Speicherbereich ist natürlich absolut tabu für andere Verwendungen.

INTERRUPT (cont.)

Beispiel 2:

```
define device 16F877
xtal 4.19
INTERRUPT intserv,hl,STACK = $80          REM Stackbereich von 80H bis 8FH

START:
    SET  RBIE          REM RB4-7 Interrupt freigeben
    ....

    SET  GIE            REM auch global freigeben

LOOP:
    ....

    GOTO loop          REM Springe nach LOOP

INTPROC

INTSERV:
    IF RBIF = 0 THEN GOTO INTSERVX  REM war nicht dieser Interr.
    ...
    ...
    RES  RBIF          REM Interrupt-Flag zurücksetzen

INTSERVX
    INTEND
```

Bemerkung: Der Befehl INTERRUPT kann nicht bei den Typen 10F2xx, 12C5xx, 12F5xx, 16C5x und 16F5x verwendet werden. (siehe auch Kapitel INTERRUPTS)

Da die Belegung des INTCON-Registers unterschiedlich sein kann, muss gegebenenfalls im Handbuch nachgesehen werden. Insbesondere dann, wenn es neben dem INTCON-Register auch ein PIE-Register gibt.

INTEND

Syntax: INTEND

Funktion: Beendet die Interruptserviceroutine.

Beschreibung: Die Interruptserviceroutine ISR wird beendet. Da diese als Unterprogramm aufgebaut ist, muss hier ein Return stehen. Das normale RET bzw. RETLW nimmt aber keine Rücksicht auf die Interrupts. Deshalb erzeugt INTEND ein RETFIE.

Beispiel:

```
define device 16c84
xtal 4.19
INTERRUPT intserv,hl,gied      REM intserv = Name des ISR
REM hl = Fallende Flanke an RB0

START:
.....

GOTO start      REM Springe nach START

INTPROC
INTSERV:
...
...
...
INTEND          REM Beendet ISR mit RETFIE
```

Bemerkung: Der Befehl INTEND kann nicht bei den Typen 10F2xx, 12C50x und 16C5x verwendet werden.

INTPROC

Syntax: INTPROC

Funktion: Teilt dem Compiler mit, dass die nachfolgende Programmsequenz bis zum INTEND zur Interruptbearbeitung verwendet wird.

Beschreibung: Identifiziert die nachfolgende Programmsequenz eindeutig als ISR, auf die sich der Compiler beim Übersetzungsvorgang beziehen kann.

Beispiel:

```
define device 16c84
xtal 4.19
INTERRUPT intserv,hl,gied      REM intserv = Name des ISR
REM hl = Fallende Flanke an RB0

START:
.....

GOTO start      REM Springe nach START

INTPROC      REM es folgt das ISR
INTSERV:
...
...
...
INTEND
```

Bemerkung: Der Befehl INTPROC kann nicht bei den Typen 10F2xx, 12C50x und 16C5x verwendet werden.

LCDCHAR

Syntax: LCDCHAR *nr, muster1, muster2, muster3, muster4, muster5, muster6, muster7, muster8*

Funktion: Schreibt ein Pixelmuster in die LCD.

Es bedeutet:

<i>nr</i>	Zeichennummer (0..7), Konstante oder Variable
<i>muster1</i>	ist das Bitmuster für die oberste Zeile dieses Zeichens, Konstante oder Variable
...	
<i>muster8</i>	ist das Bitmuster für die unterste Zeile dieses Zeichens, Konstante oder Variable

Beschreibung: Bei LCDs mit dem HD44780 Controller können 8 eigene Zeichen definiert werden. Diese liegen im Bereich der ASCII-Zeichen von 0 bis 7. Ein Zeichen besteht aus einer Pixelmatrix von 5 Punkten horizontal und 8 Punkten vertikal. Soll ein neues Zeichen definiert werden, so zeichnet man zuerst die Punktmatrix dieses Zeichens. Eine 1 wird den entsprechenden Punkt dunkel-(sichtbar), eine 0 hellsteuern (unsichtbar). Im Beispiel wird das ASCII-Zeichen 0 mit dem iL-Signet beschrieben und anschließen in Zeile 3 dargestellt.

Beispiel:

```
lcdwrite 1,1, "Test LCDCHAR"
lcdchar 0, $1f, $17, $1f, $05, $15, $15, $14, 0
lcdwrite 3,1,0
```

iL-Signet:

```
* * * * * %00011111 (1FH)
* * * * %00010111 (17H)
* * * * %00011111 (1FH)
* * %00000101 (05H)
* * * %00010101 (15H)
* * * %00010101 (15H)
* * %00010100 (14H)
* %00000000 (00H)
```

Die oberen 3 Bits in jedem Musterbyte sind 0, da es nur eine 5x8 Zeichenmatrix ist.

LDCDCLEAR

Syntax: LDCDCLEAR

Funktion: Löscht das angeschlossene LCD und setzt den Cursor auf Position 1,1

Beschreibung: Ein LCD-spezifischer Löschbefehl wird an die Anzeige übertragen

Beispiel:

START:

LDCDCLEAR

GOTO start REM Springe nach START

WEITER:

...

Bemerkung: Der Befehl LDCDCLEAR setzt das Vorhandensein der geeigneten Hardware voraus.

LCDCTRST

Syntax: LCDCTRST *n*

Funktion: Bei LCDs mit einem ST7036 wird die Kontrasteinstellung nicht über ein Potentiometer realisiert, sondern über einen entsprechenden Befehl an die LCD.

Es bedeutet:

n Zahlenwert zwischen 0 und 63 (muss Konstante sein)

Beschreibung: Ein LCD-spezifischer Befehl wird an die Anzeige übertragen um den Kontrast des Bildes einzustellen. Dieser Wert wird während der Initialisierungsphase an die LCD geschickt. Um die entsprechenden Bibliotheken für den ST7036-LCD-Controller zu laden, muss der Befehl LCDTYPE 19 bis LCDTYPE 21 ausgeführt worden sein.

Beispiel:

START:

```
LCDTYPE 19
LCDCTRST 31
LCDINIT RB,2,20
...
```

Bemerkung: Der Befehl LCDCTRST kann nur einmal im Programm verwendet werden.

LCDDELAY

Syntax: LCDDELAY *const*

Funktion: Erhöht die Verzögerungszeit bei den LCD-Befehlen.

Beschreibung: Bei den Befehlen LCDINIT, LCDCLEAR und LCDWRITE wird das BUSY-Bit nicht abgefragt. Stattdessen sorgt eine Verzögerung für das richtige Timing zwischen PIC und LCD. Laut Datenblatt sind diese Verzögerungswerte speziell bei den Befehlen LCDCLEAR und HOME starken Streuungen unterworfen. Der Compiler verzögert hier um ca. 1ms. Dies kann bei manchen Anzeigen zu kurz sein. Der Befehl LCDDELAY *const* gibt dem Compiler an, um welchen Faktor (1-9*) diese Zeit erhöht werden soll. Damit kann man die unterschiedlichsten LCDs und deren Spezifikationen berücksichtigen.

Beispiel:

START:

LCDDELAY 3 Die Verzögerung beträgt 3ms

GOTO start REM Springe nach START

WEITER:

...

* hängt von der Quarzfrequenz ab. Oft nur bis 6 oder 7 möglich (Compiler meldet Fehler)

LCDINIT

Syntax: LCDINIT *port,zeilen,spalten* (*,NBLK*) (*,CURSON*) (*,UPLINES*)

Funktion: Initialisiert eine angeschlossene LCD-Anzeige

Es bedeutet:

<i>port</i>	I/O Port, muss Konstante sein
<i>zeilen</i>	gibt die Anzahl der Zeilen auf der LCD an (1,2 oder 4)
<i>spalten</i>	gibt die Anzahl der Spalten auf der LCD an (8, 16, 20, 24, 32, 40)
<i>NBLK</i>	führende Nullen bei Dezimalzahlen werden angezeigt
<i>CURSOR</i>	schaltet den Cursor sichtbar
<i>UPLINES</i>	alternative Anschlussbelegung (siehe unten)

Beschreibung: Eine LCD wird gemäß den Angaben initialisiert. Die LCD muss über einen Controllerbaustein HD44780 angesteuert werden. Um den Bedarf an I/O-Leitungen zu minimieren, wird die Anzeige im 4-Bit-Modus angesteuert. Ebenso ist ein Rücklesen des LCD-RAMs nicht möglich. Die dazu vorgesehene Leitung R/W muss deshalb mit Masse verbunden sein.

Unterstützt werden folgende Anzeigeformate:

Zeilenzahl	Stellenzahl
1	8, 16, 20, 24, 32, 40
2	16, 20, 32, 40
4	16, 20

Die LCD kann entweder an PORT RB, RC oder RD angeschlossen werden. Dabei sind die Zuordnungen der I/O-Pins und Daten- bzw. Steuerleitungen der LCD fest vorgegeben.

Rx0	D4	(11)	(da im 4-Bit-Modus nur die oberen Leitungen verwendet werden)
Rx1	D5	(12)	
Rx2	D6	(13)	
Rx3	D7	(14)	
Rx4	(6)		
Rx5	R/S	(4)	
	GND	(1 u.5)	
	Vcc	(2)	
	Vo	(3)	(Kontrast)

Beispiel:

```
LCDINIT  RB,4,20      REM LCD initialisieren
```

START:

```
LCDWRITE 1,1,"Ing.Buero LEHMANN"
LCDWRITE 2,1,"Fuerstenbergstr. 8a"
LCDWRITE 3,1,"Telefon (07831) 452"
FOR var_a=0 to 255
    LET var_t=var_a * var_a      REM Quadrat bilden
    LCDWRITE 4,1,var_a          REM a als Dezimalwert ausge.
    LCDWRITE 4,15,var_t         REM Quadrat anzeigen
NEXT var_a
GOTO start      REM Springe nach START
```

WEITER:

...

LCDINIT (cont.)

Die alternative Anschlussbelegung!

Die LCD kann nun auch auf eine zweite bzw. dritte Art an einen Bus angeschlossen werden. Dadurch werden die Pins Rx0 und Rx1 frei, was vor allem beim RB-Port von großem Vorteil ist. Allerdings kann mit dieser Anschlussvariante kein Matrixtastenfeld parallel angeschlossen werden. Die Anschlussreihenfolge sieht dann so aus:

<u>UPLINES</u>	<u>UPLINESX</u>
Rx2 = E (Pin 6)	Rx2 = R/S (Pin 4)
Rx3 = R/S (Pin 4)	Rx3 = E (Pin 6)
restliche Belegung sowohl für UPLINES als auch UPLINESX	
Rx4 = D4 (Pin 11)	
Rx5 = D5 (Pin 12)	
Rx6 = D6 (Pin 13)	
Rx7 = D7 (Pin 14)	
(Pin 5 = RW muss auf GND liegen)	
(Pin 1 = GND)	
(Pin 2 = Vcc = +5V)	
(Pin 3 = Vo = Kontrast)	

Diese Anschlussvarianten werden durch das Anhängen des Schlüsselwortes UPLINES (upper lines) bzw. UPLINESX (upper lines crossover) realisiert.

Bsp:

LCDINIT rb,4,20,uplines

Bemerkung: Die TRIS-Register werden automatisch richtig eingestellt. Sie dürfen während des Programmlaufs aber nicht geändert werden.

Hinweis:

UPLINES wurde mit der Version 5.5-09 18.12.2003 eingeführt. Mit der Version 5.5-14 08.02.2004 kam UPLINESX hinzu. Dabei ist zu beachten, dass die bisherige Anschlussbelegung jetzt für UPLINESX gilt und für UPLINES eine neue Belegung hinzugefügt wurde (siehe Tabelle oben).

LCDTYPE n (,var)

Syntax: LCDTYPE n (,var)

Funktion: Wählt die Anschlussart und oder den eingebauten Controller der LCD aus

Es bedeutet:

n Konstante zwischen 0 und 4 sowie zwischen 16 und 21
var zusätzliche Variable im Bereich von Page 0, ist nur bei *n* =3 oder *n*=4 notwendig

Beschreibung: Vielfach wurde der Wunsch geäußert, eine LCD indirekt über einen Seriell-Parallel-Wandler (z.B. 74LS164) anzusteuern. Die unterschiedlichen Ansteuerungsarten werden durch das BASIC-Schlüsselwort LCDTYPE n eingestellt. Dabei gilt für die bisherige Ansteuerung, also nicht über den 74LS164, der Befehl LCDTYPE 0. Diese Anweisung wird per default ausgewählt und kann somit auch weggelassen werden.

LCDTYPE 1 entspricht der Angabe UPLINES in der LCDINIT-Anweisung. LCDTYPE 2 definiert UPLINESX. Man kann sowohl LCDTYPE 1 oder 2 also auch mit UPLINES oder UPLINESX arbeiten

Bei LCDTYPE 3,var und LCDTYPE 4,var wird das LCD nach folgendem Schema an den 74LS164 angeschlossen:

LCD Pin -->	74LS164 Pin	PIC	Versorgungsspannung
GND 1		Masse	
U+ 2		+5V	
Uo 3		Poti (Kontrast)	
RS 4	Q7 13		
R/W 5		Masse	
E 6		Enable	
D0 7	Q0 3		
D1 8	Q1 4		
D2 9	Q2 5		
D3 10	Q3 6		
D4 11	Q4 10		
D5 12	Q5 11		
D6 13	Q6 12		
D7 14		Masse	
	D1 1	Daten	
	D2 2	+5V	
	GND 7	Masse	
	CLK 8	Clock	
	MR 9	+5V	
	U+ 14	+5V	

Aufgrund dieser Ansteuerung ergeben sich für die Benutzung der LCD folgende Einschränkungen:

- Es sind nur ein- oder zweizeilige LCDs ansteuerbar. Die Zeichenzahl liegt bei 8, 16, 20, 24, 32 und 40 Zeichen.
- Der Cursor kann innerhalb der Zeile nicht frei platziert werden. Der Befehl LOCATE führt zu einem Syntaxfehler.
- Es muss immer LCDWRITE 1,1,... oder LCDWRITE 0,0,... geschrieben werden.
 Um also die Zeile 2 beschreiben zu können, muss Zeile 1 komplett beschrieben werden. Alle weiteren Zeichen werden auf Zeile 2 sichtbar.
- Die Ausgabe ist deutlich langsamer als bei LCDTYPE 0.
- Eine zusätzliche Variable ist notwendig.
- Ein Parallelschalten einer Matrixtastatur ist nicht möglich.
- Es kann nur der Zeichensatz von 0 bis 7FH genutzt werden. Somit sind keine Umlaute auf der LCD darstellbar.

LCDTYPE n (,var) (cont.)

LCDTYPE n muss immer vor LCDINIT stehen!

Diese Art der LCD-Ansteuerung benötigt eine zusätzliche Hilfsvariable. Diese muss auf Page 0 liegen und wird zusammen mit LCDTYPE definiert.

Bei LCDTYPE 3,var gilt: LCDINIT *port,clock,1,spalten*

Bei LCDTYPE 4,var gilt: LCDINIT *port,clock,daten,enable,1,spalten*

port legt den Port fest. *clock* legt die Leitung fest, die auf den Takteingang des 74LS164 führt. *daten* definiert die Leitung an Pin 1 des 74LS164 und *enable* ist die Leitung an den E-Anschluss der LCD (siehe auch LCDINIT).

Wird beispielsweise bei LCDTYPE 3,var *clock* auf 5 gelegt, dann sind *daten* auf 6 und *enable* auf 7 festgelegt. Diese feste Zuordnung entfällt bei LCDTYPE 4,var. Hier dürfen die drei Leitungen beliebig verteilt sein, allerdings immer auf dem selben Port. *clock*, *daten* und *enable* müssen Konstanten (8-Bit) sein.

Mit LCDTYPE 16 bis LCDTYPE 18 können LCDs mit einem KS0073 LCD-Controller angesprochen werden.

LCDTYPE 19 bis 21 ist für den LCD-Controller ST7036 ausgelegt. Bei diesem Controller ist der BASIC-Befehl LCDCTRST n mit n = 0 .. 63 nutzbar. Denn dieser LCD-Typ hat keine externe Einstellmöglichkeit für den Kontrast.

Es gilt:

16 bzw. 19 = Standardbeschaltung

17 bzw. 20 = Anschluss nach Schema UPLINES

18 bzw. 21 = Anschluss nach Schema UPLINESX

Hinweis:

LCDWRITE 0,0,.... beschreibt die LCD ab der Stelle weiter, an der der Cursor beim letzten Schreibvorgang zuletzt stand.

LCDWRITE

Syntax: LCDWRITE *y,x,"TEXT",varhex,\$,varascii,#,vardez,varbin,%,vardez,:2*

Funktion: Gibt Texte, Variablen oder Konstanten auf der LCD aus.

Es bedeutet:

y,x legt die Spalte und Zeile fest, ab der die Ausgabe auf der LCD erscheint

"TEXT" auszugebender Textstring

varhex,\$ Variable wird als Hexzahl ausgegeben

varascii,# Variable wird als ASCII-Zeichen an die LCD übertragen

varbin,% Variable wird als Binärzahl ausgegeben

vardez Variable wird dezimal ausgegeben

vardez,:2 Variable wird als Dezimalzahl mit 2 Nachkommastellen ausgegeben

Beschreibung: Die Ausgabe von Variablen erfolgt normalerweise als Dezimalzahl. Nach der Variablen kann ein Formatierungszeichen durch Komma getrennt angehängt werden. Für die hexadezimale Darstellung ist das Formatierungszeichen ein '\$', bei einer Binärzahl ein '%' und bei einem ASCII-Zeichen ein '#'. Bei der Dezimaldarstellung kann durch das Formatierungszeichen ':' die Zahl als Festkommazahl geschrieben werden. Dazu steht hinter dem Doppelpunkt die Anzahl der Nachkommastellen. Diese Angabe muss eine Konstante sein.

Die Stellenzahl ist immer fest; es gilt folgender Zusammenhang:

8-Bit Hexzahl 2-stellig

16-Bit Hexzahl 4-stellig

32-Bit Hexzahl 8-stellig

8-Bit Binärzahl 8-stellig

16-Bit Binärzahl 16-stellig

32-Bit Binärzahl nicht möglich

8-Bit Dezimalzahl 3-stellig (evt. 4-stellig)

16-Bit Dezimalzahl 5-stellig (evt. 6-stellig)

32-Bit Dezimalzahl 10-stellig (evt. 11-stellig)

Damit können Zahlen problemlos rechtsbündig untereinander geschrieben werden.

Beispiel:

```
LCDINIT RC,4,20      REM LCD initialisieren
```

START:

```
LCDWRITE 1,1,"Ing.Buero LEHMANN"      REM obere linke Ecke
LCDWRITE 2,1,"Fuerstenbergstr. 8a"     REM zweite Zeile, 1. Spalte
LCDWRITE 3,1,"Telefon (07831) 452"
FOR var_a=0 to 255
    LET var_t=var_a * var_a             REM Quadrat bilden
    LCDWRITE 4,1,var_a                  REM a als Dezimalwert ausge.
    LCDWRITE 4,15,var_t                 REM Quadrat anzeigen
NEXT var_a
GOTO start      REM Springe nach START
```

WEITER:

...

Bemerkung: Die TRIS-Register werden automatisch richtig eingestellt. Sie dürfen während des Programmlaufs aber nicht geändert werden.

Soll ab der aktuellen Cursorposition weiter geschrieben werden, muss für die Koordinatenangabe die Werte 0,0 stehen.

LCDWRITE (cont.)

Wichtige Hinweise für den Betrieb von LC-Anzeigen:

Die jahrelange Erfahrung beim Einsatz von LCDs hat gezeigt, dass die Streuungen dieser Baugruppen enorm groß ist. Sollten Sie beim Einsatz Probleme haben, versuchen Sie folgende Gegenmaßnahmen:

Problem 1:

LCD lässt sich nicht initialisieren bzw. es erscheint nichts auf dem Display.

Lösung:

Fügen den Befehl LCDDELAY ein. Falls dies bereits geschehen, erhöhen Sie dessen Wert.

Problem 2:

Nach einer bestimmten, z.T. unterschiedlich langen Zeit beginnt die Anzeige zu "spinnen". Dabei erscheinen sinnvolle und weniger sinnvolle Zeichen verteilt über das gesamte LCD.

Lösung:

Versuchen Sie auch hier zuerst mit Hilfe von LCDDELAY das Phänomen in Griff zu bekommen. Sollte das nicht möglich sein, so achten Sie bitte darauf, dass Sie die Anzeige in nicht allzu kurzen Zeitabständen beschreiben bzw. auffrischen. Dieses Problem kann behoben werden, wenn dieser Zyklus etwa alle 0,5 bis 1 Sekunde stattfindet.

Problem 3:

Es gibt eine Anzeige, die als 1 x 16 Zeichen verkauft wird. Tatsächlich ist auch der physikalische Aufbau derart gestaltet, dass 16 Zeichen in 1 Zeile liegen. Allerdings wurde hier ein IC auf der Platinenrückseite eingespart und das Multiplexverfahren geändert. Dadurch ergibt sich folgende Erscheinung: die ersten 8 Zeichen können ganz normal angesteuert werden. Die zweiten 8 Zeichen jedoch müssen so angesprochen werden, als lägen sie auf der 2. Zeile. Das Problem lässt sich also durch entsprechende Aufteilung des Ausgabestrings umgehen. Allerdings versagt die Methode, wenn Zahlen über diese Grenze hinweg angezeigt werden sollen.

LET

Syntax: LET *var* = *wert* [*operator wert*...]

Funktion: Weist einer Variablen einen (neuen) Wert zu.

Es bedeutet:

var (8/16-Bit) Variable mit dem Ergebnis der Zuweisung bzw. arithmetischen Operation

wert Operand

operator Operation (+, -, *, ...siehe weiter unten)

Beschreibung: Beim Befehl LET muss links des Gleichheitszeichens eine *var* stehen. Rechts des Gleichheitszeichens sind sowohl Variablen als auch Konstanten erlaubt. Dieses Schlüsselwort ist **obligatorisch** und darf deshalb nicht weggelassen werden.

Der Wert, der einer Variablen *var* zugewiesen werden soll, kann das Ergebnis einer komplexen mathematischen und/oder logischen Berechnung sein.

Mögliche Operatoren sind:

Addition	$a + b$	Berechnung erfolgt immer mit 16 Bit
Subtraktion	$a - b$	Genauigkeit. Das Ergebnis wird ent-
Multiplikation	$a * b$	sprechend der Zielvariablen über-
Division	a / b	nommen.
Modul	mod	$a \bmod b$ (modulo, Divisionsrest)
logisch UND	and	$a \text{ and } b$
logisch ODER	or	$a \text{ or } b$
logisch EXOR	xor	$a \text{ xor } b$

Beispiel:

```
LET Laenge = 5          REM ergibt Laenge = 5
  LET Breite = 3        REM ergibt Breite = 3
  LET Flaeche = Laenge * Breite    REM ergibt Flaeche = 15
  LET var_a = 3 + 5 / 2          REM ergibt den Wert var_a = 4
```

Bemerkung: Da keine Klammern erlaubt sind, berechnet der Basic-Compiler entgegen der üblichen Konvention jegliche Ausdrücke einfach der Reihe nach von links nach rechts.

LOCATE

Syntax: LOCATE y,x

Funktion: Positioniert den Cursor an der Stelle y,x auf der LCD.

Es bedeutet:

y,x legt die Spalte und Zeile fest, ab der die Ausgabe auf der LCD erscheint

Beschreibung: Setzt den Cursor an die Position x in der Zeile y. X und y dürfen die in LCDINIT definierten Werte nicht überschreiten. Ob ein Cursor sichtbar ist oder nicht, entscheiden die Befehle CURSON und CURSOFF.

Beispiel:

```

                                LCDINIT  RC,4,20                REM LCD initialisieren
START:
                                LCDWRITE 1,1,"Ing.Buero LEHMANN"  REM obere linke Ecke
                                LCDWRITE 2,1,"Fuerstenbergstr. 8a" REM zweite Zeile, 1. Spalte
                                LOCATE 3,1                      REM Cursor positionieren
                                LCDWRITE 0,0,"Telefon (07831) 452" REM ab der aktuellen Position schreiben
                                FOR var_a=0 to 255
                                  LET var_t=var_a * var_a        REM Quadrat bilden
                                  LCDWRITE 4,1,var_a              REM a als Dezimalwert ausge.
                                  LCDWRITE 4,15,var_t             REM Quadrat anzeigen
                                NEXT var_a
                                GOTO start                      REM Springe nach START
WEITER:
                                ...
```

LOFREQ

Syntax: LOFREQ *port, pin, freq, dauer*

Funktion: Erzeugt eine Frequenz für die Zeit *dauer* an *port, pin*

Es bedeutet:

port/O Port, kann eine Variable oder Konstante sein

pin/O Pin, muss eine Konstante sein

freq (8/16-Bit) Frequenz in Hz, muss eine Konstante sein

dauer (8/16-Bit) Länge der ausgegebenen Frequenz in Millisekunden, muss eine Konstante sein

Beschreibung: Der LOFREQ Befehl dient zur Ausgabe von Tönen an einem I/O-Pin der CPU.

Beispiel:

START:

```
LOFREQ ra.0,100,20          REM erzeugt für die Dauer von 20ms
REM einen Ton mit der Frequenz 100Hz
REM am Port ra.0
GOTO start
```

Bemerkung: Der Befehl LOFRQ ist auf 100 Hz bei einer Taktfrequenz der CPU von 4 Mhz optimiert. Einsatzgebiet ist speziell der Bereich für tiefere Frequenzen von 1 Hz - 2000 Hz. Dort liegt der Fehler knapp unter 1%. Bei 1000Hz Signalfrequenz beträgt der Fehler ca. 4,5% (957Hz) und bei 10000Hz Signalfrequenz beträgt der Fehler ca. 49% (6680Hz).

LOOKDN

Syntax: LOOKDN *var,zielwert,wert 1,wert 2,...,wert n*

Funktion: Sucht innerhalb einer Werteliste und Rückgabe der Position innerhalb der Liste

Es bedeutet:

var (8-Bit) Variable, zum Abspeichern der Position des gesuchten Wertes innerhalb der Liste, wenn dieser gefunden wurde

zielwert(8-Bit) Suchwert, kann eine Variable oder Konstante sein, nach deren Wert gesucht wird

wert1,wert2(8-Bit) eine Liste von Werten, in der gesucht werden soll, kann eine Variable oder Konstante sein

Beschreibung: Der Befehl LOOKDN vergleicht *zielwert* mit den Werten *wert1* , *wert2* usw. und übergibt bei Übereinstimmung der Variablen *var* die Position des Elementes. Ist keine Übereinstimmung vorhanden, wird der Inhalt von *var* nicht verändert.

Beispiel:

LET t = 0

START:

```
SERIN ra.0,9600,var_a      REM Empfange serielles Byte
LOOKDN var_b,var_a,65,88,93  REM Falls var_a =65 ist var_b = 0
REM Falls var_a =88 ist var_b = 1
REM Falls var_a =93 ist var_b = 2
REM Falls var_a <> 65 und 88 und 93 ist var_b =
REM unverändert
```

Bemerkung: Werden bei LOOKDN nur Konstanten als *wert1* , *wert2* usw angegeben, erzeugt der Compiler den speicherplatzsparenden Algorithmus mit den Befehlen RETLW xx. Kommt jedoch bei der Aufzählung der Werte eine Variable vor, wird ein anderer Algorithmus benutzt. Die Verwendung des RETLW-Befehls setzt voraus, dass der Programmcode in den ersten 255 Bytes des Programmspeichers steht. Falls dies nicht gewährleistet ist, sollte durch das Anfügen einer Dummy-Variablen der Compiler gezwungen werden, einen anderen Code zu erzeugen. Der Compiler überprüft diesen Umstand an dieser Stelle nicht.

Der Compiler legt die Tabellen automatisch so, dass sie soweit wie möglich am Programmanfang stehen. Dennoch darf die Gesamtzahl der Einträge (LOOKUP und LOOKDWN zusammen) nicht mehr als ca. 100 betragen.

LOOKUP

Syntax: LOOKUP *var,zeig,wert 1,wert 2,...,wert n*

Funktion: Suche innerhalb einer Werteliste und die Rückgabe des Wertes

Es bedeutet:

var (8-Bit) Variable, zum Abspeichern des gefundenen Wertes aus der Liste.

zeig (8-Bit) Position des Wertes innerhalb der Liste, kann eine Variable oder Konstante sein

wert1,wert2 (8-Bit) eine Liste von Werten, in der gesucht werden soll, kann eine Variable oder Konstante sein

Beschreibung: Der Befehl LOOKUP sucht den durch *zeig* angegebenen Wert in der nachfolgenden Liste. War die Suche erfolgreich, wird der Wert in der Variablen *var* gespeichert. Ist die Liste kürzer als die angegebene Position, wird die Variable *var* nicht verändert.

Beispiel:

START:

```
FOR var_a = 0 to 25
  LOOKUP var_b,var_a,65,66,67, ... REM Konvertiere Offset ( 0-25 ) zu ent-
  REM sprechendem ASCII-Zeichen ( A-Z )
NEXT var_a
```

Bemerkung: Es dürfen nur 8-Bit Variablen bzw. Konstanten verwendet werden. Es ist zu beachten, dass die Tabelle, falls sie nur aus Konstanten besteht, innerhalb der ersten 255 Adressen einer Seite liegen muss. Gibt es hier Probleme, hilft es, diese Anweisung weiter in Richtung Programmanfang zu verschieben.

Der Compiler legt die Tabellen automatisch so, dass sie soweit wie möglich am Programmanfang stehen. Dennoch darf die Gesamtzahl der Einträge (LOOKUP und LOOKDWN zusammen) nicht mehr als ca. 100 betragen.

LOW

Syntax: LOW *port,pin*

Funktion: Konfiguriert den entsprechenden *Port,Pin* als Ausgang und setzt ihn auf Low-Signal (0)

Es bedeutet:

port/O Port, kann eine Variable (8 Bit) oder Konstante sein

pin/O Pin, kann eine Variable (8 Bit) oder Konstante sein

Beschreibung: Durch den Befehl LOW wird der angegebene Pin auf Low-Signal gesetzt. Ist dieser Pin zum Zeitpunkt der Befehlsausführung als Eingang programmiert, wird er automatisch auf Ausgang umprogrammiert. Dieser Status bleibt auch nach der Ausführung des Befehls erhalten.

d.h. das TRIS-Register wird verändert.

Beispiel:

```
      LET var_a=1      REM Variable var_a = Pin 1
START: INP ra,var_a      REM Pin 1 ist Eingang
      IF var_a.0=1 THEN GOTO WEITER
      LOW ra,var_a      REM Pin 1 ist Ausgang und LOW
WEITER: ...
```

Bemerkung: LOW kann auch auf normale Variablen angewandt werden. Dann wird natürlich kein TRIS-Register verändert.

ON GOSUB

Syntax: ON *var* GOSUB *adr0,adr1,adr2,...,adr n*

Funktion: Springt zu dem Unterprogramm, dessen Adresse durch den Offset in *var* bestimmt wird

Es bedeutet:

var (8-Bit) Variable,

adr0, adr1 Adresse eines Unterprogramms, kann eine Variable oder Konstante sein

Beschreibung: Das Programm verzweigt zu dem Unterprogramm, deren Position in *var* steht. Hat *var* beispielsweise den Wert 2, wird die Programmausführung beim Unterprogramm mit dem Label *adr2* fortgesetzt. Nach dessen Abarbeitung kehrt das Programm zu dem Befehl zurück, der auf den ON ... GOSUB -Befehl folgt. Ist kein entsprechendes Sprungziel für den Wert in *var* vorhanden, wird der nachfolgende Befehl ausgeführt.

Beispiel:

START:

```
SERIN ra.0,9600,var_a          REM Liest seriellen Port
ON var_a GOSUB weiter6,weiter7,weiter8      REM Wenn var_a = 0, gehe zu weiter6
REM Wenn var_a = 1, gehe zu weiter7
REM Wenn var_a = 2, gehe zu weiter8
GOTO start

WEITER6:  ...                  REM Mache irgendetwas
RETURN

WEITER7:  ...                  REM Mache irgendetwas
RETURN

WEITER8:  ...                  REM Mache irgendetwas
RETURN
```

Bemerkung: Achten Sie auch hier darauf, dass beim 12C50x und 16C5x keine geschachtelten GOSUBs möglich sind. Bei den übrigen Prozessoren ist die max. Schachtelungstiefe 4 Ebenen tief.

ON GOTO

Syntax: ON *var* GOTO *adr0,adr1,...,adr n*

Funktion: Springt auf die Adresse, die durch den Inhalt von *var* angegeben ist

Es bedeutet:

var (8-Bit) Variable,

adr0, adr1 Adresse eines Labels, kann eine Variable oder Konstante sein

Beschreibung: Das Programm verzweigt zu der Programmarke, deren Position in *var* steht. Hat *var* beispielsweise den Wert 2, wird die Programmausführung beim Label *adr2* fortgesetzt. Ist kein entsprechendes Sprungziel für den Wert in *var* vorhanden, wird der nachfolgende Befehl ausgeführt.

Beispiel:

START:

```
SERIN ra.0,4800,var_a          REM Liest seriellen Port
ON var_a GOTO weiter6,weiter7,weiter8      REM Wenn var_a = 0, gehe zu weiter6
REM Wenn var_a = 1, gehe zu weiter7
REM Wenn var_a = 2, gehe zu weiter8
GOTO start
```

```
WEITER6:  ...          REM Mache irgendetwas
WEITER7:  ...          REM Mache irgendetwas
WEITER8:  ...          REM Mache irgendetwas
```

OUTP

Syntax: OUTP *port,pin*

Funktion: Konfiguriert Pin als Ausgang

Es bedeutet:

port/O Port, kann eine Variable (8 Bit) oder Konstante sein

pin/O Pin, kann eine Variable (8 Bit) oder Konstante sein

Beschreibung: Der OUTP Befehl programmiert das Datenrichtungsregister so um, dass der bezeichnete Pin zum Ausgang wird.

Bemerkung: Da das Ausgangsregister im RAM der CPU gepuffert ist, erscheint direkt nach dem OUTP Befehl derjenige Pegel am Pin, der zuletzt dort ausgegeben wurde. Wenn also mit dem Umschalten auf Ausgang sofort ein bestimmter Pegel generiert werden muss, ist es zwingend notwendig, diesen zuvor bereits zu definieren.

OUTPUT

Syntax: OUTPUT *port,var*

Funktion: Ausgabe, 8-Bit Zugriff auf Port

Es bedeuten:

port/O Port, kann eine Variable (8 Bit) oder Konstante sein

var (8-Bit) kann eine Variable oder Konstante sein

Beschreibung: Der Inhalt der Variablen *var* oder der Konstanten *var* wird auf dem angegebenen Port *port* ausgegeben. Der Zugriff erfolgt 8-Bit breit. Eine Veränderung des TRIS-Registers erfolgt nicht.

Beispiel:

START:

IF RA,0 = 1 THEN weiter

LET var_b=0

OUTPUT rb,var_b

GOTO start

WEITER:

OUTPUT rb,var_b

GOTO start

PEEK

(nicht für iL_TROLL)

(Möglichst nicht verwenden! Siehe Hinweis unten!)

Syntax: PEEK *var,adresse*

Funktion: direkter Zugriff auf eine Speicherzelle

Es bedeutet:

var(8-Bit) Variable

adresse (8-Bit) absolute Speicheradresse

Beschreibung: Der Inhalt der Speicherstelle mit der absoluten Adresse *adresse* wird in die Variable *var* übertragen.

Beispiel:

START:

LET var_a=\$1F

PEEK var_b,var_a

GOTO start

REM in b steht der Inhalt von \$1F

Hinweis:

Nur aus Gründen der Kompatibilität vorhanden; LET erlaubt die gleiche Funktionalität, da mittels DEFINE jeder Daten-Speicherplatz zugeordnet werden kann.

POKE

(nicht für iL_TROLL)

(Möglichst nicht verwenden! Siehe Hinweis unten!)

Syntax: POKE *adresse*, *wert*

Funktion: direkter Zugriff auf eine Speicherzelle

Es bedeutet:

adresse absolute Speicheradresse, Variable (8 Bit) oder Konstante

wert Variable (8 Bit) oder Konstante

Beschreibung: Die Variable oder Konstante *wert* wird in die Speicherstelle mit der absoluten Adresse *adresse* übertragen.

Beispiel:

START:

POKE \$1F, "A"

LET var_a=\$1F

PEEK var_b,var_a

GOTO start

REM in var_b steht der Inhalt von \$1F

Hinweis:

Nur aus Gründen der Kompatibilität vorhanden; LET erlaubt die gleiche Funktionalität, da mittels DEFINE jeder Daten-Speicherplatz zugeordnet werden kann.

PRINT (z.Z. nur

16F627/628, 16F818/819 und 16F87x)

Syntax: PRINT port,pin,baud, "text",var

Funktion: serielle Ausgabe von Texten und Variablen an z.B. den PC (zu Debugzwecken)

Es bedeutet:

<i>port</i>	Ausgabe von Port
<i>pin</i>	Ausgabe an Pin
<i>baud</i>	gibt Baus-Rate an
<i>"text"</i>	Ausgabe von Text, der zwischen den Anführungszeichen steht
<i>var</i>	(8/16/32-Bit) Ausgabe des Inhaltes der Variablen <i>var</i>

Beschreibung: Der Befehl PRINT hat eine vergleichbare Funktion wie der PRINT-Befehl im Standard-BASIC. Es kann ein Text gemischt mit Variablen und Steueranweisungen ausgegeben werden. Ohne eine weitere Angabe werden Variableninhalte dezimal ausgegeben. Ein \$-Zeichen nach der Variable gibt den Inhalt in hexadezimaler Darstellung aus, ein %-Zeichen in Binärdarstellung, ein '#' die Zahl als ASCII-Zeichen.

Beispiel:

PRINT rb,0,4800,var	REM sendet variable zum PC
PRINT rb,0,4800,b"text",var,\$	REM (%=binär, \$=hex,)
PRINT rb,0,4800,10,#,13,#	REM Ausgabe von Line Feed und Carrige Return

Bemerkung: Für die Umsetzung in das ASCII-Format benötigt der Compiler iL_BAS16 an dieser Stelle zusätzlichen Speicher. Für die Umsetzung nach Dezimal sind es die Speicherzellen 68H bis 6CH, nach Binär werden die Speicherzellen 69H bis 6CH benötigt. Die Umsetzung nach Hex erfordert nur den Speicherplatz 6CH. Für die Ausgabe von Textstrings und Konstanten werden keine zusätzlichen Speicher benötigt.

Es werden immer 10 Stellen (32-Bit) ausgegeben. Die führenden Nullen werden durch Leerzeichen ersetzt.

Zur Zeit können keine 32-Bit-Werte in Binärdarstellung ausgegeben werden!!!

ACHTUNG!

Um Kompatibilität zu Vorgängerversionen (bis ca. 2002) zu haben, wurden die damaligen Routinen der Laufzeitbibliothek wieder implementiert. Sie können mit PRINT_8_16 angesprochen werden. Sie erlauben nur die Ausgabe von 8- oder 16-Bit Variablen. Auch die Stellenanzahl die ausgegeben wird ist nicht fest sondern umfasst für 8-Bit Variablen 3 Stellen, für 16-Bit Variablen 5 Stellen.

Diese Routine benötigt weniger Speicher als der PRINT-Befehl, funktioniert aber nur im Zusammenhang mit den PIC 16F87x Typen. PRINT_8_16 sollte nur im Ausnahmefall verwendet werden, da keine Softwarepflege

PULSIN

ACHTUNG!!! Wurde verändert. Die Funktionsweise des 'alten' PULSIN-Befehls wird jetzt mit PULS_IN realisiert.

Syntax: PULSIN *port,pin,level,var [,TIMEOUT=x]*

Funktion: Erfasst die Länge eines Eingangsimpulses

Es bedeutet:

port/O Eingangs-Port, kann eine Variable (8-Bit) oder Konstante sein

pin/O Eingangs-Pin, kann eine Variable (8-Bit) oder Konstante sein

level(1, 0) Triggerflanke, mit der die Messung gestartet wird (1 = LO-HI; 0 = HI-LO)

var(8/16-Bit) Variable, zur Aufnahme des Messwertes

x (8/16-Bit) Timeout-Konstante (optional, default = 65535)

Beschreibung: Der Befehl PULSIN misst die Länge eines Eingangs-Impulses mit der Auflösung die durch den Quarz vorgegeben ist. Es erfolgt keine Umrechnung auf irgend eine Zeitbasis. Es handelt sich somit um relative, bezugslose Messwerte. Gestartet wird mit einer wählbaren Triggerflanke, beendet mit der hierzu inversen Flanke. Ist *level* = 0, dann startet der H-L-Übergang (fallende Flanke) die Messung, eine L-H-Flanke (steigende Flanke) beendet diese. Wird *level* durch eine Variable bestimmt, wird diese zuerst mit dem Wert 1 logisch UND verknüpft.

Beispiel:

REM misst die Frequenz an Pin RA0 (50Hz digitale Signalquelle einstellen) und gibt

REM die Abweichung an RB aus. 50Hz ergeben einen LOW-impuls von 10ms

```
TRIS ra = 255    REM Datenrichtungsregister auf Eingang
START:  PULSIN ra,0,1,var_z          REM Lese einen Impuls an ra.0 ein
.....
GOTO start
```

Bemerkung: Der benutzte *port,pin* muss vor dem ersten Aufruf des Befehls PULSIN als Eingang konfiguriert werden. Der Messwert kann in einer 8-Bit- oder einer 16-Bit-Variablen abgespeichert werden.

Kommt kein Impuls oder ist die Impulslänge zu groß, wird nach einer Timeout-Zeit das Bit 3 in ERR_ gesetzt.

Die optionale Timeout-Konstante gilt sowohl für die Impulspause als auch die Impulsbreite.

PULS_IN

ACHTUNG!!! Entspricht dem 'alten' PULSIN (VOR VERSION 4.1-00)

Syntax: PULS_IN *port,pin,level,var*

Funktion: Erfasst die Länge eines Eingangsimpulses

Es bedeutet:

port I/O Eingangs-Port, kann eine Variable oder Konstante sein
pin I/O Eingangs-Pin, kann eine Variable oder Konstante sein
level (1, 0) Triggerflanke, mit der die Messung gestartet wird (1 = LO-HI; 0 = HI-LO)
var (8/16-Bit) Variable, zur Aufnahme des Messwertes

Beschreibung: Der Befehl PULS_IN misst die Länge eines Eingangs-Impulses mit der Auflösung von 1us. Gestartet wird mit einer wählbaren Triggerflanke, beendet mit der hierzu inversen Flanke. Ist *level* = 0, dann startet der H-L-Übergang (fallende Flanke) die Messung, eine L-H-Flanke (steigende Flanke) beendet diese. Wird *level* durch eine Variable bestimmt, wird diese zuerst mit dem Wert 1 logisch UND verknüpft.

Beispiel:

REM misst die Frequenz an Pin RA0 (50Hz digitale Signalquelle einstellen) und gibt die
 REM Abweichung an RB aus. 50Hz ergeben einen LOW-impuls von 10ms

```

      TRIS rb = 0      REM Datenrichtungsregister auf Ausgang
START:  PULS_IN ra,0,1,var_z      REM Lese einen Impuls an ra.0 ein (16Bit)
      LET var_z = var_z+5      REM Runde auf
      LET var_z = var_z / 1000
      LET var_a = 0      REM Ergebnis gewichten
      IF var_z < 93 THEN LET var_a = 8
      IF var_z > 108 THEN LET var_a = 8
      IF var_a > 0 THEN GOTO weiter
      IF var_z < 96 THEN LET var_a = 4
      IF var_z > 104 THEN LET var_a = 4
      IF var_a > 0 THEN GOTO weiter
      IF var_z = 100 THEN LET var_a = 1 ELSE var_a = 2
WEITER:  OUTPUT rb,var_a      REM Gebe Ergebnis an rb aus
      GOTO start
```

Bemerkung: Der benutzte *port,pin* muss vor dem ersten Aufruf des Befehls PULSIN als Eingang konfiguriert werden. Der Messwert kann in einer 8-Bit- oder einer 16-Bit-Variablen abgespeichert werden. Bei Verwendung einer 8-Bit Variablen lassen sich nur Pulsbreiten bis 255 us erfassen, bei einer 16-Bit-Variablen hingegen 0,65535 s. Bei Verwendung einer 8-Bit-Variablen und einem Überlauf des Messwertes enthält diese das Low-Byte des internen 16-Bit Messwertes. Bei Überschreitung des Maximalwertes von 0,65535 s wird das Ergebnis 0 übergeben. Der Befehl ist auf einen Takt von 4 Mhz optimiert.

Der Eingangsimpuls mit dem Befehl PULSIN ist um den Faktor 10 kürzer, als der entsprechende Ausgangsimpuls mit dem Befehl PULSOUT.

PULSOUT

Syntax:

PULSOUT *port, pin, dauer, H*

PULSOUT *port, pin, dauer, L*

PULSOUT *port, pin, dauer*

Funktion: Ausgabe eines Rechteckimpulses für eine definierte Zeit

Es bedeutet:

port/O Ausgangs-Port, kann eine Variable (8-Bit) oder Konstante sein

pin/O Ausgangs-Pin, kann eine Variable (8-Bit) oder Konstante sein

dauer (16-Bit) Länge des Rechteckimpulses in Einheiten von 10 us, kann eine Variable oder Konstante sein

H gibt einenHigh (1) Impuls aus

L gibt einenLow (0) Impuls aus

(keine Angabe von H oder L gibt einen invertierten Impuls aus)

Beschreibung: Der Befehl PULSOUT gibt einen Rechteckimpuls mit der Auflösung von 10us aus. Der Impuls wird durch Invertieren des betreffenden Ausgangs erzeugt, weshalb der Pegel vor dem Befehl definiert sein muss. Es lassen sich Einzelimpulslängen mit einer Länge von 0 ... 0,65535 s erzeugen. Das TRIS-Register wird nicht verändert, d.h. falls dieser Pin als Eingang definiert ist, erscheint natürlich kein Impuls, obwohl das Programm einen Impuls erzeugt.

Beispiel:

REM Nicht nachtriggerbares Monoflop mit einer Impulslänge von 100ms. Die Impuls-

REM dauer wird mit einer Auflösung von 10us angegeben. Bei Zeiten unter 210us

REM (bei 4Mhz) machen sich interne Rundungsfehler stark bemerkbar

START:

HIGH rb.0

WEITER: IF ra.0 = 0 THEN GOTO weiter REM Warte bis ra.0 = 1

PULSOUT rb,0,10000,H REM Ausgabe Impuls von 100ms

REM nicht nachtriggerbares Monoflop

WARTE: IF ra.0 = 1 THEN GOTO warte

GOTO weiter

Bemerkung: Der Ausgangsimpuls mit dem Befehl PULSOUT ist um den Faktor 10 länger als der entsprechende Eingangsimpuls mit dem Befehl PULS_IN

PWM

Syntax: PWM *port,pin,pegel,dauer*

Funktion: Gibt einen Analogwert als Puls-breiten-moduliertes Signal für eine bestimmte Dauer aus

Es bedeutet:

port/O Ausgangs-Port, kann eine Variable (8 Bit, beim 18Fxxx 16-Bit) oder Konstante sein

pin/O Ausgangs-Pin, kann eine Variable (8 Bit) oder Konstante sein

pegel/Tastverhältnis, kann eine Variable (8-Bit) oder Konstante (8 Bit) sein

dauer/Dauer des Puls-breiten-modulierten Signals, kann eine Variable oder Konstante sein

Beschreibung: Der Befehl PWM erzeugt eine bestimmte Anzahl von puls-breiten-modulierten Signalen (Rechtecksignal mit einem entsprechenden Tastverhältnis) am Pin *port,pin* für die Dauer von *dauer*. *dauer* hat keinen absoluten Zeitbezug sondern ist die Anzahl der Perioden, die ausgegeben wird. Der angegebene Pin wird am Anfang der Routine als Ausgang geschaltet, am Ende (wieder) als Eingang (hochohmig) definiert.

Beispiel:

```
START:      SERIN ra.0,4800,var_a      REM  Empfange serielles Byte ( z.B. 128)

            PWM rb,0,var_a,20          REM  Ausgabe analoger Spannung ent-
            REM  sprechend dem empfangenen Byte
            REM  der Kondensator des RC-Gliedes
            REM  an rb,0 wird auf ca. 2,5V aufgeladen
            REM  ( 128 / 255 ) * 5V. Es werden in jedem
            REM  Aufruf 20 Ladezyklen generiert
            GOTO START
```

Bemerkung:

Der Wert für *pegel* muss im Bereich 1 bis 254 liegen. In Compilerversionen vor August 2007 musste diese Bereichsüberprüfung ggf. explizit erfolgen, falls *pegel* den Wert 0 oder 255 annehmen konnte. In den folgenden Compilerversionen erfolgt diese Bereichseingrenzung automatisch, d.h. aus einer 0 wird eine 1 und aus 255 wird 254.

Die PWM-Frequenz ist von der Quatzfrequenz direkt abhängig und beträgt ca. $2000 / f$. Bei einer Quatzfrequenz von 4 MHz beträgt die PWM-Frequenz ca. 500 us, bei 8 MHz somit ca. 250 us.

RANDOM

Syntax: RANDOM *var*

Funktion: Erzeugt eine Pseudozufallszahl

Es bedeutet:

var (8/16-Bit) Variable, enthält ermittelten Zahlenwert

Beschreibung: Der Befehl RANDOM erzeugt eine Zufallszahl in der genannten Variablen *var*. Als Startwert wird die Variable *var* verwendet. Dazu wählt RANDOM aus einer festen Reihe von 65535 Zufallszahlen gemäß der in der Variablen *var* abgespeicherten Position eine weitere aus. Wenn die Funktion mit dem gleichen Wert erneut aufgerufen wird, liefert sie nicht die gleiche Zahl zurück. Läuft der RTCC (Zähler), wird dieser zur Zahlengenerierung herangezogen. Durch das Lesen des momentanen Wertes wird eine "zufällige" Initialisierung erreicht. *var* kann sowohl 8-Bit- als auch 16-Bit Variable sein.

Beispiel:

START:

RANDOM var_s REM Generiert 16-Bit Zufallszahl

IF var_s < 2000 THEN GOTO tief

IF var_s < 10000 THEN GOTO hoch

GOTO start

TIEF: LOFREQ ra.0,var_s,20 REM Generiert zufälligen Ton

GOTO start

HOCH: SOUND ra.0,var_s,20 REM Generiert zufälligen Ton

GOTO start

Bemerkung:

Die Güte dieses Befehls hängt vor allem vom RTCC-Register ab. Falls der CLOCK-Befehl aktiv ist, gibt es nichts weiter zu beachten. Auch wenn der RTCC-Pin das RTCC-Register incrementiert und an diesem Pin eine Frequenz anliegt, arbeitet der Befehl sehr gut. Ansonsten sollten Sie das OPTION-Register so initialisieren, dass das RTCC-Register vom internen Befehlstakt inkrementiert wird. Dazu muss das T0CS-Bit auf Null gesetzt werden. Bei den 12C5xx-Bausteinen geschieht das einfach, indem in der DEFINE DEVICE der Schalter T0CS_INT angehängt wird.

RCTIME

Syntax: RCTIME *port, pin, pegel, var*

Funktion: Misst die Lade- bzw. Entladezeit eines RC-Gliedes.

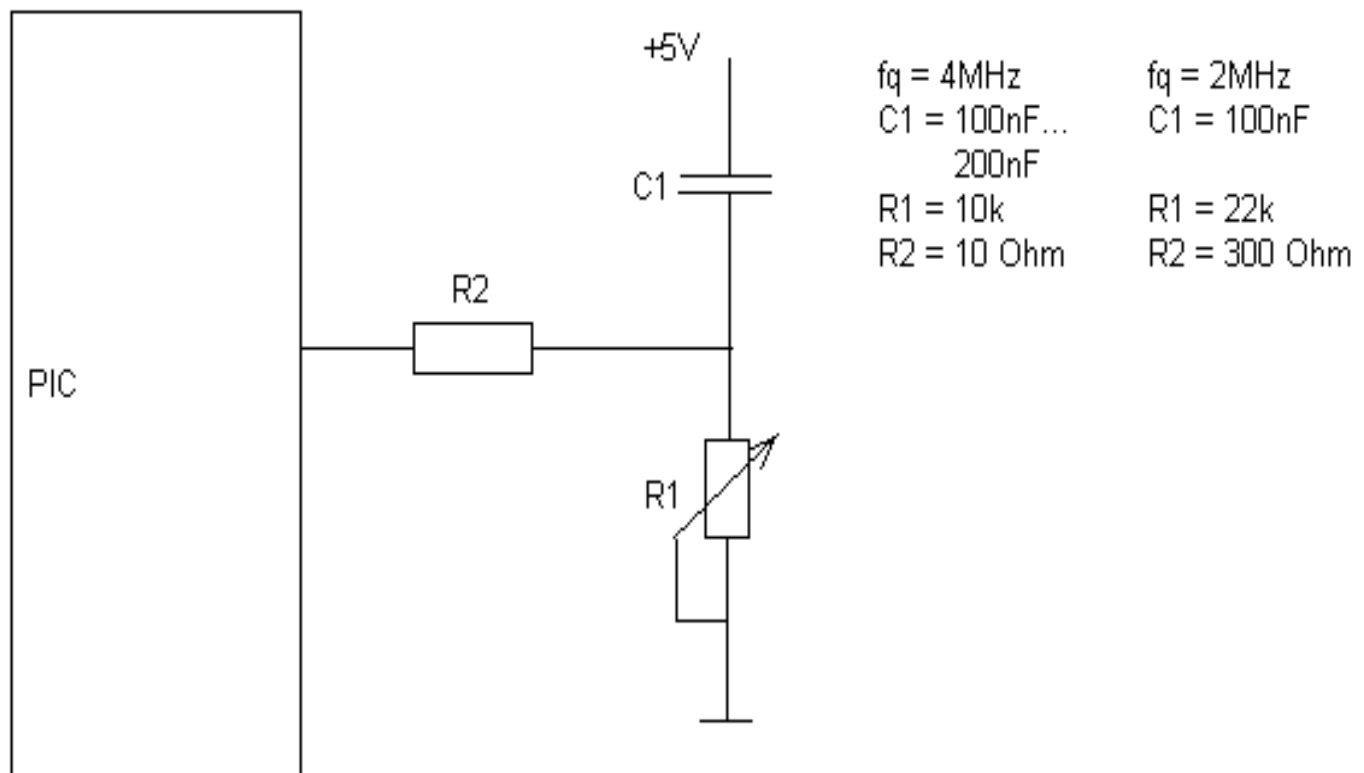
Es bedeuten:

port, pin Pin an der das RC-Glied angeschlossen wird

pegel Die Messung erfolgt, solange an dem Pin der Pegel *pegel* anliegt.

var (8 Bit) Variable, enthält den gelesenen Zahlenwert

Beschreibung: Misst die Lade- bzw. Entladezeit eines RC-Gliedes. Da die untere Schaltschwelle wesentlich niedriger liegt als die obere, empfiehlt es sich, das RC-Glied gemäß folgender Skizze aufzubauen.



Der 10-Ohm Widerstand dient als Kurzschlusschutz beim Entladen. Es gibt keinen Zeitbezug. Es hat sich gezeigt, dass für die meisten Anwendungen eine Angabe in us oder ms nicht notwendig ist. Sollte sie dennoch gebraucht werden, kann dies der Anwender selbst erledigen. Der Mehraufwand an Code und Zeit ist minimal.

Beispiel:

```
START:    HIGH ra.0      REM Kondensator laden
          WAIT 1         REM sicher laden
          RCTIME ra.0,1,var_a      REM messe Zeit bis Pegel von 1 auf 0 wechselt.
```

READDATA

Syntax: READDATA *var1* (, *var2*, *var3*,...).

Es bedeutet:

var1 8-Bit Variablen oder 8-Bit Konstanten

Funktion: Liest aus einem DATA-Feld den (die) nächsten Wert(e). Dieses Datenfeld darf bis zu 2048 Einträge beinhalten (speicherplatzabhängig).

Beschreibung: Oft werden in einem Programm Konstanten benötigt, die während des Programmlaufs in unterschiedlicher Reihenfolge den Variablen zugewiesen werden müssen. Mit DATA wird ein Feld angelegt, in dem bis zu 2048 Konstanten Platz finden. Die Elemente dieser Datentabelle können nur gelesen werden, da sie fester Bestandteil des Programmcodes sind. Der Zugriff erfolgt mit dem Schlüsselwort READDATA. Dabei wird ein interner Zeiger mitgeführt, der nach jedem Aufruf von READDATA inkrementiert wird. Für die Überwachung des Zugriffs auf das letzte Element, muss der Programmierer sorgen. Dieser Zeiger lässt sich durch den Befehl RESTORE manipulieren, so dass nicht nur sequentielle Zugriffe auf dieses Datenfeld möglich sind.

Beispiel:

START:

```
RESTORE      REM Lesezeiger auf Anfang
READDATA var_a      REM liest 1. Wert, hier 65
RESTORE label1      REM setzt Lesezeiger
READDATA var_a      REM liest 5. Wert, hier 12
READDATA var_b,var_c,var_d  REM liest die nächsten 3 Werte
```

```
GOTO start
DATA 65,66,67,"F"
```

label1:

```
DATA 12,45,32,17,25
```

Bemerkung: siehe auch DATA und RESTORE

(!! nicht für 10F2xx, 12C5xx und 16C5x !!)

READ

Syntax: READ *adr*,*var*

Funktion: Liest eine Speicherzelle aus dem EEPROM und speichert sie in *var*

Es bedeutet:

adr Adresse der gewünschten Speicherzelle, kann eine Variable oder Konstante sein
var (8-Bit) Variable, enthält den gelesenen Zahlenwert

Beschreibung: Der Befehl READ liest eine Speicherzelle aus dem 64 Byte EEPROM (nur 16C83, 16C84, 16F83, 16F84) bzw. 128 / 256 Byte beim 16F87x und speichert sie in der Variablen *var* ab.

Beispiel:

```
DEFINE ADR=$30 REM ADR to location 30H
DEFINE VALUE=$31 REM VALUE
DEFINE I1 = $32
```

START:

```
FOR I1=0 TO 63 REM only 64 bytes
LET ADR=I1 REM writes a value in each
LET VALUE=I1*2 REM memory cell
WRITE ADR,VALUE
NEXT I1
LET ADR=2 REM now read the third entry
READ ADR,VALUE REM into VALUE (must be 4)
```

Bemerkung: nur bei Bausteinen mit eingebautem Daten-EEPROM

REM

Syntax: REM oder ' (Hochkomma)

Funktion: Nachfolgender Text wird als Kommentar behandelt.

Beschreibung: Mit diesem Zeichen kennzeichnen Sie eine Bemerkung (engl.: REMark) in einer Programmzeile. Nachfolgende Zeichen werden vom Compiler als Kommentar betrachtet und deshalb einfach ignoriert.

Beispiel:

```
LET Laenge=5   REM in Laenge steht 5
```


RES

Syntax: RES

Funktion: Setzt das entsprechende Bit (Port oder Variable) auf 0. Nur für 8- und 16-Bit Werte bzw. Variablen.

Beschreibung: Im Gegensatz zum LOW-Befehl wird beim RES-Befehl das TRIS-Register nicht verändert. Damit ist dieser Befehl auch auf gewöhnliche Variablen anwendbar.

Beispiel:

START: RES var_a,0 REM Bit 0 in der Variable var_a wird gelöscht

RESTORE

Syntax: RESTORE (*label*).

Funktion: Setzt den Lesezeiger für den Befehl READDATA entweder auf den Anfang des Datenfeldes oder auf das 1. Element in einer durch *label* gekennzeichneten Zeile.

Beschreibung: Um die Daten aus einem Datenfeld nicht nur sequentiell lesen zu können, lässt sich mittels RESTORE der Lesezeiger für READDATA manipulieren.

Beispiel:

START:

```
RESTORE      REM Lesezeiger auf Anfang
READDATA var_a      REM liest 1. Wert, hier 65
RESTORE label_1      REM setzt Lesezeiger
READDATA var_a      REM liest 5. Wert, hier 12
READDATA var_b,var_c,var_d  REM liest die nächsten 3 Werte
```

```
GOTO start
DATA 65,66,67,"F"
```

label_1:

```
DATA 12,45,32,17,25
```

Bemerkung: siehe auch DATA und READDATA
(!! nicht für 10F2xx, 12C5xx und 16C5x !!)

RETURN

Syntax: RETURN

Funktion: Beendigung eines Unterprogramms und Rücksprung aus Unterprogramm ins Hauptprogramm.

Beschreibung: Das aufgerufene Unterprogramm muss durch den Befehl RETURN beendet werden. Nach dessen Abarbeitung kehrt das Programm zu dem Befehl zurück, der auf den GOSUB oder ON ... GOSUB -Befehl folgt.

Beispiel:

START:

```
FOR var_a = 0 TO 5
GOSUB LESE    REM Abfrage externes Signal
NEXT var_a
END
```

LESE:

```
IF RA.0 = 1 THEN GOTO MERKE
PULSOUT RA.1,5          REM kein Signal, Impuls ausgeben
RETURN
```

MERKE:

```
LET var_b = 1    REM Signal erkannt, merken
RETURN
```

Bemerkung: Basic-übliche Behandlung des Rücksprungs.

REVERS

Syntax: REVERS *port, pin*

Funktion: Definiert den entsprechenden Pin als Ausgang und invertiert das Signal

Es bedeutet:

port/O -Port, kann eine Variable (8) oder Konstante sein

pin/O -Pin, kann eine Variable (8) oder Konstante sein

Beschreibung: Der Befehl REVERS invertiert das Signal eines als Ausgang programmierten I/O-Pins. Ist der Pin als Eingang geschaltet, wird er automatisch zu einem Ausgang umprogrammiert. d.h. das TRIS-Register wird verändert.

Beispiel:

START:

```
LET var_a=6      REM Adresse von Port B
OUTP ra.0        REM Pin 0 ist Ausgang
OUTPUT var_a, 0   REM Pin 0 ist Ausgang, gebe 0 aus
REVERS var_a,0 REM Gebe 1 aus
REVERS var_a,0 REM Gebe 0 aus
```

Bemerkung: REVERS sollte nicht auf normale Variablen angewendet werden. Dazu ist der Befehl TOGGLE bestimmt.

SELFPROG

(nicht iL_TROLL)

Syntax: SELFPROG

Funktion: Ermöglicht das Umprogrammieren des Programmspeichers während des laufenden Betriebs. Beinhaltet auch die Debug-Funktionalität.

Beschreibung: Der Befehl SELFPROG wurde implementiert. Damit ist es bei den Bausteinen 16F818, 16F87x und 16F88 möglich, über eine serielle Leitung den Baustein neu zu programmieren. Dazu wird im unteren oder oberen Speicherbereich ein kleines SP_BIOS installiert. Das erfolgt automatisch mittels des Compilerschalters \$SELFPRG n,port,pin. Es ist jeder beliebige Digital-IO-Pin verwendbar. Dieser darf allerdings ausschließlich für diese Funktion verwendet werden und muss als Eingang definiert werden. Sollte bereits eine serielle Schnittstelle vorhanden sein, die von normalen Anwenderprogramm verwendet wird, kann diese *n i c h t* für die SELFPROG-Funktion mitverwendet werden. Der Wert n beim Schalter \$SELFPRG kann 0, 1 oder 2 sein.

0 richtet die SELFPROG-Funktion so ein, dass der Anwender selbst bestimmen kann, wann diese Funktion aktiviert werden soll. Dies erfolgt dann mit dem Befehl SELFPROG. Nachdem die Programmierung beendet wurde, wird die Adresse 0 (Kaltstart) angesprungen.

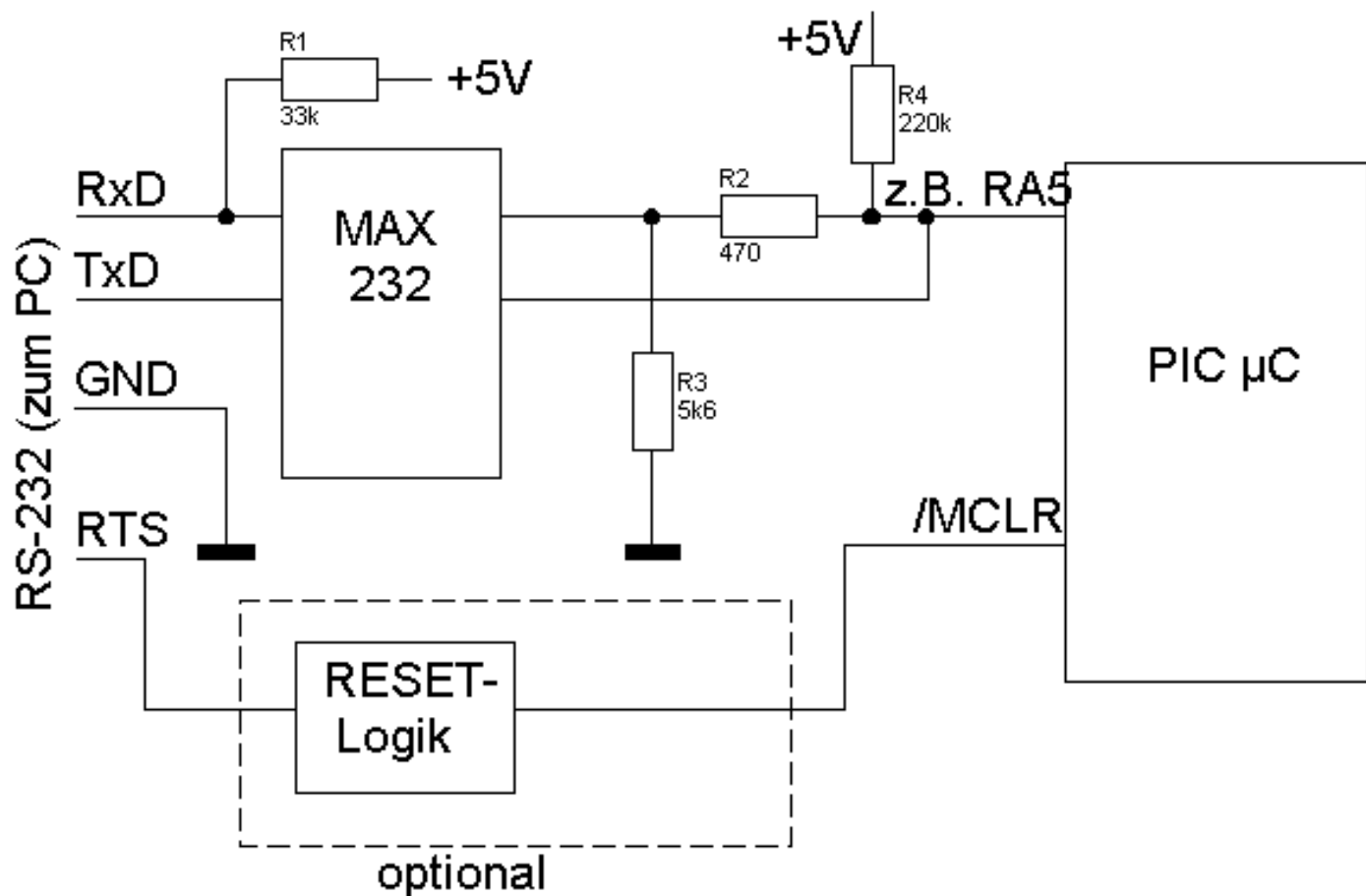
SELFPROG kann aber auch automatisch gestartet werden. Dazu muss der Wert n=1 sein. Der Compiler richtet die Funktion dann so ein, dass beim Einschalten des Prozessors (oder einem Reset, nicht vom Watchdog) ins SP_BIOS verzweigt und dort den Pegel der Schnittstelle abfragt. Ist dieser Pegel Low (über Pull-down Widerstand) wird das SP_BIOS sofort verlassen und das Anwenderprogramm gestartet. Ist der Pegel High, wartet das SP_BIOS auf Daten die in den Programmspeicher geschrieben werden. Das Anwenderprogramm wird nach der Programmierung sofort gestartet. Im Modus 2 arbeitet SELFPROG wie im Modus 0, nur dass nach erfolgter Programmierung ein Sprung auf den BASIC-Befehl erfolgt, der hinter SELFPROG steht.

Modus 3 und 4 arbeiten ähnlich den Modi 1 und 2, es wird jedoch das SPBIOS im Programmspeicher ab Adresse 0 bis FFH abgelegt. Diese Modi sind für solche Bausteine gedacht, die keine Möglichkeit haben den oberen Programmspeicher zu schützen (z.B. 16F877A).

Mode => Speicher => Einsprung => Rücksprung
0 => oben => SELFPROG => Adresse 0
1 => oben => nach Power-On => Anwenderprogramm
2 => oben => SELFPROG => hinter SELFPROG
3 => unten => nach Power-On => Anwenderprogramm
4 => unten => SELFPROG => hinter SELFPROG
Siehe auch \$SELFPRG

Die Datenübertragung hat das Format 9600,8,N. Das Einlesen und das Senden der Daten erfolgt auf dem gleichen IO-Pin. Das setzt voraus, dass die Hardware nach folgendem Schema aufgebaut sein muss.

SELFPROG (cont.)



Details sind unter
iL-TROLL
zu finden.

Damit wird aber grundsätzlich jedes empfangene Byte als Echo automatisch an den Sender zurückgesendet. Darauf muss bei der Entwicklung der entsprechenden Software Rücksicht genommen werden. Sobald das zu programmierende Wort im Flash-Programmspeicher steht, überträgt das SP_BIOS den, aus dem Speicher zurückgelesenen Wert (2 Byte = 1 Wort!). Das gesendete Wort wird somit zweimal vom Sender wieder empfangen.

Das Übertragungsprotokoll hat folgendes Format:

Es werden immer 16-Bit-Wert übertragen. Dabei wird zuerst das LOW- und anschließend das HIGH-Byte gesendet. Zuerst wird die Startadresse, danach die Anzahl der zu programmierenden Worte dem SP_BIOS mitgeteilt. Zu beachten ist, dass bei der Anzahl das High-Byte immer um 1 größer angegeben werden muss. Soll z.B. 1 Wort programmiert werden, gibt man für die Anzahl den Wert 0101H (257) an. Will man 256 Worte programmieren, so heißt der Wert 0200. Es folgen die eigentlichen Datenworte. Das 2. Echo des MSB wird erst nach erfolgter Programmierung zurückgesendet. Damit wird das Timing vom PIC vorgegeben.

In Modus 1 wird überprüft, ob die zu programmierende Anzahl ungleich Null ist. Falls dies so ist, wird der Einsprung ins SPBIOS ignoriert und das aktuelle Programm ausgeführt.

SERIN

Syntax:

```

SERIN  port,pin,baud,var 1,.var 2. ..., var n
SERIN  port,pin,baud,timeout=x,var 1,.var 2. ..., var n
SERIN  port,pin,baud,parity,var 1,.var 2. ..., var n
SERIN  port,pin,baud,parity,timeout=x,var 1,.var 2. ..., var n

```

Funktion: Einrichten eines seriellen Eingangs und Empfangen von Daten

Es bedeutet:

port I/O Eingangs-Port, kann eine Variable oder Konstante sein
pin I/O Eingangs-Pin, kann eine Variable oder Konstante sein
baud (300 - 9600*) legt Betriebsart des seriellen Kanals fest, kann eine Variable (siehe unten) oder Konstante sein
parity EVEN, EVEN7, ODD oder ODD7
timeout legt fest, nach wieviel vergeblichen Abfragezyklen diese Funktion abgebrochen werden soll (wird in ERR Bit 3 angezeigt)
var (8-Bit) Variable, enthält die empfangenen Daten

Beschreibung: An dem Angegebenen *port,pin* wird ein Zeichen mit der Geschwindigkeit von *baud* in *var* eingelesen. Das Programm verweilt in dieser Routine, bis ein Zeichen eingelesen wurde. Dazu wird auf das Startbit gewartet. Wird ein TIMEOUT definiert, wird nach einer bestimmten Anzahl von Abfragezyklen (x) abgebrochen. Dieser Abbruch wird in der ERR-Variable Bit 3 angezeigt, kann dort also abgefragt werden. Ein Abfragezyklus dauert ca. $10 \cdot 4/fq$; ist somit kein absoluter Zeitwert.

Bemerkung: Von den Parametern der seriellen Übertragung können lediglich die Baudrate und die Parität verändert werden.

Bei der SERIN-Softwareroutine wird bei 8E1 bzw 8O1 das Paritätsbit aber einfach ignoriert.

8 Datenbits, keine Parität, 1 Stopbit (8N1).

8 Datenbits, gerade Parität, 1 Stopbit (8E1).

8 Datenbits, ungerade Parität, 1 Stopbit (8O1).

7 Datenbits, gerade Parität, 1 Stopbit (7E1).

7 Datenbits, ungerade Parität, 1 Stopbit (7O1)

Bei einer Quarzfrequenz von 4 Mhz können Übertragungsgeschwindigkeiten bis zu 4800 Baud gewählt werden. Bei höheren Quarzfrequenzen entsprechend mehr, bei niedrigeren Frequenzen weniger.

Siehe auch SETBAUD

Achtung!

Was tun wenn der Parameter BAUD eine Variable ist?

Beim SERIN- und SEROUT-Befehl dürfen die Parameter PIN und BAUD auch Variablen sein. PIN muss eine 8-Bit-Variable, BAUD eine 16-Bit-Variable sein!. Der Parameter BAUD wird in eine Verzögerungszeit umgerechnet, die vom PIC-Kern, Quarzfrequenz und der Baudrate abhängig ist. Da diese Berechnung nicht zur Laufzeit erfolgen kann, muss sie der Programmierer im Vorfeld berechnen und statt der absoluten Baudrate diesen Wert als Parameter in der Variablen übergeben. Diese Berechnung führt der Compiler bei der Angabe der Baudrate als Konstante selbst durch. Das mitgelieferte Programm BAUDCALC.EXE übernimmt diese Arbeit da die Formeln kompliziert und nicht linear sind.

Bsp.

```

LET baud=329  'bei 6,144 MHz und 4800 Baud
LET baud=407  'bei 6,144 MHz und 2400 Baud

```

SERIN (cont.)

```
LET pin=2  
LET wert="A"  
SEROUT rb,pin,baud,wert
```

Interruptbetrieb:

Will man den SERIN-Befehl mittels Interruptroutinen ersetzen, dann muss man natürlich auch die entsprechenden Interrupt-Freigabebits setzen. Allen voran das GIE und das RCIE. Bei manchen PICs muss man auch noch das PEIE-Bit setzen, da hier die UART-Interrupts zuerst mit anderen Interrupts zu dem "Peripheral Interrupt" zusammengeführt werden.

*) Obergrenze der Baudrate hängt von der Taktfrequenz ab!

SEROUT

Syntax: SEROUT *port,pin,baud,var1, var2,...*
 oder SEROUT *port,pin,baud,parity,var1,var2,...*

Funktion: Einrichten eines seriellen Ausgangs und Ausgabe von Daten

Es bedeutet:

port/O Ausgangs-Port, kann eine Variable oder Konstante sein

pin/O Ausgangs-Pin, kann eine Variable oder Konstante sein

baud(300 - 9600*) legt Betriebsart des seriellen Kanals fest,
 kann eine Variable (siehe unten) oder Konstante sein

parity EVEN, EVEN7, ODD oder ODD7

var(8-Bit) Variable, enthält die auszugebenden Daten

Beschreibung: Die angegebene(n) Variable(n) *var 1, var 2,...,var n* wird/werden seriell über den entsprechenden Pin *port,pin* mit der angegebenen Übertragungsgeschwindigkeit *baud* ausgegeben.

Beispiel:

START:

```
SERIN ra.0,4800,var_b      REM  Lies seriell mit 4800 Baud
LET var_b = var_b - 1
SEROUT rb.0,2400,var_b    REM  Sende den Inhalt von b mit
REM  2400 Baud
```

Bemerkung: Bei den Übertragungsparametern kann nur die Baudrate und die Parität verändert werden.

8 Datenbits, keine Parität, 1 Stopbit (8N1).

8 Datenbits, gerade Parität, 1 Stopbit (8E1).

8 Datenbits, ungerade Parität, 1 Stopbit (8O1).

7 Datenbits, gerade Parität, 1 Stopbit (7E1).

7 Datenbits, ungerade Parität, 1 Stopbit (7O1).

Bei einer Quarzfrequenz von ca. 4 Mhz können Übertragungsgeschwindigkeiten bis 4800 Baud gewählt werden. Bei 6,144MHz sind bereits 9600 Baud möglich, mit höheren Quarzfrequenzen entsprechend mehr.

Siehe auch SETBAUD

Achtung!

Was tun wenn der Parameter BAUD eine Variable ist?

Beim SERIN- und SEROUT-Befehl dürfen die Parameter PIN und BAUD auch Variablen sein. PIN muss eine 8-Bit-Variable, BAUD eine 16-Bit-Variable sein!. Der Parameter BAUD wird in eine Verzögerungszeit umgerechnet, die vom PIC-Kern, Quarzfrequenz und der Baudrate abhängig ist. Da diese Berechnung nicht zur Laufzeit erfolgen kann, muss sie der Programmierer im Vorfeld berechnen und statt der absoluten Baudrate diesen Wert als Parameter in der Variablen übergeben. Diese Berechnung führt der Compiler bei der Angabe der Baudrate als Konstante selbst durch. Das mitgelieferte Programm BAUDCALC.EXE übernimmt diese Arbeit da die Formeln kompliziert und nicht linear sind.

Bsp.

```
LET baud=329  'bei 6,144 MHz und 4800 Baud
LET baud=407  'bei 6,144 MHz und 2400 Baud
LET pin=2
```

SEROUT (cont.)

```
LET wert="A"  
SEROUT rb,pin,baud,wert
```

*) Obergrenze der Baudrate hängt von der Taktfrequenz ab!

SET

Syntax: SET

Funktion: Setzt das entsprechende Bit (Port oder Variable) auf 1. Nur für 8- und 16-Bit Werte bzw. Variablen.

Beschreibung: Im Gegensatz zum HIGH-Befehl wird beim SET-Befehl das TRIS-Register nicht verändert. Damit ist dieser Befehl auch auf gewöhnliche Variablen anwendbar.

Beispiel:

START:

```
SET var_A,0    REM Bit 0 in der Variable var_A wird gesetzt
```

SETBAUD

(nur bei UART-Betrieb)

Syntax: SETBAUD *baud*

Funktion: Ändern der Baudrate bei Interruptbetrieb

Es bedeutet:

baud(300 - 9600*) legt Baudrate fest (Konstante)

Beschreibung:

Dieser Befehl erlaubt es im Zusammenhang mit der Hardware-Serin-Routine im Interruptbetrieb die Empfangsbaudrate einzustellen. Normalerweise wird beim Datenempfang mittels Interrupt erst dann zum Befehl SERIN (in ISR) verzweigt, wenn das 1. Byte bereits empfangen wurde. Hatte man zuvor ein Byte über den Befehl SEROUT (UART) mit einer anderen Baudrate gesendet, würde somit das erste empfangene Zeichen falsch sein. SETBAUD schafft hier Abhilfe und definiert z.B. nachdem man den letzten Wert per SEROUT ausgegeben wurde. Zur Sicherheit überprüft SETBAUD zuerst, ob der Sendepuffer leer ist, denn nur dann darf die Baudrate verändert werden.

Beispiel :

START:

SETBAUD 9600

*) Obergrenze der Baudrate hängt von der Taktfrequenz ab!

SLEEP

Syntax: SLEEP *dauer*

Funktion: Schaltet die CPU für mehrere Sekunden in den SLEEP-Mode

Es bedeutet:

dauer (8/16-Bit) Zeitdauer der Sleep-Phase in Sekunden, kann eine Variable oder Konstante sein

Beschreibung: Die CPU geht für (*dauer* * 2,3) Sekunden in den Sleepmodus, um Strom zu sparen. Dabei erfolgt nach ca. 2,3sec ein Timeout des Watchdogs. Darauf hin wird ein Reset ausgeführt. Eine Softwareroutine prüft, ob die Gesamtzeit abgelaufen ist. Ist dies noch nicht der Fall, wird der Prozessor erneut in den Sleepmodus versetzt. Ist die Zeit *dauer* abgelaufen, wird das Programm mit dem Befehl, der auf den SLEEP-Befehl folgt, fortgesetzt. Wird der PIC12C50x oder PIC16C5x verwendet, entstehen alle 2,3s an allen Portausgängen kurze Impulse, da die CPU durch einen Reset aufgeweckt wird. Dieser Reset initialisiert die Trisregister auf Eingang.

Beispiel:

START:

```
SLEEP 1565      REM  Schlafe für ca. 1 Stunde, Stromsparen
REM 3600 / 2.3 = 1565
...
GOTO xyz        REM  Hier geht's nach einer Stunde weiter
```

Bemerkung: *dauer* kann Werte zwischen 1 und 65535 annehmen, weshalb die SLEEP-Phase bei maximal 41,87 h liegt. Die zeitliche Auflösung ist etwa 2,3 s.

SLEEP verwendet den Watchdog-Timer (muss daher aktiviert sein) und den Vorteiler. Falls gleichzeitig CLOCK aktiv ist, wird der Vorteiler zum Watchdog umgeschaltet.

Wichtiger Hinweis:

Bei den PIC 12C5xx und PIC 16C5x, also alle mit 12-Bit Kern, sollte der SLEEP-Befehl ganz weit am Programmanfang stehen.

(genau: innerhalb der ersten 256 Befehlen).

SOUND

Syntax: SOUND *port,pin,frq,dauer*

Funktion: Erzeugt eine Frequenz für die Zeit *dauer* an *port,pin*

Es bedeutet:

port/O Port, kann eine Variable oder Konstante sein

pin/O Pin, muss eine Konstante sein

frq(8/16-Bit) Frequenz in Hz, muss eine Konstante sein

dauer(8/16-Bit) Länge der ausgegebenen Frequenz in Millisekunden, muss eine Konstante sein

Beschreibung: Der SOUND Befehl dient zur Ausgabe von Tönen an einem I/O-Pin der CPU.

Beispiel:

START:

```
SOUND ra.0,10000,20          REM erzeugt für die Dauer von 20ms
REM einen Ton mit der Frequenz 10kHz
REM am Port ra.0
GOTO START
```

Bemerkung: Der Befehl SOUND ist auf 5kHz bei einer Taktfrequenz der CPU von 4 Mhz optimiert. Einsatzgebiet ist speziell der Bereich für höhere Frequenzen von 1000 Hz - 10000 Hz. Dort liegt der Fehler knapp unter 1%. Bei 1000Hz Signalfrequenz beträgt der Fehler ca. 6% und bei 10000Hz Signalfrequenz beträgt der Fehler ca. 7%.

SWAP

Syntax: SWAP *var*

Funktion: Vertauscht das obere und untere Halbbyte in einer 8-Bit Variablen *var*

Beschreibung: Der Befehl SWAP vertauscht das obere und untere Halbbyte (Nibble) der Variablen *var*. Dieser Befehl ist vor allem bei der BCD-Interpretation eines Bytes notwendig.

Beispiel:

START:

```
LET var_a=$6  
SWAP var_a    REM in a steht jetzt $60
```

TXDDELAY

Syntax: TXDDELAY *const*

Funktion: Fügt am Ende der Senderoutine für 1 Byte eine Verzögerungsschleife ein.

Es bedeutet:

const 8-Bit Konstante (Defaultwert ist 125)

Beschreibung: Falls der Datenempfänger die Daten nicht so schnell übernehmen kann, wie sie der PIC sendet, reicht es oftmals aus, wenn eine Zeit von etwa der halben Breite eines Bytes gewartet wird, bevor das nächste Byte gesendet wird. Dazu wird vor dem Senden des Bytes geprüft, ob der Sendepuffer leer ist. Falls dies der Fall ist, wird noch zusätzlich eine Verzögerungsschleife eingefügt. Diese Zeit hat keinen absoluten Zeitbezug, sondern besteht aus einer einfachen Zeitschleife. Bei 4 MHz und 9600 Baud führt ein Verzögerungswert von 125 zu einer Zeit von etwa 500us. Was meistens ausreicht.

Beispiel:

```
TXDDELAY 255 REM Maximal mögliche Verzögerung
```

START:

```
SEROUT rc,6,9600,temp REM Variable TEMP wird gesendet
```

Bemerkung: TXDDELAY nur sinnvoll in Verbindung mit der USART im PIC. Bei der Implementierung einer Softwarelösung für den SEROUT-Befehl ist der Prozessor ja bis zum Schluss beschäftigt, so dass immer eine gewisse Zeit vergeht, bis das nächste Zeichen gesendet wird.

TOGGLE

Syntax: TOGGLE *port,pin*

Funktion: Invertiert das Signal eines als Ausgang definierten Pins

Es bedeutet:

port/O -Port, kann eine Variable (8) oder Konstante sein

pin/O -Pin, kann eine Variable (8) oder Konstante sein

Beschreibung: Der Befehl TOGGLE invertiert das Signal eines als Ausgang programmierten I/O-Pins. Das TRIS-Register wird nicht verändert.

Beispiel:

START:

```
LET var_a=6      REM Adresse von Port B
OUTP ra.0        REM Pin 0 ist Ausgang
OUTPUT var_a, 0   REM Pin 0 ist Ausgang, gebe 0 aus
TOGGLE var_a,0    REM Gebe 1 aus
TOGGLE var_a      REM Gebe 0 aus
```

Bemerkung: TOGGLE kann auch auf normale Variablen angewendet werden, da kein TRIS-Register verändert wird.

TRIS

(eingeschränkt bei iL_TROLL)

Syntax: TRIS *port,wert*

Funktion: Datenrichtungsregister von *port* wird mit *wert* geladen (8-Bit Zugriff)

Es bedeutet:

port/O -Port, kann eine Variable oder Konstante sein

wert(8-Bit) Richtung der Kommunikation (1=Eingang; 0=Ausgang), kann eine Variable oder Konstante sein

Beschreibung: Das Datenrichtungsregister *port* (RA, RB, RC, RD, RE, GPIO) wird mit dem Wert *wert* geladen. Dabei korrespondiert das niederwertigste Bit mit der niederwertigsten I/O-Leitung. Wird an der entsprechenden Stelle eine 1 geschrieben, wird dieser Pin als Eingang, bei einer 0 als Ausgang geschaltet.

Beispiel:

```
LET var_a=%00001111      REM Bit 0-3 von Port RA werden als Eingänge,  
REM Bit 4-7 als Ausgänge geschaltet  
LET var_b=5  
TRIS var_b,var_a
```

Bemerkung: Der Befehl TRIS ist ein byteorientiertes Kommando.
iL_TROLL kennt den Port RA nicht.

VARPTR

(Möglichst nicht verwenden! Siehe Hinweis unten!)

Syntax: `VARPTR zeiger, var`

Funktion: Ermittelt die Speicheradresse der Variablen `var` und hinterlegt in `zeiger` (8-Bit Zugriff)

Es bedeutet:

`zeiger` hier steht dann die Adresse der Variablen `var` , `zeiger` muss eine 8-Bit Variable sein
`var` Variablenname deren Adresse ermittelt werden soll

Beschreibung: Manchmal ist es von Vorteil, wenn man die Adresse kennt, in der eine Variable seine Werte ablegt. Damit kann dann auf diese Variable mittels Zeiger, also indirekter Adressierung, zugegriffen werden.

Beispiel:

```
DEFINE var8 = $25 as byte
DEFINE zeiger= $30 as byte
DEFINE wert = $31 as byte
...
LET var8=$85           REM in var8 steht jetzt der Wert $85
VARPTR zeiger,var8     REM zeiger kennt die Adresse von var8
PEEK wert,zeiger       REM in Wert steht jetzt $85
```

Bemerkung: Nur für Speicherbereich 00H bis 7FH (Bank 0) geeignet!

Hinweis:

Nur aus Gründen der Kompatibilität vorhanden; LET erlaubt die gleiche Funktionalität, da mittels DEFINE jeder Daten-Speicherplatz zugeordnet werden kann.

Die indirekte Adressierung wird über Feldvariablen realisiert.

WAIT

Syntax: WAIT *msec*

Funktion: Unterbricht die Programmausführung für eine bestimmte Zeit

Es bedeutet:

msec (8/16-Bit) Definiert die Unterbrechungszeit in ms, kann eine Variable oder Konstante sein

Beschreibung: Der Befehl WAIT unterbricht das laufende Programm für die angegebene Zeit, ohne jedoch in den Sleep-Modus zu gehen. Der Stromverbrauch wird bei diesem Befehl nicht reduziert. Die Genauigkeit der Wartezeit ist abhängig von der Qualität der Zeitbasis (Quarz, Keramik-Resonator). Bei einer kurzen Wartezeit bewirkt die Zeit für den Funktionsaufruf einen zusätzlichen Fehler.

Beispiel:

START:

```
WAIT 1000      REM Schlafe für ca. 1 s,  
REM ohne Stromsparen  
...  
GOTO xyz      REM Hier gehts nach einer Sekunde weiter
```

Bemerkung: Der Bereich der Wartezeit liegt zwischen 1 ... 65535 Millisekunden (ca. 65,5 s, Endwert hängt von der Quarzfrequenz ab.).

WRITE

(nur für PIC-Bausteine mit internem Daten-EEPROM)

Syntax: WRITE *adr,var*

Funktion: Beschreibt eine Speicherzelle im EEPROM

Es bedeutet:

adr (8-Bit) Adresse der gewünschten Speicherzelle, kann eine Variable oder Konstante sein

var (8-Bit) Variable, enthält den zu schreibenden Zahlenwert

Beschreibung: Der Befehl WRITE schreibt an die angegebene Adresse *adr* entweder den Inhalt der Variablen *var* oder den Wert (Konstante). Der EEPROM-Datenbereich hat eine Größe von bis zu 256 Byte. Da der Schreibvorgang ca. 20 msec dauert, wird vor dem Einschreiben geprüft, ob gerade ein Schreibvorgang läuft. Dadurch, dass diese Überprüfung am Anfang des neuen Schreibzykluses erfolgt, kann das Programm nach der Triggerung des Schreibvorganges sofort mit der Abarbeitung des nächsten Befehls fortfahren.

Beispiel:

```
DEFINE ADR=$30 REM ADR to location 30H
DEFINE VALUE=$31 REM VALUE
DEFINE I1 = $32

START:
FOR I1=0 TO 63 REM only 64 bytes
LET ADR=I1 REM writes a value in each
LET VALUE=I1*2 REM memory cell
WRITE ADR,VALUE
NEXT I1
LET ADR=2 REM now read the third entry
READ ADR,VALUE REM into VALUE (must be 4)
```

Bemerkung: Da bei manchen Bausteinen das interne EEPROM als eigenständiges I2C-Die implementiert ist (z.B. 12E51x, u.a.), kann dort die Laufzeitbibliothek das Ende des Schreibzykluses nicht feststellen. Deshalb muss bei diesen Bausteinen der Programmierer dafür sorgen, dass mindestens 10 ms zwischen zwei Schreibzyklen (WRITE) sind.

I2C-Schnittstelle

Die I2C-Schnittstelle kann auf verschiedene Arten konfiguriert werden.

Wird nichts angegeben, wird die Slave- und die Masterfunktion als reine Softwarelösung realisiert. Das hat den Vorteil, dass auch solche Microcontroller, die keine I2C-Hardware implementiert haben, diese Schnittstelle bedienen können. In der Regel muss man nur darauf achten, dass der Master langsamer als der Slave läuft, da der Master den I2C-Takt angibt. Dies ist gewährleistet, wenn beide Funktionen mittels iL-BAS realisiert werden.

Wird der PIC als Slave eingesetzt und mit dem Compiler-Schlüsselwort I2CHARDS diese Funktion implementiert, sollte folgendes beachtet werden:

Wenn die Anzahl der Datenbytes variieren kann, muss die Stop-Condition als Kennung des Übertragungsende herangezogen werden. Dazu fragt man das P-Bit im SSPSTAT-Register ab.

Loop1:

```
IF SSPSTAT,4 = 0 THEN GOTO Loop1  'warte bis zum Ende der Übertragung
```

ASSEMBLER (allgemeines)

Einleitung

Der Assembler iL_ASS16 ist für die Umsetzung der Mnemonics entsprechend der MICROCHIP-Notation sowie einer modifizierten Mnemonic zuständig. Es handelt sich um einen einfachen, nicht makrofähigen Assembler, der für die meisten Anwendungen ausreichend ist.

Laden des Programms

Gestartet wird der Assembler entweder direkt aus der Entwicklungsumgebung heraus oder aus der DOS-Ebene bzw. der DOS-Box.

Aus der DOS-Ebene erfolgt der Programmaufruf mit iL_ASS16 oder iL_ASS16 programmname (ohne Extension). Wird kein Programmname mit angegeben, muss dieser während des Programmlaufs nachträglich eingegeben werden. In beiden Fällen aber ohne Extension. Es ist notwendig, den Quellcode im ASCII-Format in der Datei *programmname* . **SRC** zu speichern. Der Assemblierungsvorgang wird sofort gestartet und bei fehlerfreiem Lauf der Code im INTELHEX8 oder INTELHEX16-Format abgespeichert. Diese Formate kann der dazugehörige Simulator automatisch erkennen und laden. In einer LST-Datei wird der Quellcode inklusive Kommentar und OP-Code hinterlegt. Im Fehlerfall erfolgt die Ausgabe der Meldungen sowohl auf dem Bildschirm als auch in der ERR-Datei. OBJ-, SYM- und LST-Dateien werden dann nicht erzeugt. Wird der Assembler unmittelbar nach einem Compiliervorgang mit dem BASIC-Compiler iL_BAS16 gestartet, wird zusätzlich eine DEBUG-Datei erzeugt. Mit dieser ist später der Simulator iL_SIM16 in der Lage den BASIC-Quelltext Zeile für Zeile abzuarbeiten.

Assemblerdirektiven

Assembleranweisungen

Der Assembler kennt folgende Anweisungen:

(Diese Anweisungen werden vom Assembler wie normale Befehle behandelt und müssen deshalb auch in der Befehlsspalte stehen -> mind. ein führendes Leerzeichen)

Hinweis: Die Sharewareversion unterstützt nicht die DEVICE-Anweisung. Aus diesem Grund wird in dieser Version immer die Defaultzeile angewählt.

DEVICE 16C83,XT_OSC,WDT_OFF, PROTECT_OFF

Der Aufbau einer Zeile sieht wie folgt aus:

Label **Befehl** **Argumente** **;Kommentar**

a) Label

Das Label muss an der ersten Stelle der Zeile beginnen. Steht vor dem Befehl kein Label, muss mindestens ein Leerzeichen vor dem Befehl stehen. Kommentare werden durch ein Semikolon eingeleitet.

Achtung!

Steht nur das Symbol bzw. Label in einer Zeile, wird diesem der Wert des momentanen Programmzählerstandes zugewiesen. Eine Fehlermeldung erfolgt nicht.

Bsp.

 ORG 15h
COUNT

Dem Symbol COUNT wird der Wert 15h zugewiesen. Deshalb ist diese Schreibweise nur bei Sprunglabels anzuwenden. Symbole müssen in der gleichen Zeile und mit EQU definiert werden.

b) Befehl

Neben den eigentlichen mnemonischen Befehlen für das Programm kennt der Assembler noch weitere Befehle, die jedoch in erster Linie dazu dienen, den Ablauf des Assemblierungsvorganges zu beeinflussen. Die Programmbefehle finden Sie im Kapitel 4. Die Assembleranweisungen sind:

ORG nnn

Setzt den Programmzähler auf den Wert nnn. Der nachfolgende Code wird bei dieser Adresse beginnend abgelegt.

EQU

Weist einem Symbol einen Zahlenwert zu. Z.B.

BAUD EQU 10h

nachfolgend ersetzt der Assembler das Symbol BAUD durch den Wert 10h

LIST INHX8 oder INHX16

Legt fest, ob die OBJ-Datei im Intel-Hex8 oder Intel-Hex16-Format abgespeichert werden soll. Defaulteinstellung ist INHX8

LIST /M_OBJ

Das erzeugte OBJ-File ist mit dem Microchip-OBJ-File konform.

LIST C=xxx

Legt die Zeilenbreite im LST-File fest. Zu beachten ist dabei, dass der Assembler 18 Spalten an der linken Seite hinzufügt. Dort stehen die Zeilennummer des Quellfiles, eventuell durch ein "I" gekennzeichnet, falls gerade ein INCLUDE-File gelesen wird, die Adresse des Codes und der Befehlscode selbst. Erst danach kommt die Spaltennummer und der Rest.

Assemblerdirektiven (cont.)

LIST BIN

LIST BINX

Wird einer dieser Schalter mit angegeben, erzeugt der Assembler zusätzlich eine Binärdatei mit der Extension BIN. Diese Datei beinhaltet nur die Daten für den Programmcode, also keine Angaben über den Bausteintyp oder die Bits im Konfigurationsbereich. Ebenso fehlt der Datenbereich beim 16X8x Baustein. Der Schalter BIN legt die Daten in der Folge Highbyte - Lowbyte im Binärfile ab. Der Schalter BINX vertauscht High- und Lowbyte. Welches Format vom jeweiligen Programmierer verwendet wird, muss ausprobiert werden.

LIST OBJ2HEX

Erzeugt statt einer Endung OBJ die Endung HEX. Viele Programmiergeräte benötigen die Endung HEX.

Die verschiedenen LIST-Anweisungen können durch Kommata getrennt in der gleichen Zeile stehen.
(In der BASIC-Syntax lautet der Befehl \$LIST).

IF Bedingung

ELSEIF

ENDIF

Erlauben eine bedingte Assemblierung. Ist die Bedingung wahr, so wird der Bereich zwischen IF und ELSEIF, falls vorhanden, sonst bis ENDIF assembliert. Ist die Bedingung unwahr, wird entweder der Bereich zwischen ELSEIF und ENDIF assembliert oder nichts.

IDENT nnnn

Übernimmt die Zahlenfolge nnnn und ordnet sie den ID-Speicherzellen des Controllers zu (z.Zt. nur Hexzahlen möglich).

OLDVAR

Die älteren Assemblerversionen hatten einige Symbole z.B. INDIRECT, RTCC usw. vordefiniert. Diese Funktion entfällt ab der Version 5.0. Will man weiterhin diese vordefinierten Symbole verwenden, muss der Assemblerschalter OLDVAR gesetzt sein.

Im anderen Fall muss der Anwender auch diese Symbole neu definieren.

(In der BASIC-Syntax lautet der Schalter \$OLDVAR).

DEVICE

Legt den Prozessortyp fest und die verschiedenen Einstellungen im Konfigurationsbyte des Controllers.

12C508 12C509 12E518 12E519 12F629 12C671 12C672 12F675 16C53 16C54 16C55 16C56
16C57 16C58 16C61 16C62 16C63 16C64 16C65 16C66 16C620 16C621 16C622 16E623 16E624
16E625 16C71 16C72 16C73 16C74 16C76 16C77 16F818 16F819 16C83 16C84 16F83 16F84
16F873 16F874 16F876 16F877

(Diese Liste wird laufend ergänzt!)

Der Typnummer darf auch der Begriff PIC vorangestellt werden z.B. PIC16C84

Handelt es sich um einen A-Type, wird das A einfach angehängt z.B. 16C65A.

LP_OSC	Oszillatortyp	LOW POWER
XT_OSC	"	Quarz bzw. Resonator
HS_OSC	"	HIGH SPEED
RC_OSC	"	RC-Glied
IRC_OSC	"	Interner RC (12Cxxx)

Assemblerdirektiven (cont.)

ERC_OSC " Externer RC (12Cxxx)

WDT_ON Watchdog aktiv

WDT_OFF Watchdog inaktiv

PROTECT_ON Ausleseschutz aktiv

PROTECT_OFF Ausleseschutz inaktiv

PWRTE_ON Power-up-Timer ein (nur wo vorhanden)

PWRTE_OFF Power-up-Timer aus (nur wo vorhanden)

MCLR_INT Reset wird nur intern erzeugt (12Cxxx)

MCLR_EXT Pin wird als Reseteingang benutzt (12Cxxx)

INCLUDE dateiname.ext

Durch INCLUDE-Dateien kann man bestimmte Programmmodule z.B. Definition der Symbole in eine eigne Datei schreiben. Diese werden beim Assemblierungslauf eingelesen und so behandelt, als stünde der Text direkt in der Hauptdatei. Dadurch entlastet man die Hauptdatei und die Übersichtlichkeit wird erhöht. Includedateien dürfen aber selbst keine weiteren Include-Anweisungen beinhalten. Im LST-File wird der Bereich aus einem INCLUDE-File gekennzeichnet. Da INCLUDE ein Befehl ist, darf er nicht an der Position eines Labels stehen!

(Im BASIC-Quelltext wird \$INCLUDE verwendet).

c) Argumente

Neben Zahlen im Dezimal-, Hex- und Binärformat dürfen auch in einfachen Hochkommata eingeschlossene ASCII-Zeichen verwendet werden.

z.B. 100 64H 01100100B 'A'
dez. hex bin. ergibt 65 (41H)

Argumente können auch berechnet werden. Es darf dabei allerdings nur ein Rechenzeichen bzw. Vorzeichen verwendet werden. Vor und nach dem Rechenzeichen darf max. 1 Leerzeichen stehen. Zwischen Vorzeichen und Argument darf kein Leerzeichen stehen.

richtig: VAR1+VAR2 VAR1 + VAR2 VAR1+ VAR2 VAR1 +VAR2

falsch: VAR1 + VAR2 !VAR1 + VAR2 (=zwei Operatoren) !VAR1

Erlaubte Operationen und deren Funktion sind:

- + Addition oder Vorzeichen
- Subtraktion oder Vorzeichen
- * Multiplikation
- / Division
- & logisches UND
- | logisches ODER
- ^ logisches EXCLUSIV-ODER
- << linksschieben (var1 << 4)
- >> rechtsschieben. (var1 >> 2)
- ! Negationsvorzeichen
- \$ übernimmt den aktuellen PC-Stand (\$+3)

Ist das Ergebnis einer Berechnung größer als 255 (-128, +127) erfolgt eine Fehlermeldung. Ausnahme: CALL, GOTO, ORG und EQU

Assemblerdirektiven (cont.)

Wenn der Schalter OLDVAR gesetzt ist, kennt der Assembler einige vordefinierte Symbole:

INDIRECT	für File 0 (Register für indirekte Adressierung)
RTCC	für File 1 (Real Time Clock/Counter)
PC	für File 2 (Programmzähler)
STATUS	für File 3 (Statusregister, Flags)
FSR	für File 4 (File Select Register)
RA	für File 5 (Port A)
RB	für File 6 (Port B)
RC	für File 7 (Port C), falls 16C55 oder 16C57
TRUE	0
FALSE	1

In Verbindung mit dem Statusregister sind noch folgende Symbole definiert:

C	für Carry /Borrow
Z	für Zero
DC	für DigitCarry /Borrow
PD	Power Down
TO	Time Out
PA0	Page Preselect Bit 0 (16C5x)
PA1	Page Preselect Bit 1 (16C5x)
PA2	Page Preselect Bit 2 (16C5x)
RP0	Register Page Select direct (16C71, 16C84)
RP1	Register Page Select direct (16C71, 16C84)
IRP	Register Page Select indirect (16C71, 16C84)

Assembler Grundbefehlssatz

```

ADDWF f,d INCF f,d TRIS f
ANDLW k    INCFSZ f,d  XORLW k
ANDWF f,d  IORLW k    XORWF f,d
BCF f,b    IORWF f,d
BSF f,b    MOVF f,d
BTFSC f,b  MOVLW k    ADDLW k
BTFSS f,b  MOVWF f    RETFIE
CALL k     NOP        RETURN
CLRF f     OPTION     SUBLW k
CLRW       RETLW k
CLRWDW     RLF f,d
COMF f,d   RRF f,d
DECF f,d   SLEEP
DECFSZ f,d SUBWF f,d
GOTO k     SWAPF f,d
  
```

Nachfolgend finden Sie die ausführliche Beschreibung der einzelnen Befehle des Grundbefehlssatzes.

Hinweis: f=fileregister, d=dest. (W oder 0 ->Erg.in w; 1=Erg. in f) wird d nicht angegeben, ist d=1

ADDWF f,d Verändert Z,C,DC

Addiere W und f. Ergebnis in W, wenn d=0 sonst in f

ANDLW k Verändert Z

UND-Verknüpfung von W und k, Ergebnis nach W

ANDWF f,d Verändert Z

UND-Verknüpfung von W und f, Ergebnis nach W, wenn d=0, sonst in f

BCF f,b Verändert -

Lösche Bit b in Fileregister f

BSF f,b Verändert -

Setze Bit b in Fileregister f

BTFSC f,b Verändert -

Teste Bit b in Fileregister f, überspringe nächsten Befehl wenn Bit=0 ist

BTFSS f,b Verändert -

Teste Bit b in Fileregister f, überspringe nächsten Befehl wenn Bit =1 ist

CALL k Verändert -

Rufe Unterprogramm an Adresse k auf. Rückkehradresse wird auf dem Stack abgelegt (Stacktiefe bei 16C5x 2, 16C71 u. 16C84 8 Ebenen, beim 16C5x muss der Anfang des UP auf der ersten Hälfte einer Programmspeicherseite [0..FFH] liegen)

CLRF f,d Verändert Z

Löscht das Fileregister f

CLRW Verändert Z

Löscht das Arbeitregister W

Assembler Grundbefehlssatz (cont.)

CLRWDT Verändert TO=PD=1

Löscht den Watchdogtimer

COMF f,d Verändert Z

Bildet 1er-Komplement vom Fileregister f. Ergebnis in W, wenn d=0, sonst in f

DECf f,d Verändert Z

Dekrementiert Fileregister f, Ergebnis in W, wenn d=0 sonst in f

DECFSZ f,d Verändert -

Dekrementiere Fileregister f, überspringe nächsten Befehl, wenn Ergebnis 0 ist. Ergebnis nach W, wenn d=0 sonst in f

GOTO k Verändert -

Sprung zur Adresse k. Beim 12C5xx und 16C5x ist k nur 9 Bit breit und somit der Bereich 0-1FFH. Um in eine andere Speicherseite zu gelangen, müssen zuerst die Bits PA0, PA1 und PA2 im Statusregister entsprechend gesetzt werden. Bei den übrigen Prozessoren hat eine Speicherseite den Adressraum 0-7FF. Siehe auch Kapitel 6 und 7 in der Simulatorbeschreibung.

INCF f,d Verändert Z

Erhöhe den Wert des Fileregisters f. Ergebnis nach W wenn d=0, sonst in f

INCFSZ f,d Verändert -

Erhöhe den Wert des Fileregisters f. Überspringe nächsten Befehl, wenn Ergebnis 0 ist. Ergebnis nach W, wenn D=0, sonst nach f

IORLW k Verändert Z

ODER-Verknüpfung von W und k

IORWF f,d Verändert Z

ODER-Verknüpfung von W und Fileregister f. Ergebnis nach W, wenn d=0, sonst nach f

MOVF f,d Verändert Z

Lade f in sich selbst oder nach W (zur Abfrage auf 0)

MOVLW k Verändert -

Ladet Konstante k ins Arbeitsregister W

MOVWF f Verändert -

Lade Inhalt des Arbeitsregisters ins Fileregister f

NOP Verändert -

No Operation

OPTION Verändert -

Lade Inhalt des Arbeitsregisters W ins Optionregister

RETLW k Verändert -

Rücksprung aus Unterprogramm, Arbeitsregister W wird mit k geladen (z.B. für Tabellen)

RLF f,d Verändert C

Rotiere Fileregister nach links durchs Carry, Ergebnis bei d=0 ins W- sonst ins f-Register

Assembler Grundbefehlssatz (cont.)

RRF f,d Verändert C

Rotiere Fileregister nach rechts durchs Carry, Ergebnis bei d=0 ins W-, sonst ins f-Register

SLEEP Verändert PD=0, TO=1

Stromsparmodus, fast alle Funktionen werden abgeschaltet. Beendigung beim 16C5x nur über RESET, bei 16C6x, 16C7x und 16C8x auch über Interrupts

SUBWF f,d Verändert Z,C,DC

Subtrahiert W von f. Ergebnis bei d=0 ins W-, sonst ins f-Register.

ACHTUNG. Carry wird invertiert behandelt.

SWAP f,d Verändert -

Vertauscht die beiden Halbbytes. Ergebnis bei d=0 ins W-Register, sonst ins f.

TRIS f Verändert -

Datenrichtungsregister der Ports werden mit dem Inhalt des Arbeitsregisters geladen

XORLW k Verändert Z

EXCLUSIV-ODER-Verknüpfung von W und k.

XORWF f,d Verändert Z

EXCLUSIV-ODER-Verknüpfung von W und Filereg. f. Ergebnis bei d=0 ins W-Register

Beim 12C67x, 16C6x, 16C7x, 16X8x und 16X87x kommen die nachfolgenden Befehle hinzu:

ADDLW k Verändert C,DC,Z

Addiere Literal zu w

RETFIE Verändert -

Return vom Interr. + GIE

RETURN Verändert -

Return vom UP

SUBLW k Verändert C,DC,Z

Subtr. W von Literal

Das Option- und die Datenrichtungsregister sind beim 12C67x, 16C6x, 16C7x und 16X8x im Bereich des Registerfiles angesiedelt. Aus Kompatibilitätsgründen werden diese Befehle aber auch hier als eigenständige OP-Codes mitgeführt (nur TRISA, TRISB und TRISC). Bei nachfolgenden Prozessoren kann es sein, dass diese nicht mehr unterstützt werden.

ACHTUNG!!

Nicht zu verwechseln mit dem BASIC-Befehl TRIS. Dort dürfen auch die Ports RD und RE angegeben werden.

Beim Subtraktionsbefehl wird das Carrybit genau invertiert angegeben. Der Grund liegt in der internen Abwicklung des Subtraktionsbefehls, der bei diesen Prozessoren durch eine Addition des 2er-Komplements erfolgt. Deshalb ist:

CLRF 10	f10=0	
MOVLW 1	w=1	
SUBWF 10	f10=0-1=0+FF=FF	Ergb. neg, C=0

oder

MOVLW 0FFh

Assembler Grundbefehlssatz (cont.)

MOVWF 10	f10=FF	
CLRW	w=0	
SUBWF 10	f10=FF-0 = FF	Erg. pos, C=1

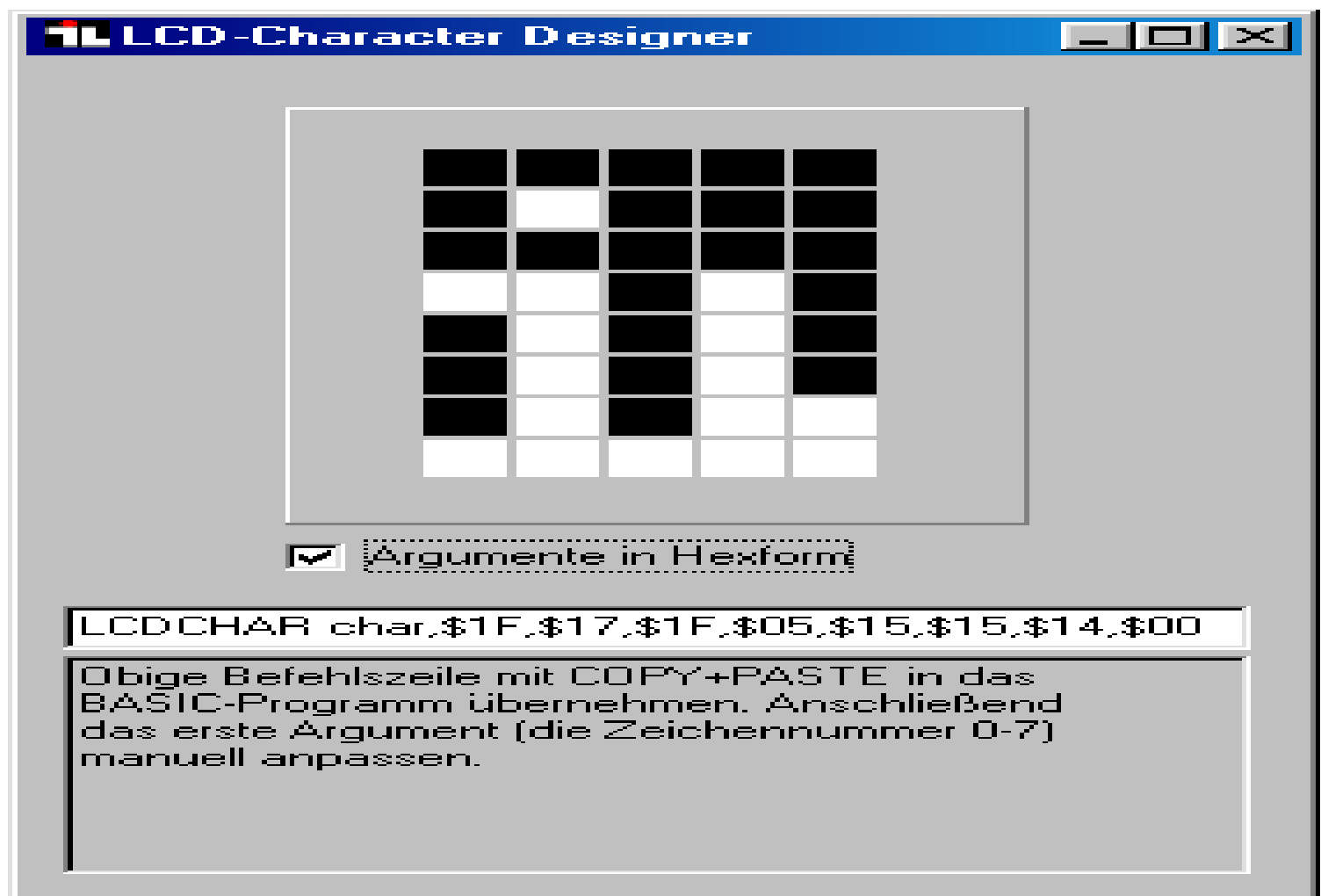
BaudCalc

BaudCalc ist ein Hilfsprogramm (Utility) um die Verzögerungskonstanten beim SERIN-und SEROUT-Befehl zu berechnen. Will man die Baudrate in diesen beiden Befehlen in einer Variablen angeben, so müsste das Programm zur Laufzeit den tatsächlichen Verzögerungswert unter Berücksichtigung der Quarzfrequenz berechnen. Dies kostet Speicherplatz und Zeit. Dieses Manko lässt sich umgehen, wenn man statt der absoluten Baudrate den tatsächlichen Verzögerungswert in die Variable schreibt. Das Berechnen dieses Wertes ist allerdings alles andere als trivial. Deshalb wurde BaudCalc.EXE entwickelt. Nach Angabe der gewünschten Baudrate, der Quarzfrequenz und des Prozessortyps (12- oder 14-Bit Kern) werden die Werte nach Drücken der Schaltfläche 'BERECHNUNG' ermittelt. Diese sind dann statt der absoluten Baudrate in die entsprechenden Variablen zu laden.

LCD Character Designer

LCD Character Designer erlaubt das einfache Erstellen beliebiger Zeichen für Standard-LCDs. Diese besitzen in der Regel acht frei definierbare Zeichen die mit der Zeichennummer 0 bis 7 angesprochen werden. An diese Stelle im Zeichengenerator schreibt der BASIC-Befehl LCDCHAR das Bitmuster das dargestellt werden soll. Leider ist das Erstellen dieses Bitmusters fehleranfällig, da auf Bitebene das Zeichen kreiert werden muss. Dieses Hilfsprogramm erleichtert den Vorgang erheblich.

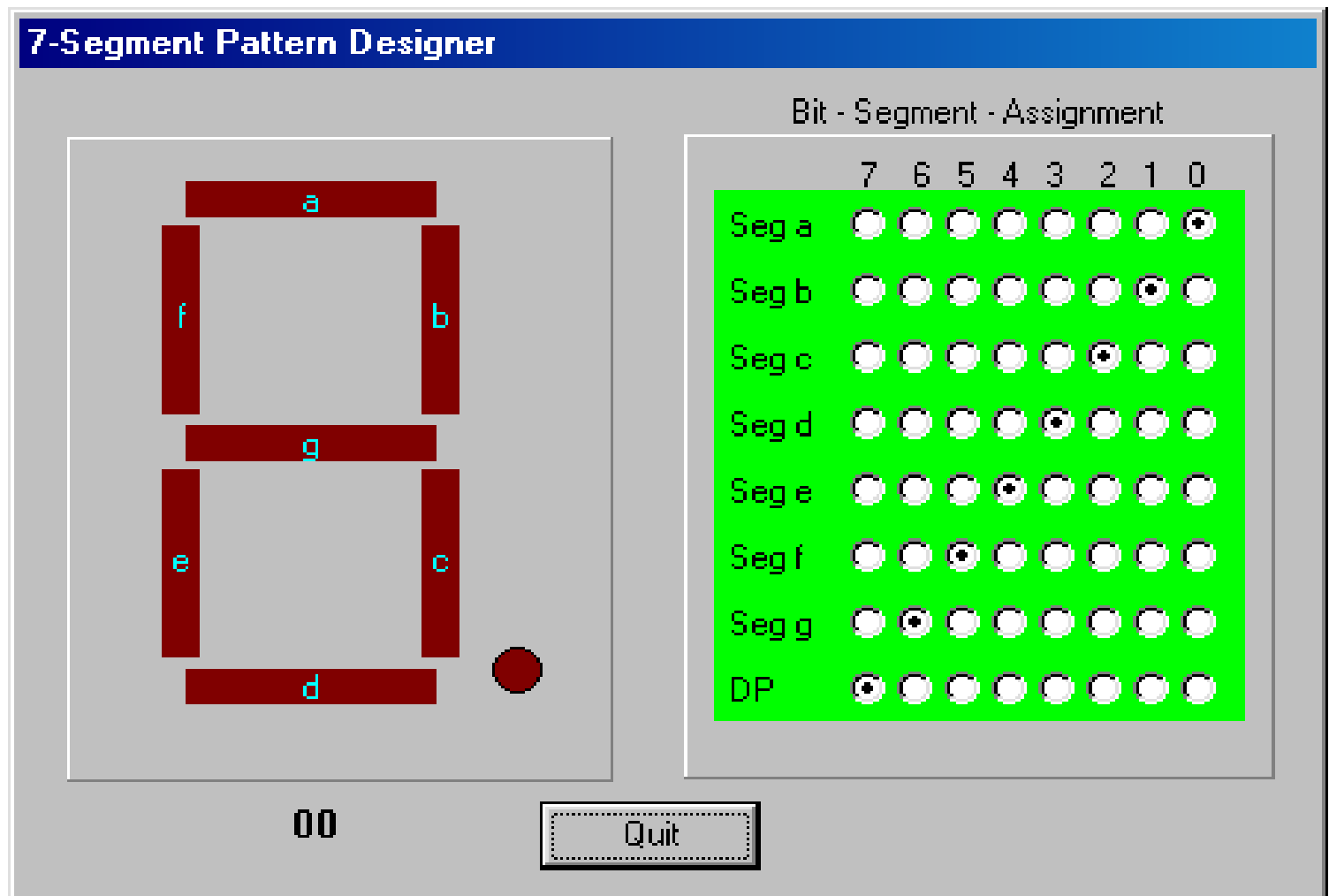
Zu kreieren des Zeichens klickt man die entsprechende Punkte an und wechselt so von hell nach dunkel bzw. umgekehrt. Jeder dunkle Punkt wird auch auf der Anzeige als sichtbarer Punkt dargestellt. Unterhalb des Editfeldes wird simultan die notwendige BASIC-Befehlszeile angezeigt. Ist das Zeichen fertig, genügt es diese Zeile mittels Maus zu markieren und per Copy-Paste in das BASIC-Programm zu übernehmen. Anschließend muss nur noch der Platzhalter *char* durch eine Zahl zwischen 0 und 7 ersetzt werden. LCDWRITE 1,1,0,# zeigt dann das selbstdefinierte Zeichen 0 an.



7-Segment Pattern Designer

Mit diesem Hilfsprogramm lassen sich die Muster auf einer 7-Segmentanzeige bequem definieren. Oft ist es so, dass ein Platinenlayout dadurch vereinfacht werden kann, wenn man Anschlüsse vertauscht. Bei einer 7-Segmentanzeige ist dies natürlich ohne weiteres möglich, da ja alle Segmente im Prinzip gleichberechtigt sind. Nur vertauscht man Leitungen, dann muss auch ein anderes Bitmuster am Port angelegt werden um das gewünschte Zeichen zu sehen.

Im rechten, grün unterlegten Feld, wird die Zuordnung zwischen Segment und Pin festgelegt. Bei ungültigen Kombinationen werden die betroffenen Zeilen rot markiert. Ist diese Zuordnung erfolgt, definiert man das eigentliche Bitmuster indem man auf der linken Seite das gewünschte Segment ein- bzw. ausschaltet. Unterhalb der "Anzeige" wird der aktuelle Hexcode sichtbar. Dieser wird manuell ins BASIC-Programm übernommen. Das kann als Variablen-, Konstanten- oder Tabelleneintrag erfolgen.



Inhalt von DEFAULT.EQU

Aufgrund der Implementierung der PIC18Fxxxx Familie mussten viele Änderungen in dieser Datei vorgenommen werden. Sie können diese Datei einfach mit einem Texteditor öffnen und über die Suchfunktion den entsprechenden Prozessor anwählen.

Unterstützte Bausteine

die aktuelle Liste finden Sie im Internet unter
www.iL-online.de

FAQs und Bemerkungen

Hier werden interessante Kundenfragen beantwortet.

Weitere interessante FAQs finden Sie auf unserer Homepage www.iL-online.de

FAQs

Frage:

Auf dem LCD erscheinen nur wirre Zeichen. Es ist keinerlei Systematik darin zu erkennen.

Antwort:

LCDs unterliegen großen Fertigungsschwankungen. Es kann sein, dass in einer Applikation eine LCD funktioniert, eine andere LCD dagegen nicht. Achten Sie darauf, dass die Verbindungsleitungen kurz sind (TTL-Pegel) und geben Sie der LCD genügend Zeit. Da die Befehle LCDWRITE und LCDCLEAR nicht das BUSY-Flag der Anzeige abfragen, sollte die LCD maximal nur 10 bis 20 mal pro Sekunde beschrieben werden. Unter Umständen hilft hier auch der Befehl LCDDELAY weiter. Auch kann es passieren, dass ihr Programm bereits losläuft und die LCD initialisieren will, obwohl diese noch im eigenen Initialisierungsmodus steckt. In solchen Fällen hilft ein WAIT 250 vor LCDINIT.

Frage:

Ich habe ein Programm geschrieben, wo ich über eine Tastatureingabe Zahlen von 1 bis 9 einlese. Anhand dieser Zahlen verzweige ich mittels ON x GOSUB zu den jeweiligen Unterprogrammen. Leider stimmt die Zuordnung nicht. Es scheint als sei alles um 1 verschoben.

Antwort:

Bei ON x GOSUB sowie bei ON x GOTO beginnt die Zählung bei 0. Das heisst, wenn x = 0 ist, dann wird der erste Eintrag angesprungen. Sie fügen also entweder am Listenanfang eine "Dummy-Adresse" ein, oder subtrahieren von x den Wert 1.

Frage:

Mein Problem ist, das ich leider kein einziges Programm mit dem PIC 12F675 zum Laufen bringen, bei dem der Ad-Wandler und der Komparator ausgeschaltet ist und der interne Oszillator verwendet wird. Der Compiler meldet keinen Fehler. Ich programmiere den Hexcode über Picstart Plus. Könnten Sie mir ein kleines Musterprogramm machen, damit ich meinen Fehler finde?

Antwort:

Das nachfolgende Testprogramm das an GP0 und GP1 LEDs ein- und ausschaltet. Beim Programmieren des 12F675 ist es ganz wichtig, dass auf der letzten Speicherstelle im Programmspeicher der Kalibrationswert steht. Dieser lautet 34xx, wobei xx ein Wert zwischen 00 und FF sein kann. Fehlt dieser Wert, läuft der Baustein nicht. Da dieser Wert beim Löschen verloren geht, sollte man vor dem Löschen den Baustein auslesen und den Wert an der Adresse 3FFH notieren. So kann dieser nachträglich wieder eingebrannt werden. Überprüfen Sie in jedem Fall vor dem Programmieren diesen Wert. Manche Programmiergeräte nehmen auf diesen Umstand keine Rücksicht, so dass man ggf. diesen Wert jedesmal manuell eingeben muss. Der Compiler iL_BAS16 bietet für diese Fälle den Befehl CALVAL xx an (für xx nur der Zahlenwert angeben, nicht den führenden Code 34). Damit wird dann allerdings jeder 12F675 mit diesem Wert programmiert, was sich in einer u.U. falschen Oszillatorfrequenz bemerkbar macht. Im Beispielpogramm sind alle Pins auf IO-Betrieb umgestellt, wobei GP3 nur als Eingang funktioniert.

```
define device
12F675, WDT_off, PROTECT_OFF, IRC_OSC, PWRTE_ON, MCLR_INT, ADCFG0, OSC2_IO, cmcfg7
tris ra, 0
loop:
set ra, 0
wait 500
set ra, 1
wait 500
res ra, 1
wait 500
```

FAQs und Bemerkungen (cont.)

```
res ra,0
    wait 500
    goto loop
```

Frage:

Beim 12C672 kann ich mit SERIN nichts empfangen. Der SEROUT-Befehl bringt die Zeichen absolut richtig heraus.

Antwort:

Vermutlich haben Sie vergessen den Empfangspin von Analoginput auf Digital-I/O umzustellen. Dies geschieht mit ADCFGx in der DEFINE DEVICE Zeile. ADCFG7 schaltet z.B. alle Pins auf Digital-I/O um.

Frage:

Ich habe eine Schaltung aufgebaut, in der der PIC 16F628 programmiert werden soll, also mittels In-Circuit-Programming. Leider funktioniert die Programmierung nicht. Der Baustein lässt sich einfach nicht programmieren. Er ist aber sicher nicht defekt, da er im Programmiergerät richtig programmiert wird. Seltsamerweise funktioniert es dann auch mit dem Programmieren in der Schaltung. Man muss also einen neuen Baustein zuerst im Programmiergerät programmieren, damit man ihn anschließend In-Circuit programmieren kann! Wiso? Was mache ich falsch?.

Antwort:

Das Problem ist ähnlich dem Problem 3. Wahrscheinlich wird in ihrer Schaltung der RB4-Pin auf High-Pegel gehalten. Das bedeutet aber, dass er nach dem Anlegen der Versorgungsspannung sofort in den Programmiermodus geht. Da bei den neuen PICs das LVP-Bit aktiviert ist, passiert dies. Nach dem erstmaligen programmieren im Programmiergerät ist dann das LVP-Bit abgeschaltet. Kann man den Baustein nicht vorab programmieren (z.B. bei SMD) schafft nur das "Herunterziehen" des Pins auf LOW-Pegel Abhilfe. (Dies gilt übrigens auch für andere PICs mit einem LVP-Eingang).

Frage

Ein Programm, das auf dem 16F84 einwandfrei läuft, läuft auf einem 16F628 nicht richtig. Die an RB angeschlossene LCD funktioniert nicht.

Antwort:

Beim 16F628 gibt es an Pin RB4 die Funktion LVP. Damit dieser Pin aber als normaler I/O-Pin arbeiten kann, muss im Konfigurationswort die LVP-Funktion abgeschaltet sein. Manche Programmiergeräte übernehmen diese Einstellungen nicht automatisch, so dass hier das Bit entsprechend manuell eingestellt werden muss. (Hinweis: Es gibt noch weitere PIC-Bausteine mit der LVP Funktion).

Frage:

Bei dem PIC 12C508 funktionieren manche Pins nicht richtig

Antwort:

Dazu muss man wissen, dass manche Bausteine bestimmte Pins nur in eine bestimmte Richtung arbeiten lassen. Es gibt auch den Fall, dass neben dem TRIS-Register noch in einem anderen Register eine Änderung vorgenommen werden muss. Dies erfolgt nicht automatisch durch den Compiler, sondern muss programmiert werden. Ein Blick ins Datenblatt des entsprechenden Prozessors erklärt manches.

Frage:

Wie kann ich den BASIC-Compiler iL_BAS16 zusammen mit dem Programmiergerät von Microchip verwenden?.

Antwort:

Dazu muss im BASIC-Programm der Schalter \$LIST /M_OBJ gesetzt werden. Die erzeugte HEX-Datei wird in die Programmiersoftware von Microchip geladen. Dann müssen die Konfigurationen (WDT, OSC usw.) manuell eingestellt werden. Nun erfolgt die Programmierung wie gewohnt.

Die neue Programmierumgebung von Microchip akzeptiert nur noch HEX-Dateien. \$LIST OBJ2HEX zwingt den Assembler dazu, die OBJ-Datei umzubenennen.

Frage:

FAQs und Bemerkungen (cont.)

Warum gibt es den BUTTON-Befehl nicht mehr?

Antwort:

Der von iL_BAS16 erzeugte Code ist so schnell, dass es sinnvoll ist, diesen Befehl, der sehr viel Code benötigt, "zu Fuß" zu programmieren. Man muss natürlich etwas mehr schreiben, der Code ist aber kürzer als beim BUTTON Befehl.

Darüberhinaus ist man viel flexibler was Timeout, Entprellung, Mindestdauer usw. angeht. Ein einfaches Beispiel:

Taste0:

```
if ra,0=1 then goto taste0
```

```
wait 10
```

```
if ra,0=1 then goto taste0
```

Frage:

Bei einem 16F87x will ich analoge Werte einlesen. Leider schwanken diese z.T. sehr stark. Zudem habe ich das Gefühl, dass der eine Kanal den anderen beeinflusst. Gibt es da ein Übersprechen, vielleicht sogar beim Platinenlayout?

Antwort:

Diese Bausteine haben einen 10-Bit-Wandler. Microchip hat den Haltekondensator vergrößert. Nun kann es sein, dass sich nach dem Umschalten der Haltekondensator nicht vollständig auf die neue Spannung eingestellt hat. Abhilfe schafft der ADDELAY Befehl, der einfach die Zeit zwischen Kanalumschaltung und Beginn der Messung vergrößert und somit dem Kondensator mehr Zeit gibt sich auf die Spannung dieses Kanals einzustellen.

Frage:

Ich will Zeichen über die serielle Schnittstelle einlesen, ohne den Befehl SERIN zu verwenden. Die Routine muss interruptgesteuert sein. Obwohl ich die Bits GIE und RCIE gesetzt habe, wird kein Interrupt generiert. Was mache ich falsch?

Antwort:

Ein Blick in das Datenbuch zeigt das Problem. Viele Interrupts, darunter der RC- und TX-Interrupt werden zum sogenannten PERIPHEREN-Interrupt zusammengefasst. Deshalb muss das PEIE-Bit ebenfalls gesetzt werden.

Bemerkungen

Laut Microchip sollte man bei LP-Typen bei Betriebsspannungen kleiner 5 V die BROWN OUT Funktion abschalten.

Die Mindestversorgungsspannung beim Programmieren und hier speziell beim Bulk-Erase ist meistens 4,5 V min. Daher kann es bei In-Circuit-Programmieren und einer niedrigen Versorgungsspannung (z.B. Batterie) zu Problemen kommen. Ideal ist es, wenn auch bei der ISP-Programmierung die Spannungen vom Programmiergerät kommen. Dazu sind allerdings u.U. schaltungstechnische Maßnahmen wegen Spannungsdifferenzen zu treffen.