Themenblatt

PIC_programmieren

Grundlagen zur PIC-Assembler Programmierung

&

Einfache Codesequenzen, kleine Funktionen und Programme

Verfasser: Stefan Lehmann

Stand: 14.04.2022

0 Änderungen und Ergänzungen

14.04.2022	Korrektur Tabelle (Registerstruktur)

Inhaltsverzeichnis

Dieses Themenblatt so eine einfache und verständliche Einführung in die Programmierung des PIC 16F84 geben. Zur Vertiefung und weitere Detailinformationen muss das Datenblatt[1] des Herstellers benutzt werden.

Für das Verständnis der Beispielprogramme in Kapitel 5 bis 11 ist dieses Themenblatt jedoch ausreichend.

1 Leider sind diese Datenblätter nicht immer fehlerfrei. In Moodle liegt eine korrigierte und leicht ergänzte Version des offiziellen Datenblatts zum Download bereit. Falls Sie keinen Zugriff auf die entsprechenden Moodleräume besitzen, nehmen Sie Kontakt mit dem Autor auf. Die E-Mail Adresse lautet: SL@iL-online.de

1 Die Hardware

Will man in Assembler programmieren, ist eine genaue Kenntnis der Hardware und der internen Struktur des Mikrocontrollers notwendig. Neben dem Schaltplan, der zeigt, wo welche Signale am Controller angeschlossen sind, muss man auch die Registerstruktur und weitere Interna des Mikrocontrollers kennen.

1.1 Der Schaltplan

Der Schaltplan (Schematic) einer einfachen Anwendung ist in Abbildung 1 zu sehen.

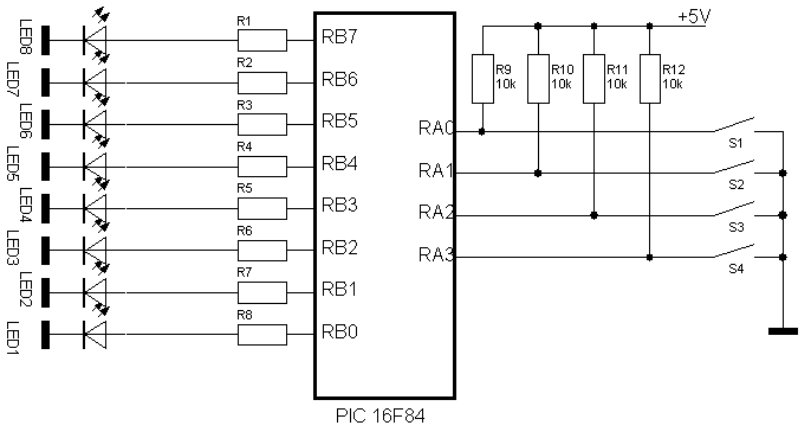


Abbildung 1: Eine einfache PIC Applikation

Die Leuchtdioden LED 1 bis LED 8 sind über Strom begrenzende Widerstände am Port RB des PICs angeschlossen. Dieser Port muss später als Ausgang arbeiten, da die Leds eine Spannungsquelle benötigen. Denn arbeitet ein Anschluss (Pin) als Ausgang, fungiert er praktisch als schaltbare Spannungsquelle oder Senke.

Auf der rechten Seite sind an die Leitungen RA 0 bis RA 3 die Schalter S 1 bis S 4 angeschlossen. Wenn einer dieser Schalter geschlossen[2] wird, verbindet er den entsprechenden Anschluss mit Massepotenzial (0 V). Damit der als Eingang arbeitende Anschluss bei offenem Schalter nicht "in der Luft hängt", sind Pullup-Widerstände angeschlossen. Somit erzeugt ein offener Schalter am Eingang einen High-Pegel, ist der Schalter geschlossen einen Low-Pegel. Dies wird als Low-Active Logik bezeichnet. Dies muss bei der Programmentwicklung berücksichtigt werden.

2 Schalter und Taster werden i.d.R. in Ruhestellung gezeichnet, hier also im geöffneten Zustand

Was nicht im Schaltplan eingezeichnet, aber beim realen Projekt in jedem Fall vorhanden sein muss, ist die Spannungsversorgung und ggf. der Taktgeber[3] sowie eine Resetlogik.

1.2 Die Registerstruktur des PICs

Im Gegensatz zum Programmieren in einer Hochsprache muss man sich bei der Assemblerprogrammierung um fasst jedes Detail selbst kümmern. Während ein Hochsprachen Compiler beispielsweise den verwendeten Variablen selbst die passende Speicheradresse zuordnet, ist dies die Aufgabe des Assemblerprogrammierers. Dazu muss er die Registerstruktur und die Adressen der speziellen Funktionsregister kennen.

Adresse	Bezeichnung	Bezeichnung	Adresse
0x00 0x01 0x02 0x03 0x04 0x05 0x06 0x07 0x08 0x09 0x0A 0x0B	indirect TMR 0 PCL Status FSR Port RA Port RB - EEData EEAdr PCLATH INTCON	indirect Option PCL Status FSR Tris RA Tris RB - EECon1 EECon2 PCLATH INTCON	0x00 0x01 0x02 0x03 0x04 0x05 0x06 0x07 0x08 0x09 0x0A 0x0B
0x0C ... 0x3F	Anwender-bereich	wird auf Bank 1 gespiegelt	0x0C ... 0x3F
0x40 ... 0x7F	nicht implementiert	nicht implementiert	0x40 ... 0x7F
Bank 0		Bank 1	

Die Bank, auf die man aktuell zugreifen kann, wird im Statusregister in dem Bit RP 0 festgelegt. Bei den PIC Mikrocontrollern mit vier statt zwei Bänken ist

3 Falls der Controller keinen internen Oszillator besitzt.

noch ein RP 1- und IRP-Bit vorhanden. Will man beispielsweise das TRIS RA verändern, muss man per Programm auf Bank 1 umschalten. Die notwendige Sequenz lautet:

```
BSF     status, rp0     ;umschalten auf Bank 1
MOVLW   11110011B       ;RA 0 und RA 1 sind Eingang
MOVWF   5               ;RA 2 und RA 3 sind Ausgang
BCF     status, rp0     ;zurück auf Bank 0
```

Manche Register sind sowohl auf Bank 0 als auch auf Bank 1 zu finden. Die Notwendigkeit ist beim Statusregister offensichtlich, da sonst ein Zurückschalten auf Bank 0 nicht mehr möglich wäre.

1.3 Register 0 für die indirekte Adressierung

Die indirekte Adressierung erlaubt bei bestimmten Programmstrukturen ein sehr effizientes und kompaktes Programmieren. Während bei der direkten Adressierung die verwendete Speicheradresse als absoluter Wert angegeben wird, zeigt bei der indirekten Adressierung ein Zeiger auf die verwendete Speicheradresse. Dieser Zeiger steht in einem Register und ist somit während des Programmablaufs veränderbar.

Beim PIC Mikrocontroller heißt dieses Zeigerregister FSR (File Select Register) und hat die RAM-Adresse 4 (sowohl auf Bank 0, als auch auf Bank 1). Es umfasst 8 Bit und erlaubt auf diese Weise neben den Adressen auf Bank 0 auch die auf Bank 1 direkt zu adressieren.

```
MOVLW   10H     ;Zeiger auf Adresse 0x10 setzen
MOVWF   fsr
MOVLW   0AAH    ;dieser Wert wird an Adresse 10
MOVWF   indirect;abgelegt.
INCF    indirect;in 0x10 steht jetzt 0xAB
```

Die indirekte Adressierung kann auf alle Befehle, die auf eine RAM-Adresse zugreifen, verwendet werden.

1.4 TMR0 Register

Das Timerregister TMR0 kann zum einen durch äußere Impulse hochgezählt werden. Dazu wird durch eine entsprechende Einstellung im Optionregister der Anschluss RA 4 mit dem TMR0 verbunden. Dabei kann zusätzlich auch noch

ein Vorteiler[4] verwendet werden, so dass nicht jeder Impuls am Eingang gezählt wird.

Das TMR0-Register lässt sich aber auch durch den internen Befehlstakt inkrementieren.

1.5 PCL

Das PCL-Register ist der untere Teil des Programmzählers. Nur diese unteren 8 Bit sind durch arithmetische bzw. logische Befehle änderbar. Obwohl das PCL ein Teil des Programmzählers ist, wird es wie ein normales 8-Bit Register angesprochen und es verhält sich auch dementsprechend. Ein 8 Bit Überlauf wird nicht auf die nächst höheren Bits des Programmzählers übertragen, sondern kommt wie bei allen übrigen Registern ins Carryflag.

Durch die Addition einer Zahl zum PCL lässt sich somit ein berechneter Sprung ausführen. Allerdings ist der Bereich, den man auf diese Weise abdecken kann, nur 256 Adressen groß. Diese Blöcke des Adressraums beginnen immer bei den Adressen 0x0000, 0x0100, 0x0200, 0x0300 etc.

Das Arbeiten mit Konstantentabellen wird mit diesem Register erst richtig möglich. Wie solche Tabellen aufgebaut werden ist in Kapitel 9 beschrieben.

1.6 STATUS

Im Statusregister sind neben dem Zeroflag, Carryflag und Digitcarryflag weitere, zur Steuerung des PICs sinnvolle Umschalt- und Zustandsbits enthalten. Welche Bits durch welche Befehle beeinflusst werden, ist aus der Befehlsübersicht auf Seite 17 zu sehen. Ein Zustandsbit bleibt solange unverändert, bis wieder ein Befehl ausgeführt wird, der das entsprechende Flag beeinflusst. Dabei wird es gesetzt oder zurückgesetzt, je nach Ergebnis der Operation.

1.6.1 Carryflag: Bit 0

Dieses Zustandsbit zeigt an, ob ein Zahlenbereichsüberlauf erfolgt ist. Dabei ist es gleichgültig, ob dieser Überlauf am oberen Ende, also bei 255 oder am unteren Ende bei 0 erfolgt. Das Übertragsbit (Carry) zeigt an, dass das Ergebnis eigentlich falsch ist, wenn nicht dieses zusätzliche Bit als Bereichserweiterung betrachtet wird.

4 Der Vorteiler kann aber auch dem Watchdog zugeordnet werden, aber nur entweder oder.

Beispiel: 255 + 1 = 0, C = 1 ;diese 1 entspricht dem Wert 256
Vorsicht ist geboten, wenn das Carry nach einer Subtraktion interpretiert werden soll. Mehr dazu unter 4.4 .

1.6.2 DC-Flag: Bit 1

Dieses Flag wird auch als Halfcarry bezeichnet. Es zeigt den Überlauf der unteren 4 Bits an. Da bei einer 8-Bit Operation dieser Überlauf nahtlos in die oberen 4 Bits einfließt hat er eher eine untergeordnete Bedeutung. Lediglich wenn man Routinen für die BCD-Arithmetik[5] implementiert, wird dieses Bit wichtig.

1.6.3 Zero-Flag: Bit 2

Das Zerobit wird gesetzt, wenn das Ergebnis einer Operation 0 ist. Interessant ist beim PIC Mikrocontroller, dass auch beim MOVF-Befehl das Zeroflag beeinflusst wird.

1.6.4 $\overline{\text{PD}}$-Flag: Bit 3

Dieses Power-Down-Bit wird im Zusammenhang mit dem Einschalten, dem CLRWDT- und dem SLEEP-Befehl verändert.

gesetzt auf 0	bei der Ausführung des SLEEP-Befehls
gesetzt auf 1	beim Einschalten des PICs und bei der Ausführung des CLRWDT-Befehls.

1.6.5 $\overline{\text{TO}}$-Flag: Bit 4

Bit 4 im Statusregister ist das Time-Out-Bit. Es wird im Zusammenhang mit dem Einschalten, einem CLRWDT-, einem SLEEP-Befehl oder durch das Ablaufen der Watchdogzeit beeinflusst.

gesetzt auf 0	wenn der Watchdog abgelaufen ist
gesetzt auf 1	beim Einschalten des PICs und bei der Ausführung der Befehle SLEEP und CLRWDT

1.6.6 RP0 und RP1: Bit 5 und 6

Mit diesen beiden Bits lassen sich die Registerbänke umschalten. Sind nur 2 dieser Bänke vorhanden, reicht das RP0-Bit. Ist es zurückgesetzt, greift man auf die Register der Bank 0 zu, bei gesetztem RP0 auf die Register der Bank 1.

RP1 RP0

5 BCD = Binary Coded Digit. Bei einer BCD-Zahl sind in einem Byte 2 Zahlen codiert. Jedes Halbbyte zählt dann von 0 bis 9.

0	0	Bank 0
0	1	Bank 1
1	0	Bank 2
1	1	Bank 3

1.6.7 IRP: Bit 7

Dieses Bit wird nur bei PICs mit mehr als 2 Registerbänken benötigt. Will man eine Adresse auf Bank 3 indirekt adressieren, reicht die 8-Bit-Adresse im FSR-Register nicht aus. Deshalb muss das IRP zusätzlich gesetzt werden.

indirekter Zugriff auf Bank 0 und 1	IRP = 0
indirekter Zugriff auf Bank 2 und 3	IRP = 1

1.7 FSR

Dieses Register arbeitet mit dem INDIRECT-Register Hand in Hand. Die ausführliche Beschreibung ist unter 1.3 zu finden.

1.8 PORT

Für die I/O-Funktionen sind die Ports eines Mikrocontroller zuständig. Beim PIC werden diese als Port RA, Port RB usw. je nachdem wie viele I/O-Anschlüsse der jeweilige Type besitzt. Beim PIC 16F84 sind dies der Port RA mit 5 (RA0 bis RA4) und der Port RB mit 8 (RB0 bis RB7) Anschlüssen.

Beim Einschalten oder bei einem Reset sind alle I/O-Leitungen zuerst als Eingang geschaltet. Dies soll verhindern, dass ein als Ausgang geschalteter Pin mit einer Stromquelle oder Stromsenke zusammen trifft und einen Kurzschluss erzeugt. Die Folgen wären ein defekter Mikrocontroller oder eine defekte Quelle bzw. Senke.

Der Zugriff auf die Portregister erfolgt über die Adressen 5 und 6 auf der Bank 0. Diese Register sind mit den übrigen gleichberechtigt d.h. Man kann deren Inhalt nicht nur lesen bzw. beschreiben sondern auch in logischen und arithmetischen Verknüpfungen verwenden.

Das Umschalten von Ein- auf Ausgang wird über das TRIS-Register gemacht. Der Zugriff auf diese Registriere erfolgt über die Bank 1.

Beispiel:

Angenommen das Programm befindet sich im Normalfall auf der Bank 0. Dann erfolgt der Schreibzugriff auf Port RB mit folgendem Befehl:

```
MOVWF   portb
```

und der Lesezugriff mit:

```
MOVF            portb, W
```

1.8.1 PORT RA und TRIS RA

Sollen RA0 und RA1 als Ausgänge, RA2, RA3 und RA4 als Eingänge arbeiten. Die passende Initialisierungssequenz sieht wie folgt aus:

```
BSF     status, rp0     ;auf Bank 1 umschalten, dort sind die
BCF     trisa, 0        ;TRIS-Register. RA0 wird Ausgang
BCF     trisa, 1        ;RA1 wird ebenfalls Ausgang
BCF     status, rp0     ;zurück auf Bank 0 schalten
...
...
BSF     porta, 0        ;setzt den Pegel an RA0 auf high
BCF     porta, 1        ;setzt den Pegel an RA1 auf low
```

Eine Besonderheit gibt es bei RA4. Dieser Pin ist bei diesem Typ als Open Drain Ausgang geschaltet. Er besitzt nur einen Transistor am Ausgang und kann mit diesem nur gegen 0 Volt schalten. Ein High-Pegel kann nur durch einen Pull-Up Widerstand erzeugt werden.

Die übrigen Pins können sowohl ein High als auch ein Low aktiv schalten.

1.8.2 PORT RB und TRIS RB

Die Initialisierung erfolgt beim Port RB analog zu Port RA.

1.8.3 Port RA Schaltung

Um die Funktionsweise eines I/O-Pins zu verstehen, muss man sich das Schaltbild genauer betrachten. Hier anhand eines Port RA Pins (Abbildung 2).

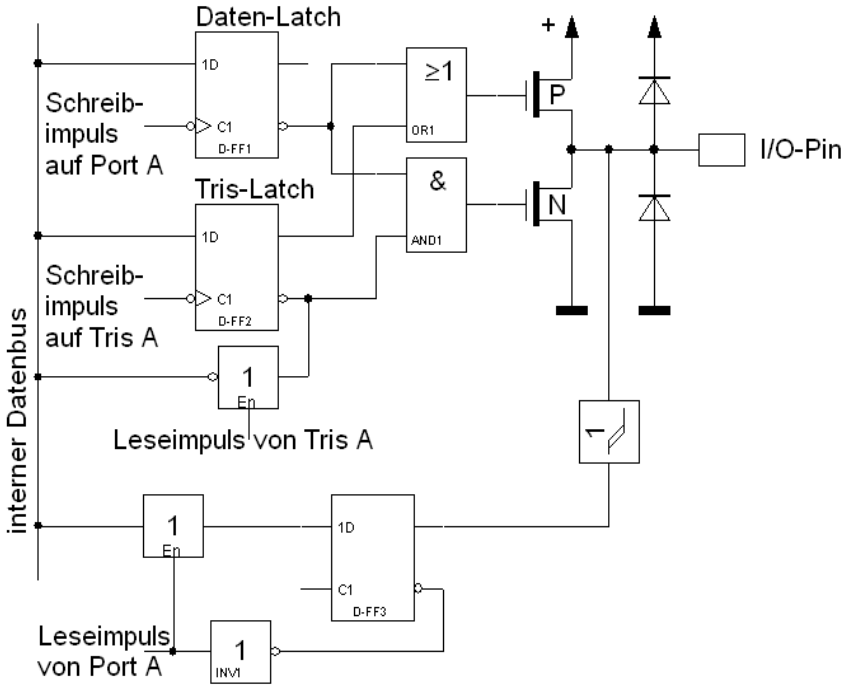


Abbildung 2: Schaltbild eines I/O-Pins

Input-Funktion

Der I/O-Pin ist mit zwei Schutzdioden gegen Störimpulse geschützt. Damit wird verhindert, dass ein negativer oder zu hoher positiver Störimpuls an der nachfolgenden Schaltung Schaden anrichtet.
Das Signal gelangt über einen Schmitt-Trigger an das D-Flip-Flop 3. Sobald ein Leseimpuls an den Port RA generiert wird (MOVF porta) verhindert das FF einen Wechsel an seinem Ausgang. Das Signal an dessen Eingang wird "quasi" eingefroren. Es gelangt über eine Verstärkerstufe an den internen Datenbus und von dort an seinen Bestimmungsort z.B. das W-Register (MOVF porta, W).

Im Eingangsmodus müssen die beiden MOSFETs am Ausgang abgeschaltet sein. Dies wird durch den Inhalt des Tris-Latchs (D-FF2) erreicht. Steht hier eine 1, so wird der Ausgang des UND-Gatters durch eine 0 am Eingang 2 immer auf 0 gehalten und so der N-MOSFET dauerhaft gesperrt. Ähnliches gilt für das ODER-Gatter. Dessen Ausgang ist durch eine 1 am unteren Eingang immer auf 1, was den P-MOSFET ebenfalls dauerhaft sperrt. In diesem Zustand sind die Ausgangspegel des Daten-Latches völlig irrelevant.

Output-Funktion

Um einen I/O-Pin als Ausgang arbeiten zu lassen, muss ins Tris-Latch eine 0 geschrieben werden. Das hat zu Folge, dass das UND- und ODER-Gatter nun

an ihrem Ausgang das ausgeben, was vom Daten-Latch kommt. Liegt dort eine 1 an, wird am Ausgang des UND-Gatters eine 0 erscheinen. Diese 0 am Gate des P-MOSFETs schaltet ihn durch und die positive Betriebsspannung wird so an den I/O-Pin gelegt. Der Ausgang des ODER-Gatters bleibt auf 0 und sperrt damit den N-MOSFET. Liegt am Ausgang des Daten-Latches eine 0, sperrt der obere MOSFET, da dessen Gate nun positiv ist. Im Gegensatz dazu wird das Gate des unteren MOSFETs positiv und dadurch schaltet dieser durch.

Arbeitet der I/O-Pin als Ausgang, können dennoch die Signale an diesem Pin gelesen werden. Damit lässt sich überwachen, ob das auszugebende Signal auch tatsächlich am Pin erscheint. Von dieser Möglichkeit macht z.B. der I²C-Bus gebrauch.

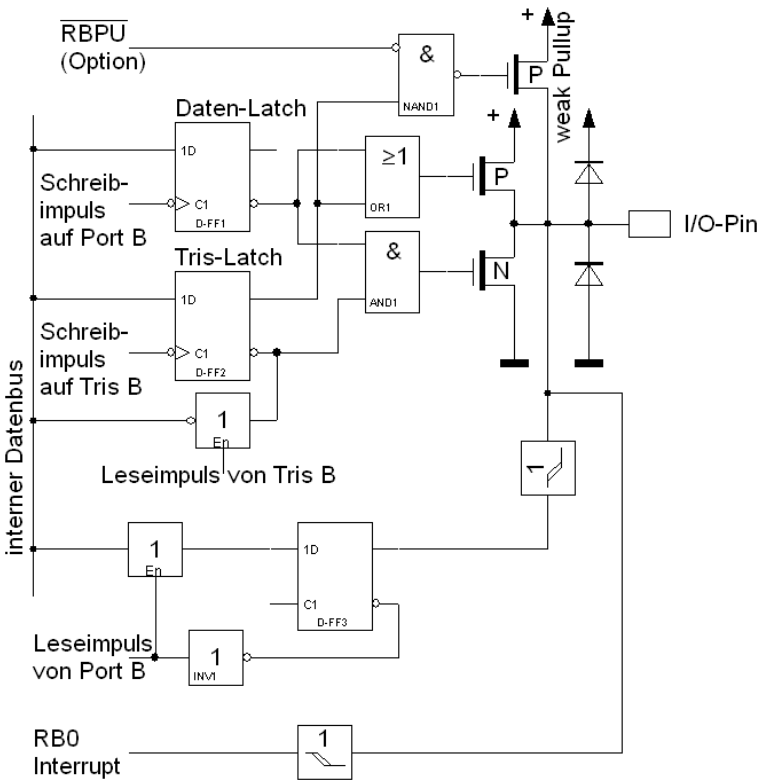


Abbildung 3: Schaltbild des RB0 Pins

Die I/O-Pins des Port RB sind ähnlich aufgebaut. Dort sind die Pins um sogenannte Weak-Pullups erweitert. Diese können durch ein Bit im OPTION-Register ein- und ausgeschaltet werden. Des weiteren ist teilweise deren Funktionalität um Interrupt-Eingänge erweitert.

1.9 PCLATH

Dieses Register kommt immer dann zum Zuge, wenn der Programmzähler in irgendeiner Weise per Befehl verändert wird. Diese Veränderungen erfolgen entweder durch Verzweigungsbefehle (GOTO, CALL) oder durch Manipulationen des PCL-Registers[6].

1.9.1 PCLATH bei Sprungbefehlen

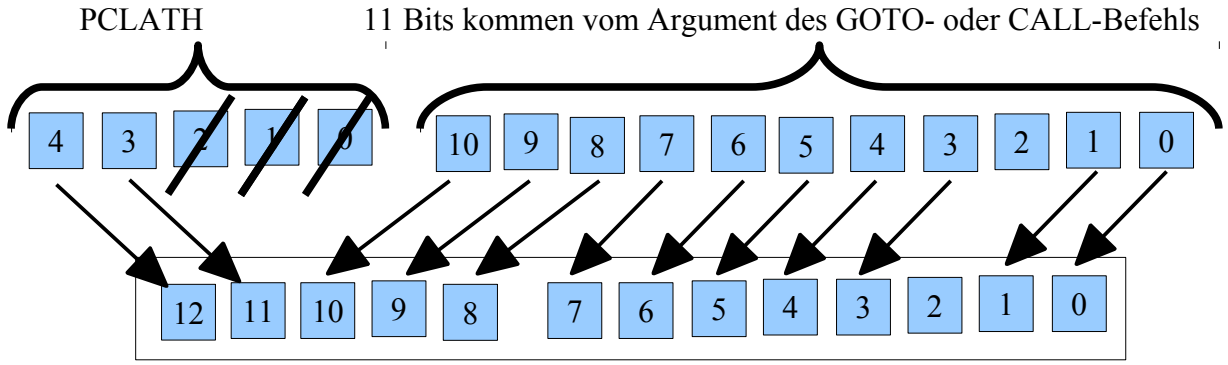


Abbildung 4: Zusammensetzen der Zieladresse bei Sprungbefehlen

Bei GOTO und CALL muss ein Sprung auf eine andere Programmseite erfolgen also aus einem 2k Block in einen anderen.

1.9.2 PCLATH bei Manipulation von PCL

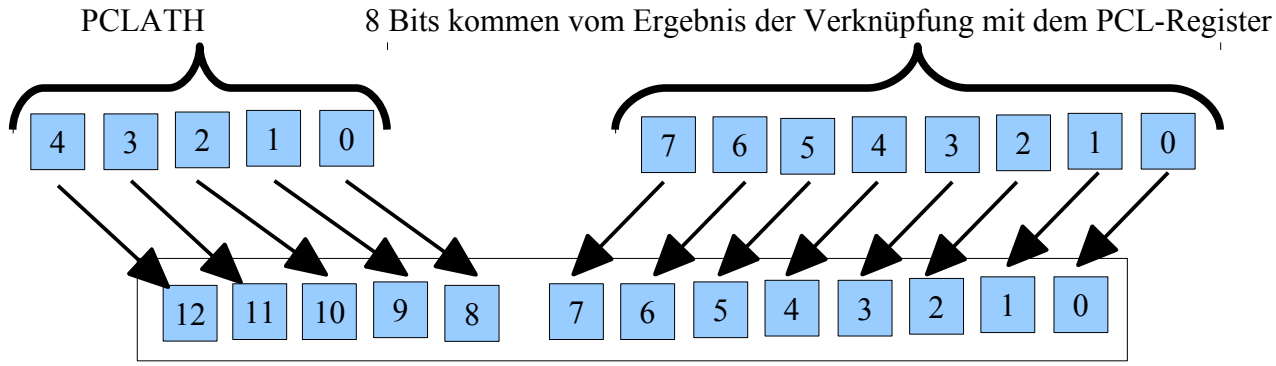


Abbildung 5: Zusammensetzen der Zieladresse bei Manipulation des PCL-Registers

Bei Zugriff auf Konstantentabellen im Programmspeicher mittels ADDWF PCL und einer Liste aus RETLW-Befehlen kommt der Mechanismus in Abbildung 5 zum Zug.

6 siehe dazu Kapitel 1.5 auf Seite 8.

1.10 INTCON

Im INTCON-Register werden die für die Interrupts relevanten Bits verwaltet. Dies sind:

Bit Nr.	Name	Funktion
7	GIE	Globales sperren aller Interrupts
6	EEIE	Interrupt erlaubt, wenn EEPROM fertig beschrieben ist
5	T0IE	Interrupt erlaubt bei einem Timerüberlauf
4	INTE	Interrupt erlaubt bei Flankenwechsel an RB0 (s. Option)
3	RBIE	Interrupt erlaubt bei Flankenwechsel an RB4 - RB7
2	T0IF	Ereignis Timerüberlauf ist eingetreten
1	INTF	Ereignis Flankenwechsel an RB0 eingetreten
0	RBIF	Ereignis Flankenwechsel an RB4 bis RB7 eingetreten

Weitere Informationen zu diesem Thema im Kapitel 12 auf Seite 47.

1.11 OPTION

Hier werden diverse zusätzliche Funktionen festgelegt. Dies sind:

Bit Nr.	Name	Funktion
7	$\overline{\text{RBPU}}$	aktivieren der Pullup-Widerstände an Port RB
6	INTEDG	legt die auslösende Flanke an RB0 fest (Interrupt)
5	T0CS	Taktquelle für den Timer 0
4	T0SE	legt zählende Flanke für Timer 0 festgelegt
3	PSA	Prescaler an Timer 0 oder an Watchdog
2	PS2	Prescalerwert
1	PS1	
0	PS0	

2 Assemblerprogrammierung

2.1 Der Befehlssatz

Betrachtet man den Befehlssatz des PIC Mikrocontrollers fällt die kleine Anzahl von Befehlen auf. Man rechnet diesen Mikrocontroller zu den sogenannten RISC Prozessoren. RISC steht für Reduced Instruction Set Computer. Die Befehle werden sehr schnell, meist in nur einem Befehlstakt abgearbeitet. Dafür müssen komplexere Befehle durch eine Befehlssequenz nachgebildet werden.

In der Tabelle 1 sind alle Befehle aufgelistet. Die exakte Beschreibung eines jeden einzelnen Befehls befindet sich im Anhang.

Der Buchstabe f bedeutet Fileregister und ist eine Adresse des Datenspeichers. Sie ist maximal 7 Bit breit und kann somit die Werte 0 bis 127 bzw. 0x00 bis 0x7F annehmen. Wird eine Fileregisteradresse angegeben, wird immer deren Inhalt verarbeitet bzw. verknüpft.

Das sogenannte Destinationbit d gibt an, wohin das Ergebnis der Verknüpfung abgelegt werden soll. Ist d = 0, kommt das Ergebnis ins W-Register, bei d = 1 ins Fileregister.

Bei den Bitbefehlen wird das zu verwendende Bit b im Befehl angegeben. Es kann Werte von 0 bis 7 annehmen.

Die Literale werden in der Übersicht durch den Buchstaben k gekennzeichnet. Microchip, der Hersteller des PIC Mikrocontrollers, bezeichnet Konstante als Literale. Sie können entweder 8 Bit lang sein, wenn es sich um Arithmetikbefehle oder logische Verknüpfungen handelt oder im Fall von Sprungbefehlen (GOTO und CALL) 11 Bit lang.

Manche Befehle beeinflussen die Flags im Statusregister. Welche dies sind, ist in der letzten Spalte aufgeführt. Ein Carryflag (C) zeigt den Über- oder Unterlauf einer 8-Bit Rechenoperation an. Hier ist aber auf die seltsame Eigenheit des PICs zu achten. Mehr dazu unter 4.4.

Das Digitcarry-Flag (DC) zeigt den Überlauf vom unteren auf das obere Halbbyte an.

Das Zeroflag (Z) wird gesetzt, wenn das Ergebnis der Operation 0 ergibt. Aller-

dings verändert auch der Befehl MOVF f,d dieses Flag. Mehr dazu unter Kapitel 4.1.

Mnemonic		Beschreibung	Befehls-takte	Flag
Byteorientierte Befehle				
ADDWF	f,d	Addiert W und Inhalt von f	1	C, DC, Z
ANDWF	f,d	UND-Verknüpfung von W und Inhalt von f	1	Z
CLRF	f	löscht den Inhalt von f	1	Z
CLRW		löscht das W-Register	1	Z
COMF	f,d	Bildet Komplement vom Inhalt von f	1	Z
DECF	f,d	Dekrementiert den Inhalt von f	1	Z
DECFSZ	f,d	Dek. f und überspringt bei 0 den nächsten Befehl	1 (2)	
INCF	f,d	Inkrementiert den Inhalt von f	1	Z
INCFSZ	f,d	Ink. f und überspringt bei 0 den nächsten Befehl	1 (2)	
IORWF	f,d	ODER-Verknüpfung von W und Inhalt von f	1	Z
MOVF	f,d	Holt den Wert aus Adresse f	1	Z
MOVWF	f	Schreibt Inhalt von W nach Adresse f	1	
NOP		Leerbefehl	1	
RLF	f,d	Rotiert den Inhalt von f nach links durchs Carry	1	C
RRF	f,d	Rotiert den Inhalt von f nach rechts durchs Carry	1	C
SUBWF	f,d	Subtrahiert W vom Inhalt der Adresse f	1	C, DC, Z
SWAPF	f,d	Vertauscht die beiden Halbbytes in Adresse f	1	
XORWF	f,d	EXCLUSIV-ODER von W und f	1	Z
Bitorientierte Befehle				
BCF	f,b	Löscht Bit b in Adresse f	1	
BSF	f,b	Setzt Bit b in Adresse f	1	
BTFSC	f,b	Testet Bit b an Adresse f und springt, wenn es 0 ist	1 (2)	
BTFSS	f,b	Testet Bit b an Adresse f und springt, wenn es 1 ist	1 (2)	
Literal und Steuerbefehle				
ADDLW	k	Addiert Literal k zum W-Register	1	C, DC, Z
ANDLW	k	Literal k wird UND verknüpft mit dem W-Register	1	Z
CALL	k	Unterprogrammaufruf an Adresse k	2	
CLRWDT		löscht den Watchdog	1	$\overline{TO}$, $\overline{PD}$
GOTO	k	Sprung zur Adresse k	2	
IORLW	k	Literal wird mit dem W-Register ODER verknüpft	1	$\overline{Z}$
MOVLW	k	Lädt das Literal ins W-Registers	1	
RETFIE		Rückkehr aus der Interruptroutine, setzte GIE	2	
RETLW	k	Rückkehr aus einem Unterprogramm, lädt k in W	2	
RETURN		Rückkehr aus einem Unterprogramm	2	
SLEEP		Geht in den Stand By Modus	1	$\overline{TO}$, $\overline{PD}$
SUBLW	k	Subtrahiert W vom Literal k	1	C, DC, Z
XORLW	k	EXCLUSIV ODER Verknüpfung von W und k	1	Z

Tabelle 1: Befehlsübersicht

In der zweitletzten Spalte werden die benötigten Befehlstakte angegeben. Übli-

cherweise werden die Befehle in einem Befehlstakt (= 4 Quarztakte) abgearbeitet. Bei manchen muss aber der in der Pipeline (Instruction Register) liegende Befehl gelöscht werden, was einen zusätzlichen Befehlstakt bedingt.

2.2 Arithmetische und logische Verknüpfungen

Bei arithmetischen und logischen Verknüpfungen sind immer zwei Argumente notwendig. Diese können entweder im Speicher stehen und sind somit während des Programmablaufs veränderbar oder einer der beiden ist ein Literal (Konstante). Um die Verknüpfung durchzuführen, muss ein Argument im W-Register stehen, das andere in einer Speicherstelle oder als Literal im Befehlscode.

Erfolgt die Verknüpfung mit dem Inhalt einer Speicherstelle, kann man mittels des sogenannten Destinationbits (d) bestimmen, ob das Ergebnis der Verknüpfung in das W-Register (d = 0) oder in die Speicherstelle, aus der das Argument kam (d=1), zurückgeschrieben werden soll.

2.3 Die Befehle GOTO und CALL

GOTO verzweigt als unbedingter Sprung zu einer absoluten Programmadresse. Dazu wird das Argument in den Programmzähler geladen. Ggf. werden zwei zusätzliche Bits aus dem PCLATH-Register mit in den Programmzähler geladen. Dies wird allerdings nur dann benötigt, wenn die Zieladresse außerhalb des aktuellen 2k großen Programmblockes (Program Page) liegt. In allen anderen Fällen bleibt das PCLATH unverändert.

Der CALL- ist fast identisch zum GOTO-Befehl. Hier wird der aktuelle Programmzählerwert, der zu diesem Zeitpunkt bereits auf den nächsten auszuführenden Befehl zeigt, auf den Stack[7] geschrieben.

7 siehe auch: Kapitel Fehler: Verweis nicht gefunden auf Seite Fehler: Verweis nicht gefunden

3 Der Stack

Der Stack des PIC 16Fxx ist als sogenannter Ringspeicher mit acht Adressen implementiert. Ein Stackpointer (3 Bits) zeigt auf den jeweils aktuellen Eintrag (TOS[8]). Jede Adresse kann den gesamten Programmzähler (nicht nur den PCL-Teil) aufnehmen.

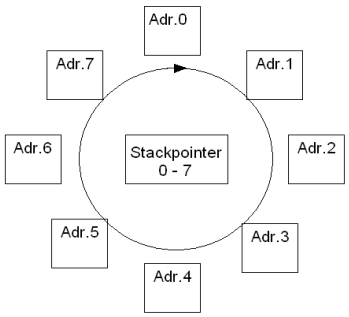


Abbildung 6: Der Stack als Ringpuffer

Der Stack ist ausschließlich für die Ablage der Rücksprungadressen vorgesehen. Durch diese Ringanordnung gehen Rücksprungadressen verloren, falls mehr als acht Unterprogramme ineinander verschachtelt werden.

8 Top of Stack

4 Inhalt einer Speicherstelle auf einen bestimmten Wert prüfen

Will man auf Gleichheit prüfen, ist ein XOR-Befehl die erste Wahl. Denn dieser verändert im Statusregister nur das Zeroflag.

4.1 Überprüfen einer Speicherstelle auf Null

Lade- bzw. Move-Befehle beeinflussen üblicherweise keine Flags. Eine Ausnahme bildet der MOVF-Befehl des PICs. Er beeinflusst das Zeroflag im Statusregister.

```
MOVF    adr[9]      ;Der Inhalt von adr wird gelesen und
                    ;in adr zurück geschrieben.

MOVF    adr, W[10]  ;Der Inhalt von adr wird gelesen und
                    ;ins W-Register geschrieben
```

Wird dieser Befehl ausgeführt und wenn danach der Inhalt von adr Null ist, wird das Zeroflag im Statusregister (Bit 2) gesetzt. Ist der Inhalt nicht Null, wird das Zeroflag auf 0 gesetzt.

Damit lässt sich der Inhalt einer Speicherzelle auf Null prüfen, ohne dass ein anderes Register verändert wird (außer dem Statusregister).

4.2 Vergleich zweier Speicherstellen

```
MOVF    adr1, W     ;ein Argument ins W-Register holen
XORWF   adr2, W     ;XOR verknüpfen und Ergebnis in
                    ;W-Register, so bleiben adr1 und
                    ;adr2 unverändert.
BTFSC   status, Zflag
GOTO    sindGleich
GOTO    sindUngleich
```

4.3 Vergleich einer Speicherstelle mit einem Literal

```
MOVLW   literal     ;das Literal kommt ins W-Register
```

9 Gleichbedeutend mit MOVF adr,1
10 Gleichbedeutend mit MOVF adr,0

```
XORWF   adr             ;das Ergebnis der XOR-Verknüp
                        ;fung kommt nach adr
BTFSC   status, Zflag
GOTO    sindGleich
GOTO    sindUngleich
```

oder:

```
MOVLW   literal         ;das Literal kommt ins W-Register
XORWF   adr,W           ;das Ergebnis der XOR-Verknüp
                        ;fung kommt ins W-Register
BTFSC   status, Zflag
GOTO    sind Gleich
GOTO    sind Ungleich
```

4.4 Prüfen, ob der eine Wert größer als der andere ist

In diesem Fall muss man mit dem Subtraktionsbefehl arbeiten. Allerdings hat dieser beim PIC so seine „Eigenheiten“. Das Carryflag wird anders gesetzt, als es dem üblichen Verständnis entspricht. Wird das Ergebnis größer oder gleich Null, wird das Carryflag gesetzt, sonst zurückgesetzt.

Dieses seltsame Verhalten ist auf die Art und Weise wie der PIC Mikrocontroller die Subtraktion durchführt und auf einen Maskenfehler zurückzuführen[11]. Die Subtraktion wird durch die Addition des Zweierkomplements realisiert.

Als Beispiel dient die Aufgabe 10 – 15 = – 5. In Hexdarstellung bedeutet die 10 ein 0x00001010, eine 15 ein 0x00001111. Das Zweierkomplement davon ist 0x11110001.

```
          0x00001010
+         0x11110001
-------------------------------------
          0x11111011          = 251
```

Dieser Binärwert entspricht dem Ergebnis – 5. Nur erfolgte bei der obigen Berechnung an keiner Stelle ein Überlauf, so dass das Carryflag nicht gesetzt ist. Und hier kommt der Maskenfehler ins Spiel. Normalerweise wird das Carry- und Digitcarry nach einer Subtraktion invertiert. Das wurde hier schlicht vergessen.

11 Dieser Maskenfehler wird heute nicht als Fehler, sondern als Feature „verkauft“.

Was passiert aber, wenn der Subtrahend kleiner als der Minuend, das Ergebnis somit positiv ist? Das zeigt die folgende Aufgabe: 15 – 10 = 5.

```
                0x00001111
          +     0x11110110          Zweierkomplement
Übertrag    1     11111100
-------------------------------------------------
                0x00000101
```

Hier wird ein Ergebnisüberlauf erzeugt, der ins Carryflag geschrieben wird.

Als letztes interessiert uns noch, was bei der Berechnung von 15 – 15 = 0 passiert.

```
                0x00001111
          +     0x11110001
Übertrag    1     11111110
-------------------------------------------------
                0x00000000
```

Das Ergebnis ist wie erwartet eine 0, aber auch hier wird das Carryflag auf 1 gesetzt.

Das bedeutet: Ist der Subtrahend kleiner oder gleich dem Minuend, wird das Carryflag gesetzt. Ist hingegen der Subtrahend größer als der Minuend, wird das Carryflag gelöscht.

Wichtig: Der Inhalt des W-Registers wird vom Literal oder dem Inhalt der Speicherstelle abgezogen!

Literal – W bzw. Registerinhalt – W

4.5 Vergleich zweier Speicherstellen auf größer / kleiner

```
MOVF    adr1, W     ;ein Argument ins W-Register holen
SUBWF   adr2, W     ;subtrahiere W vom Inhalt von adr2
                    ;und schreib Ergebnis ins ;W-Regis
                    ;ter, so bleiben adr1 und adr2 unver
                    ;ändert.
BTFSC   status, Zflag
GOTO    sindGleich  ;Wert in adr1 ist gleich dem Wert in
                    ;adr2
BTFSC   status,CFlag
```

```
GOTO    kleiner     ;Wert in adr1 ist kleiner als der in
                    ;adr2
GOTO    groesser    ;Wert in adr1 ist größer als der in
                    ;adr2
```

4.6 Ist Inhalt der Speicherstelle größer / kleiner als Literal

```
MOVLW   literal     ;das Literal kommt ins W-Register
SUBWF   adr         ;das Ergebnis der XOR-Verknüp-
                    ;fung kommt nach adr
BTFSC   status, Zflag
GOTO    sindGleich  ;Wert in adr ist gleich dem Literal
BTFSC   status,CFlag
GOTO    kleiner     ;Literal ist kleiner als Wert in adr
GOTO    groesser    ;Literal ist größer als Wert in adr
```

oder:

```
MOVLW   literal     ;das Literal kommt ins W-Register
SUBWF   adr,W       ;das Ergebnis der XOR-Verknüp-
                    ;fung kommt ins W-Register
BTFSC   status, Zflag
GOTO    sindGleich  ;Wert in adr ist gleich dem Literal
BTFSC   status,CFlag
GOTO    kleiner     ;Literal ist kleiner als Wert in adr
GOTO    groesser    ;Literal ist größer als Wert in adr
```

5 Einfache Rechenroutinen

5.1 8-Bit Addition

Bei 8-Bit Prozessoren gehört die 8-Bit Addition in der Regel zum Standardbefehlssatz. Beim PIC 16Fxx sind dies die beiden Befehle ADDLW und ADDWF. Der erste addiert eine Konstante zum W-Register. Je nach Ergebnis werden die Flags Z, DC und C im Statusregister gesetzt oder gelöscht. Generell gilt:
Ist das Ergebnis einer Operation Null wird das Z-Flag gesetzt. Ist es ungleich Null, wird es zurück auf 0 gesetzt.
Tritt ein Überlauf auf, d.h. das Ergebnis ist größer als der Zahlenbereich von 8 Bits, dann wird das Carryflag gesetzt.
Das DC-Flag (Digit Carry) wird dann gesetzt, wenn es zu einem Überlauf der unteren 4 Bits kommt.

Beispiele:

```
MOVLW   25H
ADDLW   12H
```

Am Ende dieser Programmsequenz steht im W-Register 37H. Das Zeroflag ist zurückgesetzt ebenso das Carry- und DC-Flag.

```
MOVLW   25H
ADDLW   0DBH
```

Das Ergebnis in W lautet 00H. Das Zeroflag ist gesetzt. Auch das Carryflag ist 1, da der Zahlenbereich überschritten wurde. Das DC-Flag zeigt an, dass die Addition der jeweils unteren vier Bits einen Überlauf ergeben.

```
      5H      0101B
+     0BH     1011B
```

Die Addition ergibt 16. Die unteren vier Bits können nur Zahlen zwischen 0 und 15 darstellen, deshalb entsteht ein Überlauf von Bit 3 auf Bit 4. Dieser Überlauf befindet sich im DC-Flag.

5.2 8-Bit Subtraktion

Bei den meisten Prozessoren gehört neben der Addition auch die Subtraktion zum Befehlssatz. Dort wo dies nicht der Fall ist, behilft man sich mit der Addition des Zweierkomplements. Diese Technik wird auch in einem Hardwarerechenwerk eingesetzt. Somit wird ein SUB-Befehl vom Prozessor in eine Addition des Zweierkomplements umgesetzt.

Beim PIC-Mikrocontroller muss bei den Subtraktionsbefehlen einiges beachtet werden. Das Carryflag ist invertiert, ebenso das DC-Flag. In den älteren Datenblättern wird dies noch als Designfehler bezeichnet, in den Neueren wird das Carry- zu einem Borrow-Bit umdefiniert. Gleiches gilt für das DC-Flag. Dieser Umstand ist in Kapitel 4.4 ausführlich beschrieben.

5.3 16-Bit Addition

Um bei einem 8-Bit Prozessor eine 16-Bit Addition durchzuführen benötigt man ein entsprechendes Programm. Diese Addition wird in Teilschritten durchgeführt. Zuerst werden die unteren 8 Bit addiert, danach die oberen 8 Bit. Da bei der Addition der unteren Hälfte ein Übertrag auftreten kann, muss dieser bei der Addition der oberen Hälfte berücksichtigt werden. Der PIC 16Fxx kennt keinen ADC-Befehl[12], bei dem das Carrybit gleich mit verarbeitet wird. Deshalb muss diese Berücksichtigung des Carryflags in der Software erfolgen.

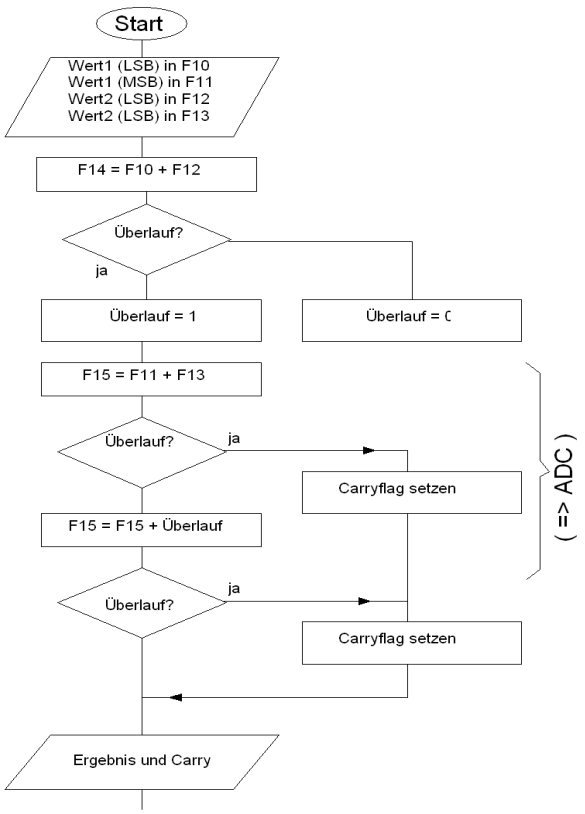


Abbildung 7: Flussdiagramm einer 16-Bit Addition

In Abbildung 7 erkennt man, dass für die Addition der unteren 8 Bits der einfache Additionsbefehl ausreicht. Erst bei der zweiten Addition muss ein eventuel-

12 ADC = Addiere unter Berücksichtigung des Carrybits die beiden Argumente.

ler Überlauf mit eingerechnet werden. Hier wird dafür eine eigene Variable (Überlauf) genutzt. Nun können sowohl bei der Addition der oberen 8 Bit als auch bei der Addition des Überlaufs ein erneuter Überlauf entstehen. Dieser wird im Carryflag gesetzt.

5.4 8-Bit Multiplikation

Die 8-Bit Multiplikation wird wie auf dem Papier durchgeführt. Einer der Werte wird immer dann zum Ergebnis addiert, wenn beim anderen an der jeweiligen Stelle eine 1 steht. Dazu nutzt man die Rotationsbefehle zusammen mit dem Carryflag. Wichtig ist auch das Linksschieben des Ergebnisses. Dabei ist darauf zu achten, dass immer eine 0 hineingeschoben wird.

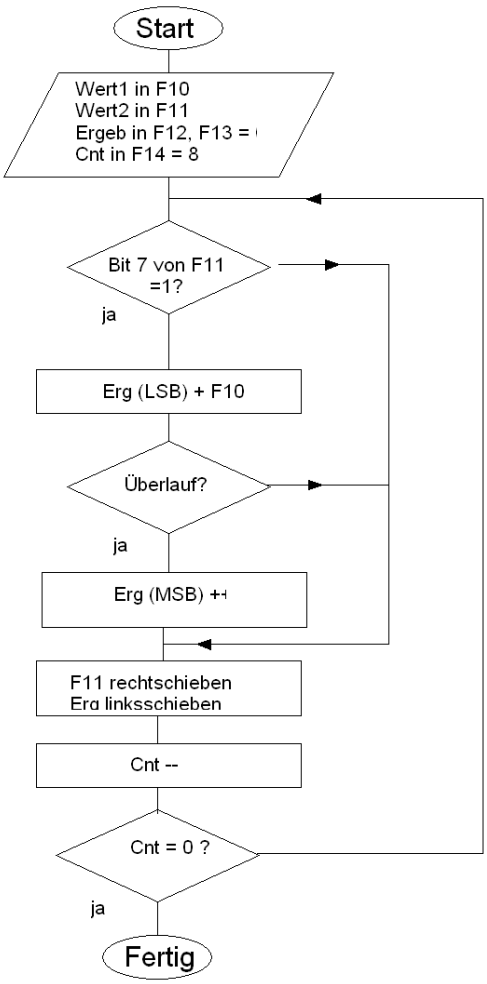


Abbildung 8: Die 8-Bit Multiplikation

5.5 8-Bit Division

Auch hier ist das Vorgehen dem schriftlichen Dividieren im Zehnersystem ähnlich. Der einzige Unterschied ist eine kleine Vorbereitung des Divisors. Dieser wird anfangs so lange nach links geschoben, bis an der höchsten Stelle eine 1 steht. Dann subtrahiert man vom Dividenden den Divisor. Ist der Rest kleiner als 0 wird die Subtraktion widerrufen und im Ergebnis eine 0 eingetragen. Im anderen Fall wird eine 1 eingetragen und mit dem Restwert weitergearbeitet.

Zahlenbeispiel:

Dividend A	=	126	=	0 1 1 1 1 1 1 0
Divisor B	=	25	=	0 0 0 1 1 0 0 1

Divisor B nach links schieben = Divisor B' = 1 1 0 0 1 0 0 0 (-74)
Es sind insgesamt 3 Schiebungen nötig. Damit steht fest, auch das Endergebnis hat 3 Stellen. Da man mit dem Startwert 1 im Zähler beginnt, steht in diesem Fall dort eine 4.

1. Schritt Wenn Zähler = 0, dann fertig
Ergebnis nach links rotieren.
A - B'

	0 1 1 1 1 1 1 0 (126)
-	1 1 0 0 1 0 0 0 (200)

Rest	1 1 0 1 1 0 1 1 0 (-74)

Da das Ergebnis negativ ist, wird das Zwischenergebnis ignoriert und im Ergebnisregister eine 0 eingetragen. Danach wird der Divisor nach rechts verschoben und der Zähler um 1 erniedrigt (3).

2. Schritt Wenn Zähler = 0, dann fertig
Ergebnis nach links rotieren.

	0 1 1 1 1 1 1 0 (126)
+	0 1 1 0 0 1 0 0 (100)
	----------
	0 0 0 0 1 1 0 1 0 (26)

Das Ergebnis ist positiv. Deshalb wird ins Ergebnisregister eine 1 eingetragen. Das Zwischenergebnis ist der neue Dividend. Der Zähler wird dekrementiert (2).

3. Schritt Wenn Zähler = 0, dann fertig
Ergebnis nach links rotieren.

A - B'		0 0 0 1 1 0 1 0 (26)
	-	0 0 1 1 0 0 1 0 (50)

	Rest	1 1 0 1 1 0 1 1 0 (-24)

Da das Ergebnis negativ ist, wird das Zwischenergebnis ignoriert und im Ergebnisregister eine 0 eingetragen. Danach wird der Divisor nach rechts verschoben und der Zähler um 1 erniedrigt (1).

4. Schritt Wenn Zähler = 0, dann fertig
Ergebnis nach links rotieren.

A - B'		0 0 0 1 1 0 1 0 (26)
	-	0 0 0 1 1 0 0 1 (25)

	Rest	0 0 0 0 0 0 0 1 (1)

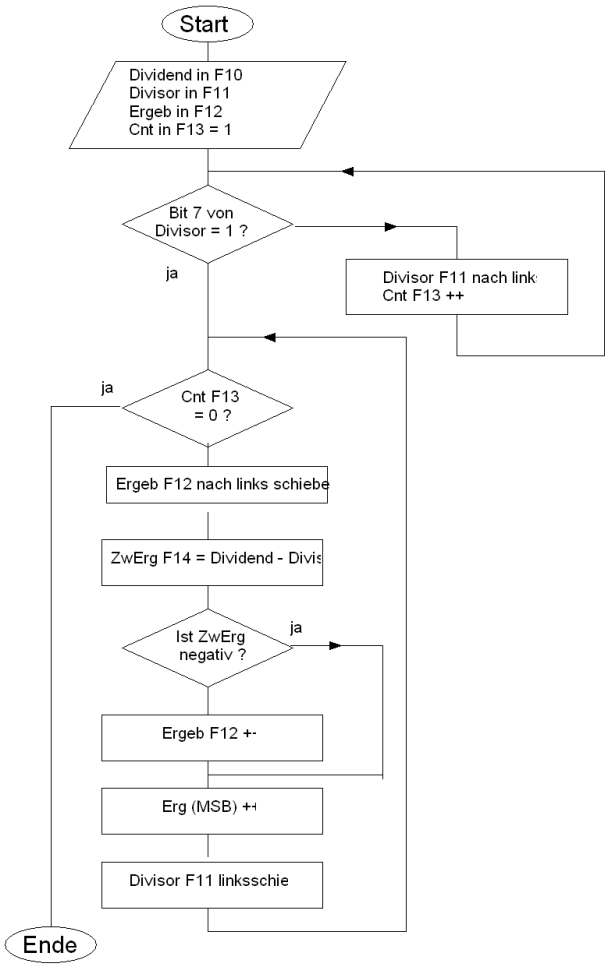


Abbildung 9: Eine 8-Bit Division

Das Ergebnis ist positiv. Deshalb wird ins Ergebnisregister eine 1 eingetragen. Das Zwischenergebnis ist der neue Dividend. Der Zähler wird dekrementiert (0).

6. Schritt Wenn Zähler = 0, dann fertig

Das Ergebnis ist 101 also dezimal 5. Im Zwischenergebnis steht der Rest, in unserem Fall 1.

126 : 25 = 5 Rest 1

In Abbildung 9 ist das dazugehörende Flussdiagramm zu sehen.

```
DIVIDIERE   CLRF     ERG          ;Variablen mit Start-
            MOVLW    1            ;werten vorbelegen
            MOVWF    SZ
DIV_L1:     BTFSC    DIVISOR      ;schiebe bis höchstes
            GOTO     DIV_W1       ;Bit auf 1 steht
            RLF      DIVISOR
            INCF     SZ
            GOTO     DIV_L1
DIV_W1:     MOVF     SZ           ;SZ = 0 ?
            BTFSC    STATUS, Z
            GOTO     DIV_END      ;ja, fertig
            RLF      ERG
            DECF     SZ           ;Schleifenzähler
            MOVF     DIVISOR, W
            SUBWF    DIVIDEND, W
            BTFSS    STATUS,C     ;Vorsicht, Carry
            GOTO     DIV_W2       ;wenn negativ
DIV_POS:    MOVWF    DIVIDEND     ;neuer Dividendwert
            BSF      ERG,0        ;1 in ERG setzen
DIV_W2      RRF      DIVISOR
            GOTO     DIV_W1

DIV_END
```

6 Zweistelliger BCD-Zähler

BCD bedeutet Binary Coded Digit. Es wird eine Zahl zwischen 0 bis 9 als Binärzahl dargestellt. Dafür benötigt man jedoch lediglich 4 Bits. Somit lassen sich in die restlichen Bits eine weitere Stelle definieren. Dadurch kann ein zweistelliger Zähler von 00 bis 99 realisiert werden. Um nun einen solchen BCD-Zähler zu realisieren kann man nicht einfach binär hoch zählen, wie es mit dem INC-Befehl möglich wäre.

Sobald an den unteren 4 Bits (unteres Nibble oder Halbbyte) eine 0x1010 steht, muss diese Stelle auf 0 gesetzt und das obere Halbbyte um 1 erhöht werden. Und wenn dort wiederum eine 0x1010 erreicht wird, muss auch diese Stelle zurück gesetzt werden sowie ein eventueller Übertrag auf mögliche weitere Stellen erzeugt werden. Somit muss sowohl das untere als auch das obere Halbbyte auf die jeweilige Zählgrenze überprüft werden.

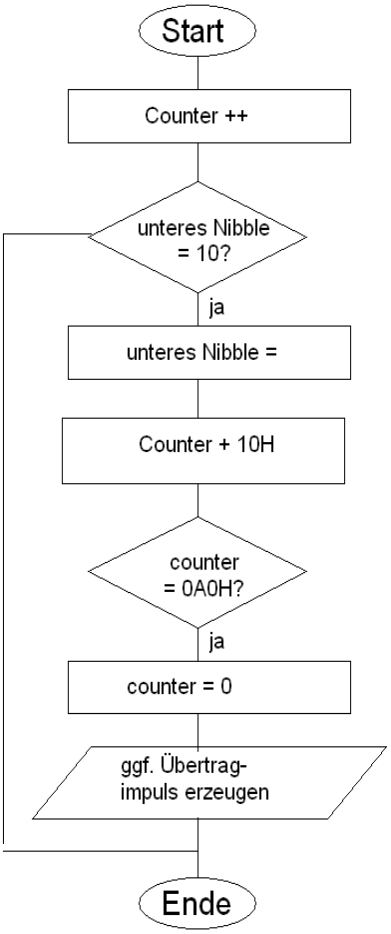


Abbildung 10: BCD-Zählroutine

Das Ablaufdiagramm für den eigentlichen Zähler ist in Abbildung 10 dargestellt. Darin wird lediglich ein Zähldurchlauf erfasst. Die Bedingung wann gezählt wird, lässt sich vor dieser Routine platzieren.

7 Siebensegment Multiplexanzeige

Anzeigen sind ein wichtiges Medium, um dem Bediener passende Rückmeldungen zu liefern. Neben einfachen Siebensegmentanzeigen (LCD oder LED) gibt es auch weitere Anzeigetypen, z.B. alphanumerische Anzeigen, die i.d.R. in der LCD- oder OLED-Technologie realisiert werden. Im Folgenden soll eine 2-stellige Siebensegmentanzeige aus LEDs besprochen werden.

Siebensegmentanzeigen besitzen sieben von einander unabhängig ansteuerbare Segmente. Diese werden üblicherweise gemäß Abbildung 11 bezeichnet.

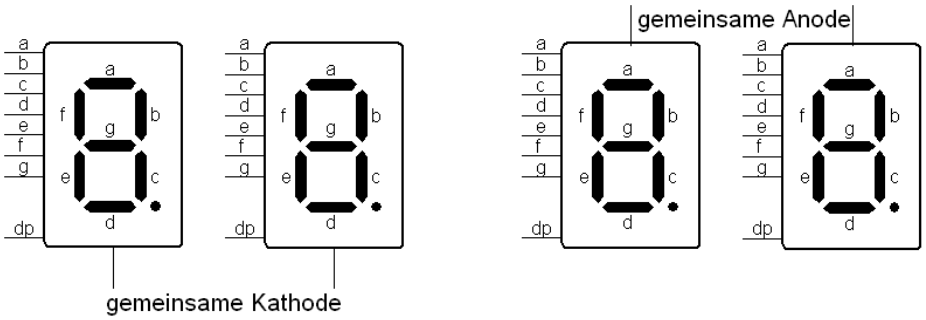
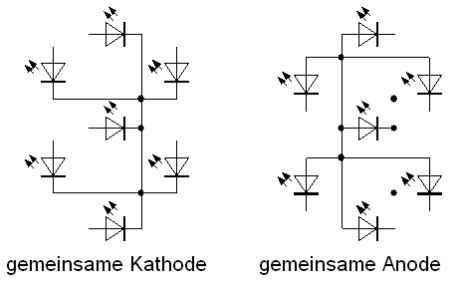


Abbildung 11: Siebensegmentanzeigen werden durch gemeinsame Anschlüsse gekennzeichnet

Um Leitungen zu sparen werden alle Anoden bzw. Kathoden der jeweils gleichen Segmenten zusammengeführt. Die freibleibenden Anschlüsse innerhalb einer Ziffer werden ebenfalls zusammengefasst.



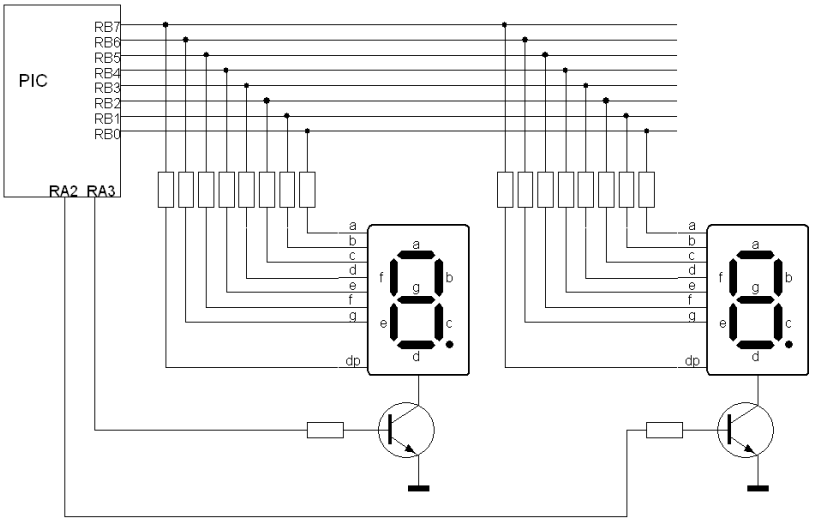


Abbildung 12: Gesamtschaltbild einer 2-stelligen Anzeige

In Abbildung 12 ist eine 2-stellige Anzeige mit der notwendigen Beschaltung dargestellt. Bei den Anzeigeelementen handelt es sich um Type mit gemeinsamer Kathode. Die Widerstände an den Segmenten dienen der Strombegrenzung. Die Ziffern werden nacheinander über die Transistoren ein- und ausgeschaltet. Damit der Eindruck einer stehenden Anzeige entsteht Wichtig ist, dass die Ansteuerung der einzelnen Element so schnell erfolgt, dass das Auge den einzelnen Bilder nicht mehr folgen kann. Diese sogenannte Bildwechselfrequenz sollte bei ca. 70 Hz liegen um Interferenzen zwischen Beleuchtung und Anzeige zu vermeiden.
Weiterhin muss dies Frequenz relativ konstant sein, um keine Helligkeitsunterschiede in der Anzeige entstehen zu lassen.

Um das zu erreichen muss die Durchlaufzeit des Programm einigermaßen konstant sein, egal was gerade abgearbeitet wird. Kann dies nicht gewährleistet werden, muss auf einen interruptgesteuerten Aufruf der Anzeigeroutine ausgewichen werden, was wiederum Auswirkung auf die Gesamtdurchlaufzeit hat (Programm-Overhead). In Abbildung 13 ist das Ablaufdiagramm für die Ausgabe einer Stelle abgebildet, d.h. um 2 Stellen anzuzeigen muss das Unterprogramm 2x aufgerufen werden. Dabei läuft ein Zeiger mit, der auf die auszugebende Adresse zeigt und auch die passende Stelle repräsentiert. Falls die Durchlaufzeit der Hauptschleife zu kurz ist, kann es passieren, dass die Anzeige zu dunkel ist. Dagegen hilft im einfachsten Fall eine kurze Verzögerung.

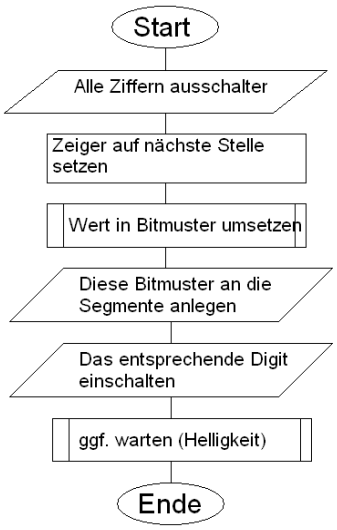


Abbildung 13

8 Flankenerkennung mittels Polling

Eine Flankenerkennung wird bei Mikrocontroller-Programmen sehr oft benötigt. Flanken erzeugende Bauteile sind z.B. Taster, Lichtschranken, Reedkontakte, Encoder etc. Je nach Anwendung werden unterschiedliche Anforderungen an eine Flankenerkennung gestellt.

Eine der wichtigsten Unterscheidungen ist, ob es sich um einen blockierende oder nicht blockierende Funktion handelt. Bei einer blockierenden Funktion verweilt das Programm so lange in ihr, bis die entsprechende Flanke am Eingang erscheint. Es können keinerlei weitere Abläufe parallel zu dieser Funktion vorgenommen werden. Dieses Verhalten ist mit modalen Meldefenstern in Desktopanwendungen vergleichbar. Auch dort muss diese Meldung zuerst quittiert werden, bevor ein anderes Fenster fokussiert werden kann.

Im Gegensatz dazu kann man mit nicht blockierende Funktionen, eine Quasiparallelität der Programme programmieren. Nachteil bei dieser Methode ist die verlängerte Zykluszeit, die eine Mindestdauer des Impulses, dessen Flanke erkannt werden soll, voraussetzt. Auch eine Zunahme der Latenzzeit ist wahrscheinlich. Das ist die Zeit, die zwischen dem Auftreten der Flanke und dem Erkennen durch das Programm, vergeht. Wird diese Latenzzeit intolerabel, muss auf die Hilfe von Interrupts zurückgegriffen werden.

Das Prinzip der Flankenerkennung mittels Polling ist der Vergleich zwischen dem aktuellen Eingangspegel und einem in der Vergangenheit liegenden Wert. Sind beide gleich, gab es keinen Flankenwechsel. Dieser liegt nur dann vor, wenn sich der neue Pegel vom alten unterscheidet. Die Abbildung 14 verdeutlicht diesen Sachverhalt.

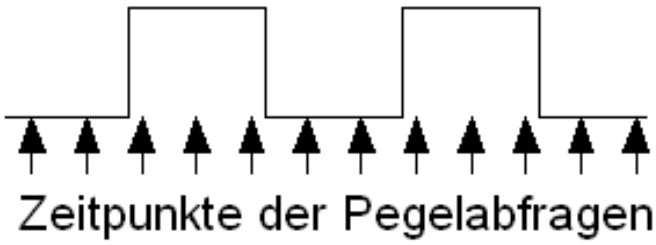
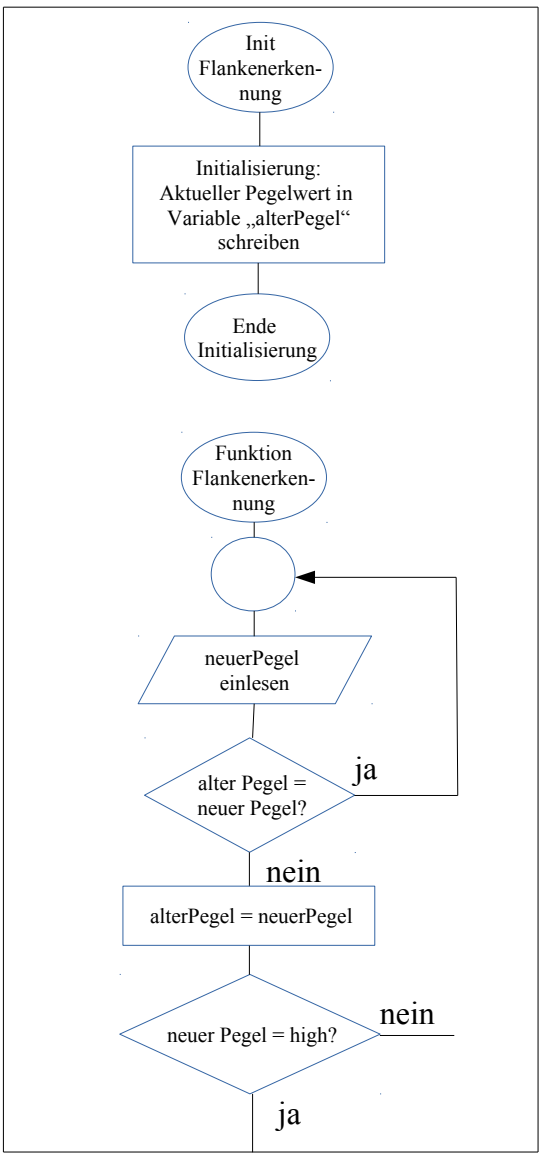


Abbildung 14: Abtastzeitpunkte bei Flankenerkennung

Blockierende Funktion:



Bei der Initialisierung wird der aktuelle Pegel in einer Variablen gespeichert. Dieser wird dann mit dem neu eingelesenen Pegel verglichen, um so einen Pegelwechsel (= Flanke) zu erkennen. Diese Initialisierung erfolgt i.d.R. beim Programmstart.

Einsprungpunkt (Ziellabel), falls es keinen Pegelwechsel gegeben hat.

Aktueller Pegel wird eingelesen

und mit altem Pegel verglichen. Sind beide gleich, fand keine Pegelwechsel statt, ansonsten wurde eine Flanke (welche ist noch unklar) gefunden.
Den neuen Pegel muss man sich merken.

Ist der neue Pegel ein High, wurde eine Low-High Flanke gefunden (Ausgang ja).
Ist aber der neue Pegel Low, handelt es sich um eine High-Low Flanke (Ausgang nein)

Man erkennt, dass die Funktion nur dann verlassen wird, wenn eine Flanke gefunden wurde. Somit wartet das Programm u.U. ewig, falls keine Flanke kommt oder der Sensor bzw. der Eingang defekt ist. Diese Art nennt man blockierende Routine.

PIC-Assemblerprogramm

```
;INIT_Flankenerkennung
Init_Flanke
        BSF     status, rp0     ;wegen TRIS auf Bank 1 umschalten
        BSF     trisx,y         ;x = a oder b, y = 0 bis 7 (Impulseingang)
        BCF     status, rp0     ;zurück auf Bank 0

        MOVF    portx, W        ;gesamter Port lesen
        ANDLW   ????????b       ;Maskenaufbau, je nach Eingangspin
        MOVWF   alterPegel      ;Pegel merken
        …
        …


;Funktion_Flankenerkennung
warte_Flanke
        MOVF    portx, W        ;aktueller Pegel einlesen
        ANDLW   ????????b       ;unwichtige Bits löschen (Maske wie oben)
        XORWF   alterPegel, W   ;die beiden Pegel vergl., Ergebnis in W
        BTFSC   status, zflag   ;beide gleich, Ergebnis = 0, Zflag = 1
        GOTO    warte_Flanke    ;keine Flanke, weiter warten
        XORWF   alterPegel      ;neuen Pegel merken[13]
        BTFSS   alterPegel, y   ;prüfe, welche Flanke es ist. Dazu das ent-
        GOTO    hl_flanke       ;sprechende Bit in der Variable „alterPegel“
        GOTO    lh_Flanke       ;prüfen.
        …
        …
```

Das Symbol trisx steht entweder für trisa oder trisb. Das sind die beiden Datenrichtungsregister auf Bank 1 Adresse 5 bzw. 6. Der Wert für y ist das Dezimaläquivalent der Maske ????????b. z.B. Maske = 00001000 => y = 3.

Für die Maske ????????b muss das entsprechende Bitmuster eingesetzt werden, um nur das gültige Eingangssignal übrig zu lassen. Der Rest wird auf 0 gesetzt. Z.B. ist der Impulseingang auf RA, 3 muss die Maske 00001000b sein.

13 Diese Vorgehensweise ist anfangs etwas undurchsichtig. Aber: wenn eine Flanke gefunden wurde, ist durch die XOR-Verknüpfung der Wert im W-Register genau gleich wie die Maske. Erfolgt nun mit diesem Wert eine XOR-Verknüpfung mit der Variable „alterPegel“, wird an genau der passenden Stelle das Bit invertiert und damit auf den Zustand des neuen Pegels gesetzt.

9 Aufbau von Konstantentabellen

Die Harvard-Architektur verhindert per se den Zugriff auf den ROM-Speicher, wo üblicherweise Tabellen mit Konstanten abgelegt werden. Diese Tabelle dienen u.a. zur Initialisierung, Linearisierung von Kennlinien oder auch als Hilfstabellen für mathematische Berechnungen.

In unseren Fall wird eine Tabelle zur Umsetzung von Binärwerten in ein Bitmuster für eine 7-Segmentanzeige entwickelt. Es gibt keinen mathematischen Zusammenhang, um aus binären Werten das Muster zu berechnen. Somit benötigt man eine Zuweisungstabelle.

```
            MOVLW   5           ;diese Zahl soll auf dem Display
                                ;angezeigt werden
            CALL    convert
            MOVWF   port
            …
            …
            GOTO    label       ;verhindert ein Eintritt nach convert
                                ;ohne ein CALL
convert
            ANDLW   15          ;damit die Tabelle nicht übersprun
                                ;gen werden kann
            ADDWF   pcl         ;Inhalt von W auf PCL addieren
            RETLW   00111111B   ;Muster für die 0
            RETLW   00000110B   ;für 1
            RETLW   01011011B   ;für 2
            RETLW   01001111B   ;3
            RETLW   01100110B   ;4
            RETLW   01101101B   ;5
            RETLW   01111101B   ;6
            RETLW   00000111B   ;7
            RETLW   01111111B   ;8
            RETLW   01101111B   ;9
            RETLW   01110111B   ;A
            RETLW   01111100B   ;b
            RETLW   00001111B   ;C
            RETLW   01011110B   ;d
            RETLW   01111001B   ;E
            RETLW   01110001B   ;F
```

10 Längenmessung mittels Impulsrad

```
;Aufgabe Nr 8 aus µP-Aufgabensammlung
;(c) Stefan Lehmann

;Deklaration des Prozessortyps
                        device 16F84
;Deklaration diverser Symbolnamen
indirekt    EQU     0       ;Register indirect
pcl         EQU     2
status      EQU     3       ;Statusregister an Adr. 3
fsr         EQU     4       ;Zeigerregister für ind. Adressierung
porta       EQU     5       ;Adresse 5 auf Bank 0
portb       EQU     6
trisa       EQU     5       ;Adresse 5 auf Bank 1
trisb       EQU     6

cflag       EQU     0       ;Carryflag
zflag       EQU     2       ;Zeroflag
rp0         EQU     5       ;Bankumschaltbit
signal      EQU     0       ;Eingangssignal an RA,0

counter0    EQU     10H     ;LSB des 4-stelligen Zählers
counter1    EQU     11H
counter2    EQU     12H
counter3    EQU     13H

ii          EQU     0CH     ;Zählvariable
alterPegel  EQU     0DH     ;für Flankenerkennung
Digit       EQU     0EH     ;Hilfregister Digitanwahl

signalmaske EQU     00000001B
digitmaske  EQU     11110001B   ;je nach Ansteuerung der Digits

signaleingang EQU   0       ;Bit 0 von Port RA

cold:
            GOTO    start   ;Unterprogramme überspringen

;Unterprogramm FLANKE
;UP liest nur den Port ein, maskiert das richtige Bit und macht eine
```

```
;XOR-Verknüpfung mit dem alten Pegel
flanke
        MOVF    porta,W         ;Port RA nach W lesen
        ANDLW   signalmaske     ;nur RA0 ist gültig
        XORWF   alterPegel, W   ;mit vorherigen Pegel vergleichen
        RETURN                  ;bei ZFlag = 1, kein Pegelwechsel

;Unterprogramm INIT_FSR
;setzt die notwendigen Variablen auf deren Startwert. Das Setzen von DIGIT ist
;eigentlich nur bei der Anzeigeroutine notwendig, schadet aber beim anderen
;Aufruf nicht.
init_fsr
        CLRF    digit           ;nur für Anzeigeroutine wichtig
        MOVLW   4               ;Anzahl der vorhandenen Stellen
        MOVWF   ii              ;ii = Laufvariable
        MOVLW   counter0        ;Startadresse in W laden
        MOVWF   FSR             ;und ins Zeigerregister laden
        RETURN

;DISP gibt den Inhalt von COUNTER auf der 7-Segmentanzeige aus
disp
        CALL    init_fsr
        DECF    ii              ;niederwert. Stelle nicht anzeigen
        INCF    fsr             ;deshalb auch Zeiger um eins weiter
disp1
        MOVF    indirekt, W     ;Wert holen
        CALL    convert         ;in Bitmuster für 7-Seg. umsetzen
        MOVWF   portb           ;Segmentinf anlegen
        CALL    digitimpuls
        MOVLW   3               ;Zeiger in 3er Schritten erhöhen
        ADDWF   digit
        INCF    fsr
        DECFSZ  ii              ;hier max. 3 x
        GOTO    disp1
        RETURN                  ;danach fertig

;DIGITIMPULS erzeugt je nach anzuzeigender Stelle einen Impuls an RA1,
;RA2 oder RA3
digitimpuls
        MOVF    digit,W         ;Digitinfo nach W lesen
        ADDWF   pcl             ;an die entsprech. Position springen
```

```
        BCF     porta,1     ;impuls an RA,1 erzeugen
        BSF     porta,1     ;Impuls wieder wegnehmen
        RETURN
        BCF     porta,2     ;Impuls an RA,2
        BSF     porta,2
        RETURN
        BCF     porta,3     ;impuls an RA,3
        BSF     porta,3
        RETURN
```

;CONVERT wandelt den Wert in W in das passende Bitmuster einer 7-Segmentanzeige um

```
convert
        ANDLW   15          ;max. 16 Tabelleneinträge (ab 0)
        ADDWF   pcl
        RETLW   00111111B   ;Muster für die 0
        RETLW   00000110B   ;für 1
        RETLW   01011011B   ;für 2
        RETLW   01001111B   ;3
        RETLW   01100110B   ;4
        RETLW   01101101B   ;5
        RETLW   01111101B   ;6
        RETLW   00000111B   ;7
        RETLW   01111111B   ;8
        RETLW   01101111B   ;9
        RETLW   01110111B   ;A
        RETLW   01111100B   ;b
        RETLW   00001111B   ;C
        RETLW   01011110B   ;d
        RETLW   01111001B   ;E
        RETLW   01110001B   ;F


start
;Initialisierung der Ports und Variablen
init:                       ;man darf einer Adresse 2 Namen geben
        BSF     status,rp0  ;auf Bank 1 umschalten
        MOVLW   11110001B   ;RA1 bis RA3 Ausgänge für Digit
        MOVWF   trisa       ;RA0 bleibt Eingang
        CLRF    trisb       ;Port RB komplett auf Ausgang
        BCF     status,rp0  ;zurück auf Bank 0
```

```
                MOVLW   15              ;Digitselekt auf 1 setzen
                MOVWF   porta
reset
                CLRF    counter0        ;Zähler auf 0 stellen
                CLRF    counter1
                CLRF    counter2
                CLRF    counter3

;Hauptschleife. Hier wird zuerst der Zählerstand angezeigt und ;dann auf die Flanke gewartet
display
                CALL    disp
main
                CALL    flanke          ;bei ZFlag = 1 ist Flanke da
                BTFSC   status, zflag   ;
                GOTO    main            ;keine Flanke, weiter warten
flanke_da
                XORWF   alterPegel      ;neue Flanke merken
                BTFSS   alterPegel,signaleingang    ;richtige Flanke?
                GOTO    main            ;war falsche Flanke
flanke_ok
                CALL    init_fsr        ;FSR für ind. Adressierung setzen
                INCF    indirekt        ;niedrigste Stelle um 2 hochzählen
counting
                INCF    indirekt        ;
                MOVF    indirekt,W      ;hochgezählter Wert in W
                XORLW   10              ;ist der Wert 10 erreicht?
                BTFSS   status, zflag
                GOTO    display         ;zurück und Wert anzeigen
overflow
                CLRF    indirekt        ;aktuelle Stelle löschen
                INCF    fsr             ;Zeiger um einen Schritt weiter
                DECFSZ  ii              ;max. 4 Durchläufe
                GOTO    counting        ;nächste Stelle bearbeiten
                GOTO    display         ;zurück und Wert anzeigen

                END
```

11 Zustandsautomat

Um Programmabläufe grafisch darzustellen greift man oft auf Ablaufdiagramme in Form von Flussdiagrammen zurück. Eine andere Darstellungsart ist das Zustandsdiagramm. Diese Diagrammart ist vor allem dann übersichtlicher, wenn ein Problem in einzelne Blöcke (Zustände) eingeteilt werden kann. Das Programm verweilt dabei solange in einem solchen Zustand, bis bestimmte Ereignisse eintreten. Diese Ereignisse entscheiden in welchen Zustand das Programm wechselt. Ein sehr einfaches Beispiel einer solchen Zustandsmaschine ist eine Ampelanlage an einer Kreuzung.

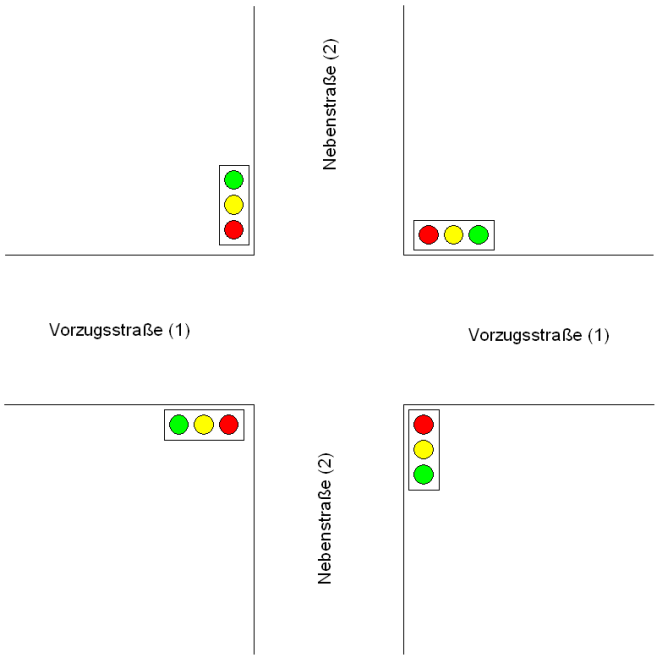


Abbildung 15: Zeitgesteuerte Ampelregelung

Zuerst formulieren wir das Problem in einfachen Sätzen:

Zustands-nummer	Übergangsbedingung und sichtbarer Zustand
S1	Anlage wird eingeschaltet
S2	Alle Ampeln werden auf rot gestellt.
S3	Nach 30 s werden die Ampeln der Vorzugstraße (1) auf rot-gelb geschaltet
S4	Nach 3 s werden die Ampeln von 1 auf grün gestellt.
S5	Nach 120 s werden die Ampeln von 1 auf gelb geschaltet
S6	Nach 3 s werden die Ampeln von 1 rot
S7	Nach 2 s werden die Ampeln der Nebenstraße (2) rot-gelb
S8	Nach 3 s werden die Ampeln von 2 auf grün geschaltet

S9 Nach 60 s werden die Ampeln von 1 gelb
S10 Nach 3 s werden die Ampeln von 1 auf rot gestellt.
S11 Nach 2 s werden die Ampeln von 2 rot-gelb
S12 Wiederholen ab Punkt S4 bis Anlage abgeschaltet werden soll.
S13 Alle Ampeln gehen auf rot, falls sie nicht schon dort sind.

Der ganze Ablauf kann mittels Zustandsdiagramm sehr gut visualisiert werden. Dazu gibt es die einzelnen Zustände die als Knoten bezeichnet werden und sogenannte Kannten, die die Übergangsbedingungen festlegen.

In Abbildung 16 ist das Zustandsdiagramm dieser zeitgesteuerten Ampelanlage dargestellt.

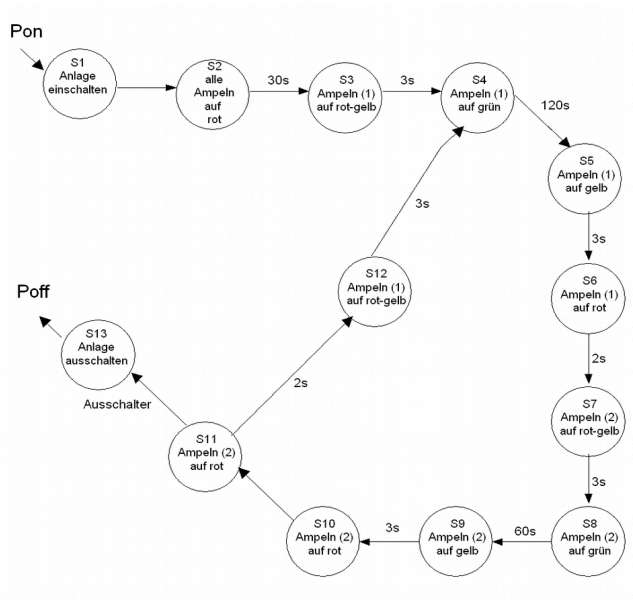


Abbildung 16: Zustandsdiagramm einer einfachen, zeitgesteuerten Ampel

Es gilt diese einzelnen Zustände in einem Assemblerprogramm zeitgesteuert ablaufen zu lassen. Doch zuerst muss festgelegt werden, welche Portpins des Mikrocontrollers die einzelnen Lampen in den Ampeln ansteuern. In dieser einfachen Ampelanlage sind die beiden Ampeln der Vorzugsstraße (1) synchron. Ebenso die beiden Ampeln der Nebenstraßen (2). Damit benötigen wir lediglich sechs Ausgänge, jeweils 3 für jede Straße. Da uns keine Vorgaben in dieser Richtung gemacht wurden, wählen wir einfach RB0 = rot für Vorzugsstraße, RB1 gelb und RB2 grün ebenfalls für diese Straße. Für die Nebenstraße legen wir RB3 = rot, RB4 = gelb und RB5 = grün fest. Damit kann die Schaltung aus Abbildung 1 für dieses Programm benutzt werden. Weiterhin legen wir fest, dass die inneren Zustände andere Nummern haben wie die äußeren Zustände

(Ampelzustände). Die internen Zustände sind bereits festgelegt, die äußeren werden wie folgt definiert:

Zustand	Ampel Straße 1	Ampel Straße 2	Port RB
0	rot	rot	0b00001001
1	rot-gelb	rot	0b00001011
2	grün	rot	0b00001100
3	gelb	rot	0b00001010
4	rot	rot	0b00001001
5	rot	rot-gelb	0b00011001
6	rot	grün	0b00100001
7	rot	gelb	0b00010001
8	rot	rot	0b00001001

In dieser Übersicht wird deutlich, dass die äußeren Zustände 0, 4 und 8 identisch sind. Ob man diese Zustände in einem zusammenfasst ist egal, da die eigentlichen Abläufe durch die inneren Zustände gesteuert werden.

Nachfolgend eine Möglichkeit diese Aufgabe mit Hilfe von Assembler zu lösen.

```
;Programm: Ampelsteuerung
                device      16F84
;Variablen und Konstantendefinitionen
trisa           EQU         5
porta           EQU         5
trisb           EQU         6
portb           EQU         6
status          EQU         3
rp0             EQU         5
zero            EQU         2
carry           EQU         0

;äußere Zustände = Ampelanzeige
azustand0       EQU         00001001b
azustand1       EQU         00001011b
azustand2       EQU         00001100b
```

```
azustand3       EQU         00001010b
azustand4       EQU         00001001b
azustand5       EQU         00011001b
azustand6       EQU         00100001b
azustand7       EQU         00010001b
azustand8       EQU         00001001b

;Festlegen des Programmstarts
                ORG         0

cold
state1
                BSF         status, rp0     ;Initalisierung
                CLRF        trisrb          ;Port B auf Ausgang
                BCF         status, rp0
state2
                MOVLW       azustand0       ;äußerer Zustand 0 an Port ausgeben
                MOVWF       portb
                CALL        wait30s         ;warte 30 Sekunden
state3
                MOVLW       azustand1       ;rot-gelb für (1) ausgeben
                MOVWF       portb
                CALL        wait3s          ;warte 3 Sekunden
state4
                MOVLW       azustand2       ;grün für (1) ausgeben
                MOVWF       portb
                CALL        wait120s        ;warte 2 Minuten
state5
                MOVLW       azustand3       ;gelb für (1) ausgeben
                MOVWF       portb
                CALL        wait3s
state6
                MOVLW       azustand4       ;rot für (1) ausgeben
                MOVWF       portb
                CALL        wait2s          ;gesamte Verkehr ruht 2 Sekunden
state7
                MOVLW       azustand5       ;rot-gelb für (2) ausgeben.
                MOVWF       portb
                CALL        wait3s
state8
                MOVLW       azustand6       ;grün für (2)
```

```
            MOVWF   portb
            CALL    wait60s     ;Hier nur 1 min Grünphase
state9
            MOVLW   azustand7   ;gelb an (2)
            MOVWF   portb
            CALL    wait3s
state10
            MOVWL   azustand8   ;rot an (2) ausgeben
            MOVWF   portb
state11
            BTFSS   einschalter ;Anlage noch eingeschaltet
            GOTO    state13
            CALL    wait2s
state12     MOVLW   azustand1   ;rot-gelb für (1)
            MOVWF   portb
            CALL    wait3s
            GOTO    state4
state13
            GOTO    state13     ;Ampeln bleiben rot bis Betriebs-
                                ;spannung weggenommen wird.
```

Dies war ein sehr einfaches Beispiel für das Programmieren einer Zustandsmaschine. In den Übungsaufgaben finden Sie eine etwas komplexere Zustandsmaschine.

12 Interrupts

In diesem Kapitel wird das Interrupt Konzept des PIC 16F84 beschrieben. Neben dieser Art der Interrupterkennung und Interruptauswertung gibt es noch viele andere Möglichkeiten. Diese werden im Themenblatt "Rechnerarchitektur" näher erläutert.
Interrupts sind asynchrone Unterbrechungen des laufenden Programms. Der Zeitpunkt der Unterbrechung ist somit nicht vorhersagbar. Dennoch kann es sein, dass zeitkritische Routinen unter keinen Umständen unterbrochen werden dürfen. Deshalb lassen sich i.d.R. Interrupts global sperren. Beim PIC wird dies durch das GIE-Bit im INTCON-Register gemacht. Weiterhin besteht die Möglichkeit die Interruptquellen individuell zu aktivieren und deaktivieren. Das sind die sogenannten Interrupt-Enable-Bits. Dazu zählen für den Timer das T0IE, das INTE für den Interrupteingang an RB0, das RBIE für die IO-Leitungen RB4 bis RB7 sowie das EEIE für das interne EEPROM. Diese Aufzählung gilt für den PIC 16F84. Andere PICs haben mal mehr mal weniger Interrupt auslösende Quellen und entsprechend die passenden Enablebits.

Sobald ein Ereignis eintritt, das einen Interrupt auslösen kann, wird das passende Interrupt-Flag gesetzt. Läuft beispielsweise der Timer über (255 auf 0), wird das T0IF-Bit gesetzt. Es wird auch dann gesetzt, wenn das T0IE-Bit nicht gesetzt ist. Somit lässt sich dieses Ereignis auch durch Polling erfassen.

Die zusammen gehörenden Bits von Interrupt-Flag und Interrupt Enable werden gemäß Abbildung 17 verknüpft. Man erkennt dabei, dass alle Interrupts die gleiche Priorität haben.

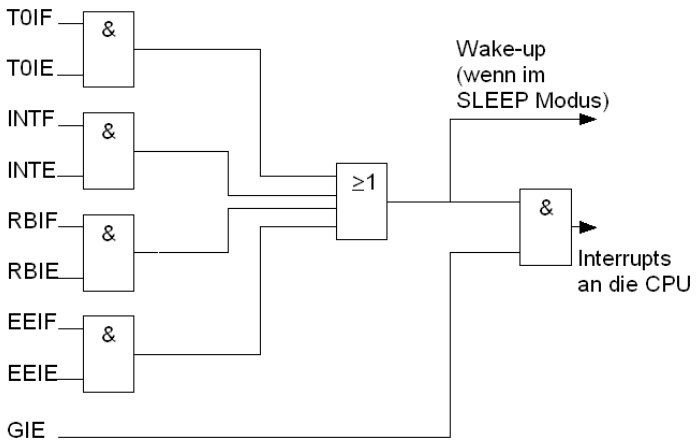


Abbildung 17: Die Interruptlogik des PIC 16F84

Sind die alle Bedingungen erfüllt (T0IE = 1, T0IF = 1, GIE = 1) verzweigt die CPU zur Adresse 0x0004 und löscht dabei das GIE-Bit. Zuvor wurde aber der aktuelle Stand des Programmzählers auf dem Stack abgelegt. An der Adresse 0x0004 hat der Assemblerprogrammierer die ISR (Interrupt Service Routine) platziert. Darin muss zuerst geprüft werden, wer den Interrupt ausgelöst hat. Das lässt sich an Hand der Interrupt-Flags erkennen. Die Reihenfolge der Abfrage bestimmt auch die Priorität der Interrupts. Nachdem dieser Interrupt bedient wurde, muss das entsprechende Interrupt-Flag gelöscht werden. Am Ende der ISR steht ein RETFIE, der den Rücksprung veranlasst und gleichzeitig das GIE wieder setzt. Sobald dieser Rücksprung vollzogen ist, wird der nächste Befehl ausgeführt. Falls zu diesem Zeitpunkt weitere Interrupts anstehen, wird nach der Befehlsabarbeitung erneut zur Adresse 0x0004 verzweigt, um auch diesen Interrupt zu bedienen.

13 Anhang

13.1 Beschreibung der Standardbefehle des PIC 16Fxx:

Es gibt drei Befehlsgruppen die sich im Aufbau ihres Befehlscodes von einander unterscheiden. Die erste Gruppe sind die Byte-orientierten Verknüpfungen mit den Fileregistern, die zweite Gruppe umfasst die Bit-orientierte Operationen der Fileregister. Die dritte Gruppe beinhaltet die Literals- (Konstanten) und Steuerfunktionen. Der Befehlscode der ersten Gruppe setzt sich zusammen aus dem Operationscode (OP-Code Bit11-6), dem Ziel-Bit (destination bit Bit 5) sowie der Adresse des Fileregisters (Bit4-0). In der zweiten Gruppe beinhaltet der Befehlscode ebenfalls den Op-Code (Bit11-8), das gewünschte Bit (Bit7-5) und die Adresse des Fileregisters (Bit4-0). Bei der dritten Gruppe umfasst der Befehlscode nur den OP-Code (Bit 11-8) sowie die 8-Bit-Konstante (Literal Bit7-0).

GRUPPE 1:

ADDWF
Addiert den Inhalt des W-Registers und den Inhalt des Fileregisters. Das Ergebnis wird entweder ins W- oder ins Fileregister geschrieben.

Flags: C, DC, Z
Codierung: 00 0111 dfff ffff

Bsp.:		vorher	nachher
ADDWF	F10	F10 = 55H; W = 12H	F10 = 67H; W = 12H
ADDWF	F10,W	F10 = 55H; W = 12H	F10 = 55H; W = 67H

ANDWF
UND-Verknüpfung des Inhalts des W- mit dem Inhalt des Fileregisters. Das Ergebnis wird entweder ins W- oder ins Fileregister geschrieben.

Flags: Z
Codierung: 00 0101 dfff ffff

Bsp.:	vorher	nachher
ANDWF F10	F10 = 55H; W = 16H	F10 = 14H; W = 16H
ANDWF F10,W	F10 = 55H; W = 16H	F10 = 55H; W = 14H

CLRF
Löscht den Inhalt des Fileregisters. Das Ziel ist hier immer das Fileregister selbst.

Flags: Z
Codierung: 00 0001 1fff ffff

Bsp.:	vorher	nachher
CLRF F10	F10 = 34H	F10 = 00H

CLRW
Löscht den Inhalt des W-Registers.

Flags: Z
Codierung: 00 0001 0xxx xxxx

Bsp.:	vorher	nachher
CLRW	W = 21H	W = 00H

COMF
Vom Inhalt des Fileregisters wird das 1er-Komplement gebildet. Das Ergebnis wird entweder ins W-Register oder ins Fileregister zurückgeschrieben.

Flags: Z
Codierung: 00 1001 dfff ffff

Bsp.:	vorher	nachher
COMF F10	F10 = 1011 0010	F10 = 0100 1101

DECF
Der Inhalt des Filregisters wird um 1 erniedrigt. Das Ergebnis wird entweder ins W-Register oder ins Fileregister geschrieben.
Flags: Z

Codierung: 00 0011 dfff ffff

Bsp.:	vorher	nachher
DECF F10	F10 = 03H	F10 = 02H
DECF F10,W	F10 = 03H; W = 7FH	F10 = 03H; W = 02H

DECFSZ

Der Inhalt des Fileregisters wird um 1 erniedrigt. Das Ergebnis wird entweder ins W-Register oder ins Fileregister geschrieben. Ist das Ergebnis 0, so wird der nachfolgende Befehl übersprungen.

Flags: keine
Codierung: 00 1011 dfff ffff

Bsp.:	vorher	nachher
DECFSZ F10	F10 = 03H	F10 = 02H
	nächster Befehl wird ausgeführt.	
bzw.		
DECFSZ F10,W	F10 = 03H	F10 = 03H; W = 02H
	nächster Befehl wird ausgeführt.	
DECFSZ F10	F10 = 01H	F10 = 00H
	nächster Befehl wird übersprungen.	
bzw.		
DECFSZ F10,W	F10 = 01H	F10 = 01H; W = 00
	nächster Befehl wird übersprungen.	

INCF

Der Inhalt des Fileregisters wird um 1 erhöht. Das Ergebnis wird entweder ins W-Register oder ins Fileregister geschrieben.

Flags: Z
Codierung: 00 1010 dfff ffff

Bsp.:	vorher	nachher
INCF F10	F10 = 03H	F10 = 04H
INCF F10,W	F10 = 03H	F10 = 03H; W = 04H

INCFSZ

Der Inhalt des Fileregisters wird um 1 erhöht. Das Ergebnis wird entweder ins

W-Register oder ins Fileregister geschrieben. Ist das Ergebnis 0, so wird der nächste Befehl übersprungen.

Flags: keine
Codierung: 00 1111 dfff ffff

Bsp.:	vorher	nachher
INCFSZ F10	F10 = 03H	F10 = 04H
	nächst. Befehl wird ausgeführt.	
bzw.		
INCFSZ F10,W	F10 = 03H	F10 = 03H; W = 04H
	nächst. Befehl wird ausgeführt.	
INCFSZ F10	F10 = FFH	F10 = 00H
	nächst. Befehl wird übersprungen.	
bzw.		
INCFSZ F10,W	F10 = FFH	F10 = 03H; W = 00H
	nächst. Befehl wird übersprungen.	

IORWF
Der Inhalt des Fileregisters wird mit dem Inhalt des W-Registers ODER verknüpft. Das Ergebnis kommt entweder ins W- oder ins Fileregister.

Flags: Z
Codierung: 00 0100 dfff ffff

Bsp.:	vorher	nachher
IORWF F10	F10 = C0H; W = 11H	F10 = D1H; W=11H
IORWF F10,W	F10 = C0H; W = 11H	F10 = C0H; W = D1H

MOVF
Der Inhalt des Filregisters wird entweder ins W-Register geladen oder wieder ins Fileregister zurückgeschrieben. Da dabei das Z-Flag entsprechend dem Registerinhalt gesetzt oder zurückgesetzt wird, kann man mittels diesem Befehl ein Register auf "0" abfragen, ohne dass andere Register verändert werden.

Flags: Z
Codierung: 00 1000 dfff ffff

Bsp.:	vorher	nachher

MOVF F10	F10 = 12H	F10 = 12H	Z=0
MOVF F10,W	F10 = 00H	F10 = 00H; W = 0	Z=1

MOVWF
Bringt den Inhalt des W-Registers in das Fileregister.

Flags: keine
Codierung: 00 0000 1fff ffff

Bsp.:	vorher	nachher
MOVWF F10	F10 = 23H; W = 6AH	F10 = 6AH; W = 6AH

NOP
Befehl hat keinerlei Einfluss auf die Register, abgesehen vom Programmzähler der um "1" erhöht wird.

Flags: keine
Codierung: 00 0000 0xx0 0000

RLF
Der Inhalt des Fileregisters wird um eine Stelle nach links geschoben. Dabei wird der Inhalt des Carry-Flags ins niederwertigste Bit (Bit 0), das höchstwertige Bit (Bit 7) ins Carry-Flag geschoben. Das Ergebnis kann entweder ins W-Register oder ins Fileregister geschrieben werden.

Flags: C
Codierung: 00 1101 dfff ffff

Bsp.:	vorher	nachher
RLF F10	F10 = 88H; C=0	F10 = 10H; C=1
RLF F10,W	F10 = 88H; C=1; W=0	F10 = 88H; C=1; W=11H

RRF
Der Inhalt des Fileregisters wird um eine Stelle nach rechts geschoben. Dabei wird der Inhalt des Carry-Flags in das höchstwertige Bit (Bit 7), das niederwertige Bit (Bit 0) ins Carry-Flag geschoben. Das Ergebnis kommt entweder ins W- oder ins Fileregister.

Flags: C
Codierung: 00 1100 dfff ffff

Bsp.:	vorher	nachher
RRF F10	F10 = 88H; C=0	F10=44H; C=0
RRF F10,W	F10 = 88H; C=1	F10=88H; C=0; W=C4H

SUBWF
Subtrahiert vom Inhalt des Fileregisters den Inhalt des W-Registers. Das Ergebnis wird entweder ins W- oder ins Fileregister geschrieben. Zu beachten ist hier die unterschiedliche Interpretation des Carry-Flags (siehe auch Kapitel 1.6.1 und 4.4)

Flags: C,DC,Z
Codierung: 00 0010 dfff ffff

Bsp.:	vorher	nachher
SUBWF F10	F10 = AAH; W = 55H	F10 = 55H W=55H C=1
SUBWF F10,W	F10 = 55H; W = AAH	F10 = 55H W=ABH C=0

SWAP
Vertauscht die oberen mit den unteren vier Bits des Fileregisterinhalts. Das Ergebnis kommt entweder ins W- oder ins Fileregister.

Flags: keine
Codierung: 00 1110 dfff ffff

Bsp.:	vorher	nachher
SWAPF F10	F10 = 74H	F10 = 47H
SWAPF F10,W	F10 = 74H	F10 = 74H; W = 47H

XORWF
Verknüpft den Inhalt des W-Registers mit dem des Fileregisters EXCLUSIV ODER. Das Ergebnis kann in das W- oder das Fileregister geschrieben werden.

Flags: Z
Codierung: 00 0110 dfff ffff

Bsp.:	vorher	nachher
XORWF F10	F10=96H; W=F0H	F10=66H Z=0
XORWF F10,W	F10=C5H; W=F0H	F10=C5H; W=35H Z=0

GRUPPE 2:

BCF

Löscht ein Bit in einem Fileregister.

Flags: keine, außer die Statusbits werden direkt gelöscht.
Codierung: 01 00bb bfff ffff

Bsp.:	vorher	nachher
BCF F10,3	F10 = FFH	F10 = F7H

BSF

Setzt ein Bit in einem Fileregister.

Flags: keine, außer die Statusbits werden direkt gesetzt.
Codierung: 01 01bb bfff ffff

Bsp.:	vorher	nachher
BSF F10,2	F10 = C0H	F10 = C4H

BTFSC

Überspringt den nächsten Befehl, wenn das Bit im Fileregister gelöscht ist. Da auch das Carry- und Zeroflag auf diese Weise abgefragt werden können, kann man in Verbindung mit einem GOTO oder CALL bedingte Verzweigungen programmieren.

Flags: keine
Codierung: 01 10bb bfff ffff

Bsp.:	vorher	nachher
BTFSC F10,2	F10 = 84H	F10 = 84H
	nächster Befehl wird ausgeführt.	
BTFSC F10,3	F10 = 84H	F10 = 84H

nächster Befehl wird übersprungen

BTFSS
Überspringe den nächsten Befehl, wenn das Bit im Fileregister gesetzt ist. Da auch das Carry- und Zeroflag auf diese Weise abgefragt werden können, kann man in Verbindung mit einem GOTO oder CALL bedingte Verzweigungen programmieren.

Flags: keine
Codierung: 01 11bb bfff ffff

Bsp.:	vorher	nachher
BTFSS F10,2	F10 = 84H	F10 = 84H
	nächster Befehl wird übersprungen	
BTFSS F10,3	F10 = 84H	F10 84H
	nächster Befehl wird ausgeführt	

GRUPPE 3:

ADDLW
Das Literal wird zum Inhalt des W-Registers addiert.

Flags: C, DC, Z
Codierung: 11 111x kkkk kkkk

Bsp.:	vorher	nachher
ADDLW 5H	W = 55H	W = 5AH; C=0, Z=0, DC =0

ANDLW
UND-Verknüpfung des W-Registers mit der Konstanten. Das Ergebnis steht im W-Register.

Flags: Z
Codierung: 11 111x kkkk kkkk

Bsp.:	vorher	nachher	
ANDLW 51H	W = 95H	W = 15H	Z = 0

CALL
Unterprogrammaufruf. Die Adresse des Unterprogramms ist nur 11 Bit breit. A11 und A12 werden aus dem PCLATH-Register (Bit 3 und 4) übernommen.

Flags: keine
Codierung: 10 0aaa aaaa aaaa

Bsp.:	vorher	nachher
CALL 3FH	PC = 0011H	PC = 003FH
		TOS[14] = 0011H

CLRWDT
Löscht den Watchdog-Timer. Ist der Watchdog aktiv (entsp. EPROM-Sicherung ein), so muss in regelmäßigen Abständen dieser Befehl ausgeführt werden, sonst spricht der Watchdog an und löst einen RESET aus.

Flags: TO, PD
Codierung: 00 0000 0110 0100

Bsp.:	vorher	nachher
CLRWDT		

GOTO
Sprungbefehl zu einer bestimmten Programmadresse. Die Adressangabe des Befehls umfasst 11 Bits und erlaubt so Sprünge an jede beliebige Programmstelle in der aktuellen 2k großen Programmspeicherseite. Ist eine Verzweigung in eine andere Seite notwendig, so müssen zuerst die passenden Bits im PCLATH-Register gesetzt bzw. zurückgesetzt werden, da diese vom Goto-Befehl in die entsprechenden Adressbits des Programmzählers kopiert werden.

Flags: keine
Codierung: 10 1aaa aaaa aaaa

Bsp.:	vorher	nachher
GOTO 14CH	PC = 102H	PC = 14CH

14 Top of Stack

IORLW
Der Inhalt des W-Registers wird mit der Konstanten ODER verknüpft. Das Ziel ist immer das W-Register.

Flags: Z
Codierung: 11 1000 kkkk kkkk

Bsp.:	vorher	nachher	
IORLW 0FH	W = AAH	W = 0AH	Z = 0

MOVLW
Die Konstante k wird ins W-Register geladen.

Flags: keine
Codierung: 11 00xx kkkk kkkk

Bsp.:	vorher	nachher
MOVLW 45H	W = 00H	W = 45H

RETFIE
Es erfolgt ein Rücksprung aus einem Interrupt-Unterprogramm. Dabei wird das GIE-Bit gesetzt und so die nächsten Interrupts zu erlauben.

Flags: keine
Codierung: 00 0000 0000 1001

Bsp.:	vorher	nachher
RETFIE	GIE = 0	GIE = 1; PC = TOS

RETLW
Beendet das Unterprogramm und lädt die Konstante ins W-Register. Mit diesem Befehl können auch Tabellenzugriffe im Programmspeicher realisiert werden.

Flags: keine
Codierung: 11 01xx kkkk kkkk

Bsp.:	vorher	nachher

RETLW 0	W = 89H	W = 00H

Beispiel für Tabellenzugriffe:

```
            MOVLW   5H          ;diese Zahl soll quadriert werden
            CALL    QUADR
            ....
            ....
            GOTO    .....
QUADR       ADDWF   PC          ;Der Inhalt von W wird zum Pro
                                ;grammzähler addiert
            RETLW   0           ;Quadrat von 0
            RETLW   1           ;      "      " 1
            RETLW   4           ;      "      " 2
            RETLW   9           ;      "      " 3
            RETLW   10H         ;      "      " 4
            RETLW   1AH         ;      "      " 5
      usw...
```

RETURN

Es erfolgt ein Rücksprung aus einem Interrupt-Unterprogramm.

Flags:	keine
Codierung:	00 0000 0000 1000

Bsp.:	vorher	nachher
RETURN		PC = TOS

SLEEP

Versetzt den Prozessor in einen stromsparenden Halt-Zustand. Alle Funktionen ruhen, die Ports behalten die zuletzt gültigen Werte und der Oszillator wird abgeschaltet. Beenden kann diesem Zustand nur ein RESET, ob durch ein "L" am MCLR-Eingang, durch Ansprechen des Watchdogs oder durch das Auftreten eines Interrupts.

Flags:	TO,PD
Codierung:	00 0000 0110 0011

Bsp.:	vorher	nachher
SLEEP		

SUBLW
Der Inhalt des W-Registers wird vom Literal k abgezogen.

Flags: C, DC, Z
Codierung: 11 110x kkkk kkkk

Bsp.:	vorher	nachher
SUBLW 5H	W = 55H	W = 50H; C=1, Z=0, DC =1

XORLW
Der Inhalt des W-Registers wird mit der Konstanten EXKLUSIV ODER verknüpft.

Flags: Z
Codierung: 11 1010 kkkk kkkk

Bsp.:	vorher	nachher	
XORLW 0FH	W = 9BH	W = 0BH	Z = 0

13.2 Binärcode Interna

Wie ist der Befehlscode aufgebaut? Jeder Befehl ist eine ganz bestimmte Bitkombination. Dabei haben die Bits bestimmte und eindeutige Bedeutungen. Damit der Befehlsdekoder einfach gestaltet werden kann, werden die Befehle in Gruppen zusammen gefasst und der Code entsprechend aufgebaut. So entscheidet beispielsweise ein Bit, das immer an der selben Stelle im Befehl steht, ob das Ergebnis der Operation ins W- oder in ein F-Register geschrieben wird.

Schaut man sich die

OP-Code	Befehle
0000 0000 0xxx	NOP, OPTION, SLEEP, CLRWDT, TRIS
0000 01df ffff	CLRF, CLRW
0000 1xdf ffff	SUBWF, DECF
0001 xxdf ffff	IORWF, ANDWF, XORWF, ADDWF
0010 xxdf ffff	MOVF, COMF, INCF, DECFSZ
0011 xxdf ffff	RLF, RRF, SWAP, INCFSZ

01xx bbbf ffff BCF, BSF, BTFSC, BTFSS

100x aaaa aaaa CALL, RETLW
101a aaaa aaaa GOTO
11xx kkkk kkkk MOVLW, IORLW, ANDLW, XORLW

13.3 Begriffserklärungen

BCD
Binary Coded Digit. Bei einer BCD-Zahl sind in einem Byte 2 Zahlen codiert. Jedes Halbbyte zählt dann von 0 bis 9

Harvard-Architektur
Rechner der zwei getrennte Adressräume (Programm- und Datenbereich) besitzt.

immediate
unmittelbar

Literal
konstanter Wert

Prefix
vorangestellte Kennung (0x12 ≙ 12 hexadezimal)

Suffix
angehängte Kennung (12H ≙ 12 hexadezimal)

Tetrade
wird auch als Nibble bezeichnet. Es werden dabei immer 4 Bits zusammengefasst. Somit besteht ein Byte aus 2 Tetraden bzw. Nibbles.

TOS
Top of Stack ist der oberste Eintrag auf dem Stack. Ein Stack ist als FILO-Speicher (First in, Last out) organisiert.

Von-Neumann-Rechner
Rechner bei dem Programm- und Datenspeicher in einem gemeinsamen Adressraum liegen.