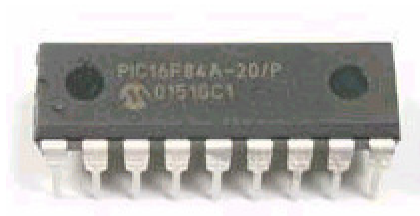
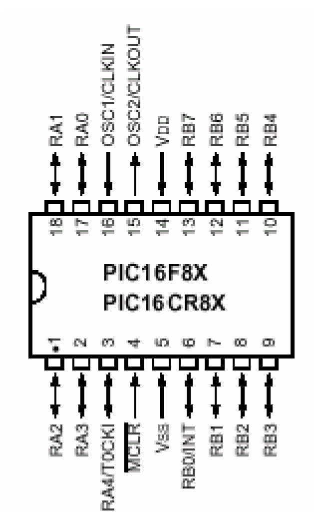
Der PIC - Microcontroller

Informations - Handbuch





(Deutsch)

PIC Prozessoren

Was ist denn überhaupt ein PIC?

Ein PIC ist ein Vertreter der Ein-Chip-Mikrocontroller. Während sich ein kompletter "Rechner" mit "normalen" Mikroprozessoren immer aus mehreren Chips (IC) zusammensetzt, hat man bei den Ein-Chip-Mikrocontrollern alles in einen Chip integriert. Darunter leidet natürlich die Gesamtleistung des Systems, aber die Ein-Chip-Mikrocontroller sollen keine Wetterprognosen machen, und sie sind auch nicht für den Aufbau von Personalcomputern gedacht. Ein-Chip-Mikrocontroller werden benutzt, um kleine Steuerungsprobleme zu lösen, die mit analogen oder diskreten digitalen Schaltungen einen hohen Aufwand erfordern würden.

Um die herkömmliche Konkurrenz (analog oder diskret aufgebaut) aus dem Rennen zu werfen, müssen sie klein, billig und einfach zu handhaben sein. Damit sind sie auch eine interessante Alternative für den Elektronikbastler. Der muß weniger Aufwand in den Entwurf und den Bau von Stromkreisen stecken. Der eingesparte Grips wird in die Entwicklung eines Steuerprogramms gesteckt.

Warum sollte man PIC-Processoren benutzen?

Es gibt keinen speziellen Grund. Sicherlich eignen sich auch andere Ein-Chip-Mikroprozessorfamilien zum Lösen vieler Probleme der Hobbyelektroniker. Jeder sollte sich aber für eine Prozessorfamilie entscheiden und mit dieser dann alle anstehenden Probleme lösen. Es ist uneffektiv, von Projekt zu Projekt auf ein anderes Pferd zu setzen, nur weil der andere Chip in diesem Fall einen kleinen Vorteil bietet. Der Arbeitsaufwand, um sich in ein neues Prozessordesign und eine andere Entwicklungsumgebung einzuarbeiten, steht normalerweise in keinem Verhältnis zum zu erwartenden Nutzen.
Ich kam durch Zufall zu PICs, nutze ihre Stärken und lebe mit ihren Schwächen, ohne mich nun noch groß um andere Prozessoren zu kümmern.

Was sind besondere Stärken der PIC-Prozessoren?

Fangen wir mal beim lieben Geld an. Was braucht man zum Einstieg in der Welt der Ein-Chip-Microcontroller?

1 - Eine Entwicklungsumgebung zum Erstellen der Programme (Assembler oder C-Compiler).
2 - Ein Programmiergerät um die geschriebenen Programme in den Prozessor zu übertragen.
3 - Die Prozessoren selbst.
4 - Unterstützung im WWW.

Beim PIC-Hersteller Microchip gibt es die Entwicklungsumgebung kostenlos zum Download .
Bauanleitungen für preiswerte Programmiergeräte (5,- ..20,- €) sind z.B. für den PIC16F84 inklusive Software im WWW kostenlos verfügbar. Z.B auf meiner Homepage .
Die für Hobbybastler geeignetsten PIC-Prozessoren kosten pro Stück unter 10,-€.
Wer mal in eine WWW-Suchmaschine PIC eintippt, wird schnell fündig. PICs sind sehr verbreitet, was sie nicht zuletzt ihrem Einsatz auf Sat-Decoder-Piraten-Karten zu verdanken haben.
Man investiert also weit unter 50,-€ für den Einstieg, und auch die Folgekosten halten sich in Grenzen.

Was sind Schwächen der PIC-Prozessoren?

Aus meiner Sicht ist die Interruptverwaltung des PIC eine Schwäche. Mit Z-80-Prozessoren aufgewachsen, bin ich es gewohnt, jeder Interruptquelle einen eigenen Interruptvektor zuzuordnen. Die PIC-Prozessoren unterstützen zwar viele Interruptquellen, haben aber nur einen Interruptvector. Das ist umständlich, wenn man in einem Programm mit mehreren Interruptquellen arbeitet. Die einzige Interruptbehandlungsroutine muß erst in Statusregistern nachschauen, welche Quelle den Interrupt ausgelöst hat, um dann die richtigen Programmschritte auszuführen.

Umfangreiche Berechnungen sind auch nicht gerade eine Lieblingsdisziplin der kleinen PICs. Sie rechnen nur mit 8-Bit-Zahlen, und beherrschen nur Addition und Sybtraktion. Es lassen sich zwar mit Softwareroutinen auch große Fließkommaberechnungen anstellen, aber das ist umständlich, langsam und macht keinen Spaß. Außerdem verbraucht z.B. eine 64-Bit Rechenroutine viele der knappen Register.

Welcher PIC-Prozessor ist der richtige?

Für den Hobbyelektroniker kommen eigentlich nur vier PIC-Typen in Frage:
- PIC 16F84
- PIC 16F876
- PIC16F628
- PIC12F6XX

Und deren nächste Verwandte: 16F83, 16F873, 16F874, 16F877, 16F627 sowie die PIC16F7x-Typen
All diese Typen sind flashbar, d.h. man kann sie nahezu beliebig oft umprogrammieren. Andere Typen (ohne 'F' in der Typenbezeichnung) sind nicht zu empfehlen, da sie entweder nur einmal programmiert werden können (OTP), oder umständlich wie EPROMS mit UV-Licht gelöscht werden. Die UV-löschbaren PICs sind obendrein recht teuer.

Eine Auflistung interessanter Typen befindet sich hier .

Ich beschäftige mich ausschließlich mit flashbaren PICs, also mit PICs der Serien PIC16F.../12F...

16F84 (A)

Der 16F84 ist mit 18-pins recht klein. Er verfügt über
- 13 digitale Ein-/Ausgänge
- einen 8-Bit Timer
- Interrups

Dieser Chip ist mit einer Taktfrequenz von bis zu 10 MHz verfügbar (als modernisierter PIC16F84A neuerdings bis 20 MHz).

Der 16F84 eignet sich für alle einfachen Anwendungen, bei denen keine analogen Werte gemessen werden müssen. Er kann Impulse ausmessen und erzeugen, kleine Tastaturen abfragen, LCD-Displays ansteuern, und auch wenn er keine Hardware für die serielle Kommunikation besitzt, so kann man doch eine serielle Schnittstelle durch ein wenig Software realisieren. (siehe hier) Im Web finden sich viele Anwendungen für den 16F84, der aber nun im PIC16F628 seinen Meister gefunden hat.

16F87x

Seit 2000 gibt es „größere“ flashbare PICs, die 16F87x-Familie. Mein Standardtyp ist der 16F876. Er verfügt über:
- max. 21 digitale Ein-/Ausgänge

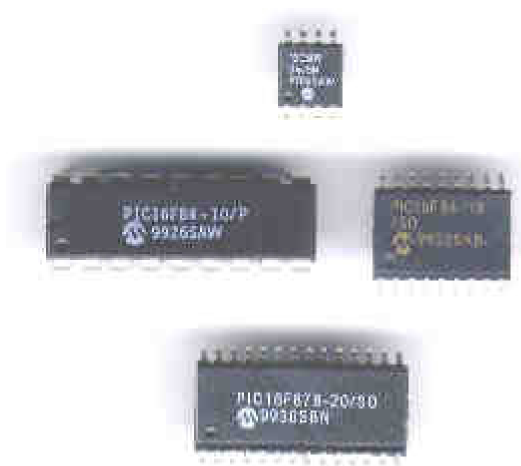
- max. 5 analoge Eingänge mit einem 10-Bit-Analog-Digital-Wandler
- drei Timer (8 und 16-Bit)
- serielle Schnittstellen (RS-232 , I²C ...)
- Capture /Compare /PWM -Hardware
- Interrups

Dieser Chip ist mit einer Taktfrequenz von bis zu 20 MHz verfügbar.

Der 16F876 ist mit 20 Pins schon etwas größer als der 16F84. Sein großer Vorteil sind die analogen Eingänge zum Messen von Spannungen. Ansonsten hat er von allem etwas mehr als sein kleiner Bruder; mehr Ein/Ausgänge, mehr Timer mehr Speicher und mehr Geschwindigkeit.
Die integrierte serielle Schnittstelle vereinfacht die Realisierung einer RS-232-Verbindung z.B. zu einem PC (verlangt aber einen zusätzlichen invertierenden Treiberbaustein).

Der PIC16F877 Hat einen zusätzlichen 8-Bit-Port, und ist mit seinen 40 Pins eigentlich schon etwas groß und recht teuer.

Eine etwas abgespeckte Version der PIC16F87x-Familie ist die PIC16F7x-Familie. Ihr fehlt ein EEPROM , mit dem man Daten dauerhaft speichern kann.



Für Leute mit kleinen Fingern oder mit wenig Platz auf der Platine gibt es die PICs natürlich auch als SMD. Auf nebenstehendem Bild sind von oben nach unten zu sehen:
- PIC12C509-04/SM (nicht flash-bar)
- PIC16F84-10/P und .../SO
- PIC16F876-20/SO

Der direkte Vergleich zwischen der DIL und der SMD-Bauform des 16F84 zeigt, daß der SMD-Typ nur die halbe Breite hat. Noch eindrucksvoller würde ein Vergleich zwischen den unterschiedlichen Gehäusebauformen des PIC16F876 ausfallen.

Die SMD-Typen haben allerdings zwei Nachteile
- das Leiterplattenlayout ist schwieriger
- das Brennen (laden des Programms in den PIC) geht nur über die ICSP - Schnittstelle, es gibt keine Brenner mit SMD-Fassungen

16F62x

Seit 2001 gibt es die 16F62x-Familie (16F627 und 16F628). Diese PICs könnten die 16F84 verdrängen, denn sie haben die gleiche Größe (18-Pin-Gehäuse) bei einer deutlich verbesserten Ausstattung:
- max. 16 digitale Ein-/Ausgänge
- max. 2 analoge Komparatoreingänge
- drei Timer (8 und 16-Bit)
- serielle Schnittstellen (RS-232)
- Capture /Compare /PWM -Hardware
- 3,5 mal so viel Arbeitsspeicher wie der 16F84
- doppelt so viel EEPROM wie der 16F84
- als 16F628 doppelt soviel Programmspeicher wie der 16F84

Dieser Chip ist mit einer Taktfrequenz von bis zu 20 MHz verfügbar.
Vor allem die 16-Bit Timer und die serielle Schnittstelle wurden am 16F84 immer vermißt. Bei Reichelt und Farnel kosten diese PICs sogar weniger als 5 €. Was will man mehr?

12F6xx

Seit 2002 gibt es die 12F6xx-Familie (12F629 und 12F675). Diese PICs sind spottbillig (3 ... 4 €) und

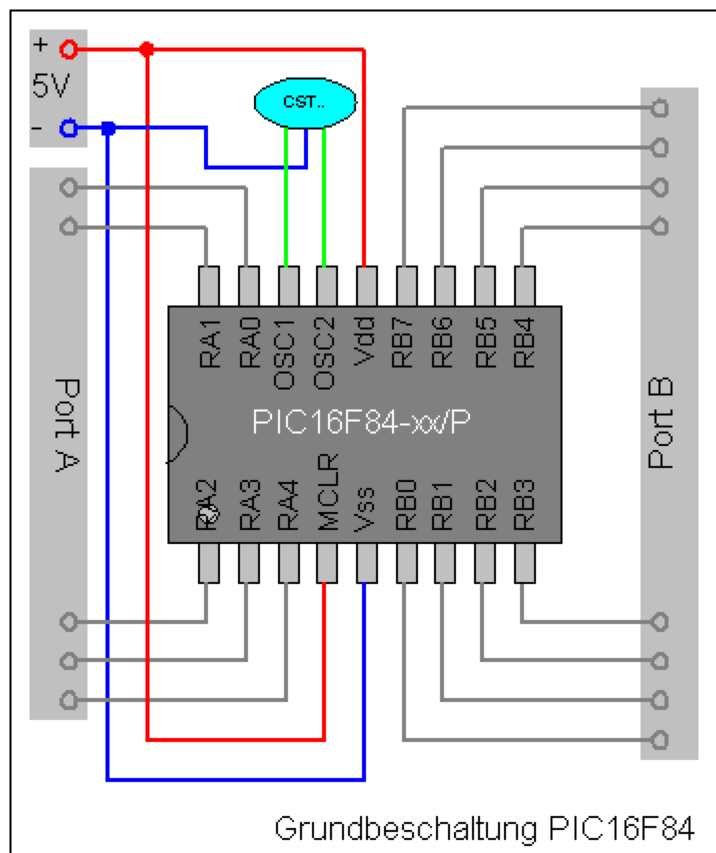
haben nur 8 Pins. Sie eignen sich gut für kleine Steueraufgaben, der 12F675 besitzt sogar einen ADC:
- max. 6 digitale Ein-/Ausgänge
- 1 Komparator + 1 ADC oder 2 Komparatoren
- 2 Time

Mit dem nächsten Softwareupdate (V2.3 Ende September 2002) werden meine Brenner warscheinlich die 12F6xx-Typen unterstützen

Es gibt auch Prozessoren der Serien PIC18Fxxx die aber für Bastelerzwecke meist überdimensioniert sind.

Was braucht ein PIC zum arbeiten an Peripherie?

Eigentlich nur eine Betriebsspannung von ca. 5 V und einen Takt.



Grundbeschaltung PIC16F84

Die Betriebsspannung ist je nach Typ in Toleranzen variierbar. Der Bereich von 4 V bis 5,5 V ist unkritisch. Bei einigen Typen kann man die Spannung bis auf 2 V absenken, ohne den sicheren Arbeitsbereich zu verlassen.
Die Stromaufnahme des Prozessorkerns ist vom Takt abhängig, überschreitet aber beim maximalen Takt nie 15 mA. Dazu kommt noch der Strom, den der PIC aufwenden muß, um über digitale Ausgänge andere Bauteile anzusteuern. Wenn man z.B. mit dem PIC 8-Leuchtdioden ansteuert, werden dafür natürlich auch einige 10 mA zusätzlich benötigt. Die Ausgangspins eines PIC können übrigens bis zu 20 mA liefern, weswegen zusätzliche Treiber meist nicht nötig sind.

Nebenstehendes Bild zeigt eine typische Grundbeschaltung für einen PIC16F84. Die mit 'Port A' und 'Port B' beschrifteten Boxen sind lediglich Steckverbinder zu den zu steuernden oder zu überwachenden Schaltungen. Der 5V-Steckverbinder liefert die Betriebsspannung, un der blaue 'CST...' ist ein Keramischer Schwinger, der den Takt für den PIC bereitstellt. Eigentlich ganz einfach.

Als Taktquellen haben sich keramische Schwinger mit integrierten Kondensatoren (Keramikresonator) bestens bewährt. Die kosten nur 1,- €, sind kleiner als ein Quarz und werden ohne zusätzliche Bauelemente einfach an den PIC angeschlossen. Ihr Frequenzfehler liegt bei 0,5 %. Wer es penibel genau mag nimmt einen Quarzgenerator oder einen Quarz. Der Quarz benötigt aber noch zusätzlich 2 Kondensatoren und evtl. einen Widerstand.
Keramikresonatoren habe ich bisher nur bis 12 MHz gefunden. Wer die 20 MHz der schnellen PICs nutzen will, ist auf Quarze und Quarzgeneratoren angewiesen.

Das nebenstehende Bild zeigt von links nach rechts:

- Quarzgenerator
- Quarz (HC18-Gehäuse)
- Keramikresonator (blaues CST-Gehäuse) (Schwinger mit integrierten Kondensatoren)

Wer gern mit SMD-Bauteilen arbeitet, findet auch Quarze und Schwinger im Kleinformat. Im Nebenstehenden Bild sieht man zwischen den beiden Normalbauformen (links: Keramikresonator, rechts: Quarz) von oben nach unten:

- SMD-Quarz
- Keramikresonator
- Keramikresonator

Wo gibt es PIC-Prozessoren und Keramikresonatoren ?

PICs sind recht populär. So kann man 16F84 z.B. bei Conrad bestellen. Dort gibt es auch Keramikschwinger bis 10 MHz.
Mit dem Hauptkatalog 2003 hat Conrad sein Angebot an Flash-PICs gründlich ausgebaut. Auch die 12F...-Typen sind nun hier gelistet.

Andere Anbieter sind da schon weiter, z.B. Farnell, Memec oder Elpro .

Auch bei Keramikresonatoren über 10 MHz sollte man mal bei Reichelt oder RS und nicht bei Conrad nachschauen.
Wer nur 4MHz oder 8MHz Keramikresonatoren braucht, findet die im Conrad-Katalog bei den Quarzoszillatoren und bei den Keramikresonatoren unter der Bezeichnung "Schwinger mit integrierten Kondensatoren".
(Conrad Bestellnummern: 4MHz: 50 31 69-88; 8MHz: 50 39 67-88)

Bei Pollin gibt es für ein Butterbrot 20MHz-Quarzoszillatoren. Die sind besonders deswegen interessant, da 20 MHz-Keramikresonatoren kaum erhältlich sind.

Sind PIC-Prozessoren empfindliche Bauelemente?

Ich habe bisher nur zwei Mal einen PIC zerstört, und ich kenne jemanden der an einem PIC eine Ausgangsleitung getötet hat. Kurz und gut, die Dinger sind sehr robust.
Trotzdem sollte man natürlich die üblichen Regeln beachten. Denn garantiert wird nur die Funktion im zuglassenen Arbeitsbereich.

Da man einen PIC, bis alles funktioniert, häufig aus seiner Fassung nimmt und wieder hineinsteckt, besteht die größte Gefahr für den PIC darin, daß man seine *Pins verbiegt* und danach wieder ausrichten muß. Das geht höchsten 3 mal gut, danach *bricht der Pin einfach ab* . Deshalb benutze ich stets ein IC-Ausziehwerkzeug (Bild rechts), um ihn aus der Fassung zu entfernen.

Ebenfalls durch das häufige Ein- und Ausstecken und die damit verbundene Routine besteht ständig die Gefahr den Schaltkreis *versehentlich falsch herum in den Sockel zu stecken*. Das quittiert der PIC dann mit starker Erwärmung die innerhalb einiger Sekunden zum Tode führt. (Auf diese Art und Weise habe ich zwei PICs verloren.) Ich kennzeichne deshalb auf den Leiterplatten die Position von Pin 1 zusätzlich mit einer auffälligen, schwarzen Markierung. Auch markiere ich gern Pin1 der PICs mit einem Tropfen Tip-Ex, auch wenn das unprofessionell aussieht.

Manuel Krüger gab mir den Tip, in der Entwicklungsphase, wenn ein PIC oft gebrannt wird, den PIC nicht einzeln zu benutzen, sondern ihn in einen Präzisionssockel zu stecken, und dieses Doppelpack dann wechselseitig in den Brenner bzw. in die Testschaltung zu stecken. Falls dann was verbiegt, ist nur der Sockel Schrott.

Was leistet die kostenlose Entwicklungsumgebung MPLAB?

Mit ihr kann man Programme im Assemblercode entwickeln. Da der Befehlssatz des PIC nicht mal 40 Befehle umfaßt, ist das auch gar nicht so schwer. Ich arbeite nur mit Assembler.
Das Resultat ist dann ein *.HEX-File, das den fertigen Code enthält. Besitzer des von Microchip vertriebenen Programmiergeräts PICstart können dieses aus der Entwicklungsumgebung heraus direkt ansteuern und das *.HEX-File in den PIC übertragen.

Gibt es Programmiersprachen für PICs?

Ja gibt es, und die lassen sich meist als Plug-In in MPLAB integrieren.

Z.B. gibt es C-Compiler (z.B. HiSoft), die kosten aber etwas Geld.

Eine interessante Alternative für Bastler ohne Programmiererfahrung scheint Parsic zu sein. Das ist ein grafisches Tool, mit dem man sich sein PIC-Programm sozusagen zusammenklicken kann. Für einfache Programme kann das interessant sein. Das kostenlose Demo kann leider nicht speichern, und die Vollversion kostet leider 100 €. Zum ausprobieren zu teuer, für professionellen Einsatz zu beschränkt, es sei denn man importiert Programmteile aus anderen Entwicklungsumgebungen. Der Ansatz ist aber lobenswert.

Unter dem Namen JAL gibt es eine Art 'Hochsprache' für PICs die etwas an Pascal erinnert.

Ich programmiere PICs ausschließlich in Assembler. Stellt mir bitte keine Fragen bei Problemen mit höheren Programmiersprachen.

Wie kommen denn nun die fertigen Programme in den PIC?

Das machen Programmiergeräte, sogenannte "Brenner" . Das sind kleine Kästen, die an die serielle Schnittstelle oder den Druckerport des PC angeschlossen werden. Der Brenner hat eine Schaltkreisfassung, in die der PIC gesteckt wird. Diese Geräte gibt es als Fertiggeräte, man kann sie aber auch selbst bauen.

Wer 150,- € zu viel hat kann den PICstart von Microchip kaufen. Dieses industrielle Gerät wird auch direkt von der Entwicklungsumgebung MPLAB unterstützt. Der PICStart kann alle am Markt verfügbaren PIC-Typen "brennen", nimmt sich dafür aber sehr viel Zeit.

Für ca. 125,- € gibt es einen 16F84-Programmiergerät bei ELV. Als Bausatz kostet es nur 80,- €. Ich hatte so ein Gerät noch nicht in den Händen, aber die Beschränkung auf den kleinen PIC ist ein großer Nachteil.

Wer lieber nur 15,- € ausgeben möchte, für den gibt es Bauanleitungen für einfache „Brenner“ die am Parallelport angeschlossen werden . Das teuerste an diesen Geräten ist die Schaltkreisfassung (man sollte sich einen 0-Kraft-Sockel gönnen), und das Steckernetzteil. Mit solch einem Brenner habe ich lange Zeit gearbeitet und hatte keine Probleme. Viele ältere dieser Brenner waren noch nicht für den 16F876 geeignet und alte „Brennersoftware lief nur im DOS-Fenster. Ich habe jetzt ein **Windowsprogramm**geschrieben, das diese Brenner komfortabel ansteuert, und alle PIC16F8xx-Typen brennen kann. Passende Brenner die sowohl den 16F84 wie auch alle 16F87x und 16F7x unterstützt sind der " **Brenner3**" und der "Brenner5" .

Wer nur mal schnell einen PIC brennen will, ohne großen Aufwand zu treiben kann zum Brenner0 oder dem "Quick and Diry-Brenner" greifen. Sowas ist in einer halben Stunde zusammengelötet und sollte mit der Tait-DPS-Software (DOS) oder meinem Windowsprogramm zu betreiben sein. Wer mehr als einen PIC brennen will, sollte aber von dieser Primitivlösung lieber Abstand nehmen.

Wer es abenteuerlich mag, für den gibt es Bauanleitungen und Software für minimalistische 16F84-Brenner, die am seriellen Port angeschlossen werden und kein Netzteil benötigen. Auch die beschränken sich oft auf die 16F84-Familie.

Ich habe mir einen etwas aufwendigeren Brenner2 gebaut, der selbst von einem PIC gesteuert wird. Er hängt am seriellen PC-Port und brennt 16F84, 16F62x wie auch 16F876 und 16F873. Die

Steuersoftware ist in Delphi geschrieben und läuft unter Windows95/98/NT. Das Ganze ist schon seit Jahren im Betrieb und wurde auch erfolgreich nachgebaut.

Momentan favorisiere ich den Brenner5 und den **Brenner3**. Das sind preiswerter Druckerportbrenner und brennen 16F84, 16F627, 16F628 sowie alle 16F87x und 16F7x. Dabei gehen sie deutlich schneller zu Werke als der Brenner2. Die Steuersoftware ist in Delphi geschrieben und läuft unter Windows95/98/NT/XP. Die Nutzung unter Win-XP oder an Notebooks ist Glückssache.

Welche Probleme kann man mit einem PIC lösen?

Große und auch ganz kleine.
Viele Bastler glauben, einen Prozessor zu benutzen, wäre wie mit Kanonen auf Spatzen zuschießen. Dem ist aber nicht ganz so.

Beispiel für ein kleines Problem
Auf einer Modelleisenbahnanlage sollen die 8 Laternen einer Straßenbeleuchtung vorbildgerecht eingeschaltet werden. Dazu müssen sie beim Einschalten unabhängig voneinander unregelmäßig flackern, bis sie schließlich alle gleichmäßig und hell leuchten.
In konventioneller Bauweise ist der Aufwand erheblich. Benutzt man aber einen PIC, so verbindet man einfach 8-Pins mit den Lampen (falls die Leistung der Lampen zu hoch ist benötigt man noch jeweils einen Treibertransistor), schließt einen Keramikschwinger an und lädt den PIC mit einem Programm, das das gewünschte Einschaltverhalten simuliert.

Beispiel für ein mittleres Problem
Ein selbstgebautes elektronisches Gerät soll über eine serielle Schnittstelle vom PC aus gesteuert werden.
Man baut in das Gerät einen PIC ein, der die serielle Schnittstelle enthält und mit seinen digitalen/analogen Leitungen mit den Steuerfunktionen des Geräts verbunden ist. Der PIC kommuniziert mit dem PC und steuert dementsprechend das Gerät.

Beispiel für ein großes Problem
Ein ferngesteuertes Modellflugzeug soll mit einem Autopiloten versehen werden, der zusätzlich zur Funkfernsteuerung das Flugverhalten nach Sensordaten stabilisiert.
Ein PIC wird in die Leitungen zwischen Fernsteuerempfänger und Rudermaschinen des Flugzeugs geschaltet. Über weitere Anschlüsse wird er mit Sensoren für den Luftdruck, die Geschwindigkeit und die Beschleunigungskräfte versehen. Er mißt die Länge der vom Boden gesendeten Fernsteuerimpulse und verändert sie nach ausgefeilten Regelgesetzen in Abhängigkeit von den Sensordaten. Die veränderten Impulse steuern die Rudermaschinen des Flugzeugs und damit das Flugzeug selbst.

All diese Beispiele sind keine Ausgeburt der Phantasie, sondern real existierende Anwendungen für PICs.

Allgemeines

Das ist eine Anleitung für die ersten Schritte in die Welt der PIC-Prozessoren:

Softwarebeschaffung

Nötige Downloads von der Microchip-Homepage:

- die Entwicklungsumgebung MPLAB (Version 5.3 ca. 11 MByte)
- die Dokumentation des Prozessors als PDF-Datei

Zum Lesen der PDF-Dateien wird der Acrobat-Reader benötigt.

Für den Bau eines Brenners (z.B. Brenner3 oder Brenner5) benötigt man

- das Layout
- die Stückliste und
- die Software des Brenners.

Hardwarebeschaffung

Bein einem günstigen Versender sollte man sich für den Anfang 2-3 PIC-Prozessoren (Typenübersicht) sowie einige Keramikschwinger bestellen. Dabei kann man ruhig zu den schnellsten Typen greifen, die kosten kaum mehr als die langsameren:

- PIC16F628-20/P & 10 MHz-Keramikschwinger/20 MHz-Quarzgenerator oder
- PIC16F84-10/P & 10 MHz-Keramikschwinger oder
- PIC16F84A-20/P oder PIC16F876-20/SP & schneller Keramikschwinger oder 20 MHz-Quarzgenerator

Da 16 bzw. 20 MHz-Keramikschwinger nicht an jeder Straßenecke zu finden sind sollte man für den 16F876, 16F628 und 16F84A auch Quarzgeneratoren in Betracht ziehen.

Alle andere Bauteile sollten in jedem gut sortierten Elektronik-Bastelladen erhältlich sein.

Bau des Brenners und der Testplatine

Nun ist es Zeit, den Brenner und eine Testplatine zu bauen. Die Platinen sollte man fotochemisch herstellen. Ein geeignetes Steckernetzteil, das ca. 15V (für den Brenner) bzw. 9V (für die Testplatine) liefert und zur Spannungsbuchse des Brenners und der Testplatine paßt, wird zusätzlich benötigt.
Wird als Nullkraftsockel ein Textool-Typ verwendet, sollte man zum Einlöten die Fassung in die Position "offen" gestellt haben, ansonsten werden einzelne Kontakte schräg fixiert, und öffnen sich später nicht mehr richtig.
Gummifüße unter den Ecken der Platinen vermeiden Kurzschlüsse durch auf dem Tisch herumliegende Drahtschnipsel und verringern das Herumrutschen.

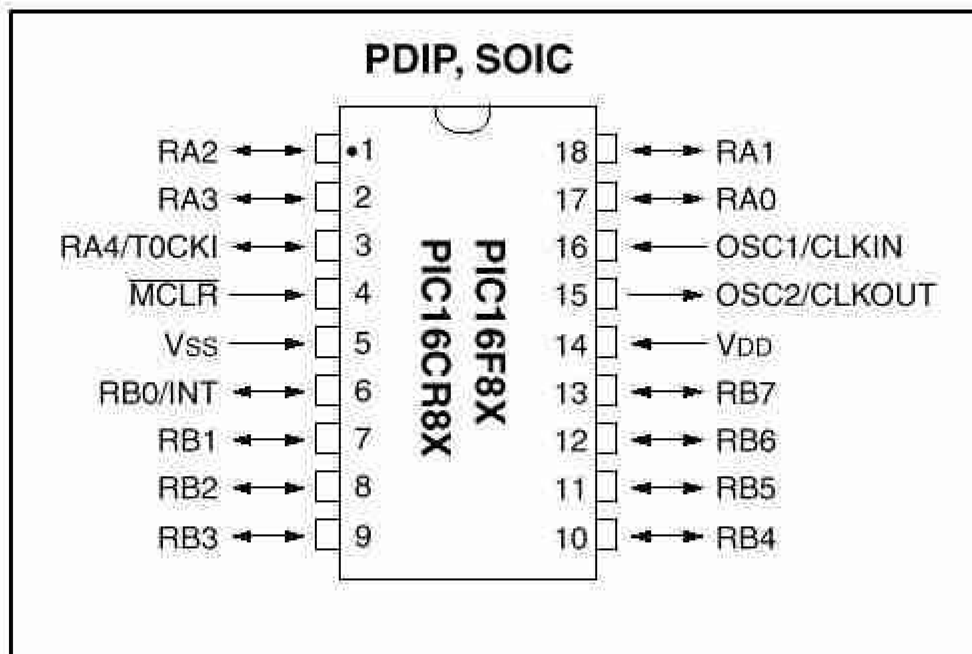
Erste Programme

Mit dem PIC macht man sich an besten durch kleine Programmierbeispiele bekannt. Solche Fingerübungen finden sich z.B. hier.

Welche Anschlüsse hat ein PIC?

Ein PIC hat stets folgende Anschlüsse:

VDD	Betriebsspannungsanschluß (+ 5 V)
VSS	Masseanschluß
MCLR	Resetanschluß (Wird dieser Pin kurz mit Masse verbunden wird der PIC in den Ausgangszustand versetzt)
OSC1/2	Anschlüsse für den Taktgenerator
Rxy	die eigentlichen Ports des PIC



Die Anzahl der Port-Pins (Rxy) ist je nach PIC-Typ verschieden. Der 16F84 besitzt z.B. 13 Portpins, der 16F876 dagegen 21.

Was sind Ports eines PIC?

Bezeichnung
Sämtliche Ein- und Ausgaben eines PIC erfolgen über seine parallelen Ports. In der Grundidee handelt es sich bei jedem Port um jeweils max. 8 Anschlüsse des PIC, die als TTL-Eingänge oder Ausgänge funktionieren.
Für den PIC benimmt sich ein Port wie ein internes Register, von dem er lesen, und in das er schreiben kann. Die Ports sind mit Buchstaben bezeichnet. Der 16F84 besitzt die Ports A und B, der 16F876 zusätzlich noch das Port C. Einige größere PICs haben auch noch ein Port D. Die zugehörigen internen Register heißen PORTA, PORTB bzw. PORTC.

Jedem Bit (0 .. 7) des Registers ist jeweils genau ein Port-Pin zugeordnet. Bei der Bezeichnung des Pins wird "Port" mit dem Buchstaben "R" ersetzt. Dadurch ergeben sich die Pin-Bezeichnungen: RB3 ist das Pin, das mit dem Bit 3 des Registers PORTB verbunden ist.

Eine Besonderheit des Ports A ist, das ihm die höchstwertigen 3 Pins fehlen. Es gibt also nur RA0 bis RA4. Die Bits 5 bis 7 des Registers PORTA sind funktionslos.

In/Out
Natürlich kann so ein Pin nicht gleichzeitig Ein- und Ausgang sein. Man muß zwischen den beiden Funktionen umschalten. In der Grundeinstellung sind die Ports Eingänge. Will man sie zu Ausgängen machen, so muß man im Programm das interne Register TRISA (für PORTA), TRISB (PORTB) bzw. TRISC (PORTC) verändern. Jedes Bit der TRIS-Register ist einem Port-Pin zugeordnet. Ist das Bit auf "1" gesetzt, funktioniert das zugehörige Pin als Eingang. Wird das Bit auf "0" gesetzt, ist das zugehörige Pin bis auf Weiteres ein Ausgang, und hat den Pegel, den das zugehörige Bit im Port-Register (z.B. PORTA) hat.
Beim Reset und beim Einschalten des PIC werden alle Bits der die TRIS-Register auf "1" gesetzt. Damit ergibt sich "Eingang" als Grundfunktion der Pins.
Eine genauere Erläuterung befindet sich hier.

Spezial
Einige Port-Pins besitzen zusätzliche Funktionen (z.B. spezielle Input/Output-Hardware). So kann RA4 als Zählereingang für den Timer des PIC benutzt werden, Pins des Port C können als serielle Schnittstellen dienen (16F87x) und die Pins des Port A können als analoge Spannungsmeßeingänge dienen (16F87x). Einzelne Bits spezieller interne Register schalten die Pins auf die Spezialfunktion um, womit sie dann nicht mehr Bestandteil des normalen Ports sind.

Das Port A des 16F876 ist nach dem Einschalten des PIC oder nach Reset als analoger Eingang konfiguriert. Will man diese Pins als digitale Bits des Port A nutzen, muß man ihre Funktion zuerst umschalten.

Programmspeicherorganisation

Alle PICs der PIC16F...-Serien haben einen internen Flash-Speicher, in dem das Programm abgelegt wird. Zum Einschreiben dieses Programmes benötigt man ein Programmiergerät. Der Inhalt des Programmspeichers bleibt nach dem Ausschalten des PIC selbstverständlich erhalten, und kann jederzeit mit dem Programmiergerät gelöscht, oder verändert oder gänzlich neu beschrieben werden.

Der Programmspeicher eines PIC ist ein Speicherblock, der an der Adresse 0000h beginnt. Die Programmspeichergröße ist typabhängig:

Typ	Programmspeichergröße [Worte]	1. Adresse	letzte Adresse
PIC16F84	1k	0000h	03FFh
PIC16F873	4k	0000h	0FFFh
PIC16F876	8k	0000h	1FFFh

Jede Zelle des Programmspeichers ist 14 Bit groß.

Zwei Zellen des Programmspeichers haben eine feste Funktion:

Adresse	Name	Funktion
0000h	Startvektor	an dieser Stelle beginnt der PIC die Programmabarbeitung nach dem Einschalten oder dem Reset
0004h	Interuptvektor	hierhin springt der PIC bei der Auslösung eines Interupt

Zum Programmspeicher gehören im weitesten Sinne noch der Programcounter (PC), der die Adresse des nächsten auszuführenden Befehls enthält, und ein Stack (8 Worte groß) in dem bei Unterprogrammaufrufen der PC des rufenden Programmteils gespeichert wird.

Datenspeicherorganisation

Der Datenspeicher ist der RAM des PIC, in dem das Programm zur Laufzeit Werte speichern kann. Mit dem Ausschalten der Betriebsspannung geht der Inhalt des Datenspeichers verloren. Sollen Werte beim Ausschalten des PIC erhalten bleiben, müssen sie im Daten-EEPROM des PIC gespeichert werden.

Der Datenspeicher des PIC ist vergleichsweise kompliziert aufgebaut. Er besteht aus mehreren Bänken, und teilt sich den Adreßbereich mit den Steuerregistern des PIC. Daten- und Programmspeicher sind völlig unabhängig, so daß in beiden Speichern die gleichen Adressen benutzt werden können.

Prinzipiell beginnt auch der Datenspeicher an der Adresse 0000h, aber die ersten Adressen sind mit Steuerregistern des PIC belegt. Die erste freie Adresse ist beim PIC16F84 die Speicherzelle 000Ch und beim PIC16F87x die Speicherzelle 0020h. Jede Speicherzelle ist 1 Byte groß.

Der Datenspeicher der PICs ist in bis zu 4 Bänke aufgeteilt. Jede Bank enhält andere Steuerregister auf den unteren Adressen. Die Bankumschaltung erfolgt durch das Setzen von 2 Bit (RP0, RP1) im Steuerregister STATUS. Dieses Steuerregister ist sinnvollerweise in allen Bänken an der selben Stelle zu finden.

Im 16F84 und im 16F876 sind einige freie Speicherzellen unabhängig von der Bankeinstellung immer zugänglich. Auf andere freie Speicherbereiche kann man nur zugreifen, wenn man in die entsprechende Bank umgeschaltet hat.

Typ	Gesamtgröße	immer zugänglich	nur in Bank 0	nur in Bank 1	nur in Bank 2	nur in Bank 3
PIC16F84	68 Byte	68 Byte (0Ch - 4Fh, 8Ch - CFh)	-	-	nicht vorhanden	nicht vorhanden
PIC16F873	192 Byte	-	96 Byte (20h - 7Fh)	96 Byte (A0h - FFh, 20h - 7Fh)	greift auf Bank 0 zu	greift auf Bank 1 zu
PIC16F876	368 Byte	16 Byte (70h - 7Fh, F0h - FFh, 170h - 17Fh, 1F0 - 1FFh)	80 Byte (20h - 6Fh)	80 Byte (A0h - EFh, 20h - 6Fh)	16 Byte (110h - 11Fh) und 80 Byte (120h - 16Fh, 20h - 6Fh)	16 Byte (190h - 19Fh) und 80 Byte (1A0h - 1EFh, 20h - 6Fh)

Daten-EEPROM

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

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

Typ	**Anzahl der EEPROM-Zellen (Bytes)**
PIC16F84	64
PIC16F873	128
PIC16F876	256

Der Hersteller gibt für die PICs eine Lebensdauer von 100 000 EEPROM-Schreibzyklen an. Die Typen PIC16F7x enthalten keinen Daten-EEPROM.

PIC-Flash-Controller-Übersicht

Typenübersicht

Noch vor wenigen Jahren gab es von Microchip nur einen PIC-Microcontroller mit Flash-Programmspeicher. Inzwischen wächst das Angebot aber erfreulich schnell. Um in der nun den Überblick zu behalten, sind die wichtigsten Eigenschaften der unterschiedlichen Typen nachfolgend aufgelistet.
Neben den hier aufgelisteten PICs gibt es auch noch PIC18Fxxx, mit denen ich mich aber nicht beschäftige.

Typ	Programm-Speicher [14-Bit-Worte]	RAM [byte]	EEPROM [byte]	Pins	I/O-Pins	ADC	USART	I2C	CCP	Timer	ca. Preis bei Conrad
12F629	1k	64	128	8	6	1xComp.	-	-	-	2	€ 3,-
12F675	1k	64	128	8	6	1 + 1xComp.	-	-	-	2	€ 4,-
16F84(A)	1k	68	64	18	13	-	-	-	-	1	€ 11,-
16F627	1k	224	128	18	16	2xComp	1	-	1	3	€ 6,-
16F628	2k	224	128	18	16	2xComp	1	-	1	3	€ 6,50
16F870	2k	128	64	28	22	5	1	-	1	3	€ 8,-
16F871	2k	128	64	40	33	8	1	-	1	3	
16F872	2k	128	64	28	22	5	1xSSP	1	1	3	€ 8,-
16F873	4k	192	128	28	22	5	1	1	2	3	€ 13,50
16F874	4k	192	128	40	33	8	1	1	2	3	€ 16,-
16F876	8k	368	256	28	22	5	1	1	2	3	€ 14,50
16F877	8k	368	256	40	33	8	1	1	2	3	€ 19,-
16F72	2k	128	-	28	22	5*	1xSSP	1	1	3	
16F73	4k	192	-	28	22	5*	1	1	2	3	€ 9,50
16F74	4k	192	-	40	33	8*	1	1	2	3	€ 12,-
16F76	8k	368	-	28	22	5*	1	1	2	3	€ 13,50
16F77	8k	368	-	40	33	8*	1	1	2	3	€ 16,-
16F87xA	wie 16F87x	wie 16F87x	wie 16F87x	wie 16F87x	wie 16F87x	wie 16F87x + 2xComp.	wie 16F87x	wie 16F87x	wie 16F87x	wie 16F87x	

Beschriftung der PICs

Auf jedem PIC befindet sich die Typenangabe mit angehangen Zusätzen:

PIC16Fxxx - AAB /CC

dabei bedeuten:

AA	die maximale Taktfrequenz in MHz:	**04**	4 MHz
		10	10 MHz
		20	20 MHz
B	Temperaturbereich	**leer**	0℃ ... 70℃
		I	-40℃ ... +85℃
CC	Gehäuse	**P**	PDIP (DIL)
		SP	PDIP (DIL)
		SO	SOIC
		SS	SSOP
		PT	TQFP
		PQ	MQFP
		ML	MLF
		L	PLCC

Legende:

Programm-Speicher:
Ist als Zahl der 14-Bit-Speicherzellen dargestellt, wobei k für den Faktor 1024 steht. So bedeutet also "2k", daß ein PIC 2048 Speicherplätze für jeweils einen 14-Bit-Befehl hat.

Pins:
Die Anzahl der Pins am Gehäuse. Darann erkennt man die physische Größe des Schaltkreises.

I/O-Pins:
Anzahl der Pins, die als Bestandteil eines Ports (PORTA..PORTC) als Ein- oder Ausgangsleitung dienen können.

ADC:
Eingänge für den Analog/Digital-Wandler. Ein PIC hat immer nur einen ADC, es können aber mehrere Eingänge wechselweise zum ADC zugeschaltet werden. Die ADCs haben normalerweise eine Auflösung von 10 Bit. Die mit "*" gekennzeichneten ADCs sind nur 8 Bit breit. Die 16F62x besitzen nur Comparatoren.

USART:
Serielle Schnittstelle, die sich z.B. als RS232 verwenden läßt.

I2C:
I2C-Bus Anschluß. Dieser Anschluß ist Bestandteile der USART, aber nicht alle USARTs unterstützen auch I2C. Deshalb ist diese Eigenschaft hier extra aufgelistet.

CCP:
Anzahl der Capture/Compare/PWM-Module. Mit diesen Modulen lassen sich Impulse Messen und Erzeugen. Außerdem können pulsweitenmodulierte Signale ausgegeben werden.

Timer:
Anzahl der Timer. Ist nur 1 Timer vorhanden, handelt es sich um einen 8-Bit-Timer. Bei 3 Timern sind 2 davon 16-Bit breit. Der Watchdogtimer ist in dieser Zahl noch nicht enthalten.

Preis
Das sind aufgerundete Einzelpreise für die teuersten Varianten der Schaltkreise. Bei der Wahl einfacherer Gehäuse, niedrigerer Taktfrequenz oder gößerer Stückzahlen (ab 3 Stk.) sind die Preise z.T. deutlich niedriger

Taktversorgung des PIC

Allgemeines zur Taktversorgung

Der PIC braucht wie jeder Mikrocontroller einen Arbeitstakt. Den kann er

- von einer externen Quelle eingespeist bekommen,
- mit einem eigenen Oszillator und ein paar externen Bausteinen selbst erzeugen, oder
- mit dem eigenen Oszillator und einem internen Widerstand selbst erzeugen

Dabei bietet nicht jeder PIC-Typ alle Möglichkeiten an. Auf welche Art und Weise der PIC nun zu seinem Takt kommt, wird beim Brennen des PIC festgelegt . Man unterscheidet folgende Betriebsarten:

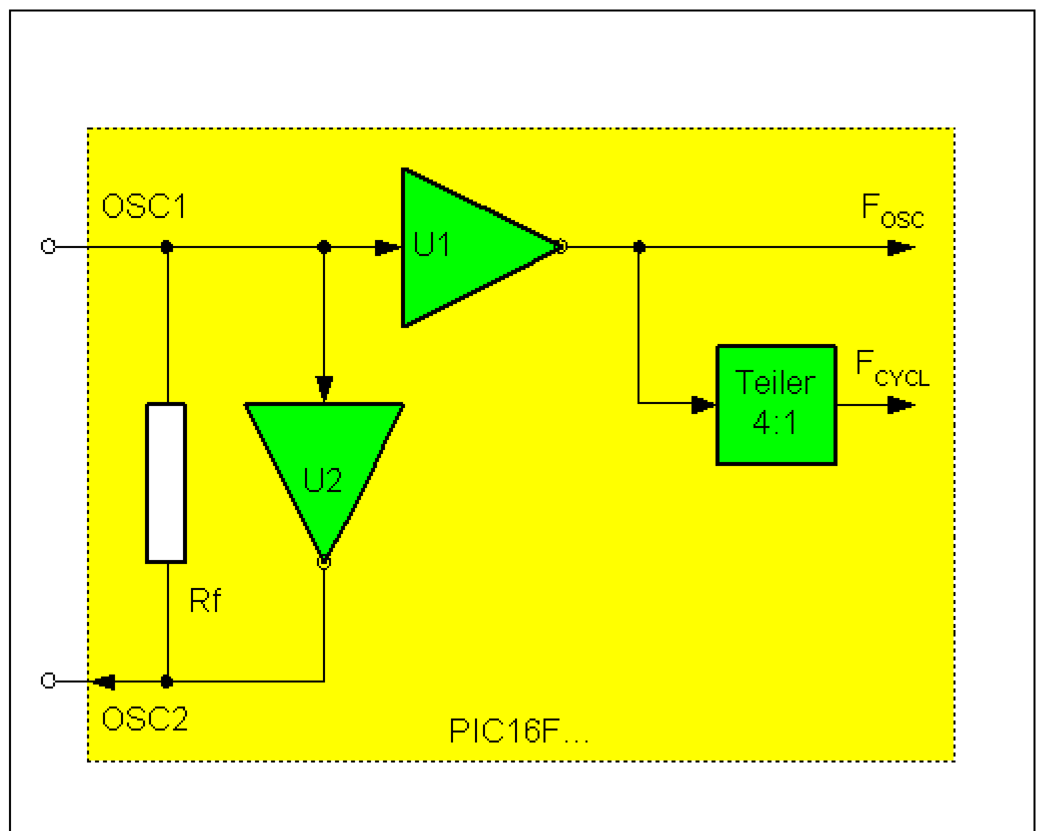
Abkürzung	Bezeichnung	Frequenzbereich	Beschreibung
LP	low power crystal/resonator	min ... 200 kHz	Quarz, Resonator oder externer Takt minimaler Stromverbrauch
XT	crystal/resonator	100kHz ... 4 MHz	Quarz, Resonator oder externer Tak
HS	high speed crystal/resonator	4 MHz ... max.	Quarz, Resonator oder externer Tak höchste Geschwindigkeit
RC	Widerstand-Kondensator	30 kHz ... 4 MHz	externer Widerstand und Kondensator
ER	externer Widerstand	10 kHz ... 8 MHz	nur 1 externer Widerstand (garantiert bis 4 MHz)
INTRC	intern	4 MHz	ohne externe Bauelemente (ca. 8% Toleranz)
EC	external clock	min ... max.	externer Takt, kein Taktausgang

wobei aber nicht jeder PIC alle Betriebsarten unterstützt:

PIC-Typ	LP	XT	HS	RC	ER	INTRC	EC
16F84(A)	X	X	X	X			
16F87x	X	X	X	X			
16F7x	X	X	X	X			
16F62x	X	X	X		X	X	X

Der interne Oszillator (LP, XT, HS)

Nebenstehendes Bild zeigt den prinzipiellen Aufbau der Oszillatorschaltung eines PIC für die Modes LP, XT und HS.



Ein am OSC1-Pin eingespeistes Signal wird durch U1 verstärkt (auf TTL-Pegel) und an die internen Schaltungen des PIC weitergeleitet. Das Signal steht inveriert und verstärkt auch am Pin OSC2 wieder zur Verfügung.

Steht ein externer Takt zur Verfügung (z.B. von einem Quarzoszillator) wird dieser einfach an OSC1 eingespeist, und OSC2 bleibt unbenutzt.

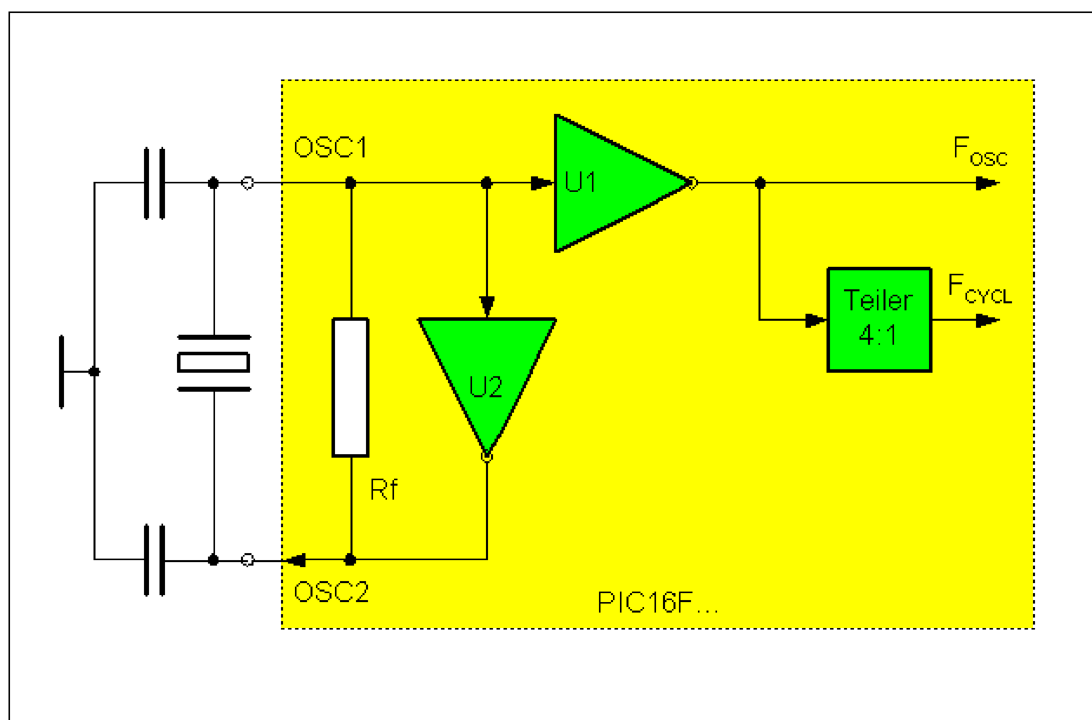
Mit der Rückkopplung über U2 läßt sich aber auch ein Oszillator aufbauen. Dazu muß ein schwingfähiges Bauelement (Quarz oder Keramikschwinger) zwischen die Pins OSC1 und OSC2 angeschlossen werden. Durch die Rückkopplung über U2 wird dieser Resonanzschwinger zum Schwingen angeregt. Die

Schwingung wird über U1 verstärkt, und an die internen PIC-Schaltungen weitergeleitet.

Der interne Widerstand Rf dient der Gegenkopplung des Inverters, um ihn je nach Taktfrequenz im stabilen Arbeitsbereich zu halten. Rf kann nicht für 10kHz und 20MHz den gleichen Wert haben. Vielmehr gibt es drei Festwerte, mit denen man sich für einen Frequenzbereich entscheidet: **LP**, **XT** und **HS**.

Quarz und Resonator (LP, XT, HS)

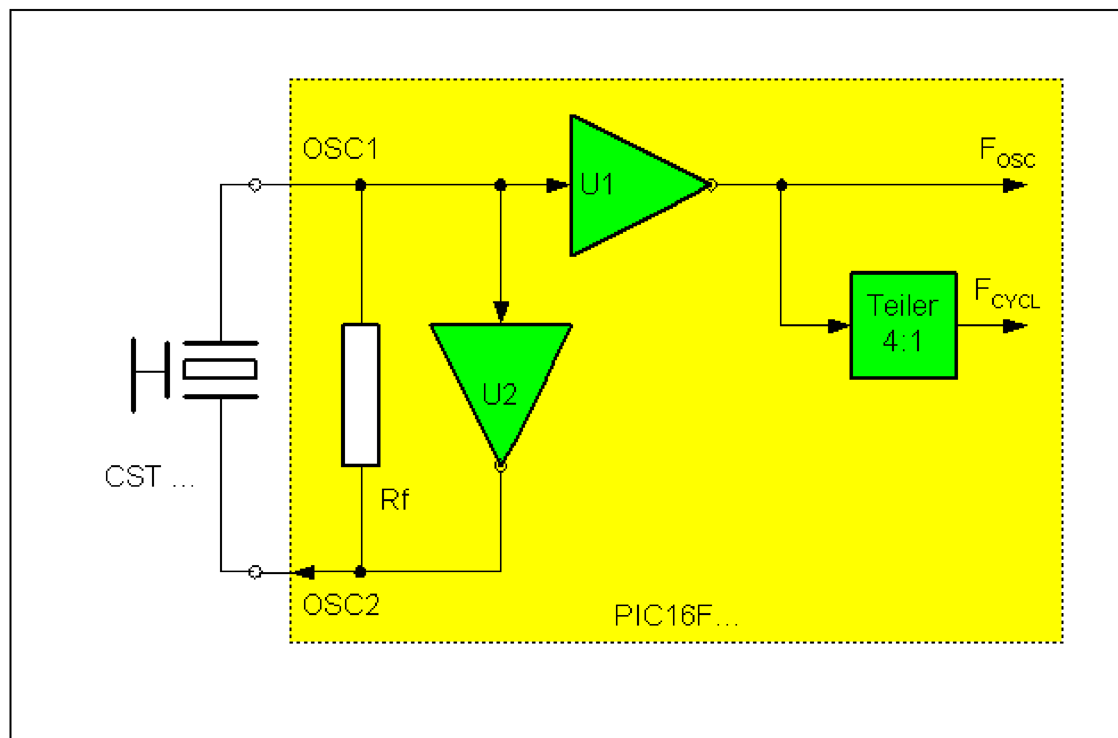
Als frequenzbestimmende Rückkoppelelemente lassen sich Quarze oder Keramik-Resonatoren einsetzen. Der Quarz oder Resonator muß an beiden Enden mit einem Kondensator an Masse angeschlossen sein.



Der Wert der Kondensatoren sollte ab Taktfrequenzen von 2MHz etwa 15pF ... 33pF betragen. Bei Frequenzen im Kiloherz-Bereich kann man sie bis zu 100 pF erhöhen.

Falls (wofür auch immer) Der Takt noch für andere Bausteine benutzt werden soll (Z.B. für einen 2. PIC), dann kann er am Pin OSC2 abgegriffen werden. An OSC1 dürfen dagegen keine weitern Verbraucher angeschlossen werden.

Besonders interessant sind Keramikresonatoren mit integrierten Kondensatoren. Die besitzen 3 Anschlußpins, der mittlere wird an Masse angeschlossen, wären die beiden äußeren jeweils an OSC1 und OSC2 angeschlossen werden. Die beiden separaten Kondensatoren entfallen.



Leider sind diese praktischen Bauteile für Frequenzen oberhalb 12 MHz kaum aufzutreiben.

Je nach Anbieter heißen diese 3-beinigen Schwinger *Schwinger mit integrierten Kondensatoren* oder *Keramikresonator*. Meist haben sie ein blaues

Gehäuse und eine Typenbezeichnung der Form 'CST x.00' wobei x für die Frequenz in MHz steht.

Der invertierende Verstärker U2 muß in einem extrem weiten Frequenzbereich das Schwingen erlauben. Um einen 20 MHz Resonator zu erregen, sind größere Ströme nötig (geringer Ausgangswiderstand) als z.B. für 32 kHz. Im low-power Mode mit 32kHz wäre dagegen die Stromaufnahme eines niederohmigen U2 reine Stromverschwendung. Deshalb gibt es für den Oszillator drei verschiedene Einstellungen, die für bestimmte Frequenzbereiche optimiert sind.

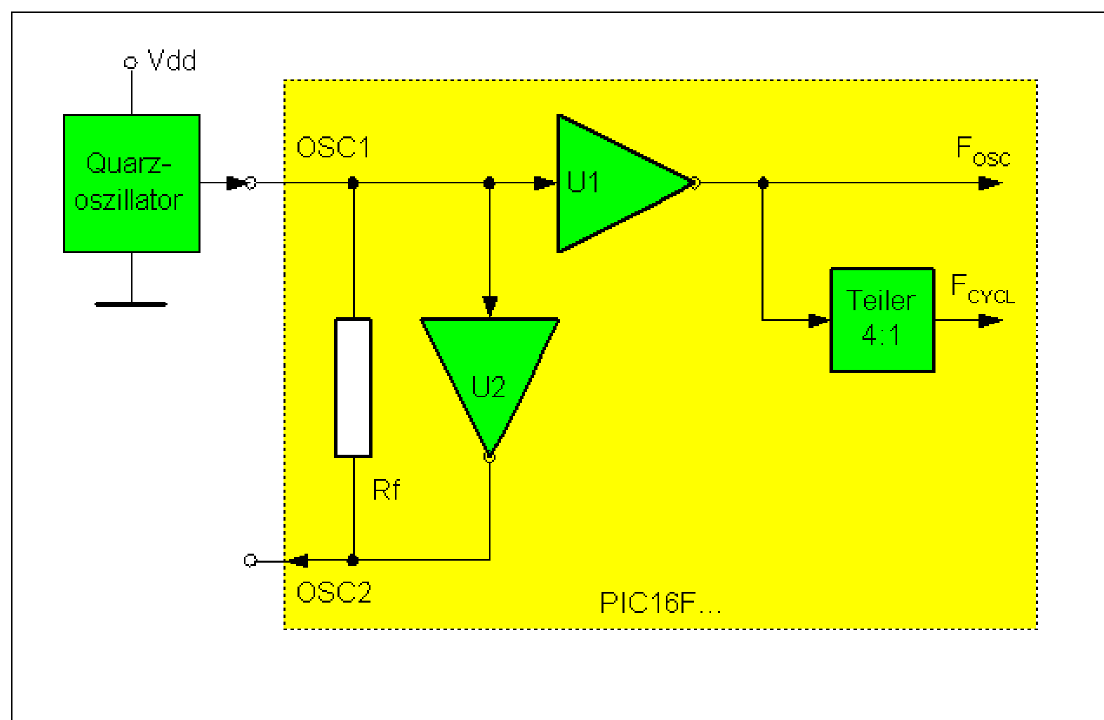
LP - low power
Im LP mode kann U2 keine 'großen' Ströme erzeugen, er muß es auch nicht, da dieser Mode nur für Takte bis zu 200 kHz gedacht ist. Der LP-Mode ist auf niedrigen Stromverbrauch hin optimiert.

XT - normal
Im XT-Mode geht U2 schon etwas stärker an die Arbeit. Der Ausgangswiderstand ist niedig genug, um Schwingungen bis zu 4 MHz sicher zu erzeugen.

HS - high speed
Um Frequenzen oberhalb von 4 MHz zu erzeugen, wird U2 niederohmig eingestellt. Das ist zwar gut für die Verarbeitungsgeschwindigkeit des PIC, hebt aber auch den Stromverbrauch des Oszillators um einige Milliampere an (typabhängig).

externe Taktquelle (LP, XT, HS, ER)

Steht ein externes Takt-Signal zur Verfügung, so wird es im Mode **LP**, **XT** oder **HS** direkt an das Pin OSC1 angeschlossen (CLKIN). Der Inverter U1 speist dieses Signal dann in den PIC ein, wärend der Ausgang OSC2 unbenutzt bleibt. An ihm ließe sich aber bei bedarf der Takt abgreifen (CLKOUT).



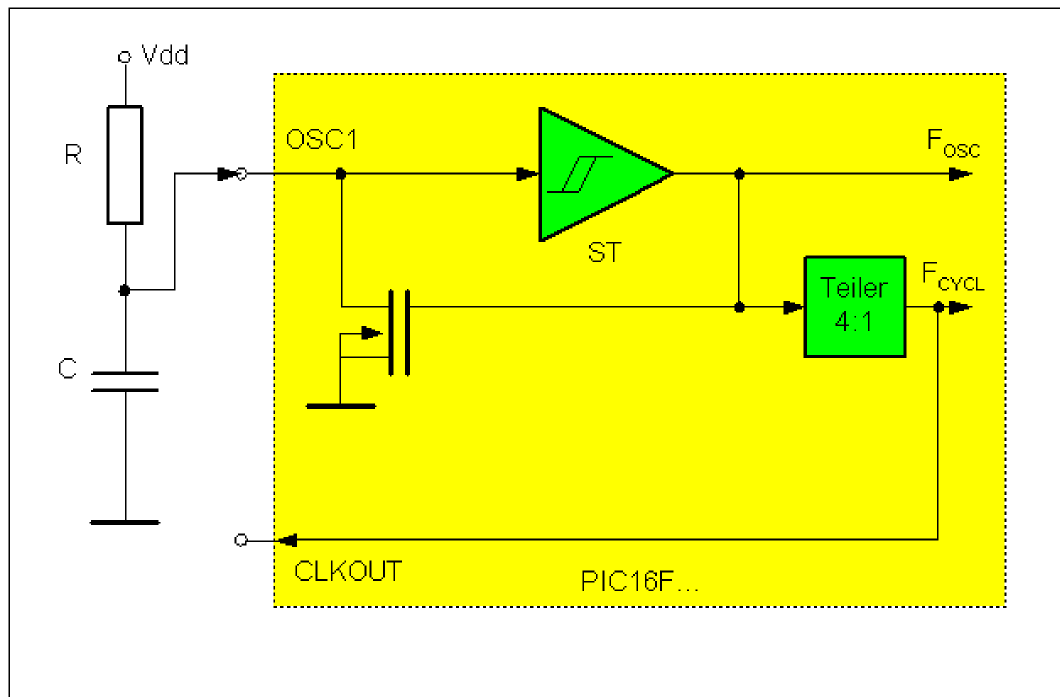
Als externe Taktquellen eignen sich z.B.

Quarzoszillatoren.

Einige Typen (16F62x) unterstützen den **EC** (external clock)-Mode. Auch in diesem Mode kann ein externer Takt direkt an OSC1 angelegt werden. Das OSC2-Pin wird dann allerdings nicht als Taktausgang (CLKOUT) verwendet, sondern kann als I/O-Pin genutzt werden.

Der RC-Oszillator (RC)

Wer bei den Kosten auf die Pfennige schauen will, und mit einem unstabilen Takt leben kann, kann auch den **RC**-Mode benutzen.



Ein Kondensator wird über einen Widerstand aufgeladen. Erreicht die Ladespannung am Kondensator die obere Schaltschwelle des Schmitttriggers (ST) am Pin OSC1, so aktiviert dieser den FET, der den Kondensator wieder entlädt. Fällt daraufhin die Kondensatorspannung unter die untere Schaltschwelle des Schmitt-Triggers, dann sperrt der FET wieder, und ein neuer Ladezyklus beginnt. Die Schwingfrequenz

hängt dabei von den Werten des Widerstands, des Kondensators sowie dem ST und dem FET ab.

Alle Werte unterliegen naturgemäß großen Toleranzen und Driften.Mit einer Taktabweichung von 25% muß man rechnen.

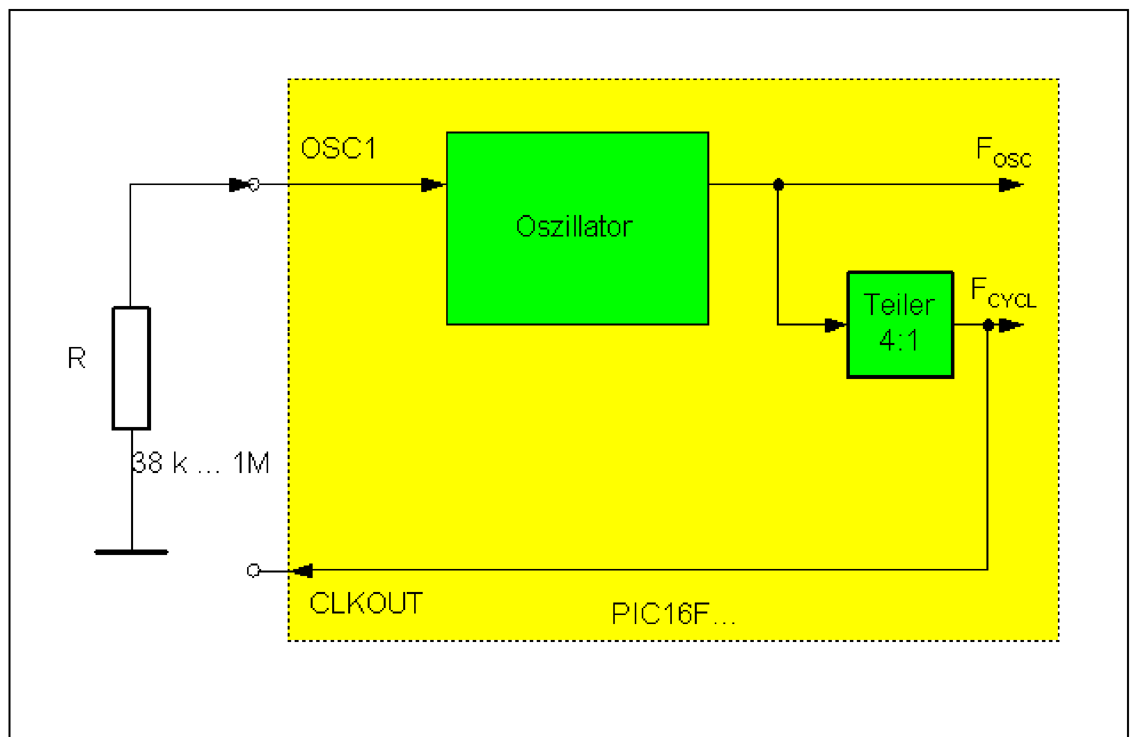
Das OSC2-Pin wird auf die Funktion CLKOUT umgeschaltet. In dieser Funktion stellt es den Zyklustakt des PIC bereit, der 1/4 des Oszillatortaktes beträgt.

Der Wert des Kondensators sollte 20pF nicht unterschreiten, ansonsten wirken sich Exemplarstreuungen und Gehäusevariante des PIC stärker auf die Taktfrequenz aus als der Kondensator. Der Widerstand sollte zwischen 5 kOhm und 100 kOhm liegen. Die Frequenz läßt sich zwischen ca. 30 kHz und 4 MHz einstellen.

++ACHTUNG++
Ist der Oszillator im **RC**-Mode, darf er **nicht mit einem externen Takt** gespeist werden. Man kann an der Schaltung leicht erkennen, daß der ST ständig versuchen würde den High-Teil des Taktes kurzzuschließen!! Das kann zur Beschädigung des PIC wie auch der Taktquelle führen.

externer Widerstand (16F62x) (ER)

Eine Billiglösung, die man verwenden kann, wenn es nicht auf Präzision ankommt, ist der **ER**-Mode. Ein Widerstand von OSC1 nach Masse stellt die Frequenz eines internen RC-Oszillators ein. Microchip garantiert den Betrieb mit Widerständen von 38kOhm bis zu 1 MOhm. Empfohlen wird maximal 4MHz.



Im laufenden Betrieb kann per Software auf einen festen 37 kHz-Takt (unabhängig vom Wert des externen Widerstands) umgeschaltet werden.

Das OSC2-Pin wird auf die Funktion CLKOUT umgeschaltet. In dieser Funktion stellt es den Zyklustakt des PIC bereit, der 1/4 des Oszillatortaktes beträgt. Es ist auch möglich, das OSC2-Pin von der

CLKOUT-
Funktion zu
entbinden, und
es als
normales IO-
Pin zu
benutzen.

interner Oszillator (16F62x 12F6xx) (INTRC)

Im **INTRC**-Mode wird ein interner 4MHz-Oszillator verwendet, der leider nicht sehr stabil ist (3,65 ... 4,28 MHz).
Im laufenden Betrieb kann per Software auf einen festen 37 kHz-Takt umgeschaltet werden.

Der interne 4MHz-Oszillator der 12F6xx-PICs läuft stabiler. Erreicht wird das durch einen exemplaabhängigen Korrekturwert, den man per Software in ein spezielles Register schreiben kann. Der Korrektuwert wird beim Hersteller für jeden einzelnen PIC ausgemessen und in die letzte Speicherzelle des Programmspeichers geschrieben.

Direkte und indirekte Adressierung

Allgemeines

Ein PIC besitzt je nach Typ bis zu mehreren hundert Byte Datenspeicher (RAM) und diverse Register zur Steuerung der Funktionen des PIC (special function register - SFR). Bereits an anderer Stelle wurde erwähnt, das dieser Speicher in mehreren parallelen Banken organisiert ist. Wie kann man nun auf die Speicherzellen zugreifen? Da gibt es zwei Methoden die direkte Adressierung und die indirekte Adressierung.

Direkte Adressierung

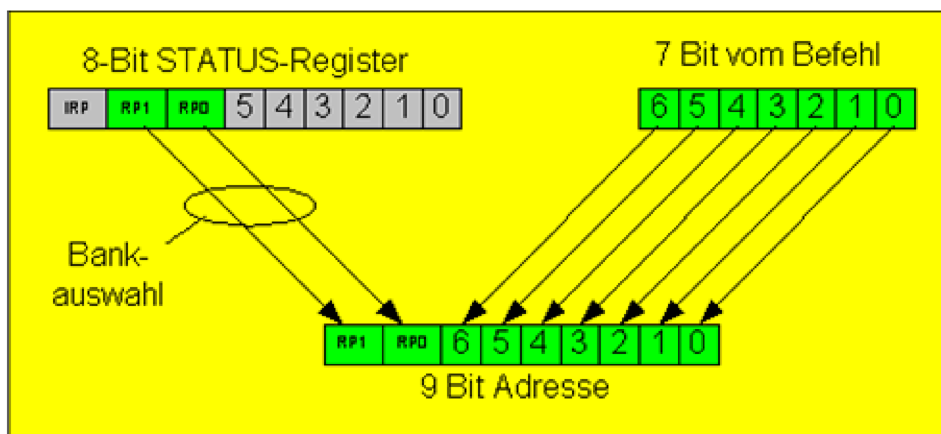
Der einfache Befehl

```
movwf     0x20
```

kopiert den Wert des Arbeitsregisters w in das Register mit der Adresse 20h (0x20). Die Adresse des Zielregisters steht direkt im Befehl. In diesem Fall redet man von direkter Adressierung.

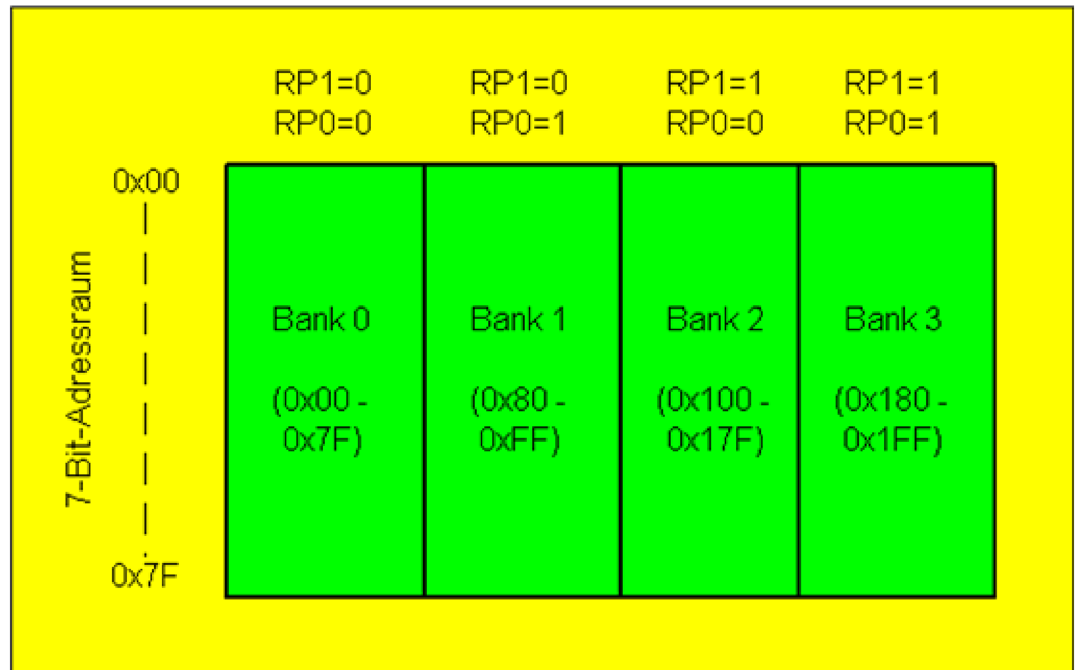
Problematisch ist allein die Tatsache, daß im Befehl nur 7-Bit lange Adressen enthalten sein dürfen. Schreibt man eine längere Adresse (z.B. 0x120) in den Befehl, so werden nur die unteren 7 Bit der Adresse verarbeitet, und der Rest ignoriert. Die folgenden Befehle sind also völlig identisch:

```
movwf     0x20
movwf     0xA0
movwf     0x120
movwf     0x1A0
```



Wie man es auch dreht und wendet, mit 7 Bit lassen sich nur 128 Adressen erzeugen. Das reicht nicht. Im PIC werden 9-Bit lange Adressen verwendet. Um aus den 7-Bit-Adressen des Befehls eine 9-Bit-Adresse zu machen, setzt der Prozessor zwei weitere Bits vor die 7 Bit. Dabei handelt es sich um die Bits **RP1** und **RP0** des STATUS-Registers.

Diese beiden Bits dienen zur Bank-Umschaltung. Der Adressraum des PIC besteht nämlich aus 4 Bänken zu je 128 Adressen (7 Bit). Innerhalb einer Bank kann man problemlos mit den Befehlen arbeiten, will man in eine andere Bank wechseln, setzt man dazu die Bits **RP1** und **RP0** im **STATUS**-Register.



Alle danach ausgeführten Befehle beziehen sich auf die nun ausgewählte Bank. Glücklicherweise ist **STATUS** in allen Bänken unter der selben Adresse (0x03) zu erreichen.

Ein Zugriff auf die Adresse 0xA0 sieht also wie folgt aus:

```
bcf     STATUS,RP1      ; RP1=0
bsf     STATUS,RP0      ; RP0=1
movwf   0x20            ; in der Bank 1 ist das die Adresse A0h  bzw. 0xA0
```

Wer will, kann in der letzten Zeile zur besseren Übersicht auch 'movwf 0xA0' schreiben, das ist egal.
Wir sind nun in der Bank 1, wenn wir wieder in die Bank 0 wollen, muß zuerst **RP0** wieder gelöscht werden.

Was für den mov-Befehl gilt, gilt natürlich genauso für alle anderen Befehle in denen Adressen verwendet werden, wie z.B incf, clrf ...

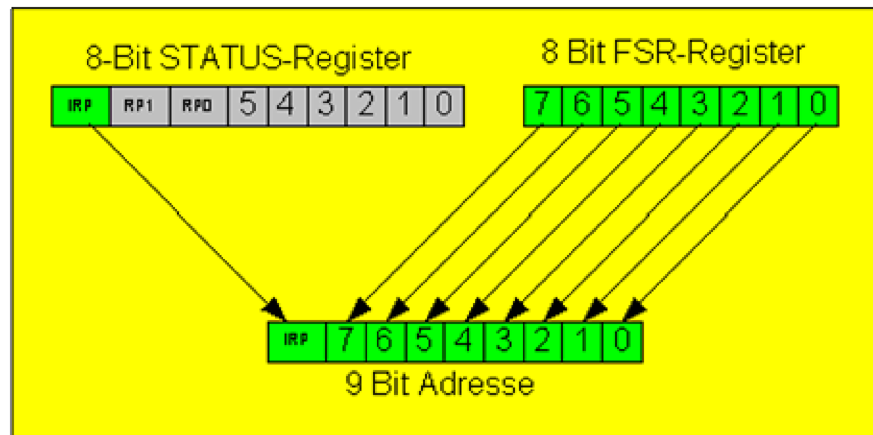
Indirekte Adressierung

Für die meisten Zwecke reicht die direkte Adressierung aus. Es kommt aber vor, das eine Adresse errechnet wird. Das ist zum Beispiel der Fall, wenn man mit

Datentabellen arbeitet, oder wenn man komplette Speicherblöcke löschen oder beschreiben möchte. Dazu dient die indirekte Adressierung.

Zur indirekten Adressierung schreibt man die gewünschte Adresse in das Register **FSR**, das sich in allen Banken unter der Adresse 0x04 findet. Im folgenden kann man auf das gewünschte Register zugreifen, indem man die auf das virtuelle Register **INDF** zugreift. Ein Register **INDF** gibt es physisch gar nicht, der PIC greift in diesem Fall immer auf das Register zu, dessen Adresse in **FSR** steht.

Der Haken daran ist, das das Register **FSR** (wie alle Register) nur 8 Bit lang ist, wärend wir bekanntlich im PIC mit 9-Bit Adressen arbeiten (2 bit zur Bankauswahl, 7 Bit zur Adressierung innerhalb der Bank). Der PIC ergänzt daher die 8 Bit zu einer vollständigen Adresse, indem er davor das Bit **IRP** aus dem **STATUS**-Register setzt.



Will man auf Speicher in den Bänken 0 und 1 zugreifen muß also **IRP** gelöscht (0) sein, will man dagegen auf die Bänke 2 und 3 zugreifen, muß man vorher **IRP** setzen (1).

Nun ein kleines Beispiel zum Löschen der Speicherzellen 40h bis 4Fh

```
        bcf       STATUS, IRP   ; Bank 0 oder 1
        movlw     0x40
        movwf     FSR           ; setzt den Zeiger auf Adresse 40h
Loop
        clrf      INDF          ; löscht die aktuelle Speicherzelle
        incf      FSR, f        ; erhöhen des Adressen-Zeigers
        btfss     FSR, 4        ; schon 4F erreicht?
        goto      Loop          ; nein, die nächste löschen
```

Lesen und Beschreiben des EEPROM und des FLASH

Allgemeines zum EEPROM

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

Der EEPROM-Speicherbereich liegt nicht im normalen Adressbereich des PIC. Er kann nur indirekt adressiert werden, was vergleichsweise umständlich ist. Da kein PIC mehr als 256 Byte EEPROM-Speicher hat, reicht eine 8 Bit Adresse aus um eine EEPROM-Zelle eindeutig zu adressieren.

Das Beschreiben des EEPROMS kann wärend des Brennens durch entsprechende Einstellung der Konfiguration des PIC verboten werden (Codeprotection).

EEPROM lesen

In einem PIC findet man die folgenden 6 Register, die zur Arbeit mit dem EEPROM und FLASH dienen:

- EECON1
- EECON2
- EEDATA
- EEDATH
- EEADR
- EEADRH

Ärgerlich ist, das diese Zellen über alle Speicher-Banken verteilt liegen. Außerdem liegen die Zellen bei jeder PIC-Familie (16F84 / 16F87x / 16F62x) an anderen Adressen und in anderen Banken. Um eine EEPROM Zelle auszulesen benötigen wir aber nur die Register **EEADR**, **EECON1** und **EEDATA**. In meinem Beispiel verwende ich die Adressen-/Banken-Einstellungen für den 16F87x.

Um Daten aus einer EEPROM-Zelle zu lesen schreibt man zuerst die Adresse der betroffenen EEPROM-Zelle in das Register **EEADR**. Danach löscht man im **EECON1**-Register das Bit 7 (**EEPGD**-Bit) und setzt das Bit 0 (**RD**-Bit). Daraufhin schreibt der PIC das Datenbyte aus der EEPROM-Zelle in das Register **EEDATA**, wo man es nun auslesen kann. Im folgenden Beispiel wird die EEPROM-Zelle 10h gelesen:

```
BSF     STATUS, RP1     ;
BCF     STATUS, RP0     ; EEADR liegt in der Bank 2
```

```
MOVLW  0x10            ; ich möchte die EEPROM-Zelle Nr. 10h auslesen
MOVWF  EEADR           ; dazu schreibe ich die Adresse 10h in EEADR

BSF    STATUS, RP0     ; EECON1 liegt in der Bank 3
BCF    EECON1, EEPGD   ; ich möchte aus dem Daten-Speicher lesen
BSF    EECON1, RD      ; EEPROM Leseprozeß starten

BCF    STATUS, RP0     ; EEDATA liegt in der Bank 2
MOVF   EEDATA, W       ; Die Daten der EEPROM Zelle nach W kopieren
```

EEPROM beschreiben

Das Beschreiben eines EEPROM ist ein komplizierter Prozess, bei dem ein isolierter Bereich des Chips schrittweise mit Elektronen gefüllt wird. Dieser Prozeß ist beim PIC automatisiert. Wir müssen dem PIC nur sagen welche Daten wir wo gespeichert haben wollen, und den Schreibprozeß starten. Dieser Schreibprozeß läuft dann im Hintergrund ab, er dauert allerdings einige Millisekunden (4 bis 8 ms beim 16F87x), was im Vergleich zum Beschreiben einer normalen Speicherzelle eine Ewigkeit ist.

Auch ist jedes Beschreiben für den EEPROM Streß. Der Hersteller gibt eine garantierte Lebensdauer von 100 000 Schreibzyklen an. Schon beim Brennen des PIC können EEPROM-Daten gleich mitgebrannt werden. Oft muß man aber im laufenden Programm Werte in den EEPROM schreiben.

Um ein versehentliches Überschreiben von EEPROM-Daten zu verhindern, hat man den Schreibprozeß kompliziert gestaltet.

Zuerst muß die Adresse der zu beschreibenden EEPROM-Zelle in das **EEADR**-Register sowie das zu schreibende Datenwort in das **EEDATA** geschrieben werden. Nach dem vorbereitenden Setzen/Löschen einiger Bits folgt diese festgelegte Folge von 5 Befehlen, die genau einzuhalten ist:

```
MOVLW  55h                ;
MOVWF  EECON2             ; schreibe 55h
MOVLW  AAh                ;
MOVWF  EECON2             ; schreibe AAh
BSF    EECON1, WR         ; starte den Schreibzyklus
```

Hält man sich nicht genau an diese Befehlsfolge, dann werden keine Daten geschrieben. Das gilt natürlich auch, wenn diese Befehlsfolge durch einen Interupt unterbrochen wird. Deshalb sollte man für diese Befehlsfolge alle Interupts verbieten.

Im folgenden Beispiel schreiben wir den Wert 4 in die EEPROM-Zelle 10h

```
BSF    STATUS, RP1        ;
BCF    STATUS, RP0        ; EEADR und EEDATA liegen in der Bank 2
MOVLW  0x10               ;
MOVWF  EEADR              ; Die Zelle 10h soll beschrieben werden
MOVLW  4                  ;
MOVWF  EEDATA             ; eine 4 wollen wir schreiben

BSF    STATUS, RP0        ; EECON1 liegt in der Bank 3
BCF    EECON1, EEPGD      ; wir wollen Datenspeicher beschreiben
BSF    EECON1, WREN       ; nun ist Schreiben erlaubt
BCF    INTCON, GIE        ; verbieten aller Interrupts
```

```
; Die folgenden 5 Zeilen müssen genau so im Code stehen!!!
    MOVLW  55h              ;
    MOVWF  EECON2           ; schreibe 55h nach EECON2
    MOVLW  AAh              ;
    MOVWF  EECON2           ; schreibe AAh nach EECON2
    BSF    EECON1, WR       ; starte den Schreibzyklus

    BSF    INTCON, GIE      ; Interrupts wieder erlauben
```

So, das wäre geschafft. Der PIC schreibt nun intern das Byte in den EEPROM, wärend unser Programm weiterläuft. Solange der Schreibprozeß läuft, darf man allerdings nicht auf den EEPROM zugreifen um z.B. weitere Daten zu schreiben. Deshalb muß man vor einem weiteren Zugriff testen, ob der Schreibprozeß beendet ist. Im einfachsten Fall schickt man das Programm in eine 10ms lange Warteschleife, innerhalb dieser Zeit wird der PIC schon fertig werden. Es geht aber auch eleganter. Am Ende des Schreibprozesses setzt der PIC das Bit **EEIF** im Register **PIR2**. Hat man dieses Bit vor Begin des Schreibens gelöscht, so kann man es nun in einer Schleife so lange Abfragen, bis es wieder gesetzt ist.

Wurde vor Begin des Schreibens das **EEIE**-Bit im Register **PIE2** gesetzt, so löst das Setzen von **EEIF** einen Interupt aus. Legt man den PIC nach dem Start des Schreibzyklus schlafen (sleep) so wacht er durch den Interupt wieder auf.

Allgemeines zum FLASH (nur für 16F87x(A) und 16F7x)

Vollen Schreib- und Lesezugriff auf den FLASH erlauben nur wenige PIC-Typen. Das sind insbesondere PIC16F87x und PIC16F87xA.
Einen Nur-Lesezugriff erlauben die PIC16F7x

Der Flash-Speicher ist der Programmspeicher des PIC. In ihm wird beim 'Brennen' das Arbeitsprogramm des PC abgelegt. Flash-Speicher ist eigentlich nichts weiter als eine verbesserte EEPROM-Technologie, die sich schneller beschreiben läßt als normaler EEPROM. Deshalb kann das Programm auch wieder gelöscht und neu geschrieben werden.

Wenn man sich einmal die technischen Daten von Flash-PICs ansieht, dann fällt auf, das der Flash-Speiche viel größer ist, als der Daten oder EEPROM-Speicher. Deshalb ist man versucht, große Datenfelder im Programmcode abzulegen, um nicht den knappen EEPROM-Speicher zu verschwenden. mit Hilfe des Befehls 'addwf PCL, f ' und vieler darauf folgender 'retlw' Befehle lassen sich so im Programmcode Datenblöcke von bis zu 128 Byte ablegen. Größere Datenfelder sind auf diese Weise aber nur kompliziert zu verwalten.

In einigen PICs (insbesondere den 16F87x-Typen) besteht deshalb die Möglichkeit, mittels spezieller Befehle den FLASH-Speicher ähnlich wie den EEPROM-Speicher zu lesen und zu beschreiben.

Das Beschreiben des FLASH kann beim Brennen durch entsprechende Einstellung der Konfiguration des PIC verboten werden (Codeprotection).
Der Hersteller garantiert für den Flash-Speicher eine Lebensdauer von 1000 Schreibzyklen (PIC16F87x) bzw. 100000 Schreibzyklen (PIC16F87xA). EEPROM-Zellen sind deutlich robusterer und überleben 100000 Schreibzyklen (PIC16F87x)

bzw. 1000000 Schreibzyklen (PIC16F87xA). Für viele Zwecke reicht aber auch eine Lebensdauer von 1000 Schreibzyklen aus.

FLASH lesen

Wollen wir eine Flash-Zelle auslesen, so müssen wir dem PIC zunächst einmal die Adresse dieser Zelle mitteilen. Wären sich eine EEPROM-Zelle mit einem 8-Bit-Wert eindeutig adressieren läßt, braucht man für den deutlich größeren Adressbereich des Flash-Programmspeichers 13 Bit. Da so eine Adresse nicht in eine Speicherzelle paßt, wird sie in zwei Teile zerschnitten und in zwei Speicherzellen geschrieben. Die unteren 8 Bit kommen nach **EEADR** und der Rest (5 Bit) nach **EEADRH**.
Danach setzt man im **EECON1**-Register das Bit 7 (**EEPGD**-Bit) und danach das Bit 0 (**RD**-Bit). Daraufhin liest der PIC die Flash-Zelle, wozu er aber zwei Arbeitszyklen braucht. Deshalb müssen nun im Progammcode zwei NOP-Befehle folgen, um auf die Beendigung des Lesezyklusses zu warten.

Wärenddessen schreibt der PIC das 14-Bit-Datenwort aus der FLASH-Zelle in die Register **EEDATA** und **EEDATH**. Die unteren 8 Bit kann man nun aus **EEDATA** auslesen, wären man die oberen 6 Bit in **EEDATH** findet. Im folgenden Beispiel wird die Programmspeicher-Zelle 120h gelesen:

```
    BSF    STATUS, RP1     ;
    BCF    STATUS, RP0     ; EEADR liegt in der Bank 2
    MOVLW  0x01            ; High Teil der Adresse 120h ist 1h
    MOVWF  EEADRH          ; den schreibe ich in EEADRH
    MOVLW  0x20            ; Low-Teil der Adresse 120h ist 20h auslesen
    MOVWF  EEADR           ; den schreibe ich in EEADR

    BSF    STATUS, RP0     ; EECON1 liegt in der Bank 3
    BSF    EECON1, EEPGD   ; ich möchte aus dem Programm-Speicher lesen
    BSF    EECON1, RD      ; EEPROM Leseprozeß starten
    NOP
    NOP

    BCF    STATUS, RP0     ; EEDATA liegt in der Bank 2
    MOVF   EEDATA, W       ; die unteren 8 Bit Programm Zelle nach W
kopieren
    MOVWF  .....           ; und irgentwohin retten
    MOVF   EEDATH, W       ; die oberen 6 Bit Programm Zelle nach W
kopieren
```

FLASH beschreiben (nur für PIC16F87x / PIC16F87xA)

Ähnlich wie EEPROM-Zellen kann man auch FLASH-Zellen beschreiben. Das soll wohl nicht der Schaffung selbstveränderlichen Programmcodes dienen, wohl aber dem Speichern von Daten im reichlich vorhandenen Programmspeicher. Der Schreibprozeß ähnelt dem EEPROM-Schreiben mit folgenden Unterschieden:

- Es wird eine 13-Bit Adresse benötigt, die in zwei Zellen geschrieben wird
- Es werden 14-Bit Daten benötigt, die in zwei Zellen geschrieben werden

- Wärend des mehrere Millisekunden langen Schreibvorgangs bleibt der Prozessor stehen.

Um ein versehentliches Überschreiben von EEPROM-Daten zu verhindern, hat man den Schreibprozeß kompliziert gestaltet.

Zuerst muß die Adresse der zu beschreibenden EEPROM-Zelle in die Register **EEADR** (untere 8 Bit) und **EEADRH** (oberer 5 Bit) geschrieben werden. Dann schreibt man das 14-Bit Datenwort in die Register **EEDATA** (untere 8 Bit) und **EEDATH** (obere 6 Bit). Nach dem vorbereitenden Setzen/Löschen einiger Bits folgt diese festgelegte Folge von 7 Befehlen, die genau einzuhalten ist:

```
    MOVLW  55h             ;
    MOVWF  EECON2          ; schreibe 55h
    MOVLW  AAh             ;
    MOVWF  EECON2          ; schreibe AAh
    BSF    EECON1, WR      ; starte den Schreibzyklus
    NOP
    NOP
```

Hält man sich nicht genau an diese Befehlsfolge, dann werden keine Daten geschrieben. Das gilt natürlich auch, wenn diese Befehlsfolge durch einen Interupt unterbrochen wird. Deshalb sollte man für diese Befehlsfolge alle Interupts verbieten.

Im folgenden Beispiel schreiben wir den Wert 204h in die Programmspeicher-Zelle 120h:

```
    BSF    STATUS, RP1     ;
    BCF    STATUS, RP0     ; EEADR und EEDATA liegen in der Bank 2

    MOVLW  0x01            ;
    MOVWF  EEADRH          ; High Teil der Adresse 120h ist 1h
    MOVLW  0x20            ;
    MOVWF  EEADR           ; Low Teil der Adresse 120h ist 20h

    MOVLW  2               ;
    MOVWF  EEDATH          ; High Teil des Datenworts 204h ist 2h
    MOVLW  4               ;
    MOVWF  EEDATA          ; Low Teil des Datenworts 204h ist 4h

    BSF    STATUS, RP0     ; EECON1 liegt in der Bank 3
    BSF    EECON1, EEPGD   ; wir wollen Programmspeicher beschreiben
    BSF    EECON1, WREN    ; nun ist Schreiben erlaubt
    BCF    INTCON, GIE     ; verbieten aller Interrupts

; Die folgenden 7 Zeilen müssen genau so im Code stehen!!!
    MOVLW  55h             ;
    MOVWF  EECON2          ; schreibe 55h nach EECON2
    MOVLW  AAh             ;
    MOVWF  EECON2          ; schreibe AAh nach EECON2
    BSF    EECON1, WR      ; starte den Schreibzyklus
    NOP
    NOP

    BSF    INTCON, GIE     ; Interrupts wieder erlauben
```

So, das wäre geschafft. Bei '`BSF    EECON1, WR`' beginnt der PIC das Datenwort in den Programmspeicher zu schreiben. Bis er damit fertig ist steht das Programm still. Danach überspringt der PIC die beiden NOP-Befehle und arbeitet weiter.

PIC-Prozessoren - Input/Output - Interfaces

Einleitung

Die 'erwachsenen' PICs (wie z.B. die PIC16F87x-Familie) besitzen eine Reihe spezieller Interfaces, deren Hardware Input- und Outputfunktionen erleichtert. Dazu zählen:

- neben den normalen digital I/O-Ports
- eine serielle Schnittstelle
- eine IIC-Schnittstelle (I2C)
- analoge Eingänge mit einem 10-Bit Analog/Digital-Wandler
- PWM-Ausgang zur Erzeugung von Pulsen mit variablem Tastverhältnis
- ein Capture-Eingang zur Messung von Pulsen

Im Folgenden wird die Nutzung dieser Schnittstellen beschrieben.

I/O-Pins (Ports)

Die einfachste I/O-Funktion des PIC verkörpern die Ports (PortA ... PortE). Sie stellen jeweils bis zu 8 digitale Leitungen bereit, die als digitaler Eingang oder digitaler Ausgang funktionieren können.

RS-232 (USART)

Die serielle Schnittstelle der PIC16F87x ist eine universelle Schnittstelle für asyncrone und syncrone Datenübertragung (USART). Am interessantesten für den Baster ist die normale asyncrone RS-232-Schnittstelle mit der sich der PIC z.B. an die COM-Schnittstellen eines Personalcomputers anschließen läßt.

IIC (I2C)-Bus

Überall, wo höchste Geschwindigkeit kein wichtiges Argument ist, erfreut sich der serielle-Zweidrahtbus I2C großer Beliebtheit. EEPROMS, ADC sowie viele ICs für Audio- und Videogeräte lassen sich mit diesem Bus Steuern.

ADC-Eingang

Mit den analogen Eingängen des PIC16F87x lassen sich positive Gleichspannungen mit einer Auflösung von 10-Bit messen. Das Entspricht bei einem Meßbereich von 5V immerhin einer Genauigkeit von bis zu 5 Millivolt.

Capture / Compare / PWM

Capture, Compare und PWM werden mit der selben Hardware realisiert, schließen sich also gegenseitig aus. Zum Glück haben die PIC16F87x aber jeweils 2 Capture/Compare/PWM-Module (CCP1, CCP2). Dadurch kann ein Moduel im Capture- oder Compare- Mode laufen, währen das andere Modul im PWM-Mode ist.

Durch Capture und Compare wird außerdem der Timer1 blockiert, wärend PWM den Timer2 benötigt.

Capture

Mit dem Capture-Mode läßt sich der Zeitpunkt genau bestimmen, zu dem ein Impuls am Port RC2 (RC1) eintrifft.

Compare

Mit dem Compare-Mode lassen sich am Port RC2 (RC1) High-Low oder Low-High Flanken mit hoher zeitlicher Präzision erzeugen

PWM

Der PWM-Ausgang ermöglicht das einfache Erzeugen von Impulsen mit einem einstellbaren Tastverhältnis. Damit lassen sich z.B. Schalttransitoren von Transvertern ansteuern.

Timer

Allgemeines zu Timern

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

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

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

PICs besitzen bis zu 3 Timer:

- Timer0 - 8 Bit , Vorteiler
- Timer1 - 16 Bit, Vorteiler
- Timer2 - 8 Bit, Vorteiler&Nachteiler

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

Timer 0

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

Seine Zählimpulse erhält er entweder über das Pin **RA4**, oder er wird mit 1/4 des PIC-Taktes gespeist. Der ausgewählte Takt kann direkt auf den Zähler gegeben werden, oder man verwendet einen Vorteiler, dessen Teilverhältnis von 1:2 bis zu 1:256 in 8 Stufen eingestellt werden kann.

Der momentane Zählerstand ist im Register **TMR0** zugänglich.

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

Die Programmierung des Timer0 erfolgt im Register **OPTION_REG** mit den Bits **T0SE**, **T0CS**, **PSA** sowie **PS2..0**.

OPTION_REG: OPTIONS REGISTER (ADDRESS 81h):

	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
Name:	RBPU	INTEDG	T0CS	T0SE	PSA	PS2	PS1	PS0
Wert:	x	x	**0 oder 1**	**0 oder 1**	**0 oder 1**	**0 oder 1**	**0 oder 1**	**0 oder 1**

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

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

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

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

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

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

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

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

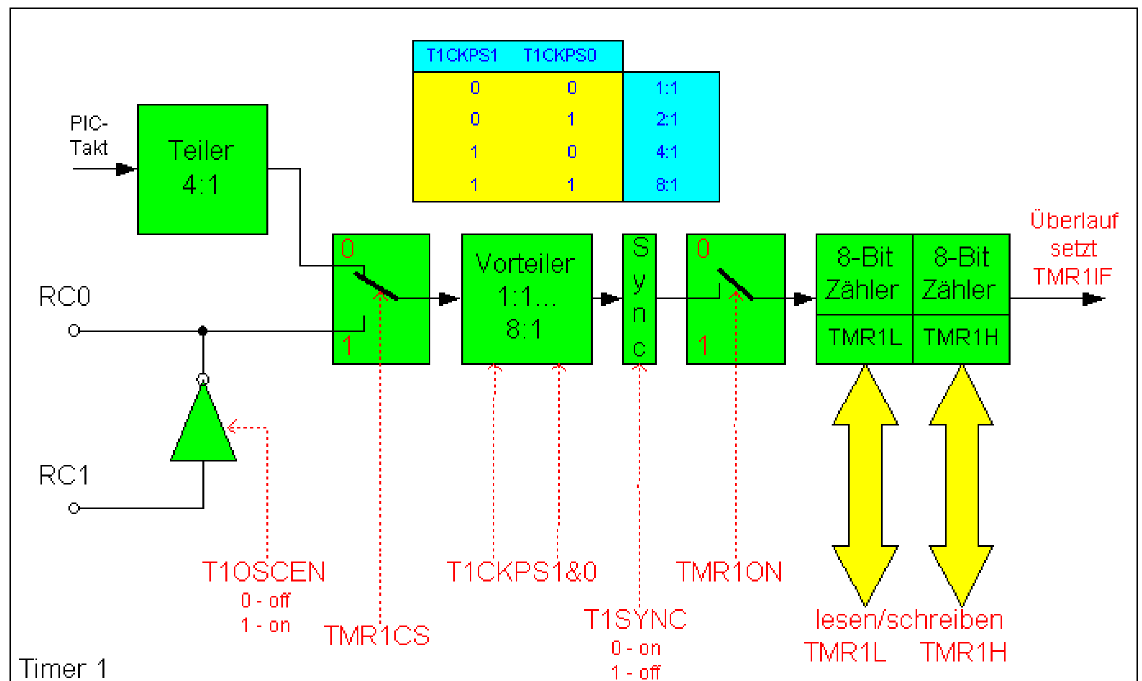
Die Zählgeschwindigkeit des Timer0 ist begrenzt auf 1/4 des PIC-Taktes. Ein mit 10 MHz getakteter PIC kann seinen Timer0 also maximal mit 2,5 MHz zählen lassen. Dabei ist es egal, ob der interne Takt oder der Takt von RA4 genommen wird. Diese Begrenzung gilt aber nicht für den Vorteiler. Dieser verträgt erfahrungsgemäß durchaus 50 MHz. Sein Teilverhältnis muß dann natürlich so hoch gewählt werden, daß am Vorteiler-Ausgang nicht mehr als 1/4 des PIC-Taktes anliegt. Eine praktische Anwendung dafür findet sich im 50-MHz-Frequenzzähler.

Den Vorteiler muß sich der Timer0 mit dem WDT (Watch dog timer) teilen. Da ein richtiges Teilen dieser Hardware kaum möglich ist, muß man sich also entscheiden, ob man den Vorteiler für den Timer0 oder für den WDT verwenden will.

Timer1

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

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



Der Zähler zählt von 0 (0000h) bis 65535 (FFFFh). Dann läuft er über und beginnt bei 0 von vorn. Beim Überlauf wird das Bit **TMR1IF** gesetzt, das auch einen Interrupt auslösen kann (wenn aktiviert). Der Zählerstand kann über die beiden Register **TMR1L** (low-Teil) und **TMR1H** (High-Teil) ausgelesen und verändert werden.

Der Eingang RC0/RC1
Das Pins RC0 kann den Timer1 als Input-Pin mit Pulsen versorgen. Es besteht aber auch die Möglichkeit, das Pin RC1 über einen Verstärker zum Pin RC0 zu schalten. dann wird RC0 automatisch Output-Pin und RC1 dient nun als Takt-Eingang. Sinn des ganzen ist es, mit RC0, RC1 und dem Verstärker einen Oszillator aufzubauen (entspricht dem Oszillator für den PIC-Takt im LP-Mode). Dazu werden RC0 und RC1 mit einem Resonator (z.B. einem Quarz) verbunden. Dieser wird durch den Verstärker zum Schwingen angeregt. Diese Schwingung is dann das Timer1 Inputsignal. Der Oszillator ist für Frequenzen bis zu einigen 100 kHz ausgelegt.

Vorteiler
Der Vorteiler des Timer1 ist immer im Signalpfad der Eingangspulse, beherrscht aber neben 8:1, 4:1 und 2:1 auch das Teilverhältnis 1:1, wodurch er sich dann nicht störend auswirkt.

Synchronisator
Der Synchronisator synchronisiert den Vorteilerausgang mit dem internen Zyklustakt (1/4 des externen PIC-Taktes). Der Synchronisator kann, aber muß nicht ausgewählt werden.

Torstufe
Durch die Torstufe kann der Eingang des Zählers für Impulse gesperrt werden.

Die Programmierung des Timer1 erfolgt im Register **T1CON** mit den Bits **T1CKPS1**&**0**, **T1OSCEN**, **T1SYNC**, **TMR1CS** sowie **TMR1ON**.

OPTION_REG: TIMER! CONTROL REGISTER (ADDRESS 10h):

	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
Name:	-	-	T1CKPS1	T1CKPS0	T1OSCEN	T1SYNC	TMR1CS	TMR1ON
Wert:	x	x	**0 oder 1**	**0 oder 1**	**0 oder 1**	**0 oder 1**	**0 oder 1**	**0 oder 1**

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

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

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

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

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

- T1CKPS1, T1CKPS0 = '00' - Teilverhältnis 1:1

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

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

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

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

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

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

Beispiel:

- Der Timer1 steht auf 01FFh.
- Wir lesen **TMR1H** = 01h.
- Der Timer incrementiert auf 0200h.
- Wir lesen **TMR1L** = 00h.
- Der ausgelesene Zählerstand ist 0100h - völlig falsch

auch ein Ändern der Lesereihenfolge bringt keine Verbesserung

- Der Timer1 steht auf 01FFh.
- Wir lesen **TMR1L** = FFh.
- Der Timer incrementiert auf 0200h.
- Wir lesen **TMR1H** = 02h.
- Der ausgelesene Zählerstand ist 02FFh - völlig falsch

Folgende Routine bringt immer das korrekte Ergebnis. Falls Interrupts benutzt werden, sollten diese wärend der Abfrage abgeschaltet werden (GIE = 0).

```
; Auslesen eines 16-Bit-Wertes aus dem laufenden
Timer1*************
TM1_read
```

```
        movf    TMR1H, w        ; High Byte auslesen
        movwf   Time_H2         ; High Byte speichern
        movf    TMR1L, w        ; Low  Byte auslesen
        movwf   Time_L2         ; Low  Byte speichern
        movf    TMR1H, w        ; High Byte noch einmal
auslesen
        subwf   Time_H2, w      ; 1. uind 2. High-Byte
vergleichen
        btfsc   STATUS, Z       ; sind die gleich (kein
Überlauf) ?
        goto    TM1_weiter      ; ja: wir sind fertig
        movf    TMR1H, w        ; High Byte noch mal
auslesen
        movwf   Time_H2         ;
        movf    TMR1L, w        ; Low  Byte noch mal
auslesen
        movwf   Time_L2         ;

TM1_weiter
```

Alternativ kann man den Timer1 natürlich auch mit TMR1ON=0 vorübergehend anhalten, aber das ist oft nicht gewünscht.

Ein ähnliches Problem ist das Schreiben in TMR1H und TMR1L:

```
; Einschreiben eines 16-Bit-Wertes in den laufenden
Timer1************

TM1_write
        clrf    TMR1L           ; Low Byte löschen =
demnächst kommt kein Überlauf
        movlw   Time_H2         ; High Byte laden
        movwf   TMR1H, f        ; High Byte schreiben
        movlw   Time_L2         ; Low  Byte laden
        movwf   TMR1L, f        ; Low  Byte schreiben
```

Beispiel für Timer0

Das folgende Beispiel entstammt dem Lernbeispiel LED-Ziffernanzeige. Hier wird der Timer0 benutzt, um alle 4 ms einen Interrupt auszulösen
Der PIC-Takt beträgt 4 MHz. Der Timer0 wird über den Vorteiler an den internen Takt (4 MHz / 4 = 1 MHz) angeschlossen. Der Vorteiler wird auf ein Teilverhältnis von 32:1 eingestellt. Damit gibt der Vorteiler alle 32 µs einen Impuls an den Timer0 ab. Nach 125 solchen Impulsen sind 4 ms vorbei. Deshalb wird der Timer0 am Beginn (und dann auch in jedem Interrupt) auf den Wert 131 eingestellt.. Nach 125 Impulsen läuft er dann jeweils über.

```
; Tektquelle: 4 MHz
;**************************************************************
; Includedatei für den 16F84 einbinden

        #include <P16f84.INC>
```

```
; Configuration festlegen
; bis 4 MHz: Power on Timer, kein Watchdog, XT-Oscillator

        __CONFIG        _PWRTE_ON & _WDT_OFF & _XT_OSC

;**************************************************************

; Variablennamen vergeben

w_copy  Equ     0x20            ; Backup für Akkuregister
s_copy  Equ     0x21            ; Backup für Statusregister

;**************************************************************
; los gehts mit dem Programm

        org     0
        goto    Init

;**************************************************************
; die Interruptserviceroutine

        org     4
intvec  bcf     INTCON, GIE     ; disable Interrupt

        movwf   w_copy          ; w retten
        swapf   STATUS, w       ; STATUS retten
        movwf   s_copy          ;

        movlw   D'131'          ; 256-125=131 ((1MHz : 32 ):
125 = 250 Hz)
        movwf   TMR0

; Intrrupt servic routine
Int_serv

;hier folgt die eigentliche Interrupt-Routine,
;die 250 mal pro Sekunde aufgerufen wird


Int_end swapf   s_copy, w       ; STATUS zurück
        movwf   STATUS
        swapf   w_copy, f       ; w zurück mit flags
        swapf   w_copy, w

        bcf     INTCON, T0IF    ; Interrupt-Flag löschen
        bsf     INTCON, GIE     ; enable Interrupt

        retfie

;**************************************************************

;Initialisierung am Anfang des Programms

Init
; 250 Hz-Timer-Interrupt einstellen
        bsf     STATUS, RP0     ; auf Bank 1 umschalten
        movlw   B'10000100'     ; internen Takt zählen,
Vorteiler zum Timer0, 32:1
        movwf   OPTION_REG
        movlw   D'131'          ; 256-125=131 ((1MHz : 32 ):
125 = 250 Hz)
        bcf     STATUS, RP0     ; auf Bank 0 zurückschalten
        movwf   TMR0
```

```
        bsf     INTCON, T0IE    ; Timer0 interrupt erlauben
        bsf     INTCON, GIE     ; Interrupt erlauben

loop    goto    loop            ; eine Endlosschleife

;**********************************************************


        end
```

PIC-Assembler Einstieg

Allgemeines

Wer schon mal in Assembler programmiert hat, egal für welchen Prozessor, sollte keine größeren Probleme haben PIC-Programme zu schreiben. Kleine Fragen, die immer wieder auftauchen sind im Folgenden behandelt:

Ein Projekt mit MPLAB erstellen

MPLAB ist die Entwicklungsumgebung für PIC-Prozessoren, die die Firma Microchip kostenlos zum download anbietet. Sie beinhaltet einen Texteditor einen Assembler und Grundeinstellungen für jeden Prozessortyp (include-Dateien).
Nach dem Start von MPLAB legt man über "Project" - "New Project..." zunächst ein neues Projekt an. Danach erstellt man mit "File" - "New" und "File" - "Save as" eine Assemblerdatei mit der Extension ".ASM". Mit dem Menüpunkt "Project" - "Edit Project" kann man dem Projekt nun die Assemblerdatei zuordnen.
Der Menüpunkt "Projekt" - "Make Project" startet den Assembler und wenn im Programm keine Syntaxfehler enthalten sind, schreibt MPLAB eine ".HEX"-Datei in der das fertige Programm enthalten ist.

Das Gerüst für ein Assemblerprogramm

Jedes Assemblerprogramm hat ein Gerüst, in dem

- der Prozessortyp festgelegt wird
- die prozessorspezifische Include-Datei festgelegt wird
- das Programm steht
- das Ende des Programms gekennzeichnet ist

Beispiel:

```
    list p=16f84                ;der Prozessortyp wird festgelegt
    include "p16f84.inc"        ;die include-Datei mit vielen Festlegungen
wird geladen
                                ;z.B. sind hier Standardnamen für wichtige
                                ;Register und Bits festgelegt
    org    0x00                 ;Startadresse nach Reset ist 0 hier startet
der PIC
    goto   main                 ;Sprung zum Hauptprogramm
    org    0x04                 ;Interruptvector ist 0x04,
                                ;bei Interrupt springt der PIC hierher
;hier muß die Interruptbehandlungsroutine stehen,
;falls Interrupts genutzt werden sollen

main
;hier steht das eigentliche Hauptprogramm

    end                         ;das Ende des Programms
```

Schreibregeln

Nur Marken und Kommentare sollten auf der ersten Position einer Zeile beginnen. Vor einem Assemblerbefehl sollte dagegen ein Tabulator stehen.
Bei Namen von Variablen und Marken unterscheidet MPLAB zwischen Groß- und Kleinbuchstaben. Also sorgfältig schreiben.
Schreibweisen der verschiedenen Zahlensysteme:

Zahlensystem	Schreibweise
hexadezimal	0x20
dezimal	D'128'
binär	B'10101010'

Welche Befehle gibt es?

Ein PIC kennt weniger als 40 unterschiedliche Befehle. Eine Liste aller Befehle findet man hier auf meiner Homepage oder in der PIC-Beschreibung, die man auf der Microchip-Homepage dowloaden kann.

Was ist das "w"? Was ist "f"? Was ist "$"?

Das Arbeitsregister, das oft auch Akku genannt wird heißt im PIC aus unerklärlichen Gründen "w" und nicht etwa "A". (Man gab mir den Tip, das "w" könne von "working register" - also Arbeitsregister abgeleitet worden sein.)

```
	clrw		; lösche den Akku
```

Dagegen steht "f" im Befehl für alle anderen Register. Natürlich muß der Name oder die Adresse des Registers dann noch genau angegeben werden.

```
	incf	0x20	; incrementiere den Inhalt des
			;   Registers mit der hexadezimalen Adresse 0x20
	clrf	STATUS	; lösche das STATUS-Register
```

Gelegentlich findet man das $-Zeichen innerhalb von Adress-Berechnungen. Dabei steht $ für den momentanen Stand des Programmcounters (PC). Der PC weist zu diesem Zeitpunkt auf den aktuellen Befehl. $ kann somit zur Brechnung von Sprungziehlen benutzt werden, ist aber viel unflexibler als die Verwendung von Marken.

```
	goto	$+2	; überspringe ein Wort im Programmspeicher
	clrw		; irgendein 1-Wort-Befehl der übersprungen wird
	incw		; irgendein Befehl zu dem gesprungen wird
```

das gleiche lässt sich sauberer wie folgt programmieren:

```
	goto	ziel	; überspringe ein Wort im Programmspeicher
	clrw		; irgendein 1-Wort-Befehl der übersprungen wird
ziel
	incw		; irgendein Befehl zu dem gesprungen wird
```

Wo sind die Flags?

Auch ein PIC kennt die üblichen Flags wie z.B. C (Carry) und Z (Zero) die eine Überlauf oder ein Nullergebnis der vorangegangenen Operation kennzeichnen. Sie sind aber als einzelne Bits des Registers STATUS versteckt.
Falls also bei gesetzten Zero-Flag ein Sprung zur Marke "Nirwana" ausgeführt werden soll, muß das Entsprechende Bit des Registers STATUS getestet werden:

```
    btfsc     STATUS,Z  ; teste das Bit Z (Zero) im Register STATUS (bt -
bit test),
                        ; wenn es nicht gesetzt ist ignoriere den nächsten
Befehl
                        ; (sc - skip if clear)
    goto      Nirwana   ; ansonsten springe nach Nirwana
```

PIC-Assembler - Befehle

Befehlsübersicht

ADDLW , ADDWF , ANDLW , ANDWF , BCF , BSF , BTFSC , BTFSS , CALL , CLRF , CLRW , CLRWDT , COMF

DECF , DECFSZ , GOTO , INCF , INCFSZ , IORLW , IORWF , MOVF , MOVLW , MOVWF , NOP

RETFIE , RETLW , RETURN , RLF , RRF , SLEEP , SUBLW , SUBWF , SWAPF , XORLW , XORWF

Befehlsübersicht nach Gruppen

Kopierbefehle (MOV...)
Löschbefehle (CLR...)
Setzen und Löschen einzelner Bits, Bitverschiebungen, Bitvertauschungen (BSF, BCF, RLF, RRF, SWAPF)
Aritmetische Operationen (ADD.., SUB.., COM..., AND..., IOR..., XOR...)
Increment und Decrement (INC... und DEC...)
Steuerbefehle und Anderes (NOP, BTFSC, BTFSS, GOTO, CALL, RETURN, RETLW, RETFI, SLEEP, CLRWDT)

Quellen

Immer wieder werde ich nach einem deutschsprachigen Buch gefragt, in dem die Programmierung der PICs erläutert ist. Ich kenne keines, obwohl es bestimmt gute Bücher dieser Art gibt. Ich habe alles was ich weiß aus den Veröffentlichungen von Microchip. Für jeden PIC-Typ gibt es auf der Microchip-Homepage ein Datenblatt, das auch die Programmierung beschreibt. Ich empfehle jedem, der des Englischen mächtig ist, diese Datenblätter zu nutzen. Das sind die Originalquellen, und sie bieten deshalb den höchsten Standard an Fehlerfreiheit.

Jede Sekundärliteratur, also jedes Buch jeder Zeitschriftenartikel und z.B. meine Homepage sind naturgemäß fehlerträchtiger als das Orginal.

Trotzdem möchte ich nachfolgend in deutsch die Befehle des PIC erläutern. Wer mir guten Gewissens ein tolles deutsches Buch zu diesem Thema empfehlen kann, soll das gern per eMail tun. Ich bin gern bereit an dieser Stelle gute Sekundärliteratur zu erwähnen.

Allgemeines

Den PIC betrachtet man am Besten als einen 8-Bit Prozessor, mit nur einem Arbeitsregister - dem Register 'W'.

Es gibt Ein-Adressbefehle und Zwei-Adressbefehle. Ein Ein-Adressbefehl bezieht sich nur auf das Arbeitsregister oder nur auf eine Speicherzelle. Ein Beispiel hierfür ist der Löschbefehl, der eine Speicherzelle oder W auf den Wert 0 setzt. Zwei-Adressbefehle arbeiten immer mit dem Arbeitsregister und einer Speicherzelle. Ein Beispiel ist der MOV-Befehl, der den Wert einer Speicherzelle in das Arbeitsregister kopiert (oder umgekehrt).

Somit ist es nicht möglich, den Wert einer Speicherzelle unter Umgehung von W direkt in eine andere Speicherzelle zu kopieren. Hier muß der Wert zunächst aus der Speicherzelle nach W kopiert werden,

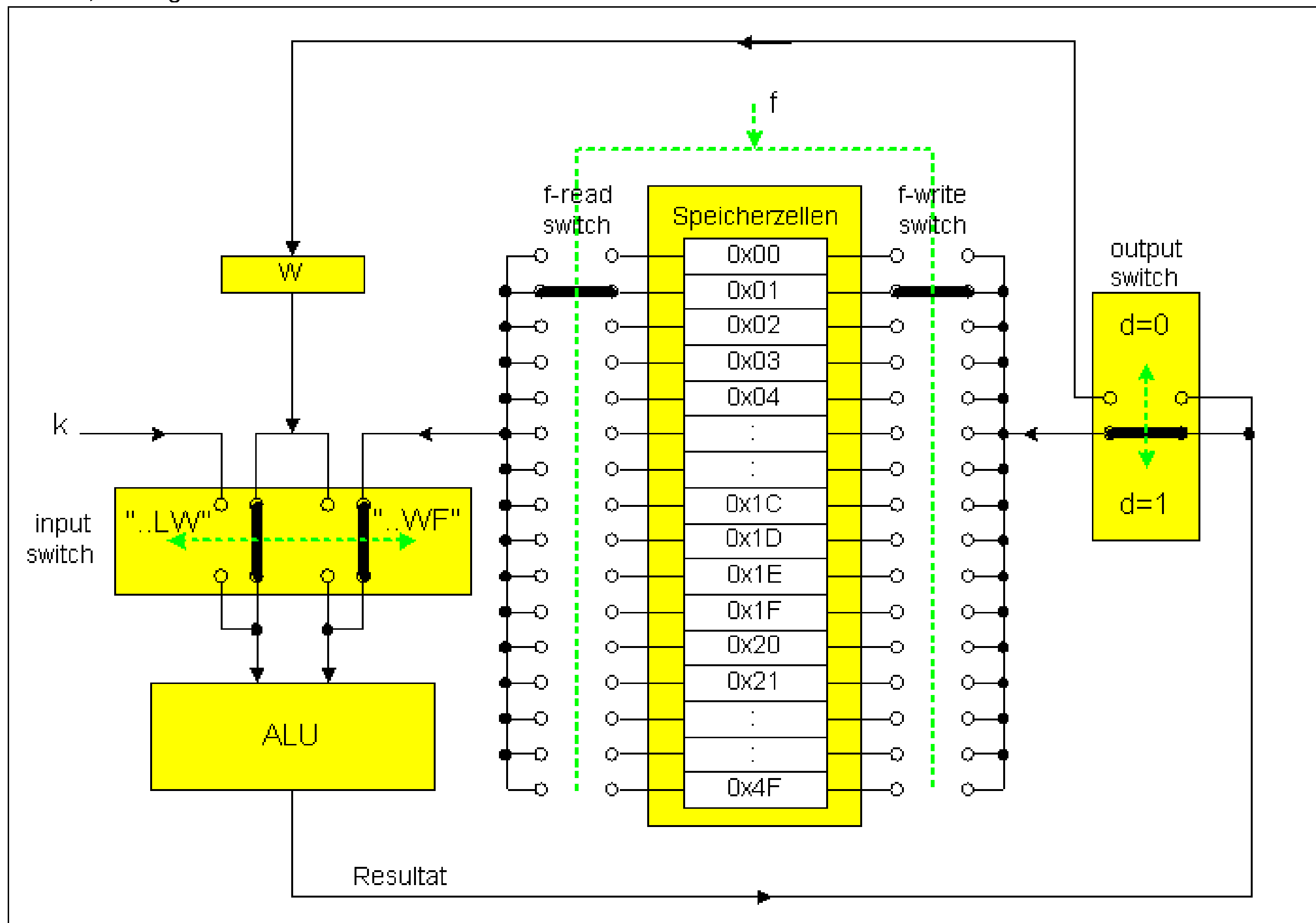
um anschließend von W in die andere Speicherzelle kopiert zu werden. Dafür werden also zwei Befehle benötigt.

Zu den Zwei-Adressbefehlen gehören auch die "literal"-Befehle, das sind Befehle mit Zahlen. Hier ersetzt ein fester Zahlenwert die Speicherzelle. So kann man z.B. eine Zahl in das Arbeitsregister laden, oder eine Zahl zum Arbeitsregister dazuaddieren. Der Zahlenwert ist dann Teil des Befehlscodes, und muß nicht vorher in einer Speicherzelle stehen. Literal-Befehle arbeiten immer nur mit dem Arbeitsregister zusammen. Man kann also keine Zahl direkt in eine Speicherzelle laden, sondern nur in das Arbeitsregister, von dem es mit einem weiteren Zwei-Adressbefehl (MOV) in die Speicherzelle kopiert werden muß.

Für eine normale Addition nach dem Muster a+b=c würde man eigentlich drei Adressen benötigen (je eine für a, b und c) solche Befehle gibt es aber nicht. Auch Additionen und Subtraktionen sind Zwei-Adressbefehle. Der Wert aus dem Arbeitsregister W wird zunächst mit dem Wert aus einer Speicherzelle addiert. Die resultierende Summe wird dann (je nach Befehl) ins Arbeitsregister oder in die selbe Speicherzelle geschrieben. Einer der beiden Ausgangswerte wird dabei also vernichtet. (a := a+b)

Multiplikation, Division oder komplizierteres gibt es als Assemblerbefehl gar nicht. Benötigt man so eine Operation, muß man auf ein Unterprogramm zurückgreifen, das diese Funktionen als Algoritmus realisiert. Solche Algoritmen gibt es aber schon fertig z.B. bei Microchip.

Das kling alles nach einer Beschränkung auf ein Minimum, und das war auch die Idee der Designer. Der PIC ist ein RISC-Prozessor, der nur über einen minimalen Befehlsvorrat verfügt. Das macht den Chip schnell und billig. Gerade einmal 35 unterschiedliche Befehle muß man als Programmierer kennen, mehr gibt es nämlich nicht.



Die Grafik zeigt den möglichen Datenfluß im Prozessor bei einem Zweiadressbefehl. Die zwei Eingänge der ALU ("Rechenzentrale" des Chip) werden entweder von einer Zahl (k) und W oder von W oder einer Speicherzelle gespeist. Das steuert ein "input swich", der bei allen Befehlen der Form "...LW" (literal-Befehl) in der linken und bei "...WF"-Befehlen in der rechten Position steht.

Das Resultat der Operation geht aus der ALU über den "output switch" in eine Speicherzelle oder zurück in W. Dieser "output switch" wird vom Wert des Bits "d" gesteuert. Welche Speicherzelle genau in die Operation einbezogen ist, steuern der "f-read" und "f-write switch". Beide stehen in der selben Position, die durch den Wert "f" vorgegeben ist.
"k", "d", "f" sowie "..LW" und "..WF" stammen aus dem Befehl, den der Prozessor gerade abarbeitet. Der Befehl enthält darüberhinaus noch die Art der mathematischen Operation (Addition, Subtraktion ...) die die ALU ausführen soll.

In der momentanen Schalterstellung werden die Werte aus W und der Speicherzelle 01h miteinander verknüpft (z.B. addiert). Das Resultat wird wieder in der Speicherzelle 01h gespeichert.

Speicherzellen (am Beispiel PIC16F84)

Wie schon erwähnt besitzt der Prozessor neben dem 8-Bit-Arbeitsregister W noch **Speicherzellen** , die auch 8-Bit breit sind. Jede Speicherzelle besitzt eine eigene Adresse, mit der sie angesprochen (addressiert) wird. Im 16F84 gibt es Speicherzellen von der Adresse 00h bis zur Adresse 4Fh und noch eine weite Gruppe von 80h bis CFh. Diese beiden Gruppen werden als Bank 0 und Bank 1 bezeichnet.

Viele dieser Speicherzelle dienen nur der Speicherung von 1-Byte Daten. Andere werden zur Steuerung des Prozessors benutzt. Als reine Speicherzellen werden im 16F84 die Zellen mit den Adressen 0Ch bis 4Fh und 8Ch bis CFh verwendet. In Wirklichkeit verbergen sich hinter diesen zwei Adressgruppen die selben Speicherzellen. Die Zelle mit der Adresse 0Ch hat nämlich auch die Adresse 8Ch, die Zelle 0Dh kann auch mit 8Dh addressiert werden u.s.w. Somit stehen uns 68 freie Speicherzellen zur Verfügung.
In größeren PICs hat jede Bank eigene Speicherzellen. Im 16F84 sind nur die zur Steuerung des PIC benütigten Speicherzellen 00h-0Bh und 80h bis 8Bh in den beiden Bänken verschieden.

Die 68 Byte nehmen sich neben den mehrere 100MByte großen Hauptspeichern moderner PCs recht bescheiden aus, aber für viele Zwecke reicht das aus.

Von 00h bis 0Bh und 80h bis 8Bh liegen Speicherzellen, die zur Steuerung des Prozessors verwendet weren. Diese **Steuerregister** werden wie normale Speicherzellen beschrieben und gelesen. Ihre Funktionen sind im Datenblatt des PIC erläutert. Da ihre Addressierung mit Hilfe der hexadezimalen Adresse umständlich ist (wer kann sich schon die ganzen Zahlen merken) sind für die Steuerregister leicht zu merkende Aliasnamen eingeführt worden. Die Zuordnung der Aliasnamen zu den physischen Adressen steht im der *.INC-Datei, die für den PIC16F84 z.B. P16f84.INC heißt. Um die Aliasnamen verwenden zu können, muß im Assemblerprogram das INC eingebunden werden. Das geschieht mit folgender Zeile am Anfang des Programms:
#include <P16f84.INC>

Auch für die einfachen Speicherzellen kann man beliebige Aliasnamen festlegen. Dazu dient im Assemblerprogramm der EQU-Befehl. Das ist eigentlich gar kein richtiger Befehl, sondern nur ein Hinweis für den Assembler, das in Zukunft eine bestimmte Speicherzelle mit einem bestimmten Namen addressiert wird:
Temp Equ 0x10
Diese Zeile legt für die Speicherzelle mit der Adresse 10h (hier Schreibweise 0x10) den neuen Namen Temp fest. Man muß bei diesen Namen übrigens die Groß- und Kleinschreibung beachten!

Flags

Flags sind einzelne Bits im Prozessor, die sich Besonderheiten eines Rechenergebnisses merken. Sie werden für die Ablaufsteuerung des Programms dringend gebraucht. Alle Flags stehen im Register STATUS (Adresse 03h). An dieser Stelle mögen die beiden Flags genügen, die in den Bits 0 und 2 des STATUS-Registers stehen:

bit 0 : Das **Carry**-Bit
Mit seiner 8-Bit Datenbreite kann der PIC (ohne spezielle Algoritmen) nur mit Zahlen von 0 bis 255

rechnen. Wird der Wert 255 überschritten, so fängt der PIC wieder bei 0 an. So ergibt die Berechnung 255+1 das lustige Ergebnis 0 und 250+20 ergibt 14. Allerdings fällt dem PIC dieses Überlaufen auf, und er setzt in diesem Fall das Carry-Bit auf 1. Bei einer Addition ohne Überlauf bleibt das Carry-Bit dagegen auf 0.
ACHTUNG: Bei einer Subtraktion verhält sich das Carry-Bit genau verkehrt herum. 20-5=15 setzt Carry auf 1 aber 20-25=251 löscht das Carry-Flag.

bit 2 : Das **Zero**-Bit
Ergibt eine Operation zufällig genau Null, so wird das Zero-Bit auf 1 gesetzt. Ergibt die Operation ein anderes Ergebnis, so geht das Zero-Bit auf 0.

Nicht alle Operationen, von denen man es erwarten würde, beeinflussen die Flags. So beeinflußt der INCF-Befehl, der den Inhalt einer Speicherzelle um 1 erhöht, zwar das Zero-Bit, aber nicht das Carry-Bit. In der unten folgenden Beschreibung der einzelnen Befehle sind die beeinflußten Flags jeweils aufgelistet.

Schreibregeln

Einige Zeichen werden im folgenden Text immer wieder auftreten. Deshalb folgt nun ihre Erläuterung:

f	eine Speicherzelle
d	Ergebins wird gespeichert in: d=0: Arbeitsregister W d=1: Speicherzelle
W	das Arbeitsregister
k	ein Zahlenwert von 0 ... 255 (bei CALL und GOTO: 0..2047)
b	ein Zahlenwert von 0 bis 7

Die Befehle

1. **Kopierbefehle (MOV...)**
2. **Löschbefehle (CLR...)**
3. **Setzen und Löschen einzelner Bits, Bitverschiebungen, Bitvertauschungen (BSF, BCF, RLF, RRF, SWAPF)**
4. **Aritmetische Operationen (ADD.., SUB.., COM..., AND..., IOR..., XOR...)**
5. **Increment und Decrement (INC... und DEC...)**
6. **Steuerbefehle und Anderes (NOP, BTFSC, BTFSS, GOTO, CALL, RETURN, RETLW, RETFIE, SLEEP, CLRWDT)**

Kopierbefehle (MOV...)

Mit MOV-Befehlen werden Werte im Prozessor "transportiert" also von einer Speicherzelle oder dem Arbeitsregister in eine andere Speicherzelle kopiert. Der Begriff "kopieren" ist dabei genauer als das englische "move" was bewegen bedeuten würde. Die Speicherzelle, aus der der Wert kommt behält diesen Wert nämlich beim MOV-Befehl unverändert bei, und eine Kopie ihres Inhalts wird in der Ziel-Speicherzelle angelegt.
Mit dem MOVLW-Befehl lassen sich feste Zahlenwerte in die Speicherzellen bringen.

MOVF	Kopiere den Inhalt der Speicherzelle f nach...
Syntax:	MOVF f,d
Bedeutung:	wenn d=0: Der Inhalt der Speicherzelle f wird in das Arbeitsregister kopiert wenn d=1 Der Inhalt der Speicherzelle wird in die selbe Speicherzelle kopiert. Es passiert also gar nichts. Allerdings kann man dadurch prüfen, ob in der Speicherzelle der Wert 0 steht, da dann das Zero-Flag gesetzt werden würde.
Beispiel:	MOVF PORTA,0 ;Das Register PORTA wird in das Arbeitsregister kopiert
Flags:	Z

MOVWF	Kopiere den Inhalt von W in die Speicherzelle f
Syntax:	MOVWF f
Bedeutung:	Der Inhalt des Arbeitsregisters W wird in die Speicherzelle f kopiert
Beispiel:	MOVWF PORTA ;Das Arbeitsregister wird nach PORTA kopiert

MOVLW	Kopiere einen Zahl (L) nach W
Syntax:	MOVLW k
Bedeutung:	Die Zahl k wird in das Arbeitsregisters W geschrieben
Beispiel:	MOVLW 5 ; Der Wert 5 wird in das Arbeitsregister geschrieben

Löschbefehle (CLR...)

Mit Löschbefehlen lassen sich Speicherzellen oder das Arbeitsregister auf 0 setzen. Dabei wird das Zero-Flag gesetzt.

CLRF	Lösche die Speicherzelle f
Syntax:	CLRF f
Bedeutung:	In die Speicherzelle f wird mit der Wert 0 geschrieben.
Beispiel:	CLRF PORTA ;In das Register PORTA wird 0 geschrieben
Flags:	Z=1

CLRW	Lösche W
Syntax:	CLRW
Bedeutung:	Das Arbeitsregisters W wird mit 0 beschrieben
Beispiel:	CLRW ;In das Arbeitsregister wird 0 geschrieben

Setzen und Löschen einzelner Bits, Bitverschiebungen, Bitvertauschungen (BSF, BCF, RLF, RRF, SWAPF)

Mit den BSF und BCF-Befehlen lassen sich einzelne Bits in einer beliebigen Speicherzelle auf 1 oder 0 stellen. Dabei werden Flags **nicht** beeinflußt.
Die Bits innerhalb einer 8-Bit-Speicherzelle tragen die Nummern 0 bis 7, wobei Bit Nr. 0 den kleinsten Wert hat (LSB = 1) wärend Bit Nr.7 den höchsten Wert besitzt (MSB = 128). Einzelne Bits im Arbeitsregister W lassen sich nicht direkt setzen.

BSF	Ein Bit (das Bit Nr. b) in einer Speicherzelle f setzen
Syntax:	BSF f,b
Bedeutung:	In der Speicherzelle f wird das Bit b auf 1 gesetzt
Beispiel:	BSF PORTA,3 ; Im Register PORTA wird das Bit Nr.3 auf 1 gesetzt ; (das dritt"kleinste" Bit mit dem Wert 4). ; War PORTA vorher Null, so ist jetzt PORTA=4.

BCF	Ein Bit (das Bit Nr. b) in einer Speicherzelle f löschen
Syntax:	BCF f,b
Bedeutung:	In der Speicherzelle f wird das Bit b auf 0 gesetzt
Beispiel:	BCF PORTA,3 ; Im Register PORTA wird das Bit Nr.3 auf 0 gesetzt ; (das dritt"kleinste" Bit mit dem Wert 4). ; Was PORTA vorher 6, so ist jetzt PORTA=2.

RLF	Alle Bits der Speicherzelle f um eine Position nach links verschieben
Syntax:	RLF f,d
Bedeutung:	In der Speicherzelle f werden alle Bits auf die nächst höhere Position verschoben. Bit 6 wandert zur Position 7, Bit 5 zur Position 6, ..., und Bit 0 zur Position 1. In die Position 0 wird der Wert des Carry-Flags eingetragen. Das Bit 7 wird in das Carry-Flag geschoben. Es ist ein ein linksherum-Rotieren der Speicherzelle f durch das Carry-Register um eine Position. wenn d=0: Das Ergebnis der Verschiebung wird Arbeitsregister W gespeichert. Die Speicherzelle f bleibt unverändert. wenn d=1 Das Ergebnis der Verschiebung wird wieder in der Speicherzelle f gespeichert.
Beispiel:	RLF 0x20,1 ; Im Register 20h rotiert alles um 1 Position nach links. ; War in 20h vorher der Wert 6 und das Carry-Flag=1, so steht ; jetzt in 20h der Wert 13 und C-Flag=0.
Flags:	C

RRF	Alle Bits der Speicherzelle f um eine Position nach rechts verschieben
Syntax:	RRF f,d
Bedeutung:	In der Speicherzelle f werden alle Bits auf die nächst niedrigere Position verschoben. Bit 1 wandert zur Position 0, Bit 2 zur Position 1, ..., und Bit 7 zur Position 6. In die Position 7 wird der Wert des Carry-Flags eingetragen. Das Bit 0 wird in das Carry-Flag geschoben. Es ist ein ein rechtsherum-Rotieren der Speicherzelle f durch das Carry-Register um eine Position. wenn d=0: Das Ergebnis der Verschiebung wird Arbeitsregister W gespeichert. Die Speicherzelle f bleibt unverändert. wenn d=1 Das Ergebnis der Verschiebung wird wieder in der Speicherzelle f gespeichert.
Beispiel:	RRF 0x20,1 ; Im Register 20h rotiert alles um 1 Position nach rechts. ; War in 20h vorher der Wert 13 und das Carry-Flag=0, so steht ; jetzt in 20h der Wert 6 und C-Flag=1.
Flags:	C

SWAPF	Obere und untere 4 Bit der Speicherzelle f austauschen
Syntax:	SWAPF f,d
Bedeutung:	Die obere und die untere Hälfte des Wertes in f werden ausgetauscht. wenn d=0: Das Ergebnis des Tauschs wird Arbeitsregister W gespeichert. Die Speicherzelle f bleibt unverändert. wenn d=1 Das Ergebnis des Tauschs wird wieder in der Speicherzelle f gespeichert
Beispiel:	SWAPF 0x20,1 ; Die obere und untere Hälfte des Werts in 20h ; werden vertauscht, War vorher in 20h der Wert 25h ; gespeichert, so steht nun eine 52h dort.

Aritmetische Operationen (ADD.., SUB.., COM..., AND..., IOR..., XOR...)

Das sind die Befehle zur "Datenverarbeitung". Hier wird also gerechnet.

ADDWF	Das Arbeitsregister mit einer Speicherzelle addieren
Syntax:	ADDWF f,d
Bedeutung:	Der Inhalt von W wird mit dem Inhalt von f addiert. wenn d=0:

	Das Ergebnis wird Arbeitsregister W gespeichert. Die Speicherzelle f bleibt unverändert. wenn d=1 Das Ergebnis wird wieder in der Speicherzelle f gespeichert
Beispiel:	ADDWF 0x20,1 ; Der Inhalt der Speicherzelle 20h wird mit dem ; Inhalt von W addiert. Die Summe wird in der ; Speicherzelle 20h abgelegt.
Flags:	C, DC, Z

ADDLW	Das Arbeitsregister mit einer Zahl addieren
Syntax:	ADDLW k
Bedeutung:	Der Wert von W wird um k erhöht.
Beispiel:	ADDLW 5 ; Der Wert von w wird um 5 erhöht.
Flags:	C, DC, Z

SUBWF	W von einer Speicherzelle abziehen
Syntax:	SUBWF f,d
Bedeutung:	Vom Wert, der in f steht, wird W abgezogen. wenn d=0: Das Ergebnis wird Arbeitsregister W gespeichert. Die Speicherzelle f bleibt unverändert. wenn d=1 Das Ergebnis wird wieder in der Speicherzelle f gespeichert.
Beispiel:	SUBWF 0x20,1 ; Der Inhalt der Speicherzelle 20h wird um den ; Inhalt von W vermindert. Das Ergebnis wird in der ; Speicherzelle 20h abgelegt.
Flags:	C, DC, Z

SUBLW	W von einer Zahl abziehen
Syntax:	SUBLW k
Bedeutung:	Von der Zahl k wird der momentane Wert von W abgezogen. Das Ergebnis wird wieder in W gespeichert.
Beispiel:	SUBLW 5 ; Die Differenz von k und W wird in W abgelegt. ; War W vorher 3, so ist W nun 2.
Flags:	C, DC, Z

COMF	eine Speicherzelle invertieren (Complement bilden)
Syntax:	COMF f,d
Bedeutung:	Alle Bits der Speicherzelle f werden invertiert. wenn d=0: Das Ergebnis wird Arbeitsregister W gespeichert. Die Speicherzelle f bleibt unverändert. wenn d=1 Das Ergebnis wird wieder in der Speicherzelle f gespeichert.
Beispiel:	COMF 0x20,1 ; Der Wert im Register 20h wird invertiert. ; Stand in 20h vorher 0, so steht dort jetzt 0FFh.
Flags:	Z

ANDWF	W und eine Speicherzelle mit der UND-Funktion verknüpfen
Syntax:	ANDWF f,d
Bedeutung:	Der Wert, der in f steht, wird mit W UND-verknüpft. wenn d=0: Das Ergebnis wird Arbeitsregister W gespeichert. Die Speicherzelle f bleibt unverändert. wenn d=1 Das Ergebnis wird wieder in der Speicherzelle f gespeichert.

Beispiel:	ANDWF 0x20,1 ; Der Inhalt der Speicherzelle 20h wird mit dem
	              ; Inhalt von W UND-verknüpft. Das Ergebnis wird in der
	              ; Speicherzelle 20h abgelegt.

Flags:	Z

ANDLW	W und eine Zahl mit der UND-Funktion verknüpfen
Syntax:	ANDLW k
Bedeutung:	Die Zahl k wird mit dem momentane Wert von W UND-verknüpft. Das Ergebnis wird wieder in W gespeichert.

Beispiel:	ANDLW 5 ; W wird mit 5 UND-verknüpft. Das Resultat
	             ; wird wieder in W gespeichert.
	             ; War W vorher 11h, so ist W jetzt 01h.

Flags:	Z

IORWF	W und eine Speicherzelle mit der ODER-Funktion verknüpfen (inclusive-ODER)
Syntax:	IORWF f,d
Bedeutung:	Der Wert, der in f steht, wird mit W ODER-verknüpft. wenn d=0: Das Ergebnis wird Arbeitsregister W gespeichert. Die

	Speicherzelle f bleibt unverändert. wenn d=1 Das Ergebnis wird wieder in der Speicherzelle f gespeichert.
Beispiel:	IORWF 0x20,1 ; Der Inhalt der Speicherzelle 20h wird um den ; Inhalt von W ODER-verknüpft. Das Ergebnis wird in der ; Speicherzelle 20h abgelegt.
Flags:	Z

IORLW	W und eine Zahl mit der ODER-Funktion verknüpfen (inclusive-ODER)
Syntax:	IORLW k
Bedeutung:	Die Zahl k wird mit dem momentane Wert von W ODER-verknüpft. Das Ergebnis wird wieder in W gespeichert.
Beispiel:	IORLW 5 ; W wird mit 5 ODER-verknüpft. Das Resultat ; wird wieder in W gespeichert. ; War W vorher 11h, so ist W jetzt 15h.
Flags:	Z

XORWF	W und eine Speicherzelle mit der Exclusiv-ODER-Funktion verknüpfen
Syntax:	XORWF f,d
Bedeutung:	Der Wert, der in f steht, wird mit W EX-ODER-verknüpft. wenn d=0: Das Ergebnis wird Arbeitsregister W gespeichert. Die Speicherzelle f bleibt unverändert. wenn d=1 Das Ergebnis wird wieder in der Speicherzelle f gespeichert.
Beispiel:	XORWF 0x20,1 ; Der Inhalt der Speicherzelle 20h wird um den ; Inhalt von W EX-ODER-verknüpft. Das Ergebnis wird in der ; Speicherzelle 20h abgelegt.
Flags:	Z

XORLW	W und eine Zahl mit der Exclusiv-ODER-Funktion verknüpfen
Syntax:	XORLW k
Bedeutung:	Die Zahl k wird mit dem momentane Wert von W EX-ODER-verknüpft. Das Ergebnis wird wieder in W gespeichert.
Beispiel:	XORLW 5 ; W wird mit 5 EX-ODER-verknüpft. Das Resultat ; wird wieder in W gespeichert. ; War W vorher 11h, so ist W jetzt 14h.
Flags:	Z

Increment und Decrement (INC... und DEC...)

Increment- und Decrement-Befehle erhöhen bzw. verringern den Wert einer Speicherzelle oder des Arbeitsregisters jeweils um 1. Sie eignen sich damit zum Aufbau von Zählschleifen. Läuft eine Speicherzelle beim Incrementieren über (255+1=0) oder beim Decrementieren unter (0-1=255) wird das Carry-Flag nicht gesetzt. Ist das Ergebnis von Increment oder Decrement aber 0 (255+1=0 oder 1-1=0), so wird das Zero-Flag gesetzt.

INCF	Erhöhe den Wert aus der Speicherzelle f um 1
Syntax:	`INCF f,d`
Bedeutung:	wenn d=0: Der Wert in f wird mit 1 addiert, und das Ergebnis in W gespeichert. wenn d=1: Der Wert in f wird mit 1 addiert, und das Ergebnis wieder in f gespeichert.
Beispiel:	INCF 0x20,1 ; Der Inhalt der Speicherzelle mit der Adresse 20h wird um 1 erhöht. ; War er vorher z.B. 45, so ist er jetzt 46.
Flags:	Z

DECF	Verringere den Wert aus der Speicherzelle f um 1
Syntax:	`DECF f,d`
Bedeutung:	wenn d=0: Vom Wert in f wird 1 abgezogen, und das Ergebnis in W gespeichert. wenn d=1: Vom Wert in f wird 1 abgezogen, und das Ergebnis wieder in f gespeichert.
Beispiel:	DECF 0x20,1 ; Der Inhalt der Speicherzelle mit der Adresse 20h wird um 1 erniedrigt. ; War er vorher z.B. 45, so ist er jetzt 44.
Flags:	Z

Eine Besondere Form der DEC... und INC...-Befehle beinhaltet einen relativen Sprung: Falls das Ergebnis der Incrementierung oder Decrementierung 0 ist, wird der folgende Befehl übersprungen. Damit lassen sich einfach Schleifen aufbauen, die eine bestimmte Anzahl von Zyklen durchlaufen werden müssen. Da die Auswertung des Null-Zustandes schon intern erfolgt, wird das eigentliche Zero-Flag **nicht** beeinflußt.

INCFSZ	Erhöhe den Wert aus der Speicherzelle f um 1. Falls das 0 ergibt, dann ignoriere den nachfolgenden Befehl.
Syntax:	`INCFSZ f,d`
Bedeutung:	wenn d=0: Der Wert in f wird mit 1 addiert, und das Ergebnis in W gespeichert. wenn d=1: Der Wert in f wird mit 1 addiert, und das Ergebnis wieder in f

	gespeichert. Ist das Ergebnis der Addition Null, dann wird der nächste Befehl im Programm übersprungen, und mit dem übernächsten weitergebacht.
Beispiel:	`INCFSZ 0x20,1  ; Der Inhalt der Speicherzelle mit` `                ; der Adresse 20h wird um 1 erhöht`

DECFSZ	Verringere den Wert aus der Speicherzelle f um 1. Falls das 0 ergibt, dann ignoriere den nachfolgenden Befehl.
Syntax:	DECFSZ f,d
Bedeutung:	wenn d=0: Vom Wert in f wird 1 abgezogen, und das Ergebnis in W gespeichert. wenn d=1: Vom Wert in f wird 1 abgezogen, und das Ergebnis wieder in f gespeichert. Ist das Ergebnis der Subtraktion Null, dann wird der nächste Befehl im Programm übersprungen, und mit dem übernächsten weitergebacht.
Beispiel:	`DECFSZ 0x20,1  ; Der Inhalt der Speicherzelle mit` `                ; der Adresse 20h wird um 1 erniedrigt`

Beispiel für eine Programmschleife, die 5 Mal durchlaufen wird:

```
    MOVLW    5        ; 5 ins Arbeitsregister laden
    MOVWF    0x20     ; die 5 wird in die Speicherzelle 0x20 kopiert
LOOP                  ; eine Einsprungmarke
;    weitere Befehle in der Schleife
;    können hier eingefügt werden
    DECFSZ   0x20,1   ; der Wert in der Speicherzelle 20h wird um 1
verringert
    GOTO     LOOP     ; Sprung zur Marke LOOP

    nächster Befehl
```

Die ersten beiden Zeilen sin Vorbereitung. Von "LOOP" bis "GOTO LOOP" reicht die Schleife.
Der DECFSZ-Befehl wird immer am Schleifenende ausgeführt. Ist das Ergebnis nicht 0 (sondern 4, 3, 2 oder schließlich 1), so wird der darauf folgende "GOTO LOOP"-Befehl ausgeführt, und der Prozessor macht ab er oben stehenden LOOP-Marke weiter. Beim fünften Mal ist das Ergebnis des DECFSZ-Befehle Null, und der folgende Befehl wird ignoriert. Der GOTO-Befehl wird also nicht ausgeführt, und das Programm wird mit den Befehlen nach der GOTO-Zile fortgesetzt.

Steuerbefehle und Anderes (NOP, BTFSC, BTFSS, GOTO, CALL, RETURN, RETLW, RETFI, SLEEP, CLRWDT)

Der NOP-Befehl tut gar nichts (no operation)

NOP	Einen Takt lang gar nichts tun
Syntax:	NOP
Bedeutung:	Der Prozessor legt für einen Arbeitszyklus eine Pause ein
Beispiel:	NOP ; nichts ändert sich
Flags:	keine

Die Bittestbefehle (BTF..) sind bedingte Sprünge. In Abhängigkeit vom Wert eines beliebigen Bits einer beliebigen Speicherzelle wird der auf diesen Befehl folgende Befehl ausgeführt oder übergangen. Ist dieser (bedingt ausgeführte) Befehl ein Sprungbefehl (GOTO), ergeben die beiden Befehle zusammen einen bedingten Sprung..
Mit den Bittestbefehlen werden in der Regel Sprünge in Abhängigkeit von den Flags realisiert. Die Flags sind einzelne Bits in der Speicherzelle mit dem Bezeichner STATUS (Speicherzelle 0x03 oder 03h). Das Zero-Bit ist dort das Bit Nr. 2 und das Carry-Flag ist das Bit Nr. 0 .
Die addressierung des Zero-Flag wäre demzufolge "STATUS,2" und des Carry-Flag "STATUS,0". Um das zu vereinfachen, wurden für 0 und 2 die Bezeichner C und Z eingeführt. Damit ist das Zero-Flag mit "STATUS,Z" und das Carry-Flag mit "STATUS,C" zu addressieren.

BTFSC	Übergehe nachfolgenden Befehl, wenn Bit=0 (bit test f, skip if clear)
Syntax:	BTFSC f,b
Bedeutung:	wenn in der Speicherzelle f das Bit Nr. b den Wert 0 hat, dann übergehen den nachfolgenden Befehl.
Beispiel:	BTFSC STATUS,Z ; prüfe das Zero-Flag GOTO Marke1 ; bei Z=1 wird diese Zeile ausgeführt: Sprung zu Marke1 GOTO Marke2 ; bei Z=0 wird diese Zeile ausgeführt: Sprung zu Marke2

BTFSS	Übergehe nachfolgenden Befehl, wenn Bit=1 (bit test f, skip if set)
Syntax:	BTFSS f,b
Bedeutung:	wenn in der Speicherzelle f das Bit Nr. b den Wert 1 hat, dann übergehen den nachfolgenden Befehl.
Beispiel:	BTFSS STATUS,C ; prüfe das Carry-Flag GOTO Marke1 ; bei C=0 wird diese Zeile ausgeführt: Sprung zu Marke1 GOTO Marke2 ; bei C=1 wird diese Zeile ausgeführt: Sprung zu Marke2

GOTO	Unbedingter Sprung
Syntax:	GOTO k
Bedeutung:	Springe im Programm zur Adresse k. (k ist 11 Bit lang) Normalerweise ist k eine Marke. Dadurch wird dann ein Bezeichner und keine nackte Zahl verwendet.
Beispiel:	GOTO Marke1 ; Springe zur Marke1 im Programm. . . Marke1 ; hier geht es weiter.

CALL	Sprung in ein Unterprogramm
Syntax:	CALL k
Bedeutung:	Speichere die Adresse des nachfolgenden Befehls im Stack. Dann springe zur Adresse k. (k ist 11 Bit lang) Normalerweise ist k eine Marke. Dadurch wird dann ein Bezeichner und keine nackte Zahl verwendet. An der Adresse k steht ein Unterprogramm, das mit dem Befehl RETURN oder RETLW enden muß.
Beispiel:	CALL Marke1 ; Springe zur Marke1 im Programm. . . Marke1 . ; hier beginnt das Unterprogramm . . RETURN ; hier endet das Unterprogramm, ; springe zurück an die Stelle direkt hinter CALL

Am Ende eines Unterprogramms kehr man mit RETURN und RETLW autiomatisch zu der Stelle im Hauptprogramm zurück, von der das Unterprogramm gerufen wurde.
RETLW setzt dabei W auf einen bestimmten Wert. Falls ein Unterprogramm mehrere mögliche "Enden" hat, kann man so leicht erkennen, welchen Ausgang das Unterprogramm genommen hat.

RETURN	Rückkehr aus einem Unterprogramm
Syntax:	RETURN
Bedeutung:	Hole den bei CALL gespeicherten Wert aus dem Stack, und springe zu dieser Adresse. Dort geht das Programm weiter, das dieses Unterprogramm aufgerufen hatte.
Beispiel:	RETURN ; hier endet das Unterprogramm

RETLW	Rückkehr aus einem Unterprogramm mit einem bestimmten Zahlenwert in W
Syntax:	RETLW k
Bedeutung:	Schreibe zuerst die 8-Bit-Zahl k in das Arbeitsregister W. Hole dann den bei CALL gespeicherten Wert aus dem Stack, und

	springe zu dieser Adresse. Dort geht das Programm weiter, das dieses Unterprogramm aufgerufen hatte.
Beispiel:	RETLW 1 ; hier endet das Unterprogramm ; das Unterprogramm endet mit einer 1 in W.

RETFI wird nur benötigt, wenn man mit Interupts arbeitet.

RETFIE	Rückkehr aus der Interuptroutine
Syntax:	RETFI
Bedeutung:	Setze das Bit "GIE" auf 1 (nun sind Interupts wieder erlaubt). Hole den bei der Auslösung des Interupts gespeicherten Wert aus dem Stack, und springe zu dieser Adresse. Dort geht das Programm weiter, das durch den Interupt unterbrochen wurde.
Beispiel:	RETFI ; hier endet die Interutroutine.

SLEEP und CLRWDT sind nur nötig, wenn man den Watch-Dog-Timer nutzt.

SLEEP	Prozessor in den Stand-By-Modus schalten
Syntax:	SLEEP
Bedeutung:	Der Prozessor wird in den Schlafmodus versetzt (stand-by). Dazu werden der WDT und sein Vorteiler auf 0 gesetzt. Das TO-Flag (inverses Time-out-Status-Bit)wird gesetzt. Das PD-Flag (inverses Power-Down-Status-Bit)wird gelöscht. Der Taktgenerator stoppt.
Beispiel:	SLEEP ; jetzt wird geschlafen
Flags:	TO, PD

CLRWDT	Löschen des Watch-Dog-Timers
Syntax:	CLRWDT
Bedeutung:	Der Watch-Dog-Timer und der Vorteile des Watch-Dog-Timers werden auf 0 gesetzt. Die Statusflags TO (inverses Time-out-Status-Bit) und PD (inverses Power-Down-Status-Bit) werden gesetzt.
Beispiel:	CLRWDT ; WDT und Prescaler=0.
Flags:	TO, PD

PIC-Prozessoren - Interrupt

Einleitung

Das Programm eines Prozessors oder Microcontrollers enthält oft in einer ständig durchlaufenen Schleife alle nötigen Routinen.

So ein Programm hat es aber schwer, auf plötzlich auftretende Ereignisse zu reagieren. Soll der Microcontroller z.B. auf das Drücken einer an einem Pin angeschlossenen Taste reagieren, so muß er diese Taste in kurzen Abständen immer wieder abfragen, um keinen kurzen Tastendruck zu verpassen. Diese Methode der Ereignis-Überwachung nennt sich Polling. Polling ist ausreichend, wenn nur ein oder zwei externe Signale überwacht werden müssen, und der Microcontroller sowieso nichts anderes zu tun hat.

```
; Beispiel für Polling
; Pin RB0 wird überwacht

loop
    btfss   PORTB, 0        ; ist RB0 high?
    goto    loop            ; nein: in der Warteschleife
bleiben

; und weiter im Programm    ; ja: Programm fortsetzen
```

Kompliziert wird das Polling aber, wenn der Microcontroller eigentlich ständig mit komplexen Aufgaben ausgelastet ist, und wärend der Ausführung dieser Aufgaben auch noch zyklisch die Ereignis-Eingänge überwachen soll. In diesem Fall sollte anstelle des Polling der Interrupt verwendet werden.

Beim Interrupt überwacht die Hardware des Microcontrollers mögliche auftretende Ereignisse. Tritt ein Ereignis ein, unterbricht der Microcontroller seine normale Arbeit, und springt in eine spezielle Programmroutine, zur Behandlung dieses Ereignisses - die Interrupt-Routine. Mögliche externe Ereignisse für PIC-Prozessoren sind

- Signal am Pin RB0
- Pegelwechsel an einem der Pins 4..7 des Port B
- Überlauf des Timers 0
- Überlauf des Timers 2
- EEPROM schreiben abgeschlossen
- USART: Zeichen seriell empfangen
- USART: Zeichen seriell gesendet
- ADC ist mit der Wandlung fertig
- Parallel-Port Interrupt
- SSP Interrupt

- CCP Interrupt
- TMR2=PR2
- Bus collisions Interrupt (I2C-Bus)

Welche Interruptquellen ein PIC unterstützt, hängt natürlich von der vorhandenen Hardware ab. Ein 16F84 hat keine USART und folglich auch keine USART-Interrupts.

Flags und Enable-Bits

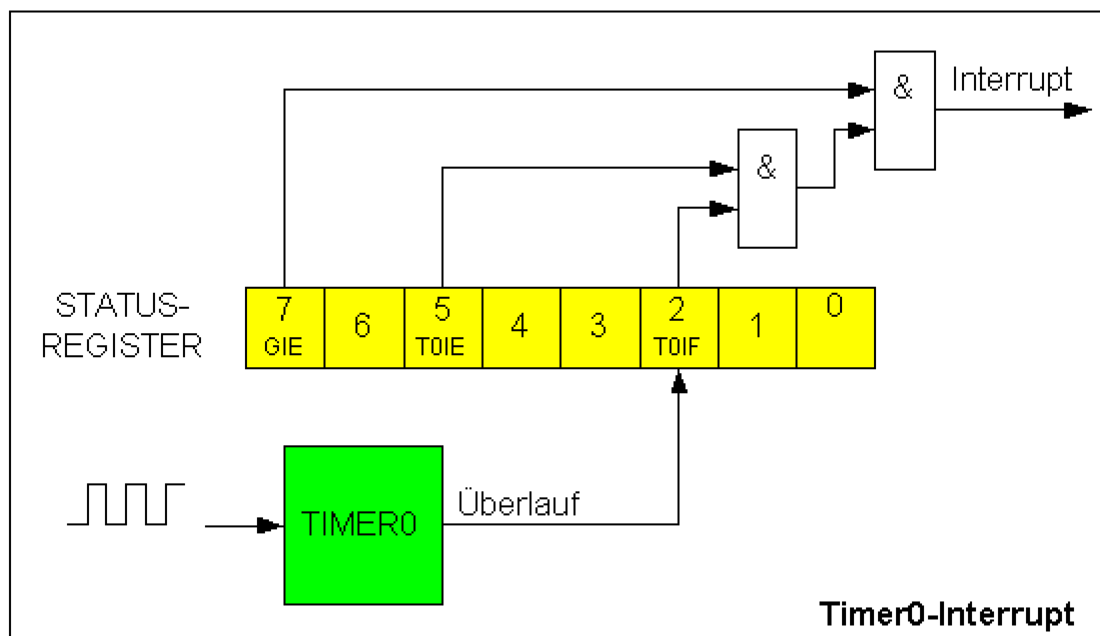
Für jede Interruptquelle gibt es in einem Register ein Interrupt-Bit - das Interrupt-Flag. Tritt ein Ereignis auf (z.B. der Überlauf des Timer0), dann wird das zu diesem Ereignis gehörende Interruptflag auf 1 gesetzt.

Es lösen immer nur die Ereignisse einen Interrupt aus, für die die Interrupterzeugung auch eingeschaltet wurde. Zum Einschalten der benötigten Interrupts dienen in einigen Steuerregistern Enable-Bits für jede Interruptquelle. Diese sind standardmäßig auf 0 (also aus) gesetzt. Wird ein solches Enable-Bit durch das PIC-Hauptprogramm auf 1 gesetzt, dann ist das zugehörige Interrupt-Flag in der Lage, einen Interrupt auszulösen.

Beispiel:
Jedes mal, wenn der Timer0 überläuft setzt er das Bit **T0IF** (Bit 2 im Register **INTCON**) auf 1. Das hat aber keine weiteren Auswirkungen, wenn nicht auch das zugehörige Enable Bit **T0IE** (Bit 5 in **INTCON**) gesetzt wurde.

Außer diesen Enable-Bits, mit denen jede Interruptquelle ein und aus geschaltet werden kann, gibt es auch noch einen Interrupt-Hauptschalter, das Bit **GIE** (Bit 7 in **INTCON**) - general interrupt enable. Auch dieses Bit muß also vom Hauptprogramm zunächst auf 1 gestellt werden.

Läuft nun der Timer0 über, wird im Microcontroller ein Interrupt ausgelöst.



Timer0-Interrupt

Interrupt-Auslösung

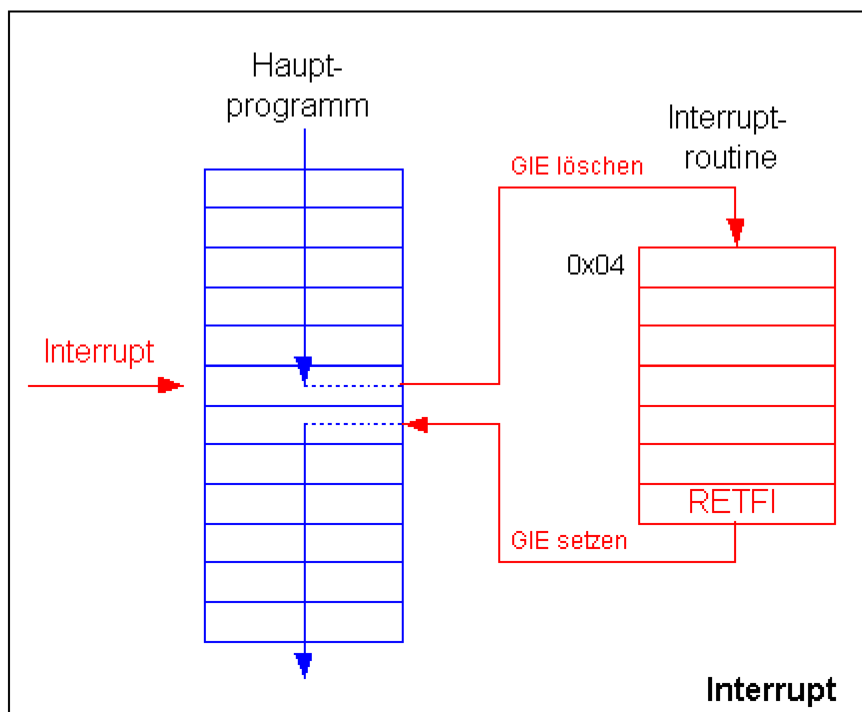
Bei einem Interrupt beendet der Microcontoller noch den gerade anstehenden Befehl des Hauptprogramm, dann speichert er die Adresse des nachfolgenden Befehls (um später wieder zurückzufinden) und springt zur Adresse 04h (0x04). Je nach Takt wird dafür etwa eine Mikrosekunde benötigt.

An dieser Adresse 04h muß also die Interrupt-Behandlungsroutine beginnen.

Gleichzeitig wird **GIE** (der Interrupt-Hauptschalter) auf 0 gesetzt, um einen weiteren Interrupt wärend des Interrupts zu verhindern.

Es folgt nun die Interrupt-Behandlungsroutine.
In dieser Routine muß das auslösende Interrupt-Flag wieder auf 0 zurückgesetzt werden! Wird dies vergessen, löst das noch immer aktive Flag nach dem Interrupt-Ende sofort einen neuen Interrupt aus. Das Resultat wäre ein PIC, der in einer Endlosschleife von Interrupts fest hängt.

Diese Routine muß mit dem Befehl **RETFI** enden. Dies ist ein erweiterter **RETURN**-Befehl. Der Microcontroller springt wieder in das Hauptprogramm (er hat sich ja die Adresse des nächsten Behehls des Hauptprogramms gemerkt) und schaltet das **GIE**-Bit wieder ein, wodurch neue Interrupts wieder erlaubt sind.



Sichern der 'Programm-Umgebung'

Die Interrupt-Routine wird also an eine 'zufällige' Stelle im Hauptprogramm eingefügt, wenn ein Ereignis auftritt.

Das Hauptprogramm verläßt sich darauf, daß nur es selbt das **STATUS**-Register und das Register **w** (Akku) verändert. Das geht auch gut, bis man Interrupts nutzt. Viele normale Befehle einer Interrupt-Routine werden z.B. das Zero-Flag oder das Carry-Flag im **STATUS**-Register verändern. Tritt im folgenden Programmausschnitt

```
    decf    counter, f
    btfsc   STATUS, Z
    goto    Null
```

zwischender 1. und der 2. Zeile ein Interrupt auf, der das Zero-Flag verändert, dann versagt an dieser Stelle das Programm. Auch das **w**-Register wird kaum unverändert eine Interruproutine überstehen.

Um solchen Problemen vorzubeugen, muß die Programm-Umgebung des Hauptprogramms 'gerettet' werden wenn ein Interrupt ausgelöst wird, und der gerettete Zustand muß wieder hergestellt werden, wenn der Interrupt endet. Zu den zu rettenden Werten zählt außer dem (automatisch geretteten) Programcounter:

- das Akku-Register **w**
- das **STATUS**-Register
- das **PCLATCH**-Register

Wärend das **STATUS**-Register und **w** in allen PICs vorkommt, ist **PCLATCH** nur bei größeren PICs (16F87x) von Interesse.

Für einen 16F84 sollte die Interruptbehandlungsroutine stets mit dem folgenden Code beginnen:

```
    movwf   w_temp        ;status retten
    swapf   STATUS,w
    movwf   status_temp
```

und wie folgt enden:

```
    swapf   status_temp,w
    movwf   STATUS
    swapf   w_temp,f
    swapf   w_temp,w
    retfie
```

Für einen 16F87x sollte die Interruptbehandlungsroutine stets mit dem folgenden Code beginnen:

```
    MOVWF   W_TEMP
    SWAPF   STATUS,W
    CLRF    STATUS
    MOVWF   STATUS_TEMP
    MOVF    PCLATH, W
    MOVWF   PCLATH_TEMP
    CLRF    PCLATH         ; Bank 0
```

und wie folgt enden:

```
    MOVF    PCLATH_TEMP, W
    MOVWF   PCLATH
    SWAPF   STATUS_TEMP,W
    MOVWF   STATUS
    SWAPF   W_TEMP,F
    SWAPF   W_TEMP,W
    retfie
```

Die Verwendung von SWAPF-Befehlen wirkt zwar umständlich, aber einfache MOVF-Befehle verändern das Zero-Flag und können deshalb nicht verwendet werden.

Damit sind die wichtigsten Register geschützt. Verändert man in der Interruptroutine weitere Register, die das Hauptprogramm unverändert benötigt (TRISA, TRISB, FSB ...) so muß man sie innerhalb der Interruptroutine ebenfalls retten und am Ende wiederherstellen.

Beispiel

Im folgenden Beispiel soll immer dann, wenn der Eingang RB0 von 0 auf 1 geht, per Interrupt eine LED am Port RA0 umgeschaltet werden (an-aus-an- ...). Das ließe sich natürlich auch einfacher realisieren, aber es geht um die Demonstration der Interrupt-Programmierung.

Das Programm muß aus Hauptprogramm und Interruptrotine bestehen. Da die Interruptrotine auf der Adresse 04h beginnt, ist davor nicht genug Platz für das Hauptprogramm. Deshalb wird auf der Adresse 00h ein Sprung zum Hauptprogramm erfolgen, welches hinter der Interruptroutine stehen muß.

Das Hauptprogramm hat folgende Aufgaben zu erfüllen:

- Port RA0 auf output stellen
- Port RB0 als Interupteingang für 0-1-Flanke einstellen
- Interrupt vom Port RB0 erlauben
- Interrupts generell erlauben (GIE)
- danach in einer Endlosschleife warten

Die Interruptroutine hat folgende Aufgaben zu erfüllen:

- Sichern der Programm-Umgebung
- Umschalten der LED
- Löschen des Interruptflags von Port RB0
- Wiederherstellen der Programmumgebung
- Beenden des Interrupts

Das Port **RB0** kann sowohl bei einer 0-1 wie auch bei einer 1-0-Flanke einen Interrupt auslösen. Das wird mit dem Bit **INTEDG** im **Options**-Register eingestellt.

Das erlauben von **RB0**-Interrupts erfolgt mit dem **INTE**-Bit (Interupt-Enable) im **INTCON**-Register.

Das vom RB0 gesetzte Interuptflag, das in der Interruproutine wieder gelöscht werden muß ist **INTF** in **INTCON**-Register.

```
        list p=16f84
;**************************************************************
;*
;* Pinbelegung
;* ----------------------------------
;* PORTA:  0 LED
;*         1
;*         2
;*         3
;*         4
;* PORTB:  0 Eingang
;*         1
;*         2
;*         3
;*         4
;*         5
;*         6
;*         7
;*
;**************************************************************
;
; sprut (zero) Bredendiek 05/2002
;
; Demo für Interrupt
;  0-1-Flanke an RB0 ändert LED an RA0
;
; Taktquelle: 4 MHz
;
;**************************************************************
; Includedatei für den 16F84 einbinden
 #include <P16f84.INC>

; Configuration festlegen
; bis 4 MHz: Power on Timer, kein Watchdog, XT-Oscillator

        __CONFIG _PWRTE_ON & _WDT_OFF & _XT_OSC

;**************************************************************
; Variablennamen vergeben

w_copy  Equ    0x20               ; Backup für Akkuregister
s_copy  Equ    0x21               ; Backup für Statusregister

;**************************************************************
; los gehts mit dem Programm

        org    0
        goto   Init               ; Sprung zum Hauptprogramm

;**************************************************************
; die Interuptserviceroutine

        org  4
```

```
intvec
        movwf  w_copy              ; w retten
        swapf  STATUS, w           ; STATUS retten
        movwf  s_copy

        incf   PORTA, f            ; invertieren von RA0

Int_end
        bcf    INTCON, INTF        ; RB0-Interupt-Flag löschen
        swapf  s_copy, w           ; STATUS zurück
        movwf  STATUS
        swapf  w_copy, f           ; w zurück mit flags
        swapf  w_copy, w
        retfie

;**************************************************************
; das Hauptprogramm

Init
; Port RA0 auf Ausgabe stellen
        clrf   PORTA               ; LED aus
        bsf    STATUS, RP0         ; auf Bank 1 umschalten
        movlw  B'11111110'         ; PortA RA0 output
        movwf  TRISA
        bcf    STATUS, RP0         ; auf Bank 0 zurückschalten

; RB0-Interrupt einstellen
        bsf    STATUS, RP0         ; auf Bank 1 umschalten
        bsf    OPTION_REG, INTEDG ; 0-1-Flanke an RB0
        bcf    STATUS, RP0         ; auf Bank 0 zurückschalten

        bsf    INTCON, INTE        ; RB0-interupt erlauben
        bsf    INTCON, GIE         ; Interupt generell erlauben

loop    goto   loop                ; eine Endlosschleife

;**********************************************************

        end
```

Anfängerfallen bei der Programmentwicklung

Was sind Fallen?

Gerade der erfahrene Umsteiger nutzt seinen Erfahrungsschatz und setzt manchmal Dinge voraus, die nicht unbedingt zutreffend sind. Daraus entstehen kleine tückische Programmierfehler, die von einem betriebsblinden Programmierer schwer zu finden sind. Keine dieser Fallen ist ein Fehler des Prozessors oder eine Böswilligkeit der Entwickler. Es sind nur die kleinen Unterschiede zum Gewohntem, die beim ersten Mal viel Zeit kosten können.
Eigentlich ist das auch alles in den Handbüchern beschrieben, aber wer liest die schon genau durch, besonders wenn es um scheinbar einfache Dinge geht.
Im folgenden liste ich die Fallen auf, in die ich selber schon getappt bin:

Die Taktraten-Falle

Microchip preist die Eigenschaft seiner Controller, fast alle Befehle in nur einem Zyklus ausführen zu können. Mit diesem Zyklus ist aber nicht ein Takt sondern 4 Takte gemeint.
Ein NOP-Befehl benötigt auf einem mit 10 MHz getakteten PIC also keineswegs nur 100 ns sondern 400 ns. Wer Zeitschleifen aufbaut sollte das berücksichtigen.

Der Timer läßt sich übrigens auch nur mit maximal 1/4 des Prozessortaktes füttern, wenn man den internen Takt verwendet.

Die Interrupt-Falle

Die meisten Prozessoren retten zum Beginn eines Interrupts selbsttätig den Programmzähler und das Statusregister mit den Flags. PIC-Prozessoren zeigen hier falsche Bescheidenheit und beschränken sich auf den Programmzähler.
Viele Befehle in der Interruptbehandlungsroutine verändern aber die Flags. Man könnte sich im Hauptprogramm nie mehr auf die Stellung eines Flags verlassen, sobald ein Interrupt benutzt wird, es sei den man rettet zu Beginn der Interruptroutine die Flags und schreibt sie am Ende des Interrupts wieder zurück.
Dazu kann man aber nicht einfach den mov-Befehl verwenden, da dieser das Z-Flag verändern kann. Microchip schreibt deshalb eine etwas umständlich aussehende aber absolut notwendige Variante vor, um den Programmstatus über einen Interrupt hinwegzuretten.
Für einen 16F84 sollte die Interruptbehandlungsroutine stets mit dem folgenden Code beginnen:

```
    movwf   w_temp        ;status retten
    swapf   STATUS,w
    movwf   status_temp
```

und wie folgt enden:

```
    swapf   status_temp,w
    movwf   STATUS
    swapf   w_temp,f
    swapf   w_temp,w
    retfie
```

Für einen 16F87x sollte die Interruptbehandlungsroutine stets mit dem folgenden Code beginnen:

```
    MOVWF   W_TEMP
    SWAPF   STATUS,W
    CLRF    STATUS
    MOVWF   STATUS_TEMP
    MOVF    PCLATH, W
```

```
    MOVWF   PCLATH_TEMP
    CLRF    PCLATH
```

und wie folgt enden:

```
    MOVF    PCLATH_TEMP, W
    MOVWF   PCLATH
    SWAPF   STATUS_TEMP,W
    MOVWF   STATUS
    SWAPF   W_TEMP,F
    SWAPF   W_TEMP,W
    retfie
```

Die Analog-Falle (Port A)

Umsteiger vom kleinen 16F84 zum 18F876 freuen sich der weitgehenden Assemblercode-Kompatibilität. Allerdings ist PortA gemeinerweise nach dem Zuschalten der Betriebsspannung oder nach dem Reset als analoge Eingänge konfiguriert. Man muß also noch die Pins des PortA auf digital umstellen, und schon läuft der 16F86-code es sei denn man ist noch in die Arbeitsspeicherfalle getappt.

```
; 16F876: alle ADC-Eingänge auf digital I/O umschalten
    BSF STATUS, RP0    ; auf Bank 1 umschalten
    BSF ADCON1, PCFG3  ; PCFG3=1
    BSF ADCON1, PCFG2  ; PCFG2=1
    BSF ADCON1, PCFG1  ; PCFG1=1
    BSF ADCON1, PCFG0  ; PCFG0=1
    BCF STATUS, RP0    ; auf Bank 0 zurückschalten
```

Die Analog-Falle betrifft übrigens auch den PIC16F628. Hier liegen die Komperatoreingänge nach Reset an den Pins, die eigentlich dem Port A zugedacht sind.

```
; 16F628 alle Comparatoreingänge auf Digital umschalten
; alles in der Bank 0
    BSF CMCON, CM0
    BSF CMCON, CM1
    BSF CMCON, CM2
```

Die Arbeitsspeicher-Falle

Noch eine kleine Fußangel für Umsteiger vom 16F84 zum 16F876:
Wird ein bewährtes 16F84 Assembler-Programm einfach für den 16F876 benutzt, so kommt es vor, das es fast funktioniert. Dieses "fast" kann mit den verschobenen Adressen der frei verfügbaren Speicher-Zellen zusammenhängen. Beim 16F84 beginnt der vom Programmierer nutzbare Bereich bei 0x0C. Der große Bruder hat hier aber noch interne Register. Erst ab 0x20 darf der Nutzer sich austoben.

Die Daten-Tabellen-Falle

Oft werden Konstanten in einer Tabelle im Programmspeicher abgelegt. Das folgende Beispiel ist ein Ausschnitt aus einem Programm, in dem ein PIC16F84 eine 7-Segment LED-Anzeige ansteuert. Welche Leuchtsegmente zur Darstellung der Ziffern 0 .. 9 eingeschaltet werden müssen, ist in 10 Programmzeilen mit RETLW-Befehlen festgelegt.

Um z.B. die Ziffer 4 darzustellen, wird "w" mit dem Wert "4" geladen, und dann die Zeile "addwf PCL,f" mit einem Call-Befehl "call Segmente" angesprungen. Der addwf-Befehl erhöht den

Programmcounter zusätzlich um den Wert 4 und bewirkt dadurch einen Sprung in die Zeile "retlw B'11010100' ; 4". Hier wird der Wert B'11010100' nach "w" geladen und ein Return ausgelöst, der zur rufenden Routine zurückführt.

```
     ; 7-Segment-Tabelle
Segmente
      addwf PCL, f
      retlw B'00011000' ; 0
      retlw B'11011110' ; 1
      retlw B'00110010' ; 2
      retlw B'01010010' ; 3
      retlw B'11010100' ; 4
      retlw B'01010001' ; 5
      retlw B'00010001' ; 6
      retlw B'11011010' ; 7
      retlw B'00010000' ; 8
      retlw B'01010000' ; 9
```

Das ist trickreich und funktioniert meistens.

Bei der Verwendung socher Tabellen muß man aber darauf achten, daß der addwf-Befehl nur 8-bittig arbeitet, der Programcounter aber viel länger ist. Innerhalb der Tabelle, darf keine Befehlsadresse der Form xxxFF auftauchen. Der addwf-Befehl kann solche Adressen nicht überspringen, sondern springt stattdessen zu einer Adresse 256 Byte vor dem gewünschten Ziel. Die Adressen muß man deshalb im *.lst-file prüfen und evtl. durch "org"-Befehle und Sprünge manipulieren.

Die I2C-Falle

Die 'großen' PIC16F87x verfügen über eine umfangreiche Hardware zur seriellen Kommunikation. Dabei ist auch eine I2C-Schnittstelle.

Da die beiden Pins des PIC, die das I2C-Port darstellen keine internen Hochziehwiderstände besitzen, muß man diese extern anschließen. Ansonsten interpretiert der PIC den externen Low-Pegel als aktiven I2C-Bus, den gerade ein anderer I2C-Master benutzt. Solange jemand anders auf dem Bus zu senden scheint, übernimmt der PIC aber nicht die Kontrolle über den Bus.

Die externen Hochziehwiderstände präsentieren dem PIC einen sauberen High-Pegel, und der I2C-Master übernimmt dann gern die Buskontrolle

PIC-Projekte

Auf dieser Seite Stelle ich einige meiner PIC-Projekte vor. Einige sind schon abgeschlossen, einige sind noch in der Entwicklung.

RC-5-Adapter	Wandlung von Phillips Fernbediencodes für Marantz-Slim-Line
MD-Beschrifter	"Beschriftung" von MDs im Marantz-MD-Recorder vom PC aus
PIC-Brenner1	der klassische Parallelportbrenner für 16F84 (veraltet und deshalb vom Brenner3 abgelöst)
PIC-Brenner3	der Parallelportbrenner für PIC16F84, PIC16F870/871/872/873/874/876/877, PIC16F7x mit Windows95/98/NT-Software
PIC-Brenner5	verbesserter Brenner3, auch für sehr schnelle PCs geeignet
PIC-Brenner0	der billiger Parallelportbrenner für PIC16F84, PIC16F8xx, PIC16F7x (nur für Geizige)
PIC-Brenner2	PIC-Brenner für 16F84, 16F876 und 16F873 mit Windows95/98/NT-Software
Bus628	Grundgerüstzum Bau von Geräten mit PIC16F628/84
ARINC-Interface (in Entwicklung)	Empfänger für das serielle ARINC-429-Protokoll (Datenverbindung in Flugzeugen) mit Datenausgabe über RS232
Modell-RC-Gameport-Adapter1	Adapter mit dem man eine Modellflugzeug-Fernsteuerung an den Gameport eines PC anschließen kann
Modell-RC-Gameport-Adapter3	Billigadapter mit dem man eine Modellflugzeug-Fernsteuerung an den Gameport eines Windows-PC anschließen kann (in Arbeit)
Leitungs-Zuordner	findet heraus, welche Leitung vom Ende eines Kabels zu welcher Leitung an seinem Anfang gehört
RC-5 Fernbedienungstester	Zeigt den Code an, den eine IR-Fernbedienung aussendet (nur RC-5 wie z.B. Philips)
Frequenzzähler 4 Hz ... 50 MHz	ein 8-stelliger Frequenzzähler mit PIC16F84 und LCD-Anzeige
Frequenzzähler 50 MHz ... 1 GHz	ein 10-stelliger Frequenzzähler mit PIC16F84, externem Vorteiler und LCD-Anzeige (in Arbeit)
Frequenzanzeige für UKW-Empfänger	eine 9-stellige Frequenzanzeige für UKW-Empfänger mit einer ZF von 10,7 MHz
Frequenzgenerator 0 .. 20 MHz	Ein Frequenzgenerator mit den DDS Schaltkreis AD7008
Logger	Datenaufzeichnungim Flash-Speicher des 16F876

PIC-Programmiergeräte

Hat man sein PIC-Programm fertig entwickelt und mit MPLAB in ein HEX-File umgewandelt, muß dieser Code noch in den PIC "hineingebrannt" werden. Dazu dient ein PIC-Programmiergerät - ein Brenner.

Alle Programmiergeräte funktionieren nach dem Prinzip der ICSP. Dabei wird der PIC mit nur 5 Leitungen an den Brenner angeschlossen und über diese die zu Brennenden Daten in den PIC geschrieben:

- Betriebsspannung +5V
- Masse
- Programmierspannung +12V
- Taktleitung
- serielle Datenleitung

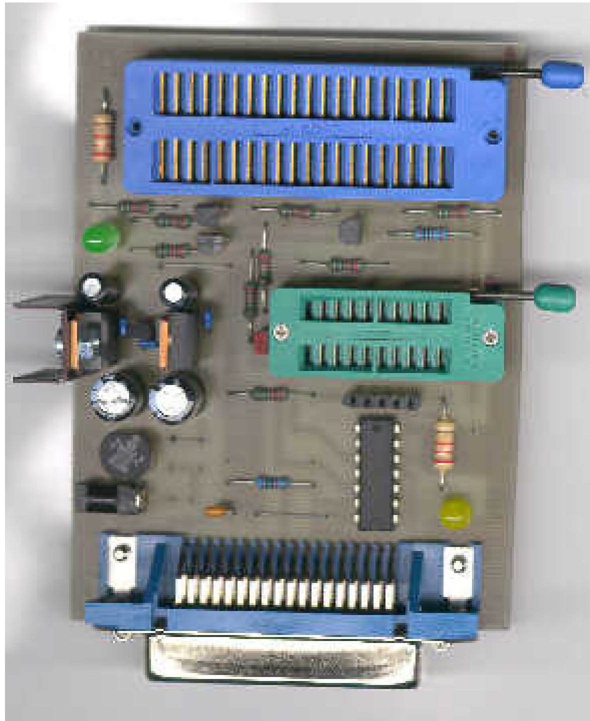
Es gibt industielle Brenner wie den PICstart von Microchip, aber auch Eigenbaulösungen. Einige Links verweisen auf Beispiele für Brenner, die am Druckerport oder am Seriellen Port des PC angeschlossen werden.

Falls jemand einen Brenner bauen möchte aber noch unentschlossen ist, welcher Brenner der richtige ist, empfehle ich meinen **Brenner3 oder den Brenner5.**

Hier stelle ich alle meine Lösungen vor
- den klassischen Brenner1 für den Druckerport und seine verbesserten Nachfolger **Brenner3 und Brenner5**
- der Billig-Brenner0 für den Druckerport
- den Brenner2 für den Serialport

Brenner5

- PIC16F84(A), PIC16F87x(A), PIC16F7x, PIC16F627/628, (PIC12F629/675 über Adapter)
- Anschluß an Druckerport des PC



Der Brenner5 ist der Nachfolger des Brenner3, und funktioniert genau wie dieser. Kleine Modifikationen tragen neuen Erfordernissen Rechnung:

- verbesserte Eignung an schnellen PCs
- verbesserte Stromversorgung
- verbesserte ICSP-Tauglichkeit

Brenner5 läuft zwar notfalls mit der alten Tait-DOS-Software (nur PIC16F84) ist aber eigentlich für meine Windowssoftware PBrenner (alle PIC16F8xx-, PIC16F62x und PIC16F7x-Typen) gedacht.

Hier ist die Beschreibung.

Brenner3

- PIC16F84(A), PIC16F627/628, PIC16F87x(A), PIC16F7x, (PIC12F629/675 über Adapter)
- Anschluß an Druckerport des PC

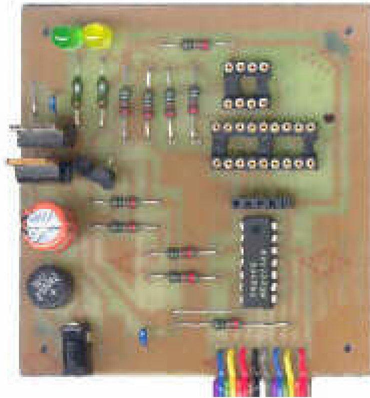
Der Brenner3 ist der legitime Nachfolger des Brenner1, und funktioniert genau wie dieser. Es ist lediglich die unnütze 8-polige Fassung gegen einen großen Sockel für die PIC16F87x und PIC16F7x ausgetauscht worden.

Brenner3 läuft zwar notfalls mit der alten Tait-DOS-Software (nur PIC16F84) ist aber eigentlich für meine Windowssoftware PBrenner (alle PIC16F8xx-, PIC16F62x und PIC16F7x-Typen) gedacht.

Hier ist die Beschreibung.

Brenner1 (veraltet)

- nur PIC16F84(A), PIC16F627/628, PIC12F629/675
- Anschluß an Druckerport des PC



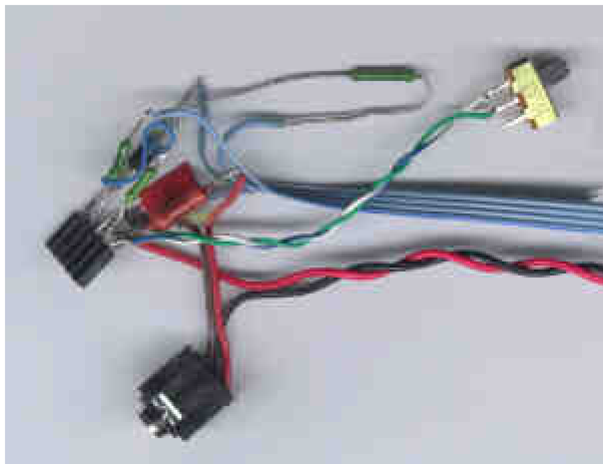
Der gute alte Parallelportbrenner nach dem David-Tait-Verfahren ist schon ein Standard für die 16F84-Programmierung.
Wer sich auf diese "kleinen" PICs beschränken möchte und bereit ist, mit dem Brennprogramm in der DOS-Box zu arbeiten findet **hier die Beschreibung**.

Ein passendes Windowsprogramm gibt es bei mir zum Download. Dieses Programm kann auch die größeren Flash-PICs brennen, die aber nicht in die Fassungen des Brenner1 passen. Deswegen entwickelte ich eine neue Platine Brenner3, die den Brenner1 ablöst, und neben dem 16F84/627/628 auch alle 16F87x und PIC16F7x akzeptiert.

Die 8-polige Fassung dient dem Brennen 8-pin Flash-PICs 12F6xx. Die alten 12Cxxx.unterstützt meiner Brenner dagegen nicht!

Brenner0

- PIC16F84(A), PIC16F627/628, PIC16F87x(A), PIC16F7x, PIC12F629/675
- Anschluß an Druckerport des PC



Der Brenner0 ist ein Billigbrenner für den Parallelport, der einen erweiterten "Quick and Dirty-Brenner" nach Tait darstellt. Er verzichtet auf Treiber oder Schalttransistoren, so daß sein Preis nur von der PIC-Fassung und vom Druckerportstecker bestimmt wird.

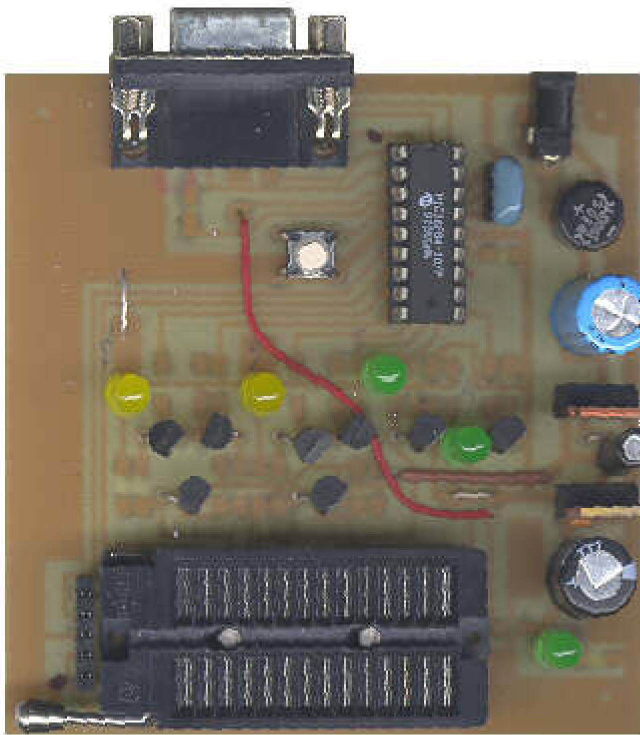
Der Brenner0 ist weniger für jemanden gedacht, der sich intensiv mit PICs beschäftigen möchte, sondern eher für jemanden, der mal einen PIC brennen muß, und für den es sich deshalb nicht lohnt einen richtigen Brenner zu bauen.

Brenner0 und Quick and Diry Brenner laufen mit der Tait-Software (pp.exe) im DOS-Fenster oder mit meiner Windowssoftware PBrenner, die ursprünglich für den Brenner1 geschrieben wurde.

Hier befindet sich die Beschreibung.

Brenner2

- Typen: PIC16F84(A), PIC16F876, PIC16F73
- Anschluß an Serialport des PC



Ich brauchte dringend einen Brenner für 16F876. Gleichzeitig ärgerte ich mich darüber, das die Software für 16F84-Parallelportbrenner nur in der DOS-Box lief.
Das Resultat ist ein Brenner für 16F84, 16F876 und 16F873, der an das serielle Port des PCs angeschlossen und mit einem Windows-Programm bedient wird.
Der Brenner2 ist nicht gerade minimalistisch. Ein paar Bauelemente ließen sich problemlos einsparen (T4, T7, R13, R15, R20, R22, D5, R25) oder durch Brücken ersetzen (D1 .. D4).

Der Nullkraftsockel nimmt alle drei möglichen PIC-Typen auf. Für ICSP (das Programmieren eines in eine Schaltung bereits eingelöteten PIC) steht neben der Fassung eine 5-polige Buchsenleiste zur Verfügung.

Der PC sendet Kommandos über die RS232-Schnittstelle, die der Steuer-PIC auf dem Brenner2 empfängt. Dieser steuert dann den eigentlichen Programmiervorgang, wozu er sich aber noch einer Reihe Transistoren, und Widerstände (SMD-Bauformen auf der Platinenunterseite) bedient. LEDs zeigen an, ob der PIC mit Betriebs- (grün) und Programmierspannung (gelb) versorgt wird.

In der aktuellen Version ist der Brenner nur unwesentlich langsamer als ein Parallelportbrenner. Ein 16F84 wird in 15 Sekunden vollständig programmiert
Der Brenner läuft unter Win95, Win98 und unter NT.

Der Brenner kann selbständig erkennen ob ein 16F84 oder ein 16F87x in die Fassung eingesetzt ist. Alle üblichen Konfigurationsdetails (Oszillatortyp, Codeprotection...) können manuell eingestellt oder aus dem HEX-File eingelesen werden.

Die zugehörigen Files liegen hier.

Hinweis:
Der Nachbau des Brenners2 ist vergleichsweise kompliziert. Falls man auch einen Parallelportbrenner akzeptieren kann, empfehle ich den leistungsfähigeren Brenner3 anstelle des Brenners2.

PBrenner V2.3
Die Software PBrenner unterstützt ab der Version V.2.3 auch den Brenner2. Allerdings ist diese Unterstützung noch in der Erprobungsphase. Das Brennen von 16F84, 16F84A, 16F87x

und 16F62x funktioniert recht gut. Das Brennen von 16F7x, 16F87xA und 12F6xx ist noch zu langsam und unzuverlässig.

Brenner nach AN589 / FA-Brenner

- PIC16F84(A), PIC16F627/628, PIC16F87x, PIC16F87xA, PIC16F7x, PIC12F629/675 (je nach IC-Fassung)

Microchip hat in der Application-Note AN589 einen eigenen Vorschlag für einen einfachen Parallelportbrenner gemacht. Im Dokument zur AN589 (AN589a.pdf auf der microchip homepage) findet sich neben einer Beschreibung und dem Stromlaufplan auch eine Software zur Progranmmierung des PIC16C84.

Da die Zeitschrift Funkamateug einen AN589-kompatiblen Brenner veröffentlicht hat, habe ich meine PBrennersoftware auch um die AN589-Ansteuerung erweitert. Damit ist nun eine große Palette von PICs brennbar.

Da ich selber nie einen AN589-Brenner aufgebaut habe, kann ich kein Platinenlayout anbieten.

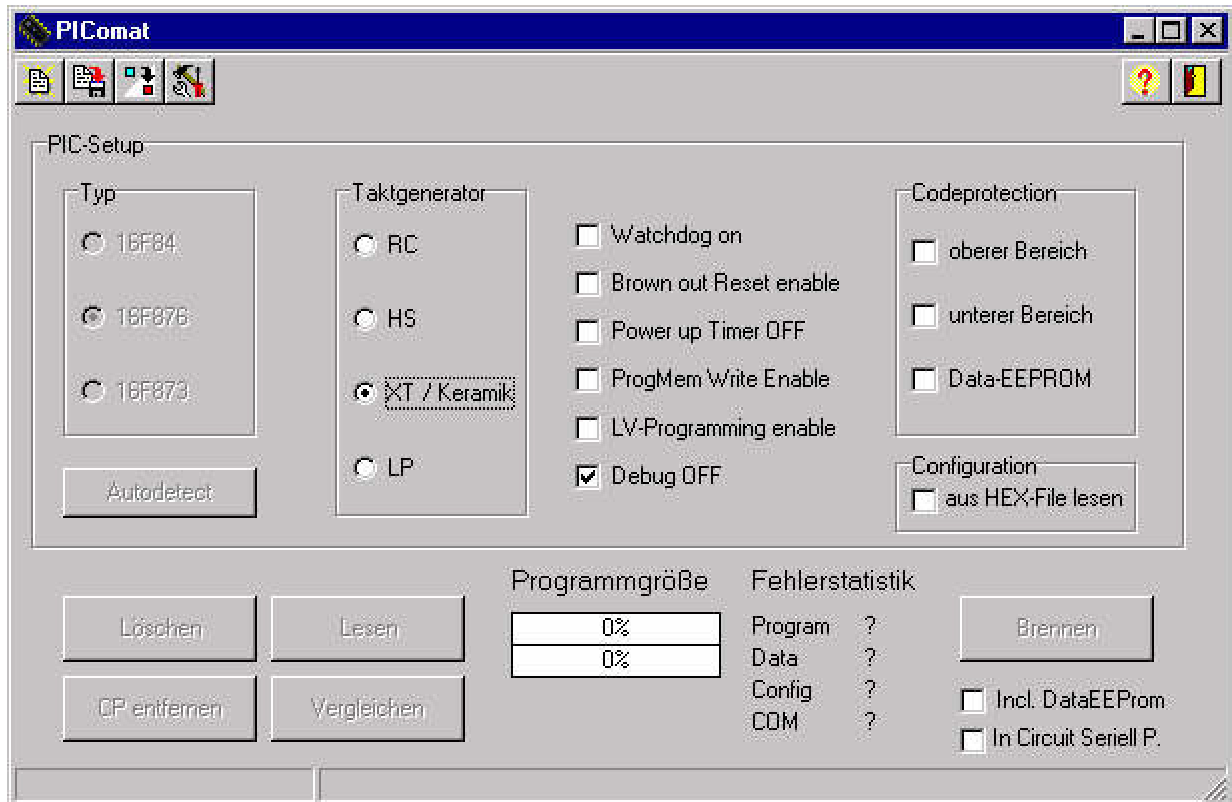
Hinweis:

- In die nebenstehende Schaltung sollte ein 1nF-Kondensator zwischen Pin 17 des 74LS244 und Masse eingefügt werden.
- Die in der Entwicklung befindlichen 8-Pin-Flash-PICs (PIC12F6xx) wird der AN589-Brenner vielleicht nicht brennen können.
  - Anschluß an Druckerport des PC

Brenner-Software

Was nutzt der schönste Brenner, wenn seine Bedienung über kryptische DOS-Kommandozeilen-Programme erfolgt?

Für alle meine Brennertypen liegt ein komfortables Windows-Steuerprogramm zum Download bereit.



PBrenner für

- Brenner3
- Brenner 5
- Brenner1
- Brenner0
- Q&D-Brenner
- AN589/FA-Brenner
- (Brenner2 im Experimentalstadium)

PIComat (Brenner52) für

- Brenner2

Beide Programme ähneln sich in Optik und Bedienung. PBrenner wird im Gegensatz zu PIComat kontinuierlich weiterentwickelt, und unterstützt deutlich mehr PIC-Typen.

ICSP - In Circuit Serial Programming

- das Brennen des PIC in der fertigen Schaltung
- der Anschluß von PICs an Brenner ohne passende Fassung
-

ICSP - Allgemeines

Wer seine PICs im Brenner programmieren möchte, um sie dann aus der Brennerfassung zu nehmen und in die Anwendungsschaltung einzusetzen, der braucht hier nicht weiterzulesen - es sei denn der Brenner hat keine passende Fassung für diesen PIC.

Das Prinzip des Brennens im separaten Brenner stößt spätestens bei PICs im SMD-Gehäuse an seine Grenzen. Hier gibt es nämlich keine geeigneten Schaltkreisfassungen. Alle meine Brenner besitzen aber eine 5-polige Buchsenleiste, die mit "ICSP" beschriftet ist. Die bietet eine Lösung: das Brennen des PIC in seiner Anwendungsschaltung

Außerdem lassen sich alle möglichen zusätzlichen IC-Sockel über den ICSP-Steckverbinder an den Brenner anschließen, und damit dann auch z.B. PICs im DIL-8 oder PLCC-44-Gehäuse brennen.

Wie funktioniert ICSP?

Auch wenn der PIC beim Brennen mit allen Pins in der Fassung des Brenners steckt, elektrisch sind mit dem Brenner nur 5 Pins verbunden. Das ist möglich, da der PIC mit hilfe einer seriellen Datenübertragung programmiert wird - dem In Circuit Serial Programming (ICSP).

Dazu benötigt man:

1. eine Leitung für die +12V-Programmierspannung
2. eine Leitung für die +5V-Betriebsspannung
3. eine Masseleitung
4. eine Datenleitung
5. eine Taktleitung

Diese 5 Leitungen des Brenners werden an folgende Pins des PIC angeschlossen:

Nr.	Leitung des Brenners	Pin des PIC
1	Leitung für die +12V-Programmierspannung	MCLR/Vpp (der Reset-Anschluß)
2	Leitung für die +5V-Betriebsspannung	Vdd
3	Masseleitung	Vss
4	Datenleitung	RB7
5	Taktleitung	RB6

Um in den Programmiermodus zu gelangen, wird zunächst die 5V-Betreibsspannung eingeschaltet und die Pins MCLR, RB6 und RB7 mit Masse verbunden. Dann wird MCLR schnell von Masse auf die Programmierspannung von 12 V gezogen. Dabei müssen RB6 und RB7 noch auf Masse gehalten werden.

Danach kann der Brenner den PIC über die Pins RB6/RB7 auslesen und neu programmieren.

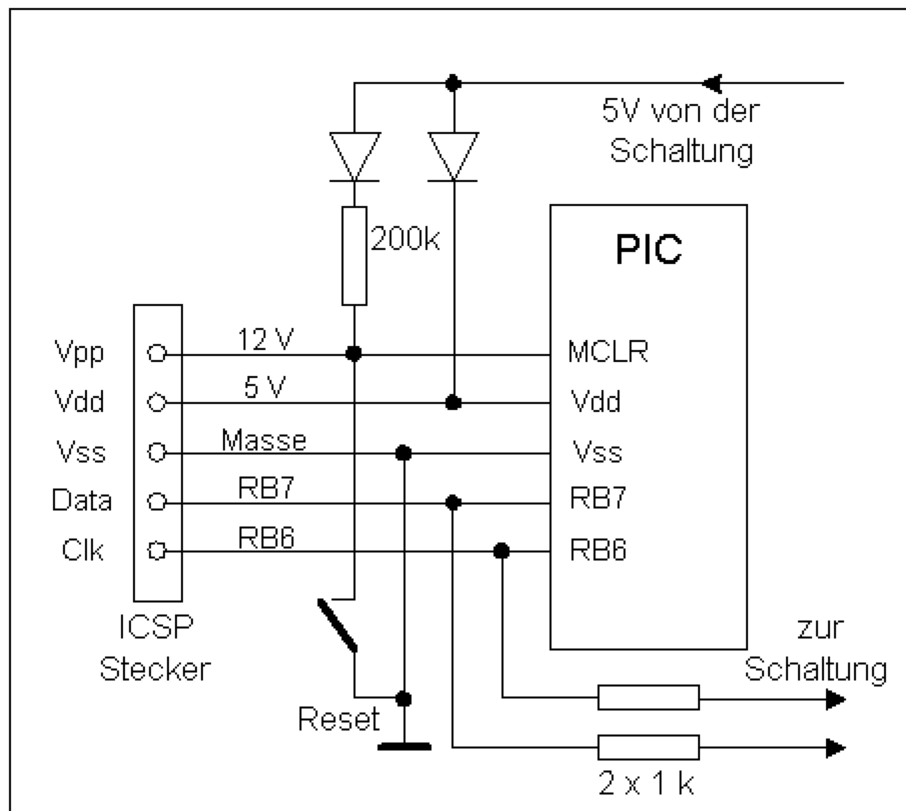
An der ICPS-Buchse stehen also alle Signale zur Verfügung, um einen PIC zum Programmieren an den Brenner anzuschließen. Ein Beispiel für die Nutzung der ICPS-Buchse sind Adapter, mit dem eine **zusätzliche Schaltkreis-Fassung an einen beliebigen Brenner** angeschlossen werden kann, wenn dieser über die ICPS-Buchse verfügt.

Die Hauptanwendung der ICPS-Verbindung ist aber eine andere: **Das Brennen eines PIC, der bereits in seine Anwendungsschaltung eingebaut ist.**

Dazu verfügt die Leiterplatte der Anwendungsschaltung des PIC auch über eine ICPS-Buchse. Brenner und Anwenderschaltung werden über ein 5-poliges Kabel miteinander verbunden, und der PIC wird "zuhause" gebrannt. Das lästige umstecken des PIC zwischen Anwendungsschaltung und Brenner entfällt, und das komfortable Brennen von PICs im SMD-Gehäuse wird überhaupt erst möglich.

Was ist beim Entwurf ICSP-tauglicher Schaltungen zu beachten?

Die 5 Pins, die zum ICSP an den Brenner angeschlossen werden müssen dienen ja nicht exclusiv zum Brennen, sie werden meist auch in der Anwendungsschaltung verwendet. Um zu verhindern, das sich Brenner und Anwendungsschaltung gegenseitig in ihrer Funktion stören, sind einige Dinge beim Entwurf der Anwendungsschaltung zu beachten:



Programmierspannung MCLR/Vpp
Dieser Anschluß ist am schwierigsten.
Meine Brenner1/3 ziehen dieses Pin über 10 kOhm auf Masse, oder mit einem Transistor (+ 1kOhm) auf +12V.
In der Anwenderschaltung wird diese Pin mit einem Hochziehwiderstand auf 5V gehalten, oder mit einem Resettaster kurzfristig auf Masse gelegt. Das der ResetTaster beim Brennen keinesfalls gedrückt werden darf ist damit klar!!

Schwierig ist aber auch der Hochziehwiderstand. Beim Brennen trennt nur

dieser Widerstand die 5-V-Versorgung der Anwendungsschaltung von den 12V des Brenners. Hier kann deshalb eine Diode vor Schäden durch Überspannung schützen. Der 5V-Hochziehwiderstand muß deutlich größer sein (mindestens 20 X) als der 10kOhm Widerstand, der im Brenner das MCLR-Pin mit Masse verbindet. Ansonsten kann der Brenner MCLR nicht sauber auf Masse ziehen. Deshalb sollte ein wenigsten 200 kOhm großer Widerstand zwischen dem Reset/MCLR-Pin und +5V eingesetzt werden.

Betriebsspannung Vdd
Beim Brennen speist der Brenner den PIC mit der nötigen Betriebsspannung. Ist der PIC der einzige Spannungsverbraucher in der Anwenderschaltung, kann die +5V-Leitung des Brenners direkt mit dem Vdd-Pin des PIC verbunden werden. Vor dem Anschluß des Brenners muß dann unbedingt die normale Betriebsspannung des PIC abgeschaltet werden.

Sind neben dem PIC noch andere Bauelemente mit der 5-V-Versorgung der Anwendungsschaltung verbunden, würde der Brenner bei Brennen die gesamte Anwendungsschaltung in betrieb nehmen. Bei größeren Schaltungen könnte das den Brenner überlasten. Eine Entkopplung mit Dioden oder ein Jumper in der +5V-Leitung trennen dann besser die beiden potentiellen 5-V-Quellen.

Masseverbindung Vss
Das ist die einzige unkritische Verbindung. Normalerweise wird die Masse des Brenners direkt mit der Masse des PIC und damit auch mit der Masse der Anwenderschaltung verbunden.

Takt- und Datenleitung RB6 und RB7
Wer in der Anwendungsschaltung auf diese beiden Pins verzichten kann, sollte sie exclusiv der ICSP-Schnittstelle zur Verfügung stellen. Werden die beiden Pins aber benötigt, sollten sie mit der ICSP-Buchse direkt, aber mit dem Rest der Schaltung über Widerstände von wenigstens 1 kOhm verbunden werden. Ist so ein 1 kOhm Widerstand für die Applikationsschaltung zu groß, helfen nur noch Jumper, die vor dem Brennen geöffnet werden müssen, um den PIC von der restlichen Schaltung zu trennen.

Das ICSP-Kabel

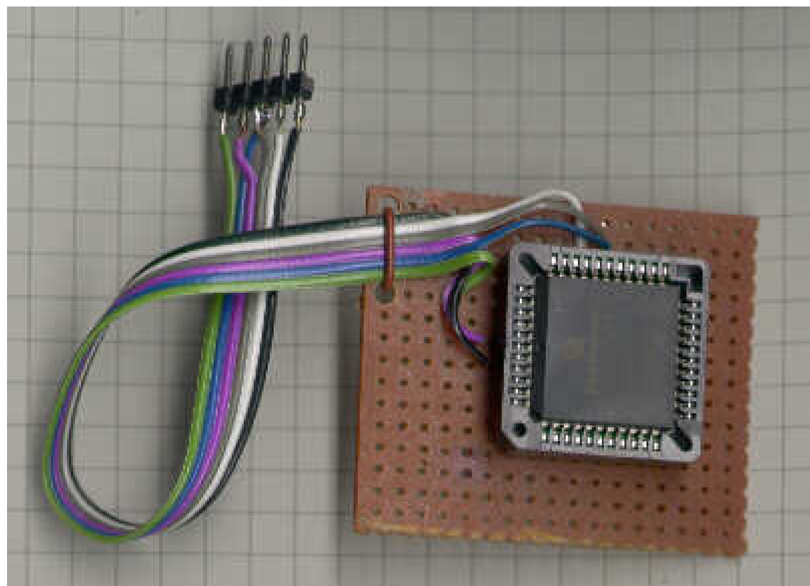
Ist es wirklich nötig, über ein einfaches Kabel Worte zu verlieren? **JA ES IST NÖTIG**.
In der Belegung des Kabels gibt es eine Schwachstelle. Die für störende Einsteuungen sehr empfängliche **CLK**-Leitung muß dringend von den anderen Leitungen abgeschirmt werden. Dazu ist nun keine komplette Schirmung nötig, aber eine separate Masseleitung zwischen **CLK** und **DATA** ist wenigstens erforderlich. Aus diesem Grunde verwende ich in letzter Zeit stets 6-poliges Hosenträgerkabel mit 2 Masseleitungen: eine zwischen **Vdd** und **DATA** und eine weitere zwischen **DATA** und **CLK**. Das ist in den untenstehenden Stromlaufplänen deutlich zu sehen.

Natürlich gibt es auch andere Lösungen, um ein Übersprechen auf die **CLK**-Leitung zu vermeiden, z.B. kann man die **CLK**-Leitung vom restlichen Kabel getrennt verlegen, wie es im Foto vom DIL-40-Adapter zu sehen ist. Man kann auf die Masseleitung zwischen **Vdd** und **DATA** auch verzichten, ihre Funktion erfüllt ja auch die Masse zwischen **DATA** und **CLK**. Wer High-endig baut, verwendet vielleicht ein 10-poliges Flachbandkabel, in dem jede zweite Leitung auf Masse liegt .....
Da will ich keine weiteren Vorschriften machen, hauptsache **CLK** ist vor Einstreuungen geschützt.

Wer diese einfache Regel mißachtet, wird feststellen, daß schon die Autodetect-Funktion meiner Brennersoftware nicht funktioniert. Vom Brennen ganz zu schweigen.

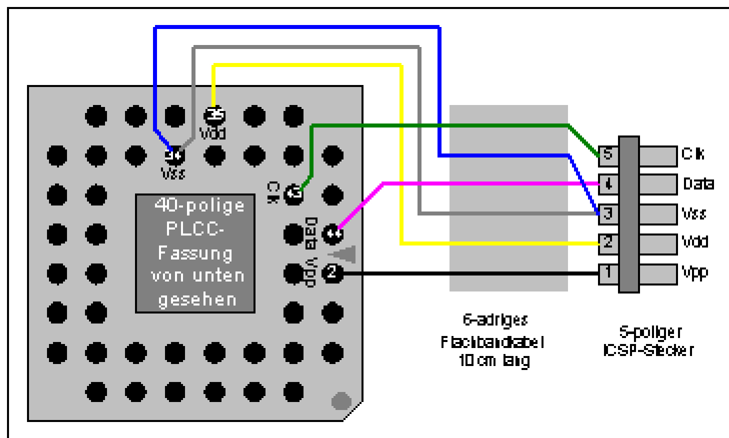
Wie lang darf ein ICSP-Kabel eigentlich sein?
Es sollte so lang wie nötig und so kurz wie möglich sein. Wer nur einen zusätzlichen Sockel adaptieren will, kommt mit 10 cm aus. Für ICSP sollten 20 cm auch genügen. Wenn CLK ordentlich geschirmt ist, sollte aber auch 1/2 Meter kein Problem sein.

ICSP-Adapter für PICs im PLCC-44-Gehäuse



Keiner meiner Brenner hat einen PLCC-44-Sockel, um PIC16F871, PIC16F84(A), PIC16F877(A), PIC16F74 oder PIC16F77 im quadratischen PLCC-44-Gehäuse zu brennen. All diese Chips lassen sich aber über einen einfachen Adapter an den 5-poligen ICSP-Steckverbinder der Brenner1/2/3/5 anschließen, und dann brennen.

.



Der nebenstehende Stromlaufplan stellt die Verbindung zwischen dem 5-poligen ICSP-Stecker und dem PLCC-44-Sockel dar. Der Sockel ist so dargestellt, daß man von unten auf die Lötpins schaut.

Wichtig ist die zusätzliche Masseleitung (blau) im Flachbandkabel. Sie schirmt die CLK-Leitung gegen Einstreuungen von den anderen Leitungen (insbesondere DATA) ab.

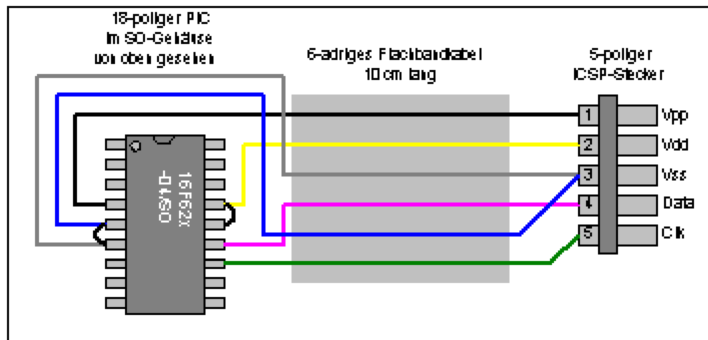
ICSP-Adapter für PICs im SO-18-Gehäuse

Für SMD-Gehäuse wie SO-18 (PIC16F84(A), PIC16F62x) gibt es keine Fassungen, die man auf eine Brennerplatine löten könnte. Auch ein richtiger Adapter läßt sich nicht bauen, aber man kann den Schaltkreis direkt an ein ICSP-Kabel löten.

All diese Chips lassen sich damit an den 5-poligen ICSP-Steckverbinder der Brenner1/2/3/5 anschließen, und dann brennen. Eleganter wäre es hier natürlich, den PIC in seine Schaltung einzulöten, und dort via ICSP zu brennen.

Die Ausführung im

nebenstehenden Foto und die einzelnen Leitungsfarben entsprechen **nicht** den Farben im untenstehenden Stromlaufplan.

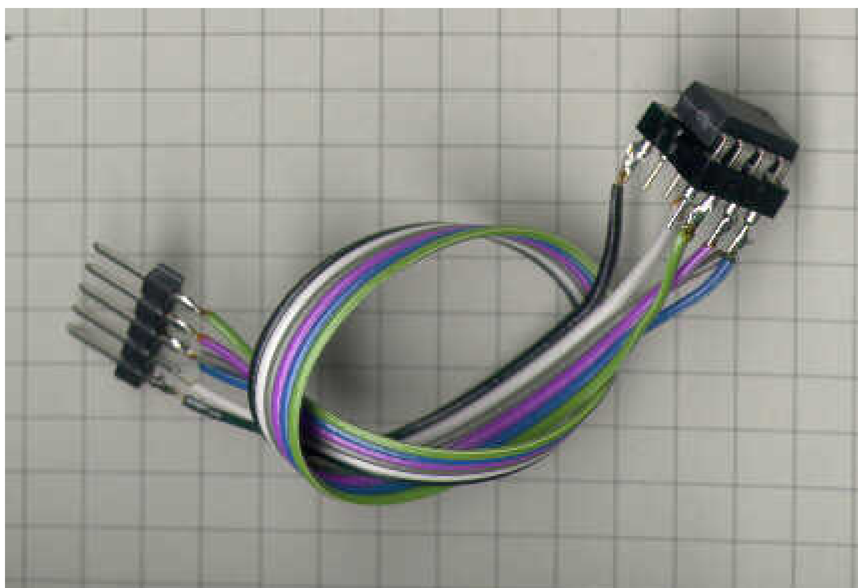


Der nebenstehende Stromlaufplan stellt die Verbindung zwischen dem 5-poligen ICSP-Stecker und dem SO-18-Gehäuse da. Der PIC ist in der Draufsicht dargestellt.

Beim 20-poligen SSOP-Gehäuse, muß die Beschaltung entsprechend abgeändert werden.

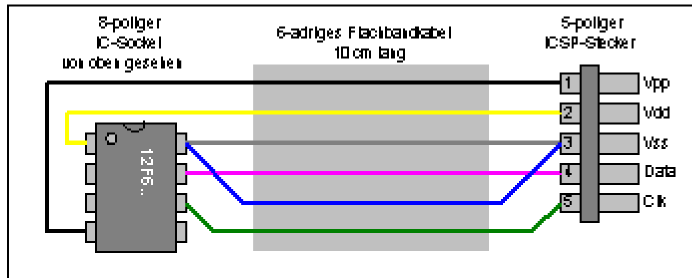
Wichtig ist die zusätzliche Masseleitung (blau) im Flachbandkabel. Sie schirmt die CLK-Leitung gegen Einstreuungen von den anderen Leitungen (insbesondere DATA) ab.

ICSP-Adapter für PICs im DIL-8-Gehäuse



Einen DIL-8-Sockel, um PIC12F629 und PIC12F675 im niedlichen DIL-8-Gehäuse zu brennen hat von meinen Brennern z.Z. nur der veraltete Brenner1.
An alle anderen Brenner (Brenner2/3/5) lassen sich diese Chips aber über einen einfachen Adapter an den 5-poligen ICSP-Steckverbinder anschließen, und dann brennen.

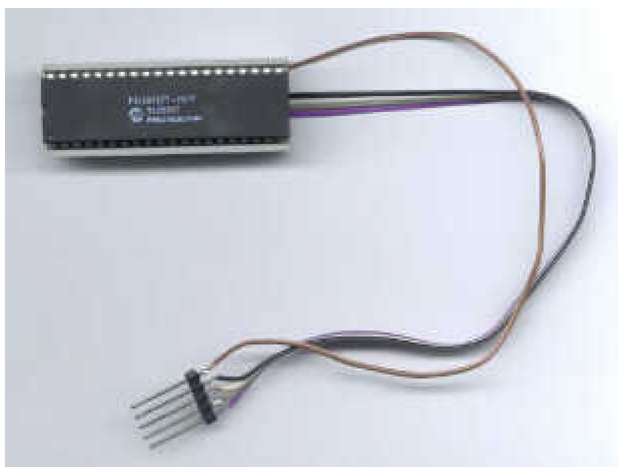
.



Der nebenstehende Stromlaufplan stellt die Verbindung zwischen dem 5-poligen Stecker und dem DIL-8-Sockel dar. Der Sockel ist in der Draufsicht dargestellt.

Wichtig ist die zusätzliche Masseleitung (blau) im Flachbandkabel. Sie schirmt die CLK-Leitung gegen Einstreuungen von den anderen Leitungen (insbesondere DATA) ab.

ICSP-Adapter für PICs im DIL-40-Gehäuse

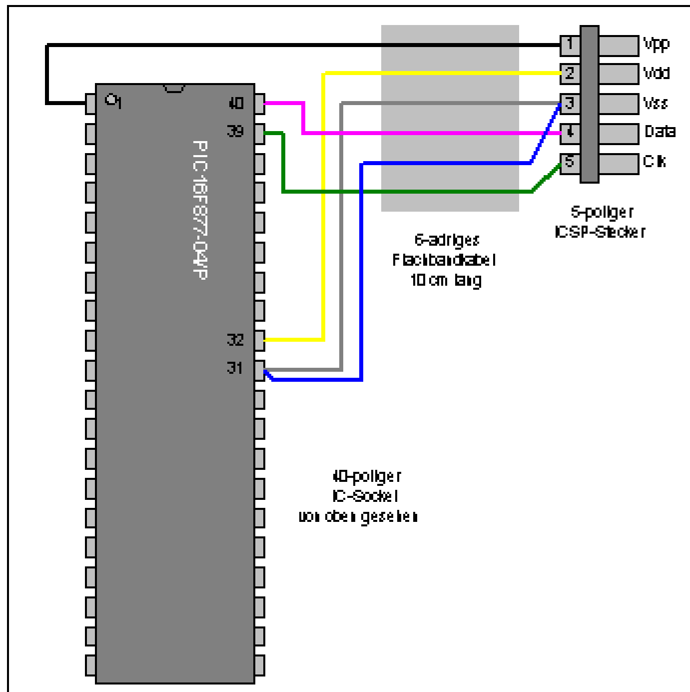


Von meinen Brennern können nur die Brenner3 und Brenner5 mit einer Fassung für die DIL-40-Gehäuse bestückt werden um PIC16F871, PIC16F84(A), PIC16F877(A), PIC16F74 oder PIC16F77 im DIL-40-Gehäuse zu brennen.

Man kann die aber auch mit preiswerten DIL-28-Sockeln bestückt haben. In diesem Fall, oder wenn man einen Brenner1 oder Brenner2 benutzt, lassen sich DIL-40-PICs über einen einfachen Adapter anschließen und brennen.

Das nebenstehende Foto zeigt noch eine alte Version, die nicht in allen Details mit den darunter

stehendem Stromlaufplan exakt übereinstimmt.



Der nebenstehende Stromlaufplan stellt die Verbindung zwischen dem 5-poligen ICSP-Stecker und dem DIL-40-Sockel dar. Der Sockel ist in der Draufsicht dargestellt.

Wichtig ist die zusätzliche Masseleitung (blau) im Flachbandkabel. Sie schirmt die CLK-Leitung gegen Einstreuungen von den anderen Leitungen (insbesondere DATA) ab.

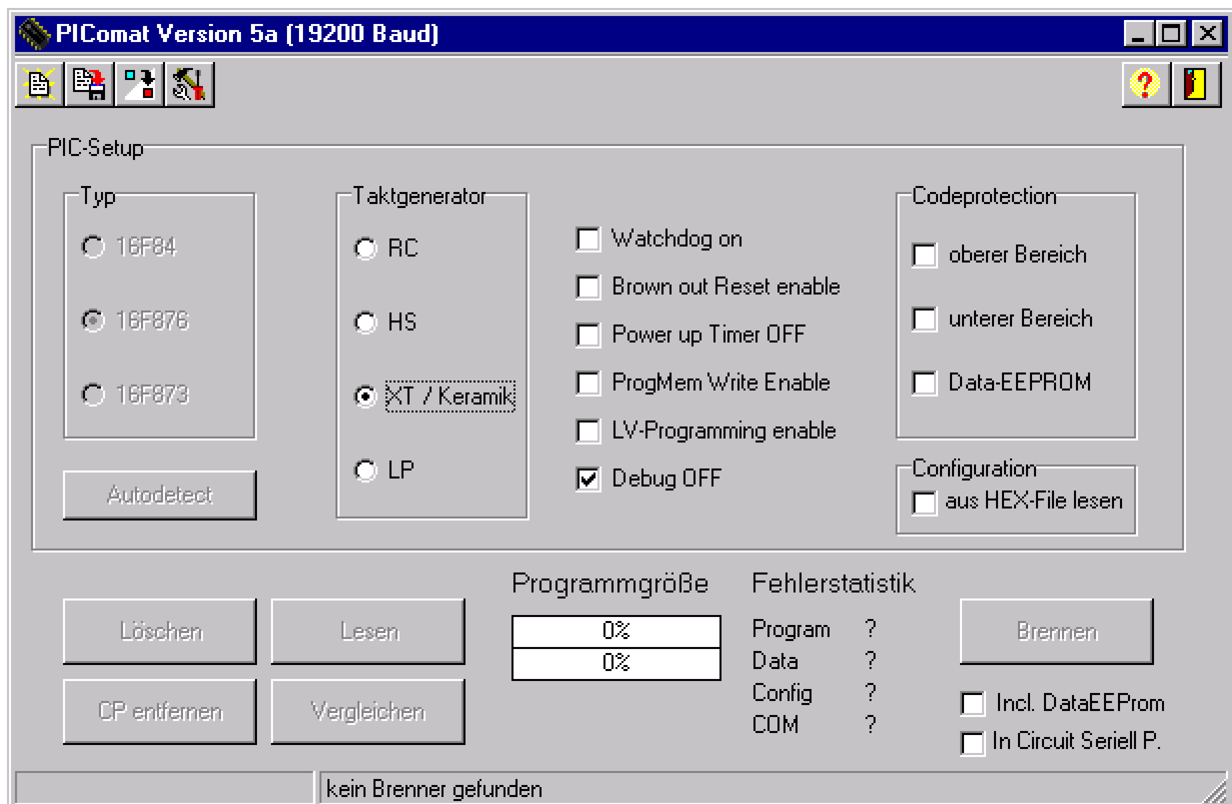
PIC-Prozessoren - Configuration

Einstellungen des Brennprogramms

Was ist die Konfiguration eines PIC?

Beim Brennen eines Programms in einen PIC müssen eine Reihe von Optionen ausgewählt werden, die auf die spätere Funktion des PIC einen großen Einfluß haben. Bei alter DOS-Software (PP.EXE) erfolgt das durch Kommandozeilenparameter, bei meiner Windowssoftware für Brenner2 und Brenner3 durch grafische Auswahlfelder. Alternativ können diese Parameter auch schon im Programmquelltext festgelegt werden. Beim Brennen werden diese Einstellungen im Konfigurationsregister des PIC abgelegt.

Der folgende Screenshot des Steuerprogramms für den Brenner2 (ähnlich Brenner3) zeigt die Konfigurationsmöglichkeiten für einen PIC16F876/873. Einige dieser Optionen entfallen beim PIC16F84:



Taktgenerator	allgemeine Einstellungen	Codeprotection
LP	Watchdog an	oberer Bereich
XT	Brown out Reset enable	unterer Bereich
HS	Power up Timer off	oberer + unterer Bereich
RC	ProgMem Write Enable	Data-EEPROM
ER (PIC16F62x)	LV-Programming enable	
INTRC (PIC16F62x)	Debug off	
EC (PIC16F62x)	RA5=MCLR(PIC16F62x)	
	RA6=CLKOUT	

Welche Taktgeneratoreinstellung soll ich benutzen?

Ein PIC kann mit einer großen Zahl unterschiedlichster Taktquellen in einem weiten Frequenzbereich betrieben werden. Man kann sich leicht vorstellen, daß es unterschiedlicher Einstellungen der Takterzeugungsschaltung des PIC bedarf um mit einem RC-Glied, einem Quarz oder einem Keramikresonator einen Takt zu generieren. Auch ist ein 32 kHz Takt mit einem 20 MHz-Takt nicht zu vergleichen. Deshalb gibt es 4 unterschiedliche Taktgeneratoreinstellungen: **LP**, **XT**, **HS** und **RC.**
Die Wahl der richtigen Einstellung ist sehr wichtig, denn mit einer falschen Einstellung schwingt der Generator vielleicht nicht und der PIC verweigert die Arbeit. Noch dümmer ist es, wenn der Generator nur manchmal schwingt, was rätselhafte Aussetzer hervorruft.

LP, XT und HS sind für den Betrieb mit Quarzen, Keramikresonatoren oder externen Taktquellen vorgesehen. Diese Betriebsarten unterscheiden sich im Frequenzbereich. Die von mir angegebenen Minimalfrequenzen stellen meist keine technischen Begrenzungen dar, sondern kennzeichnen den sinnvollen Bereich. (Beispiel: Der HS-Mode geht auch bei 1 MHz, aus Gründen des Stromverbrauchs sollte aber der XT-Mode benutzt werden.)

LP - Low Power Crystal (ca. 32 kHz - 200 kHz)
Quarz, Keramikresonator oder externe Taktquelle mit geringer Frequenz.
Diese Einstellung ist für den energiesparenden Betrieb bei niedrigem Takt reserviert.
Diese Betriebsart ist für die 10-MHz-Variante PIC16F84-10 (und schnellere Typen) nicht garantiert.

XT - Crystal / Resonator (ca. 100 kHz - 4 MHz)
Quarz, Keramikresonator oder externe Taktquelle mittlerer Frequenz.
Das ist die Standardeinstellung für die meisten Quarze und Keramikresonatoren. Nur bei sehr hohem Takt sollte man auf HS ausweichen, da dieser zu einem deutlich höheren Stromverbrauch führt.

HS - High Speed Crystal / Resonator (ca. 4 MHz - 20 MHz)
Quarz, Keramikresonator oder externe Taktquelle mit sehr hoher Frequenz.
Diese Einstellung ist bei hohem Takt nötig, sie erhöht aber den Stromverbrauch des PIC. So benötigt z.B. ein 16F84-10 im LP/XT-Mode <2mA, im HS-Mode dagegen 10mA.

RC - Resistor / Capacitor (ca. 30 kHz - 4 MHz)
Das ist die Billiglösung. Anstelle eines Quarzes oder eines Keramikresonators (2,-DM) kann man auch einen Widerstand und einen Kondensator verwenden. Je nach verwendeten Bauelementewerten läßt sich ein Takt von einigen 10 kHz bis zu einigen MHz einstellen. Allerdings sollte man keinen genauen oder stabilen Takt erwarten Toleranzen von 25% sind normal. Ich halte das nur für eine Notlösung, insbesondere wenn man die Preise für Keramikresonatoren bedenkt.
++VORSICHT++
Im RC-Mode darf der PIC nicht von einer externen Taktquelle (z.B. separater Quarzoszillator) gespeist werden!! Andernfalls kann es zur Zerstörung des PIC kommen!!

Die folgenden Modes unterstützen nur die PIC16F62x.:

ER - External Resistor (nur PIC16F62x; ca. 10 kHz - 8 MHz / 37 kHz)
Eine Billiglösung, die man verwenden kann, wenn es nicht auf Präzision ankommt. Ein Widerstand von RA7 nach Masse stellt die Frequenz eines internen RC-Oszillators ein. Microchip garantiert den Betrieb mit Widerständen von 38kOhm bis zu 1 MOhm. Empfohlen

wird maximal 4MHz.
Mit dem OSCF-Bit im PCON-Register kann auf einen festen 37 kHz-Takt (unabhängig vom Wert des externen Widerstands) umgeschaltet werden.

INTRC - internal RC-Oscillator (nur PIC16F62x;   4 MHz / 37 kHz)
Es wird ein interner 4MHz-Oszillator verwendet, der leider nicht sehr stabil ist (3,65 ... 4,28 MHz).
Mit dem OSCF-Bit im PCON-Register kann auf einen festen 37 kHz-Takt umgeschaltet werden.

EC - External Clock In (nur PIC16F62x; ca. 0 kHz - 20 MHz)
Wenn ein externer Takt vorhanden ist, kann dieser im EC-Mode dem 16F62x dierekt in Pin OSC1/RA7 eingespeist werden.

Der Watchdog

Auch ein PIC kann abstürzen. Damit das ohne schlimme Folgen bleibt, löst der Watchdog in so einem Fall automatisch ein Reset des PIC aus.

Der Watchdog ist ein kleiner Timer im PIC, der einen eigenen internen RC-Taktgeber besitzt, und somit von der Funktion des PIC unabhängig ist. Dieser Watchdogtimer (WDT) löst, wenn er eingeschaltet ist, alle 18 ms (Toleranzbereich: 7 ms ... 33 ms) ein Reset des PIC aus. Der Reset kann nur verhindert werden, wenn das im PIC laufende Programm den WDT immer wieder zurücksetzt, bevor er einen Reset auslösen kann. Dieses Rücksetzen erfolgt mit dem Befehl clrwdt (oder sleep).

Das ordnungsgemäß laufende Programm muß also so programmiert werden, daß in kurzen Abständen (z.B. alle 5 ms) der Befehl clrwdt ausgeführt wird, dann kommt der WDT nicht dazu ein Reset auszulösen. Stürzt das Programm aber ab, wird der WDT nicht mehr gelöscht und startet den PIC neu.

Falls 18 ms zu kurz sind, kann man den Vorteiler des TIMER1 benutzen und so einen WDT-Zyklus von bis zu 2,3 s einstellen. Dann steht der Vorteiler aber dem TIMER1 nicht mehr zur Verfügung.

Ohne besondere Notwendigkeit sollte man den WDT nicht verwenden, da er die Programmentwicklung kompliziert.

Brown out Reset (nur 16F87x)

Ein kurzer Aussetzer in der Betriebsspannung des PIC kann zum Absturz oder zum Weiterarbeiten mit verfälschten Werten führen. Die "Brown out Reset-Option" überwacht die Betriebsspannung. Falls die Betriebsspannung für 0,1 ms oder länger unter 4 V fällt, wird sofort ein Reset ausgelöst. Ist die Betriebsspannung wieder im sicheren Bereich wartet der PIC noch 72 ms und startet dann neu.

++ACHTUNG++
Wer beabsichtigt, seinen PIC mit einer Betriebsspannung von 4V oder daruinter zu betreiben, darf diese Funktion natürlich nicht aktivieren.

Der Power up Timer

Normalerweise beginnt der PIC beim Zuschalten der Betriebsspannung sofort mit der Abarbeitung seines Programms. Manchmal benötigen andere Teile der elektronischen Schaltung, in die der PIC eingebaut ist, aber etwas Zeit, um ihren Anfangszustand einzunehmen. Deshalb kann der PIC auf Wunsch seine Arbeit mit einer festen Verzögerung von ca. 72 ms nach dem Einschalten der Betriebsspannung aufnehmen.

Diese Option zu aktivieren kann eigentlich nie schaden, aber oft die Stabilität des Systems erhöhen.

ProgMem Write Enable (nur 16F87x)

Im 16F87x können Programme nicht nur auf den Datenspeicher und den Daten-EEPROM sondern auch auf den Programm-Speicher zugreifen. Programmcode kann gelesen und geschrieben werden. Man wird das weniger für die Entwicklung selbstmodifizierenden Codes als vielmehr zum Ablegen großer Datenblöcke benutzen. (Der Hersteller garantiert nur 1000 Schreibzyklen für den Programmspeicher aber 100 000 für den Daten-EEPROM.)

Das Lesen des Programmspeichers ist immer möglich, schreiben kann man aber nur in Speicherbereiche die nicht codeprotected sind.

Die Option "ProgMem Write Enable" muß zusätzlich aktiviert werden, um das Schreiben in den Programmspeicher zu erlauben, ansonsten ist nur das Lesen möglich.

LV-Programming Enable (nur 16F87x)

Alle herkömmlichen PICs benötigen zum Brennen eine 12-V-Spannung am MCLR-Eingang. Die modernen PIC16F876/873 lassen sich alternativ auch mit nur 5 V (LV steht für low voltage) programmieren. Das vereinfacht in der Großproduktion die Umprogrammierung in fertigen Geräten.

Beim Einsatz der 5-V-Programmierung wird allerdings das Pins RB3 (PortB Bit 3) für die Programmierfunktion blockiert, und steht dann nicht mehr für andere I/O-Funktionen zur Verfügung.

Wenn man auf das LV-Programming verzichtet, hat man also ein I/O-Pin mehr zur Verfügung. Deshalb, und weil meine Brenner LV-Programming generell nicht unterstützt, sollte man die Option "LV-Programming Enable" normalerweise nicht auswählen.

Debug Off (nur 16F87x)

Die PIC16F87x-Familie unterstützt einen Debug-Mode, bei dem das Programm des schon in die Schaltung eingesetzten PICs debugt wird. Dazu benötigt man ein In-Circuit-Debuger-Modul, das man bei Microchip erwerben kann. An dieses Modul wird der Pic über das ICSP-

Kabel angeschlossen (In Circuit Serial Programming) und dann vom MPLAB-Programm aus debugt.

Normalerweise wird man diesen Mode nicht nutzen, schon deshalb nicht, weil man das dazu benötigte In-Circuit-Debuger-Modul nicht besitzt. Da der Debug-Mode die beiden Pins RB6 und RB7 blockiert, sollte man normalerweise Debug OFF wählen.

RA5=MCLR (nur 16F62x)

Ber 16F62x vereinigt viele Funktionen in einem kleinen Gehäuse. Da heist es, mit den Pins sparsam umzugehen.
In vielen Schaltungen wird das RESET-Pin (MCLR) gar nicht benötigt. Beim Einschalten wird der PIC durch das Power-on-Reset in den Anfangszustand gesetzt, falls ein Absturz auftritt, kann der Watchdog den PIC wieder zum Leben erwecken. Deshalb kann man das MCLR-Pin wahlweise auch als zusätzliches IO-Pin mit der Bezeichnung RA5 nutzen. Ob man nun lieber ein Reset-Pin (Feld ankreuzen) oder ein IO-Pin (Feld nicht ankreuzen) hätte, entscheidet man durch setzen dieser Option. Die default-Einstellung von PBrenner ist IO-Pin.

RA6=CLKOUT (nur 16F62x)

Ber 16F62x vereinigt viele Funktionen in einem kleinen Gehäuse. Da heist es, mit den Pins sparsam umzugehen.
Der Ausgang des Taktgenerators wird nur benötigt, wenn man einen Quarz, Keramikresonator oder RC-Glied verwendet. Benutzt man dagegen den internen RC-Generator oder den ER-Mode (externer Widerstand), dann ist der Taktausgang meistens überflüssig. Dann ist es möglich, aus dem CLKOUT-Pin ein zusätzliches IO-Pin mit der Bezeichnung RA6 zu machen.
Ob man nun lieber ein CLKOUT-Pin (Feld ankreuzen) oder ein IO-Pin (Feld nicht ankreuzen) hätte, entscheidet man durch setzen dieser Option. Die default-Einstellung von PBrenner ist IO-Pin.

Was ist Codeprotection

Schutz des Programmspeichers
So wie sich ein Programm in einen PIC hineinschreiben läßt, so kann man es auch wieder auslesen. Das ist manchmal unerwünscht, z.B. wenn man viel Zeit und Nerven in die Entwicklung einen guten Programms investiert hat und nun befürchtet, jemand anders könnte das Programm nun einfach klauen.
Deshalb kann man den Programmspeicher eines PIC vor dem Wiederauslesen schützen.

Bei den großen PIC16F876/873 ist der Programmspeicher jeweils in Teile aufgeteilt, für die sich der Leseschutz getrennt aktivieren läßt. Beim PIC16F84 schützt man immer den gesamten Programmspeicher.

geschützte Bereiche	**PIC16F84**	**PIC16F873**	**PIC16F876**	**PIC16F627**	**PIC16F628**
oberer Bereich	nicht möglich	0F00h - 0FFFh	1F00h - 1FFFh	keine Wirkung	0400h - 07FFh
unterer Bereich	nicht möglich	0800h -	1000h -	0200h -	0200h -

		0FFFh	1FFFh	03FFh	07FFh
oberer + unterer Bereich	0000h - 3FFFh	0000h - 0FFFh	0000h - 1FFFh	0000h - 03FFh	0000h - 07FFh

Ein Leseschutz läßt sich mit einem Brennprogramm wieder aufheben, allerdings löscht man dabei sinnvollerweise automatisch den gesamten Programmspeicher. Die Sicherheit des schützenswerten Programms ist also gewährleistet.

Wärend der Programmentwicklung rate ich von Codeprotection ab, da das immer nötige Gesamtlöschen beim Neuprogrammieren den Flash-Speicher des PIC unnötig belastet. Ob jemend sein fertiges Programm schützen will, wenn er programmierte PICs an andere weitergibt ist eine persönliche Entscheidung.

++Anmerkung++
Ist im 16F87x ein Programmspeicherbereich geschützt, so kann das im PIC laufende Programm diesen Bereich zwar noch lesen,aber nicht mehr beschreiben.

Schutz des Daten-EEPROMs (nur 16F87x)
Im PIC gibt es einen EEPROM-Bereich für Daten auf die das Programm des PIC zugreifen kann. Hier lassen sich z.B. Kalibrierwerte oder der letzte Gerätezustand vor dem Ausschalten speichern.

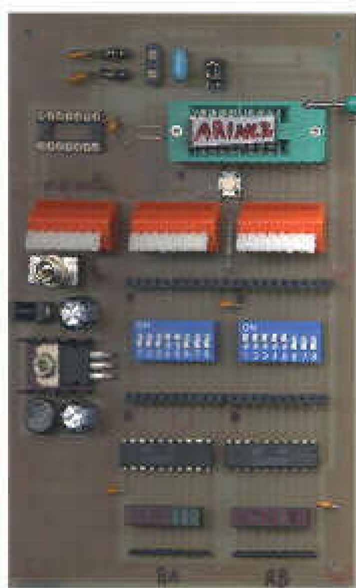
Dieser Bereich kann beim Brennen des PIC, aber normalerweise auch vom PIC-Programm beschrieben werden. Das Schreiben des PIC-Programms in den Daten-EEPROM kann mit dieser Option aber verboten werden.

PIC-Test-Platinen

Eine universelle Testplatine vereinfacht die Entwicklung einer PIC-Applikation sehr. Eine solche Testplatine enthält im günstigsten Fall:
- einen 0-Kraft-Sockel
- Steckerleisten auf die alle PIC-Ein/Ausgänge geführt sind
- eine Takterzeugung für den PIC
- die Stromversorgung
- LEDs für die Anzeige der PIC-Ausgänge
- Schalter für die Eingabe von Werten zum PIC

Jede Komponente kostet Geld, und so entscheidet man sich in der Praxis meist für einen Kompromiß.

16F84 / 16F62x - Testplatine



Meine 16F84/62x-Testplatine ist recht komfortabel. Der IC sitzt in einem 20-poligem Nullkraftsockel, dem ein Kontaktpaar entfernt wurde (18-polige Nullkraftsockel sind kaum zu bekommen).

Beide Ports sind direkt auf Klemmen und Buchsenleiste gelegt. Diese Klemmen waren vorübergehend schwer beschaffbar. Nun sind sie aber bei Reichelt im Katalog.

Über 8-polige DIL-Schalter kann jeweils eine zweite Buchsenleiste sowie eine Leuchtdiodenzeile zugeschaltet werden. Die Leuchtdioden haben eigene Treiberschaltkreise, und belasten so die Signale am PIC kaum.

Die Buchsenleiste eines jeden Ports ist 10-polig. Die mittleren 8 Pins sind für den Port selbst reserviert. Pin 1 führt Masse und Pin 10 liegt auf Betriebsspannung. Dadurch ist es möglich andere Platinen mit diesem Stecker am PIC anzuschließen und gleichzeitig mit Spannung zu versorgen. Da Port A nur 5 Bit breit ist, bleiben hier 3 Pins ungenutzt.
Wird in die zweite Buchsenleistenreihe eine 8-fach Widerstandsmatrix (1 ... 10 kOhm) so eingesetzt, so daß der gemeinsame Anschluß im Massepin der Buchsenleiste steckt, kann mit dem Dil-Schalter ein binärer Code an das Port gelegt werden.

Die Takterzeugung ist so ausgelegt, das sowohl Keramikschwinger, Quarze wie auch Quarzgeneratoren eingesetzt werden können. Der jeweilige Typ wird mit Jumpern ausgewählt, und in eine Fassung gesteckt.

- Quarz: HC-18-Sockel
- Keramikschwinger: 3-polige Buchsenleiste
- Quarzgenerator: 14-polige Schaltreisfassung

Als Betriebsspannung genügt der Platine ein Steckernetzteil, das Gleich- oder Wechselspannung zwischen 8 V und 20 V bereitstellt. Auch die direkte Einspeisung von 5 V Gleichspannung ist möglich (linke Klemmleiste Pin 1).

Meine Originalplatine ist mit Treiberschaltkreisen 8287 bestückt, die kaum noch beschaffbar sind:

- Stromlaufplan
- Bestückungsseite
- Bestückungsplan

- Stückliste
- Layout im CDR-3-Format
- Layout im GIF-Format (300 dpi) (56k) Platinengröße: 3,7 x 5,5 inch = 9,4 x 14 cm

Als Alternative kann man die Treiberschaltkreise 74LS245 einsetzen, dann müssen die folgenden Dateien benutzt werden

- Stromlaufplan
- Bestückungsplan
- Stückliste
- Layout im GIF-Format mit 300 dpi Platinengröße: 3,7 x 5,5 inch = 9,4 x 14 cm

Die Variante mit dem Treiberschaltkreis 74LS245 ist mit einem ICSP-Steckverbinder ausgestattet, der es erlaubt, den PIC in der Testplatine zu programmieren. Die Variante mit 74LS245 wurde von mir nie praktisch aufgebaut. Über Erfahrungen mit dem Nachbau wäre ich dankbar.

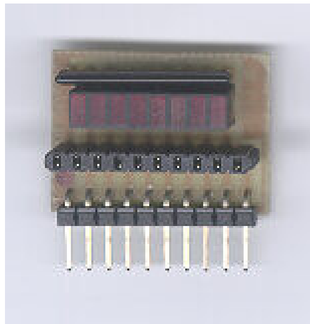
TIP:
Verwendet man Nullkraftsockel von Textool, so sollte man den Hebel an der Fassung vor dem Einlöten auf "offen" stellen.

16F84/62x-Mini-Platine

Das ist die Minimalversion einer Testplatine. Der PIC sitzt in einer Normalen IC-Fassung. Beide Ports sind auf die schon oben beschriebenen 10-poligen Buchsenleisten geführt. Als Taktgenerator kann nur ein Keramikschwinger eingesetzt werden. Als Stromversorgung muß 5 V Gleichspannung über eine 10-polige Buchsenleiste oder über eine zusätzliche 2-polige Buchsenleiste eingespeist werden.
Der Siebkondensator für die Betriebsspannung kann als SMD oder als normaler stehender Typ bestückt werden.

- Stromlaufplan
- Bestückungsseite
- Leiterseite
- Bestückungsplan
- Stückliste
- Layout im CDR-Format (Corel Draw)
- Layout im GIF-Format (11 kByte) Platinengröße: 2,1 x 1,8 inch = 5,3 x 4,6 cm

LED-Platine



Diese kleine Platine kann auf die Buchsenleiste der 16F84-Mini-Platine gesteckt werden. Die Leuchtdioden zeigen dann den Ausgangspegel des PIC-Ports an. Der Port selbst ist durchgeschliffen, und steht an der Buchsenleiste der Platine zur Verfügung.

- Stromlaufplan
- Stückliste
- Bestückungsplan
- Layout im CDR-Format (10 kB)
- Layout im GIF-Format (300 dpi) (4 kB) Platinengröße: 1,1 x 0,8 inch = 2,8 x 2 cm

LCD-Tastatur-Platine

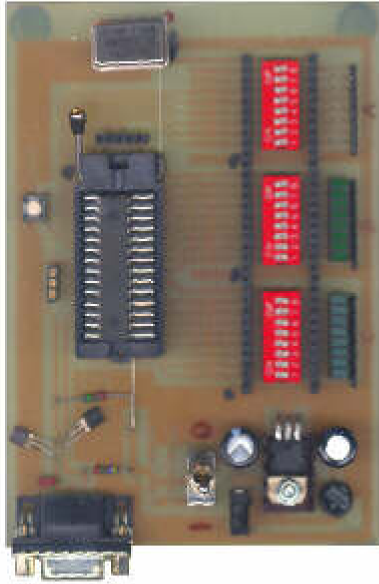
Blinkende Leuchtdioden sind zwar ganz nett, aber viel besser ist doch eine alphanumerische LCD-Anzeige . Auch eine Matrixtastatur, zur Zahleneingabe verspricht einen akzeptablen Bedienkomfort. Sowohl die LCD-Anzeige wie auch die Tastatur benötigen relativ viele Anschlußpins des PICs. Deshalb nutze ich einen einfachen Adapter, der die Tastatur über Widerstände vom Display entkoppelt, und es erlaubt, beide gemeinsam an ein 8-Bit Port, also z.B. PortB, an den PIC anzuschließen.

Die Platine besitzt Buchsenleisten für das Display und die Tastatur, eine Reihe SMD-Widerstände und ein Kabel mit 10-poligem Steckverbinder, der auf die Buchsenseisten der Testplatinen paßt.

Die Stromversorgung des Displays erfolgt von der Testplatine aus, die Kontrasteinstellung des Displays ist mit Widerständen auf einen guten Wert eingestellt.

- Ansicht ohne Display und Tastatur
- Platinenrückseite (der schwarze "Gnubbel" ist ein Gummifuß)
- Stromlaufplan
- Bestückungsplan
- Stückliste
- Layout im CDR-Format (12 kByte)
- Layout im GIF-Format (300 dpi) (8 kByte) Platinengröße: 1,8 x 1,2 inch = 4,6 x 3 cm

16F876-Testplatine



Die 16F876-Testplatine war ein überhasteter Entwurf und müßte dringend überarbeitet werden, aber die neuen PICs waren angekündigt, und ich brauchte schnell eine neue Testumgebung.

Das Grundkonzept ist das der 16F84-Platine.

Die Klemmleisten sind schwer beschaffbar und teuer. Deshalb werde ich sie erst auf einer überarbeiteten Platine bestücken.

Die Treiberschaltkreise der für die LED-Zeilen sind weggefallen. Das hat aber zur Folge, das hochohmige Eingangssignale stark belastet werden. In diesem Fall ist die dem PIC näher stehende Buchsenleiste zur Signaleinspeisung zu verwenden, und der DIL-Schalter des Eingangs auf OFF zu stellen, um die LEDs vom PIC zu trennen. Dann sieht man allerdings auch die Signale nicht mehr an der LED.

Als Taktgenerator hatte ich ursprünglich nur noch Keramikschwinger vorgesehen. Als ich den 16F876 aber bis 20 MHz ausreizen wollte, mußte ich feststellen, daß 20 MHz-Schwinger nicht so ohne weiteres zu bekommen waren. Deshalb habe ich einen Steckplatz für einen Quarzoszillator nachgerüstet.

Zu spät habe ich bemerkt, daß der Ausgang und Eingang der seriellen Schnittstelle invertiert werden müssen, um wenigstens RS232-ähnliche Signale zu erzeugen. Normalerweise übernimmt das ein separater Treiberschaltkreis, der sich auch um den normgerechten Pegel (-12V / +12V) kümmert. Ich habe nachträglich zwei Transistoren "integriert".

So ist diese Platine noch ein Provisorium, aber Provisorien halten sich bekanntlich am längsten. Die Version, die in den folgenden Dateien beschrieben ist, enthält keinen Sockel für Quarzgeneratoren:

- Stromlaufplan
- Bestückungsseite
- Stückliste
- Bestückungsplan
- Layout im CDR-3-Format (20 k)
- Layout im GIF-Format (300 dpi) (53k) Platinengröße: 4,8 x 3,7 inch = 12,2 x 9,4 cm

ACHTUNG
Noch ein Tip. 28-polige Nullkraftsockel gibt es überwiegend in der breiten Ausführung mit 15 mm Reihenabstand, wärend der PIC nur 7,5 mm Reihenabstand hat. Beim Kauf sollte man unbedingt darauf achten, daß der schmale PIC auch in die breite Fassung paßt!

PIC-Lernbeispiele

Diese Seite enthält einfache Beispiele für die Anwendung von PICs. Diese sind weniger als praktische Anwendungen gedacht (eine Uhr hat ja schon jeder) sondern sie sollen als Lernübungen dazu dienen, sich mit den Prozessoren vertraut zu machen.

Die meisten dieser Lernübungen wurden primär für den PIC16F84 geschrieben, laufen aber (evtl. mit kleinen Änderung) auch auf dem PIC16F876 oder dem PIC16F628.

Lauflicht	8 Leuchtdioden bilden ein einfaches Lauflicht (Pin-Einstellung, Warteschleifen, Zyklen)
Tastatur	Eine Matrixtastatur wird abgefragt (In- and Out-Befehle)
LCD-Display	Darstellung auf einem intelligenten LCD-Display
Tastatur und LCD an einem Port	die Verbindung von Tastatur und LCD-Beispiel (#define, PIC16F628)
LED-Ziffernanzeige	eine 7-Segment LED-Ziffernanzeige wird angesteuert (Timer, Interupt, Datentabelle)
LED-Stopp-Uhr	Eine Stopp-Uhr mit 7-Segment Ziffernanzeige
RS-232-Interface	eine RS-232-Schnittstelle mit einem PIC16F84
Spannungsmessung mit 16F876	eine Eingangsspannung (0..5V) wird gemessen und mit LEDs angezeigt (ADC) **nur 16F87x**
spannungsgesteuerte PWM	eine Eingangsspannung (0..5V) steuert das Tastverhältnis einer Rechteckschwingung (ADC, PWM) **nur 16F87x**
EEPROM lesen	ein LED-Muster wird aus dem EEPROM gelesen und angezeigt (EEPROM, PIC16F628)
David Taids WALK	4 Leuchtdioden bilden ein einfaches Lauflicht (Programmanalyse,

Die Beispiele können vom Internet von der URL :

http://www.sprut.de/electronic/pic/programm/

heruntergeladen werden, sie sind inkl. der Beschaltung und weiteren Informationen verfügbar.

PBrenner

Ein Windowsprogramm für Parallelport-PIC-Brenner

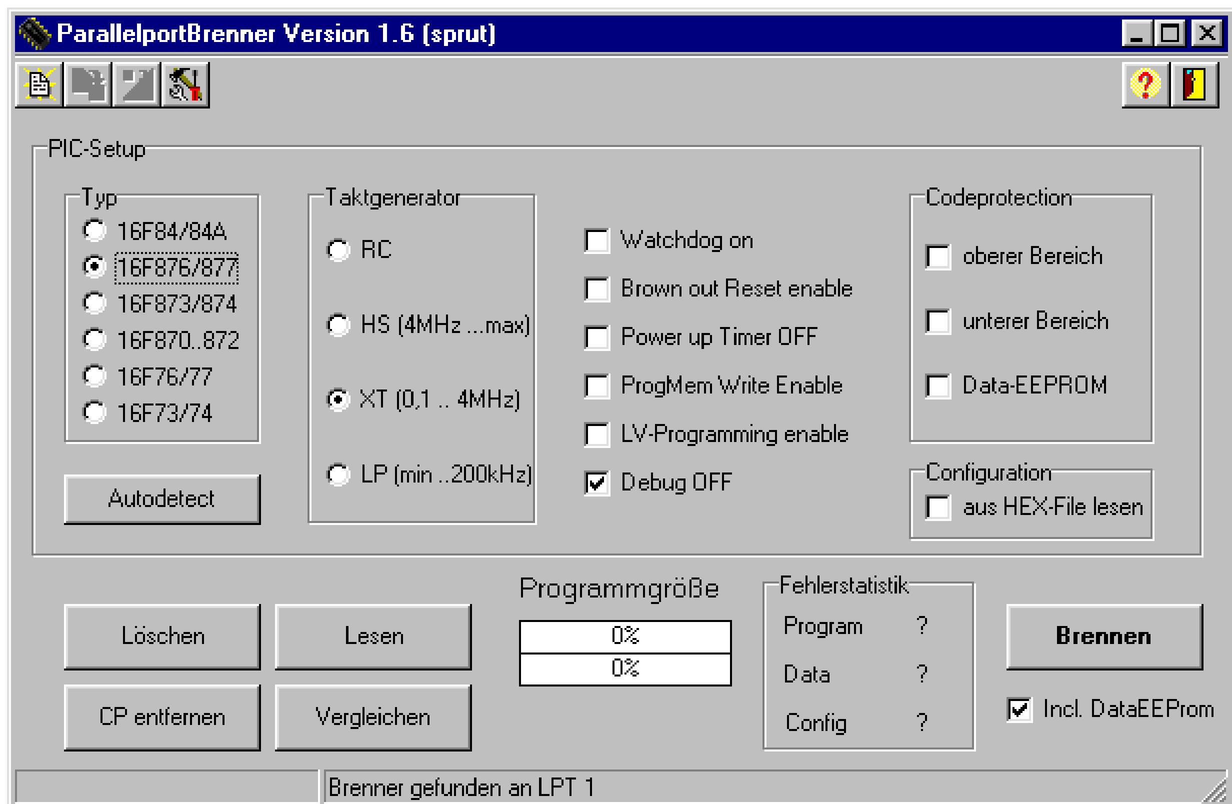
Achtung XP-User
Die Versionen ab 2.1 sind XP-fähige Version die in vielen Bereichen überarbeitet und erweitert wurden.

Problem
Wer sich mit PIC-Prozessoren beschäftigen möchte, benötigt einen PIC-Brenner - ein Gerät, mit dem sich ein fertig entwickeltes Programm in den PIC-Prozessor einspeichern läßt.
Auf meiner Brennerseite biete ich dazu mehrere Lösungen an. Die preiswerten Parallelportbrenner (Brenner0 , Brenner5 , Brenner3 und Brenner1) sind kompatibel zum klassischen Tait-Brenner, dem Standard für das hobbymäßige PIC16F84-Brennen.

David Tait hat sich aus der PIC-Brenner-Szene verabschiedet, und hinterließ für sein Brennerkonzept ein DOS-Kommandozeilen-Programm, das für Neueinsteiger umständlich zu bedienen ist, und neuere PIC-Typen nicht kennt.
Deshalb hatte ich zunächst den Brenner2 entwickelt, der über die Serielle-PC-Schnittstelle angesteuert wird, und für den ich ein hoffentlich benutzerfreundliches Windows-Programm schrieb. Der Nachteil des Brenners2 ist sein aufwendiger Aufbau, so benötigt er selbst einen schon programmierten PIC16F84 als Steuerprozessor.

Lösung
Der Tait-Brenner ist aber aufgrund seines einfachen Aufbaus weit verbreitet. Deshalb schrieb ich ein Windowsprogramm, das den Tait-Brenner steuert, und gleichzeitig seine Beschränkung auf den PIC16F84 aufhebt. Ermöglicht wurde das durch eine Delphi-Komponente, die Alexander Weitzman als Freeware verteilt. Dieses Tool ermöglicht eine sichere Kontrolle des Parallelports unter Win95/98/NT/2000. Inzwischen benutze ich aber eine Komponenta von Zloba Alexander, die unter XP besser funktioniert.
Ich habe schon vorher ähnliche Programme gesehen, aber diese scheiterten oft unter WinNT, und stellten somit für mich nie eine dauerhafte Lösung dar.

Das Programm PBrenner besitzt die Bedienoberfläche des Programms für den Brenner2, und ist deshalb hoffentlich leicht zu bedienen. Wer mit den ganzen PIC-Setup-Optionen nicht zurecht kommt, informiere sich bitte hier über die Konfiguration eines PIC .



Voraussetzungen
- ein Brenner3 oder Brenner5 (bevorzugte Variante) oder Brenner1 oder
- Parallport-Brenner nach Tait (für PP.exe) oder
- ein Brenner0 oder
- ein Brenner nach AN589 oder
- ein "Quick and Dirty-Brenner" nach Tait oder
- Brenner2 (noch im Experimentierstadium)

Soll ein Parallelportbrenner (Brenner1/3/5, AN589 ...) verwendet werden so sollte vor dessen Aufbau die Funktionsfähigkeit der Software entsprechend der im Softwarepaket PBrenner ab der Version 2.0 enhaltenen Help-Datei geprüft werden.

Möglichkeiten

- unterstützt **16F84(A), 16F87xA-, 16F87x- und 16F7x-Typen sowie 16F627/628 und 12F629/675**
- Brennen, Löschen , Auslesen, Vergleichen,
- Entfernung des Speicherschutzes bei gleichzeitigem Löschen
- komfortable Konfiguration des PIC
- Anzeige des zu Brennenden Codes als HEX-Listing oder reassembliert

---

Realisierung
Der Brenner wird (zunächst ohne PIC) an den Druckerport des PC angeschlossen und mit der Spannungsquelle verbunden. Dann wird das Programm gestartet. Der Brenner wird erkannt. Falls kein Brenner1 oder Brenner3 (oder Tait-Typ-3) oder Brenner5 verwendet wird, muß noch die richtige Hardware eingestellt werden, bevor ein PIC in die Fassung gesteckt wird.

Download

Die aktuelle Version V2.4b

unterstützt die neuen Typen PIC16F873A..877A sowie PIC12F629/675.
Es gibt 2 Versionen der Software. Die Standardversion V2.4b ist für Win 95/98/XP geeignet, sowie für WinNT und Win2000 wenn der Anwender Administratorrechte besitzt

- PBrenner V2.4b + Readme-Datei + Help-Datei + Treiber (590 kByte)

Die 2. Version V2.4bU läuft unter Win95/98 nicht aber unter XP. Dafür läuft sie unter Win2000/WinNT mit Hauptbenutzer-Rechten, wenn sie erst einmal mit Admin-Rechten installiert wurde.

- PBrenner V2.4bU + Readme-Datei + Help-Date + Treiber (590 kByte)

Die Vorgängerversion V2.2

Ab Version V2.2 war die erste XP-taugliche Version. Im Vergleich zur Version V1.8 verfügt sie über ein Hilfe-System, unterstützt Brenner nach AN589 und unterstützt das Brennen der ID-Information. Der integrierte Reassembler wurde nun standardmäßig freigeschaltet. Natürlich läuft diese Version auch unter Win95/98/ME sowie NT (Admin-Rechte erforderlich)

- PBrenner V2.2+ Readme-Datei + Help-Datei + Treiber (deutsche Version) (533 kByte)
- PBrenner V2.1+ Readme-File + Driver (english Version) (514 kByte)

Die alte Version V1.8c

ist die ehemalige alte Standardversion für Win95/98/NT, die aber nicht XP-tauglich ist. Dafür erfordert sie unter NT keine Admin-Rechte. (außer zur Installation)

- PBrenner V1.8d + Readme-Datei + Dokumentation + Treiber (434 kByte)

HINWEIS:
Bitte darauf achten, daß die Grafikkarte auf "kleine Schriftarten" eingestellt ist (in Windows unter: Systemsteuerung - Anzeige - Einstellungen). Ansonsten kann es zu Problemen mit der Grafikausgabe kommen.

---

bekannte Probleme

Die Programm-Version V2.3 ist mit dem PIC16F84, PIC16F84A, dem PIC16F873/876, dem PIC16F874A, dem PIC16F627/628, dem PIC12F675 und dem PIC16F877 unter Win95, Win98 erprobt.
Die Programm-Version V1.8 ist mit dem PIC16F84, PIC16F84A, dem PIC16F873/876, dem 16F627/628 und dem PIC16F877 unter Win95, Win98 und WinNT erprobt. Die Geschwindigkeit ist überzeugend, besonders, wenn der PIC nicht vollständig neu gebrannt werden muß.

Ein angeschlossener Brenner (außer Quick and Dirty) wird beim Programmstart manchmal nicht erkannt, wenn ein PIC im Brenner steckt (vor allem AN589).

Ist im Setup der Quick und Dirty-Modus eingestellt, brennt die Software blind, und kann das Ergebnis nicht überprüfen. Das Programm weiß nicht einmal, ob ein PIC im Brenner eingesetzt wurde. Auch das Fehlen des Brenners wird nicht erkannt. Das ist kein Fehler, sondern liegt an der Natur des Q&D-Brenners.

Einige wenige PCs machen immer wieder Probleme. In der Dokumentation von PBrenner (ab V2.0) ist eine einfacher Kompatibilitätstest beschrieben. Mit dem läßt sich auch ohne fertigen Brenner prüfen, ob PBrenner auf dem eigenen PC lauffähig ist.

Einige moderne Druckertreiber behindern PBrenner. Verantwortlich sind vor allem die Funktionen zur Druckerstatusüberwachung. Falls PBrenner nicht funktioniert, sollte man probehalber diese Funktionen abschalten, oder den Druckertreiber entfernen.

In die Platine des Brenners1 kann nur der 16F84, 16F62x sowie 12F6xx eingesetzt werden, die anderen Typen brenne ich über den ICSP-Steckverbinder. Eine neue Brenner-Platine mit Fassungen für alle 16F8xx-Typen ist der Brenner3 , und der Brenner5 die den Brenner1 endgültig ablösen.

Einige Compiler erzeugen recht exotische HEX-Files, die gelegentlich die 'Programmspeicher-Füllstandsanzeige' austrixen (immer ein gelber Balken). Das hat keine negativen Auswirkungen auf das Brennen. Eine grafische Anzeige der wirklichen Speicherbelegung kann mit dem Button 'grafische Speicherdarstellung' aufgerufen werden.

Entwicklungsgeschichte
V2.4a (04.10.2002) / V2.4b (10.10.2002) / Help-File angepaßt (25.10.2002)
- Win NT/2000-Problem von V2.4 hoffentlich behoben

V2.4 (15.09.2002)
- Editor für Bandgap- und OSCAL-Werte der PIC12F629/675
- stabilere Arbeit mit dem Brenner2

V2.3 (08.09.2002)
- Unterstützung der PIC16F873A/874A/876A/877A, PIC12F629/675
- Bugfix für PIC16F72
- geänderte Codeprotection-Einstellung für 16F870..873 (Chip wurde modifiziert)
- beschleunigtes Lesen, Vergleichen, Brennen
- verbesserter Reassembler
- versuchsweise Unterstützung des Brenner2

V2.2 (27.05.2002)
- Produktpflege: Autodetect zeigt codeprotection an

V2.1 / V2.1e / V1.8d (11.02.02)
- Bugfix: Fehler bei der PIC16F627 / PIC16F76/77 Behandlung behoben

V2.0T2e (15.01.02)
- da die Version V2.0T2e sehr stabiel zu laufen scheint, stelle ich die englische Version bereit

V2.0T2 (12.12.01)
- Betaversion einer gründlich überarbeiteten Programmfassung mit XP-Unterstützung und AN589-Tauglichkeit

V1.8c (18.11.01)
- Bugfix: Fehler bei der PIC16F84(A)-Takteinstellung behoben (dieser Fehler trat nur bei Ver. 1.8(b) auf)

V1.8b (15.11.01)
- Bugfix: Fehler in der PIC16F7x-Programmierung behoben

V1.8 (31.10.01)
- zusätzliche Unterstützung der PIC-Typen 16F627 und 16F628.
- Verbesserung der 16F84A-Unterstützung
- neue Fenster für grafische Speicheranzeige und Anzeige des geladenen HEX-Files

V1.7b (09.10.01)
- Bugfix: Fehler in der EEPROM-Adressierung/EEPROM-Vergleich behoben

V1.7a (28.09.01/03.10.01)
- Verbesserte Erkennung eines vorhandenen Brenner5
- ein in der Fassung steckender PIC behindert die Brennererkennung nicht mehr so konsequent
- Toleranz gegenüber EEPROM-Daten mit mehr al 8 Bit

V1.6 (03.07.01)
- zusätzliche Unterstützung der PIC-Typen 16F870..16F872 und 16F73/74/76/77.
- Unterstützung von Kommandozeilenparametern für PIC-Typ und HEX-Datei-Name

V1.5 (21.05.01)
- Toleranz gegenüber Nonsens-Programmcodes mit mehr al 14 Bit (wird von einem BASIC-Compiler ausgegeben)

V1.4 (03.04.01)
- verbesserte Stabilität auf schnellen PCs

V1.2 (25.01.01)
- Bug beseitigt: PIC16F874 wird nun richtig behandelt

V1.1 (18.10.00)
- individuelle Behandlung des PIC16F84A (verschieden vom 16F84)
- Löschen des EEPROM-Datenbereichs des 16F84 möglich

V1.0 (12.10.00)
- kleinere Änderungen

V0.4 (09.10.00)
- unter WinNT/2000 sind keine Administrator-Rechte mehr erforderlich (außer zur Treiberinstallation)
- die gewählte Hardwarevariante wird bis zum nächsten Programmstart in einer ini-Datei gespeichert
- rudimentäre Dokumentation im Word-Format

V0.3 (07.10.00)
- Unterstützung Der PIC-Typen 16F84 , 16F84A, 16F873 , 16F874, 16F876, 16F877

(grau geschriebene Typen nicht erprobt)
- Quick and Dirty-Brenner unterstützt
- Bug in der Codeprotection-entfern-Routine für 16F87x behoben

V0.2 (05.10.00)
- Unterstützung verschiedener Tait-Brenner-Varianten

V0.1 (04.10.00)
- Urversion zum Testen nur für den Brenner1

PIC-Links

Wichtiger Hinweis:
Mit Urteil vom 12. Mai 1998 hat das Landgericht Hamburg entschieden, dass man durch die Ausbringung eines Links die Inhalte der gelinkten Seite gegebenenfalls mit zu verantworten hat. Dies kann - so das LG - nur dadurch verhindert werden, dass man sich ausdrücklich von diesen Inhalten distanziert. Von www.sprut.de-Seiten führen Links zu anderen Seiten im Internet. Für all diese Links gilt: Ich möchte ausdrücklich betonen, dass ich keinerlei Einfluss auf die Gestaltung und die Inhalte der gelinkten Seiten und Foren haben. Aus diesem Grunde distanziere ich mich hiermit ausdrücklich von allen Inhalten gelinkter Seiten und machen mir ihre Inhalte nicht zu Eigen. Diese Erklärung gilt für alle Links und für alle Inhalte der Seiten, zu denen bei mir gelinkte Inhalte führen.

Microchip

Homepage
Entwicklungsumgebung MPLAB-IDE und andere Tools
Übersicht über die Dokumentation zu PIC16... Microcontrollern
Übersicht über die FLASH-Microcontroller
Dokumentation PIC16F84
Dokumentation PIC16F87x
Dokumentation PIC16F628

Programmiergeräte

David Taits Programmiergerätelinks
David Taits Programmiergeräte (ftp)
David Taits Brenner für den Parallelport
Parallelportbrenner
Einfacher Brenner für den Serialport
J. Aichingers Brenner für Serial- und Parallelport mit Windowsprogramm

Anwendungen

PIC als Flugdatenschreiber (Datenlogger) im Modellflugzeug (D.Meissner)
PIC als Flugdatenschreiber (Datenlogger) - Ergänzungen (D Meissner))
J Aichinger Motorsteller für Elektromodellflugzeuge mit PIC-Controller

andere PIC-Seiten

Parsic: grafische Programmierung von PICs
JAL - eine Programmiersprache für PICs

Bauelementequellen

Conrad-Electronik (Keramikschwinger, PIC16F84)
Reichelt
RS
ELPRO
Memec GmbH (moderne PIC-Prozessoren z.B. PIC16F87x)
Farnell (moderne PIC-Prozessoren z.B. PIC16F87x; 18polige 0-Kraft-Sockel)
Nessel-Elektronik (chaotische Webseite, aber gutes Angebot an FETs)

Leiterplattenlayoutentwicklung

EAGLE 3.5 von der Firma CadSoft (nichtkomerziell bis zum halben Eurokartenformat (10x8 cm) bei 2 Ebenen frei nutzbar)
TARGET (voll funktionsfähige Demoversion ist auf 100 Pins/Pads sowie auf 2 Ebenen begrenzt)

Spezifikationen/Datenblätter

Intel-HEX-Format

Andere Links

Modellsportverein Siewisch
Wilhelm Gecks Seite für elektrischen Flugmodellantrieb (Motoren, Luftschrauben, Akkus)

meine Homepage

meine email-Adressel

Die Nutzung der I/O-Pins (Ports)

Allgemeines

Die einfachste I/O-Funktion des PIC verkörpern die Ports (PortA ... PortE). Sie stellen jeweils bis zu 8 digitale Leitungen bereit, die als digitaler Eingang oder digitaler Ausgang
funktionieren können.
Wieviele Ports ein PIC besitzt, hängt von der Zahl der zur Verfügung stehenden Anschlußpins, also von der Gehäusegröße ab.

Gehäuse	Typ (Beispel)	Port A	Port B	Port C	Port D	Port E
8-Pin Gehäuse	12F6xx	X				
18-Pin Gehäuse	16F84 16F62x	X	X			
28-Pin Gehäuse	16F873 16F876	X	X	X		
40-Pin Gehäuse	16F877	X	X	X	X	X

Mit Ausnahme des Port A besitzen alle Ports jeweils 8 Pins. Das Port A besitzt je nach PIC-Typ 5 Pins, 6 Pins (12F6xx) oder 8 Pins (16F62x).

Einige Pins stehen nicht exclusiv für die Ports zur Verfügung, sondern können auch anderen Funktionen wie z.B. seriellen Ein-/Ausgängen zugewiesen werden.

Jedes Port-Pin kann sowohl als Eingang wie auch als Ausgang initialisiert werden. Das erfolgt durch das Setzen von Bits in den **TRISx**-Registern. Jedes Port besitzt ein eigenes **TRISx**-Register, in diesem Register ist jedem Port-Pin ein Bit zugeordnet. Steht dieses Bit auf 1, dann ist daszugehörige Pin ein Eingang. Steht das Bit aber auf 0, dann ist das Pin ein Ausgang. Nach einem Reset oder nach dem Einschalten, sind die **TRISx**-Register auf 0xFF gesetzt, womit alle Port-Pins zunächst Eingänge sind.

Das setzen von Ausgangspins auf High (1) oder Low-Level (0) erfolgt durch Beschreiben der **PORTx**-Register. Jedes Port besitzt ein eigenes **PORTx**-Register, in diesem Register ist jedem Port-Pin ein Bit zugeordnet. Ist ein Port-Pin als Ausgang initialisiert, dann führt es den Pegel des zugehörigen Bits im **PORTx**-Register.

Das Abfragen von Eingangs-Port-Pins erfolgt durch Lesen der **PORTx**-Register. Dabei werden die an den Port-Pins anliegenden elektrischen Pegel (High/Low) in die zugehörigen Bits der **PORTx**-Register kopiert. Als Ausgang initialisierte Pins lesen dabei ihre eigenen Ausgangspegel, die verändern sich also im **PORTx**-Register nicht.

Initialisierung

Beim Einschalten der Betriebsspannung (Power-On-Reset) oder beim Reset (**MCLR**=0) werden alle Bits der **TRIS**-Register auf den Wert 1 gesetzt. Damit sind alle I/O-Ports automatisch als Eingang konfiguriert.

Eine Besonderheit stellen Pins da, die neben der Port-I/O-Funktion auch als analoge Inputs benutzt werden können (z.B. Port A & Port E beim 16F87x oder 16F62x). Diese Pins werden beim Reset als analoge Eingänge initialisiert, und können als digital I/O erst benutzt werden, wenn sie von alnalog auf digital umgestellt wurden.

Dies erfolgt durch Setzen der Bits **PCFG3**...**PCFG0** im Register **ADCON1** (Adresse 0x9F). In Detail ist das hier beschrieben.

```
; alle ADC-Eingänge auf digital I/O umschalten
    BSF      STATUS, RP0      ; auf Bank 1 umschalten
    BSF      ADCON1, PCFG3    ; PCFG3=1
    BSF      ADCON1, PCFG2    ; PCFG2=1
    BSF      ADCON1, PCFG1    ; PCFG1=1
    BSF      ADCON1, PCFG0    ; PCFG0=1
    BCF      STATUS, RP0      ; auf Bank 0 zurückschalten
```

Um den Eingangspegel an einem als Eingang initialisiertem Pin zu lesen, greift man einfach auf das zugehörige **PORTx**-Register zu. Daraufhin wird das gesamte Port (also alle zugehörigen Pins) eingelesen, ihre digitalen Werte in das **PORTx**-Register geschrieben, und dieser Wert an das Programm weitergegeben.

```
; Einlesen des Port B in das Akkuregister W
    MOVFW     PORTB           ; Port B lesen und nach W kopieren
```

Auch der Zugriff auf ein einzelnes Bit eines **PORTx**-Registers führt zum Einlesen des gesamten Ports in das **PORTx**-Register, was aber unproblematisch ist.

```
; Abfrage des Pins 0 des Ports B (RB0)
    BTFSC     PORTB, 0        ; Port B Pin 0 lesen
    GOTO      RB0istHigh      ; Sprung fall RB0 High ist
    GOTO      RB0istLow       ; Sprung fall RB0 Low ist
```

Um ein Pin eines Ports als Ausgang zu nutzen, muß es erst als Ausgang initialisiert werden, wozu das zugehörige Bit im **TRISx**-Register dieses Ports auf '0' gesetzt werden muß. Danach kann High und Low durch setzen und löschen des zugehörigen **PORTx**-Bits ausgegeben werden.

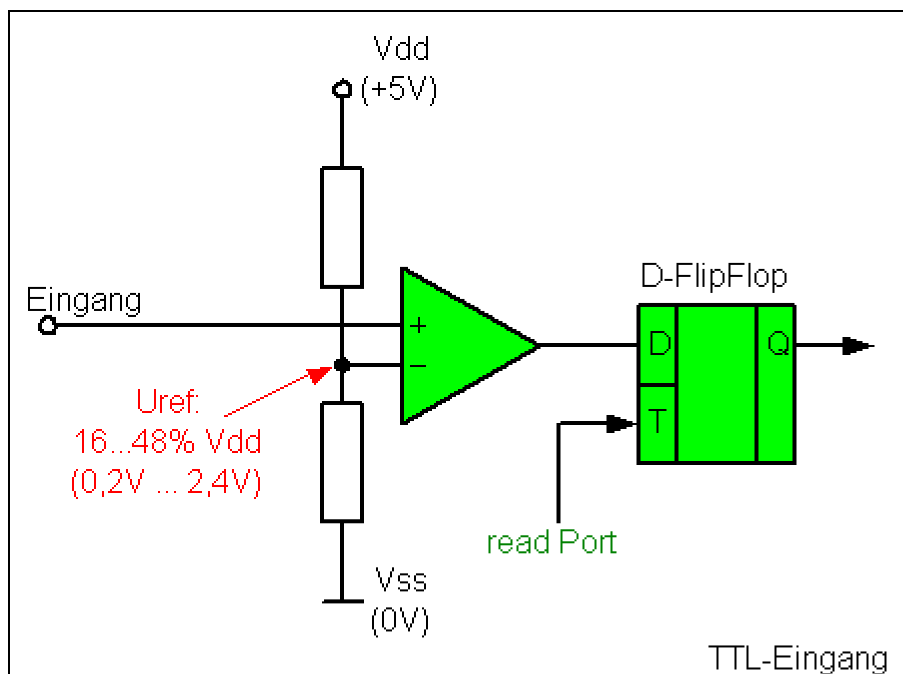
```
; Pins 0 des Ports A (RA0) und das gesamte PortB auf Ausgang konfigurieren
    BSF      STATUS, RP0      ; auf Bank 1 umschalten
    BCF      TRISA, 0         ; RA0 auf Ausgang einstellen
    CLRF     TRISB            ; Port B auf Ausgang umschalten
    BCF      STATUS, RP0      ; auf Bank 0 zurückschalten
    BSF      PORTA, 0         ; High-Pegel an RA0 ausgeben
    MOVLW    0x0F
    MOVWF    PORTB            ; RB0..RB3: High-Pegel, RB4..RB7: Low-Pegel
```

Eingänge

Es gibt zwei unterschiedliche Arten von Eingangspins:

- TTL-Eingangs-Pin
- TTL-Eingang mit Pull-up-Widerstand
- Schmitt-Trigger Eingangs-Pins

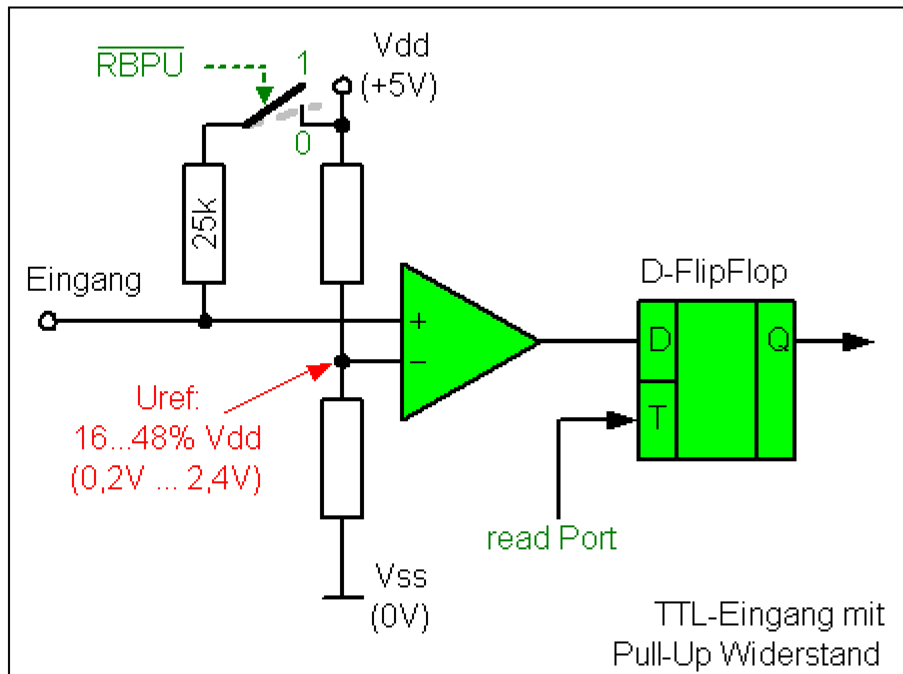
Die Ports C, D und E haben Schmitt-Trigger Eingänge.
Die Ports A und B haben TTL-Eingänge mit Ausnahme des Pins 4 von Port A (RA4), welches ebenfalls ein Schmitt-Trigger Eingang ist.



TTL-Eingang

Ein TTL-Eingang erkennt eine Spannung unter 16% der Betriebsspannung (<0,2V) als Low-Pegel und eine Spannung über 48% der Betriebsspannung (>2,4V) als High Pegel. Der Bereich dazwischen ist nicht definiert. In der Realität sind die Eingänge nicht ganz so pingelig, aber die oben angegebenen Werte sind die vom Hersteller garantierten.

So ein Eingang eignet sich nur zum Einlesen sauberer digitaler Pegel.

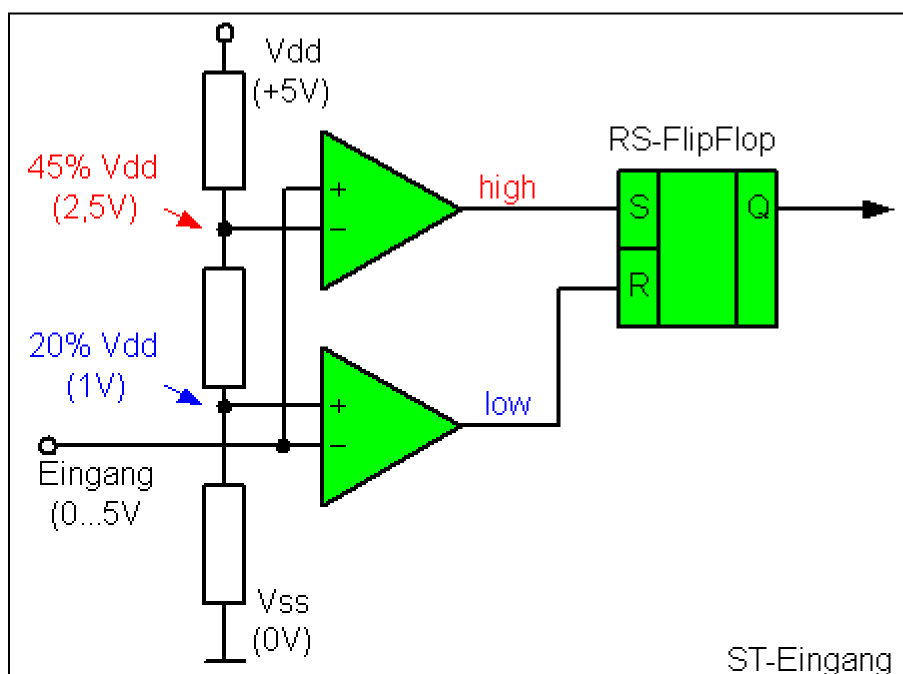


TTL-Eingang mit Pull-Up Widerstand

Bei einigen Ports kann man einen zusätzlichen Pull-Up-Widerstand aktivieren. Dieser garantiert, das ein offener Eingang auf definiertem High-Pegel liegt. Das ist z.B. beim Anschluß einfacher Schalter oder Taster hilfreich.

Alle Pins von Ports B besitzen solche Pull-Up-Widerstände, die mit dem Steuerbit RBPU geneinsam ein- ('0') oder ausgeschaltet ('1') werden können.

Beim 12F6xx besitzen alle I/O-Pins außer GP3 Pull-Up-Widerstände, die aber individuell über Steuerbits ein- oder ausgeschaltet werden können.



Schmitt-Trigger-Eingang (ST)
Der ST-Eingang schaltet auf Low-Pegel um, wenn die Eingangsspannung unter 20% der Betriebsspannung (<1V) fällt. Steigt die Eingangsspannung dann wieder über 45% der Betriebsspannung (2,25V), schaltet der Eingang wieder auf High-Pegel. Bei Schwankungen zwischen diesen beiden Schwellwerten, behält der Eingang seinen alten Wert bei. Es gibt also keinen verbotenen Bereich.

ST-Eingänge eignen sich gut, um Eingangssignale zu verarbeiten, die keine perfekten TTL-Pegel haben, sondern sich stetig (und nicht Sprunghaft) ändern. Ein ST-Eingang kann z.B. mit einer verrauschten sinusförmigen Spannung gespeist werden, die er in saubere High und Low-Signale wandelt. Ein normaler TTL-Eingang würde im verbotenen Spannungsbereich zwischen 0,2V und 2,4V undefinierte, und vielleicht sogar unstabile Werte liefern.

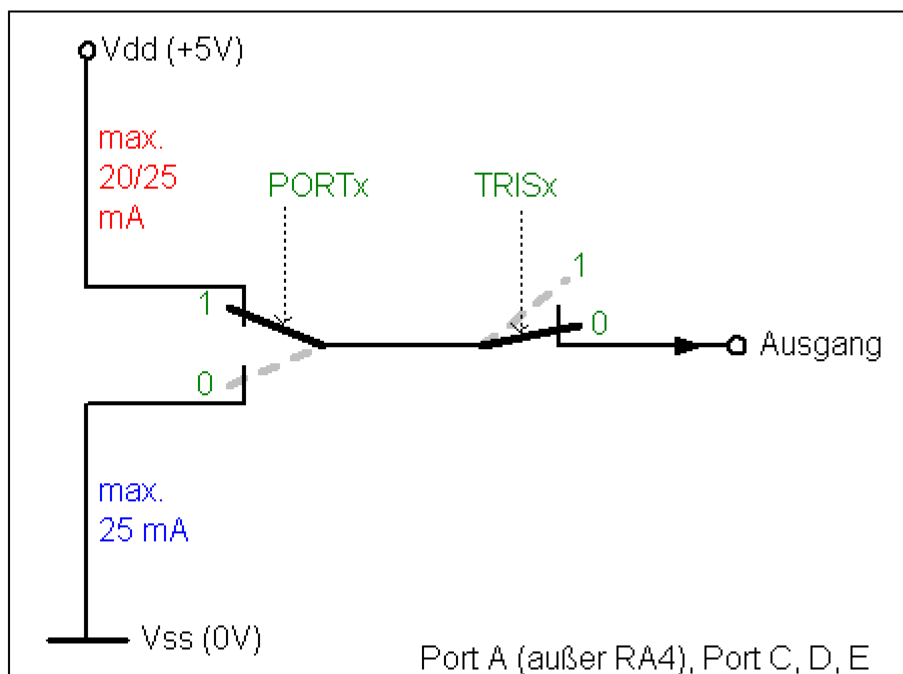
Schutzdioden
IO-Pins des 16F62x und des 12F6xx besitzen Schutzdioden nach Vdd und Vss, die Über- und Unterspannungen an den Pins verhinden sollen.

Ausgänge

Es gibt zwei unterschiedliche Arten von Ausgangspins:

- TTL-Ausgangs-Pin
- Open Drain-Ausgang-Pins

Die Ports B, C, D und E haben TTL-Ausgänge.
Das Port A hat auch TTL-Ausgänge, mit Ausnahme des Pins 4 von Port A (RA4), welches einen Open Drain-Ausgang hat.

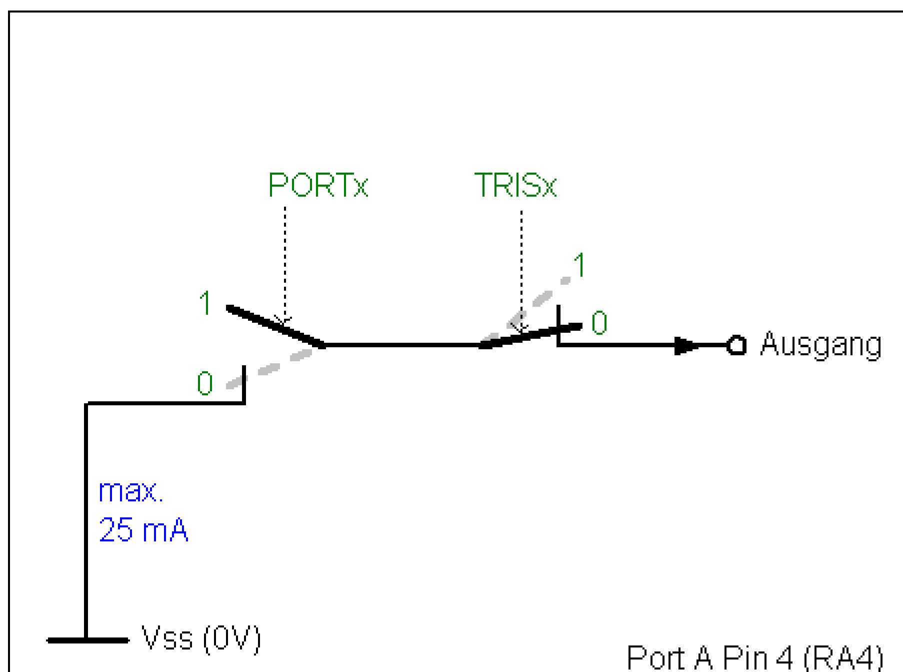


Natürlich sind in den PICs die Ausgänge mit Logikgattern und MOSFETs aufgebaut, aber zur vereinfachten Darstellung benutze ich Stromlaufpläne mit Schaltern.

Port A (außer RA4), Ports B, C, D, E
Der TTL-Ausgang kann sein Ausgangspin sauber auf Vdd (+5V) wie auch auf Vss (0V) legen. Der jeweilige Pegel wird vom zugehörigen Bit im **PORTx**-Register bestimmt. Die Belastbarkeit eines einzelnen Pins beträgt dabei

für den 16F84 25 mA (Low-Pegel) bzw. 20 mA (High-Pegel). Alle anderen Flash-PICs erlauben 25mA bei beiden Pegeln.

Mit dem zugehörigen Bit im **TRISx**-Register kann die Ausgangsfunktion abgeschaltet werden.



Pin RA4

Der Open-Drain-Ausgang kann sein Ausgangspin nur sauber auf Vss (0V) ziehen. Das erfolgt, wenn das zugehörigen Bit im **PORTx**-Register den Wert 0 hat. Die Belastbarkeit des Pins beträgt dabei 25 mA.

Liegt das **PORTx**-Bit dagegen auf 1, dann wird der Ausgangspegel nur durch die externe Beschaltung bestimmt.

Der Vorteil dieser Beschaltung ist ein erhöhter zulässiger Spannungspegel am Pin von max. 8,5V. Außerdem lassen sich Open-Drain-Ausgänge bedenkenlos parallel schalten.

Mit dem zugehörigen Bit im **TRISx**-Register kann die Ausgangsfunktion

abgeschaltet werden.

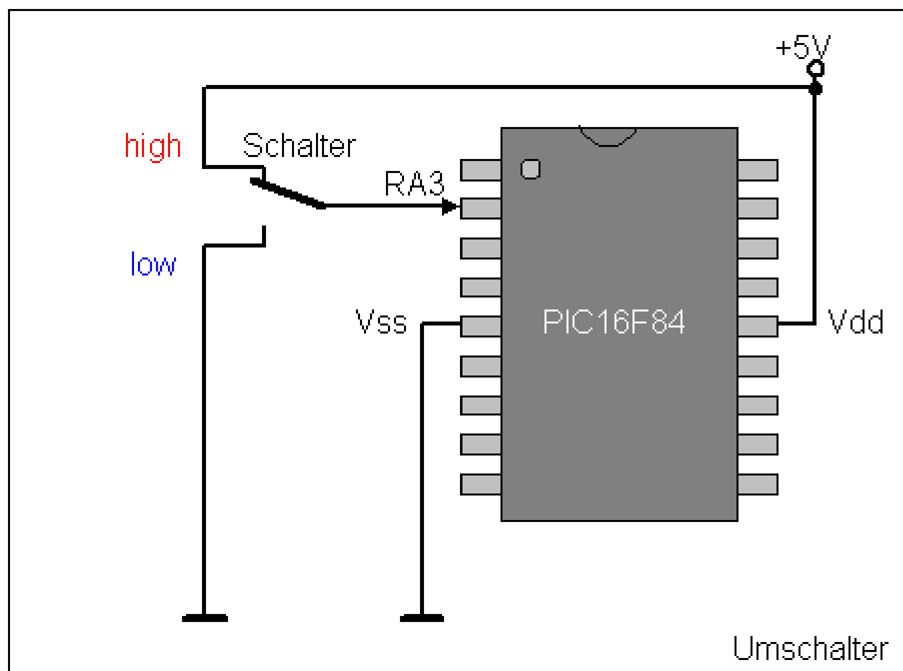
Die Belastbarkeit eines einzelnen Pins beträgt für den 16F84 25 mA (Low-Pegel) bzw. 20 mA (High-Pegel). Alle anderen Flash-PICs erlauben 25mA bei beiden Pegeln.

Der Gesamtstrom aller Pins darf aber eine Summe nicht überschreiten, die von PIC zu PIC etwas verschieden ist:

- 16F62x: 200 mA
- 12F6xx: 125 mA
- 16F87x: je 200 mA für PortA+B+E und PortC+D
- 16F7x: je 200 mA für PortA+B+E und PortC+D
- 16F84 High-Pegel: PortA: 50 mA / PortB: 100 mA
  16F84 Low-Pegel: PortA: 80 mA / PortB: 150 mA

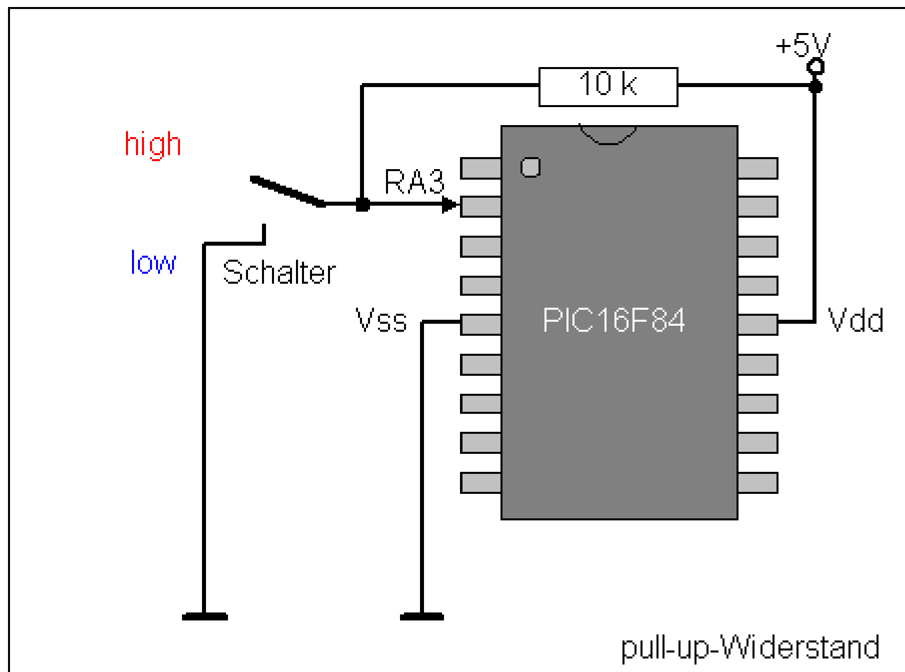
Beispiele für Eingangsbeschaltungen

Es folgen einige einfache Eingangsbeschaltungen für I/O-Portpins.



Die einfachste Signalquelle für ein Input-Pin ist ein Umschalter, der das Pin mit 0V oder 5V verbindet.

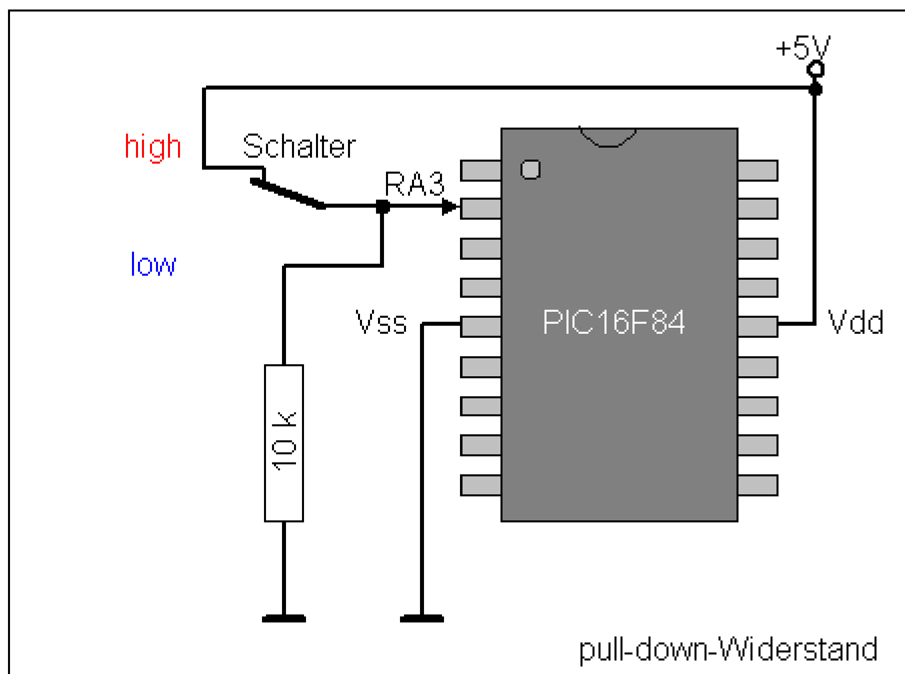
Falls ein Entprellen des Schalters nötig ist, dann erfolgt das hier (wie auch beiden weiteren Schalter-Beispielen) am Besten per Software.



Kostensparend ist der Einsatz eines einfachen Schließers oder Öffners anstelle des teureren Umschalters. In diesem Fall wird aber der Einsatz eines Hochziewiderstandes (pull-up) nötig. Dieser sorgt bei offenem Schalter für den High-Pegel.

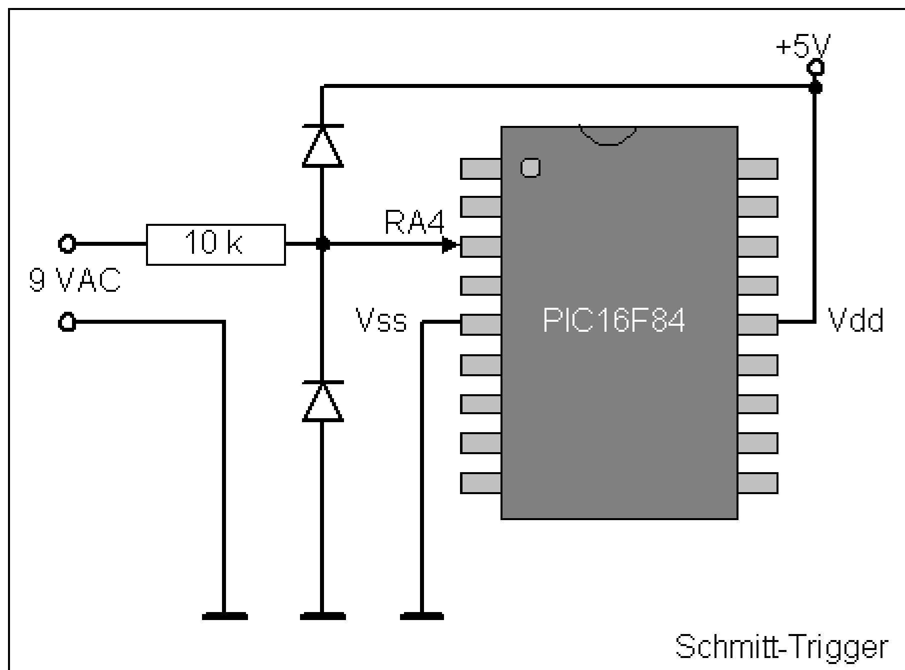
Der Wert des Widerstandes ist unkritisch (1k ... 100k).

Einige Portpins (z.B. das gesamte Port B) besitzen interne Hochziewiderstände, die eingeschaltet werden können. Das erspart den externen Widerstand.



Anstelle des Hochziehwiderstandes kann auch ein Runterziehwiderstand (pull-down) verwendet werden, wenn der Schalter mit +5V verbunden wird.

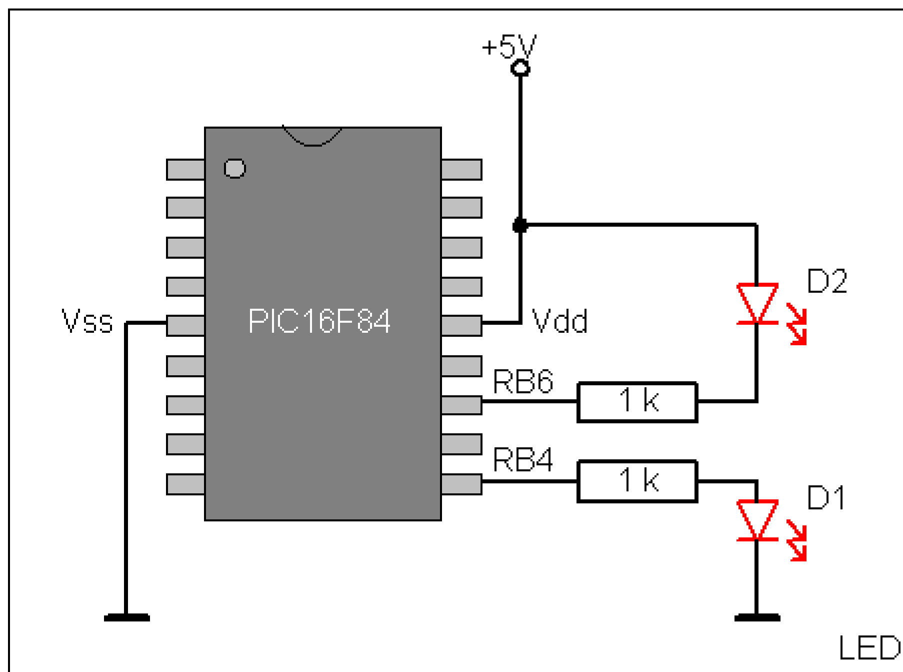
Der Schmitt-Trigger-Eingang RA4 kann wie ein normaler TTL-Eingang beschaltet werden, er bietet aber darüberhinaus auch die Möglichkeit nicht TTL-konforme Signalpegel zu verarbeiten.



Im nebenstehenden Beispiel soll ein Ablauf im Programm mit der 50 Hz Netzfrequenz synchronisiert werden (z.B. im Rahmen einer Phasenanschnittsteuerung o.ä.). Eine 9V-Wechselspannung wird über einen Vorwiderstand und 2 Shottkydioden in eine Spannung verwandelt, die zwischen 0V und 5V pendelt. Der ST-Eingang wandelt das Signal in saubere High und Low Werte um. Die Dioden dienem dem Eingang als Schutz vor Überspannung, der Wiederstand schützt seinerseits die Dioden vor Überstrom.

Beispiele für Ausgangsbeschaltungen

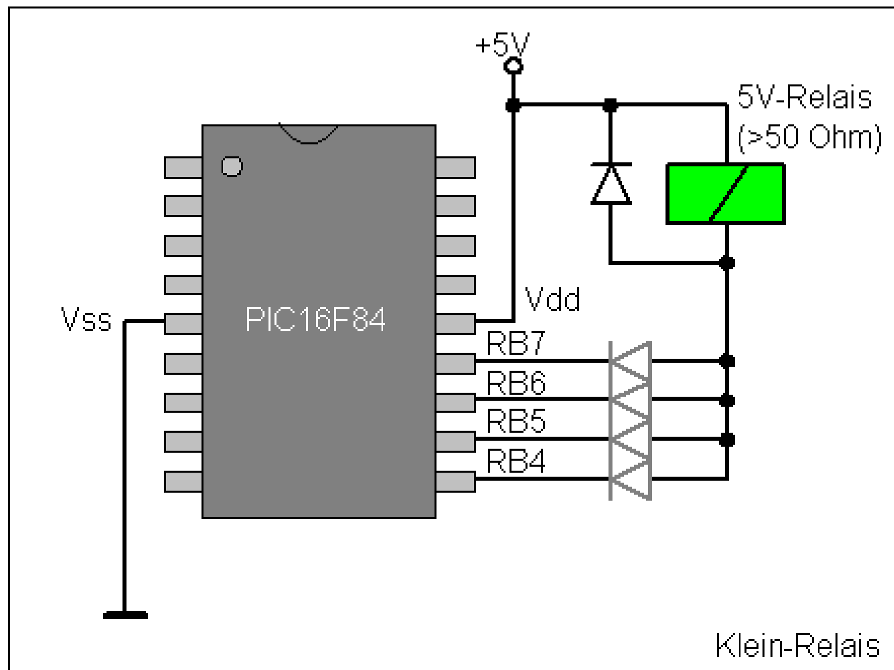
Im Folgenden werden einige Beispiele für Ausgangsbeschaltungen aufgezeigt.



Ein PIC-Ausgang kann einen Strom von 20 ... 25 mA bereitstellen. Das reicht aus, um kleinere Lasten direkt zu treiben.

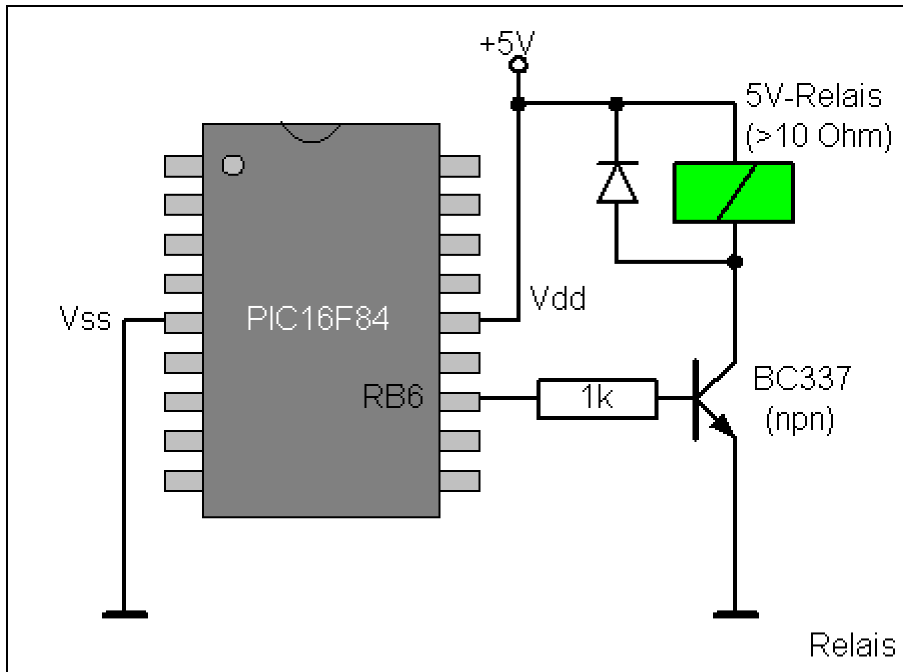
Im nebenstehenden Bild sind zwei Möglichkeiten für den Anschluß von Leuchtdioden gezeigt. D1 leuchtet auf, wenn RB4 Hig-Pegel führt, D2 leuchtet wenn RB6 Low-Pegel hat. Die Größe des Vorwiderstandes hängt von der gewünschten Leuchtstärke ab. Bei 1 Kiloohm fließen durch eine LED ca. 3,5 mA, was normalerweise ausreicht, aber nicht sehr hell ist. Bei Bedarf kann der Widerstand bis auf ca. 220 Ohm verringert werden (15 mA).

Die Werte gelten für rote (und in etwa auch für gelbe und grüne) LEDs. Blaue LEDs benötigen eine höhere Flußspannung. Hier müssen Widerstandswerte entsprechend den technischen Daten der LEDs festgelegt werden, die Wiederstände fallen hier relativ klein aus.



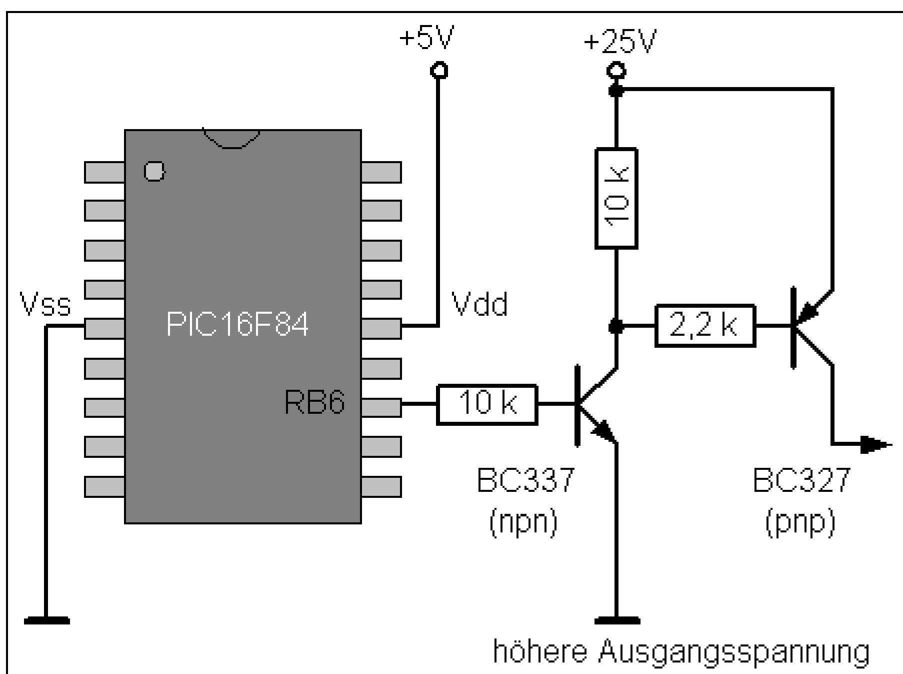
Um ein Relais anzusteuern reichen die max. 25 mA eines Pins meist nicht aus. 5V-Relais haben normalerweise Spulenwiederstände von 180 Ohm und darunter.

Eine Notlösung ist das Parallelschalten von Ausgängen. Damit lassen sich durchaus 100 mA erzeugen, es wird aber eine große Disziplin beim Programmieren erwartet. Alle zusammengeschalteten Pins müssen immer den gleichen Pegel führen, ansonsten werden einzelne Pins überlastet.
Die grauen Dioden dienen nur dem Schutz der Pins vor Programmierfehlern, und können von selbstbewußten Bastlern weggelassen werden. Die schwarze Diode (anti-parallel zum Relais) schützt den PIC vor den Induktionsspitzen hoher Spannung, die beim Abschalten des Relais auftreten.



Der elegantere und sicherere Weg zum Ansteuern niederohmiger Lasten ist ein Transistorverstärker. Ein Billigtyp wie der BC337 eignet sich schon, um Lasten bis zu 500 mA anzusteuern. Der Wert des Widerstands hängt vom nötigen Schaltstrom ab. Mit 1 Kiloohm ist man auf der sicheren Seite, wird nur ein Schaltstron von 100 mA benötigt, darf der Widerstand auch auf 4,7 kOhm anwachsen.

Die Diode ist für den Schutz des Transistors nötig.



Soll der PIC einen Verbraucher ansteuern, der eine höhere Spannung als Vdd benötigt, läßt sich auch das mit einem Transitorverstärker bewältigen.
Im gezeigten Beispiel sollte der Lastwiderstand groß genug sein, um den pnp-Transistor nicht zu überlasten (>100 Ohm).