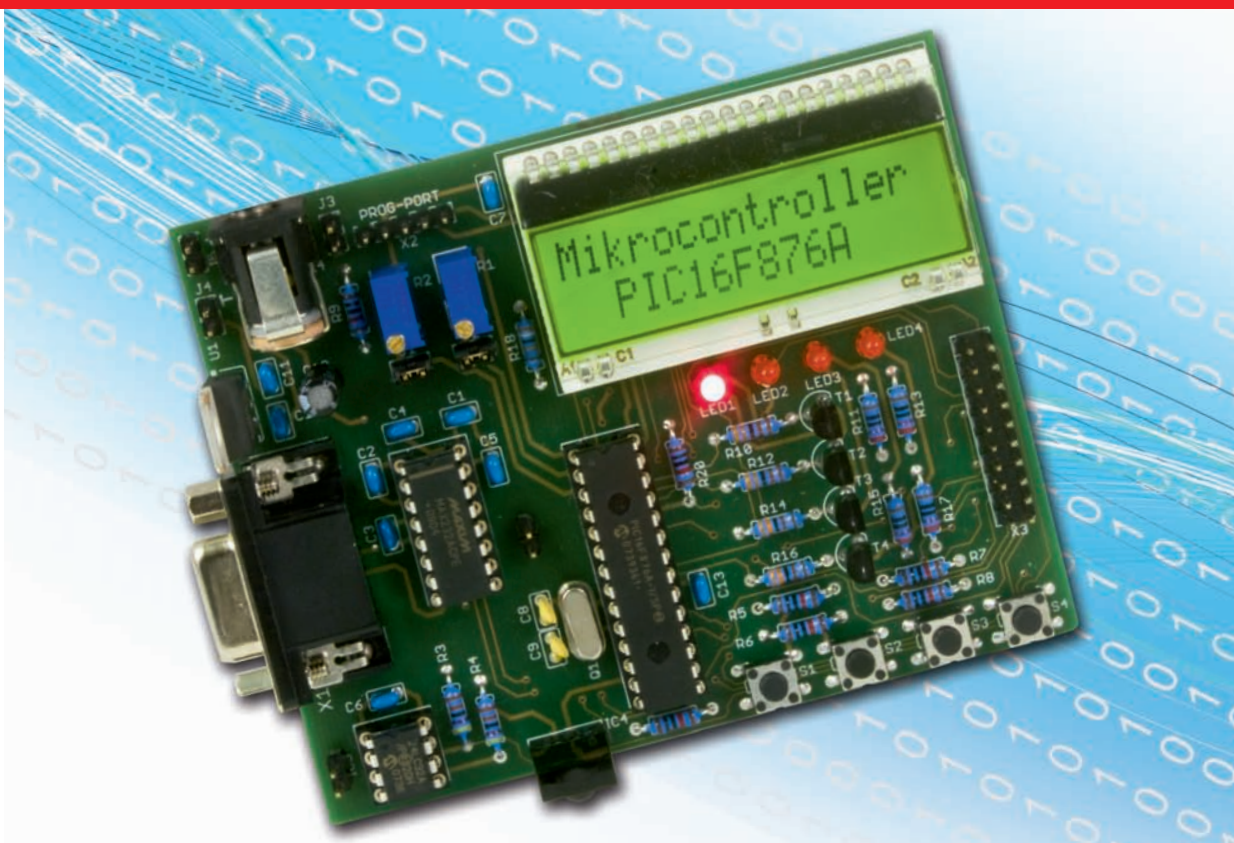
FRANZIS
PC & ELEKTRONIK



Michael Hofmann

Mikrocontroller für Einsteiger

Schaltungen entwerfen und Software programmieren

Auf CD-ROM:

- Beispielprogramme
- Layoutdaten
- Schaltpläne
- Datenblätter

Vorwort

Mikrocontroller sind in fast allen elektronischen Geräten zu finden. Dieses Buch zeigt, dass die Verwendung und Programmierung eines Mikrocontrollers nicht schwer ist. So manchem sträuben sich die Haare, wenn das Wort *Assembler* im Zusammenhang mit der Programmierung fällt. Nach der Lektüre dieses Buchs werden Sie feststellen, dass es nicht so kompliziert ist, wie Sie vielleicht vermutet haben.

Verschiedene Beispiele zeigen die Möglichkeiten eines Mikrocontrollers. Hierfür wird der *PIC16F876A* von Microchip verwendet. Dieser Typ verfügt über verschiedene Schnittstellen und Features und man kann damit eine große Bandbreite der Funktionalität vorstellen. Die Beispiele werden zeigen, wie Eingänge abgefragt und Ausgänge geschaltet werden. Sie werden auch lernen, wie man ein Display ansteuert, um Text und Daten anzuzeigen. Sie werden auch erfahren, wie man analoge Signale misst, diese in einem seriellen EEPROM speichert und anschließend über einen PC ausliest. Ein weiteres Beispiel zeigt die Steuerung der Ausgänge mit einer Infrarotfernbedienung. Ein Teil der Beispiele kann mit der Entwicklungsumgebung *MPLAB* simuliert werden. Daher wird keine funktionsfähige Hardwareschaltung benötigt. Da aber jeder Mikrocontroller irgendwann einmal in einer Schaltung eingesetzt werden soll, wird ein kleines Entwicklungs-Board vorgestellt, mit dem alle Beispiele in einer realen Umgebung getestet werden können. Um die Arbeit und die Fehlersuche für den Aufbau zu erleichtern, können Sie über meine Homepage (www.edmh.de) eine unbestückte Leiterplatte erwerben. Falls Sie eigene Erweiterungen vornehmen wollen, finden Sie das Layout im Eagle-Format auf der beiliegenden CD-ROM. Darauf finden Sie auch verschiedene Unterlagen zum Ausdrucken, z. B. eine Befehlsübersicht und Registerbeschreibungen. Für die Programmierung des Mikrocontrollers wird der *ICD2* von Microchip verwendet. Auf der CD-ROM finden Sie aber auch eine sehr einfache Schaltung, mit der der PIC über die serielle Schnittstelle programmiert werden kann.

Ich wünsche Ihnen viel Spaß und Erfolg bei der Mikrocontrollerprogrammierung. Für Kritik, Lob und Verbesserungsvorschläge bin ich immer offen und freue mich daher auch über eine Rückmeldung an meine E-Mail-Adresse info@edmh.de.

Michael Hofmann

Inhalt

4 Die Programmierschnittstelle

In den vorherigen Kapiteln wurden bereits einige Grundlagen für die Programmierung erklärt. In diesem Kapitel geht es nun darum, wie man den Mikrocontroller mit dem Programmiergerät verbinden und wie die endgültige Schaltung beschaffen sein muss, damit das geschriebene Programm in der fertigen Hardware funktioniert.

Um das Programm in den PIC zu laden, werden 3-4 Pins zur Verfügung gestellt. Diese können bei Bedarf auch nach dem Programmieren für andere Zwecke als Ein- oder Ausgang verwendet werden. Es handelt sich dabei um folgende Pins:

- PGC = Programming Clock = Takt für die Programmierdaten
- PGD = Programming Data = Daten für die Programmierung
- VPP = Programming Voltage = hohe Programmierspannung ca. 11 bis 13 V
- PGM = Low-Voltage Programming Enable = Freigabe-Pin, damit der PIC ohne die hohe Programmierspannung programmiert werden kann.

Neben diesen Pins werden noch weitere für die Versorgungsspannung (VDD und VSS) benötigt. Falls der Mikrocontroller nicht in einer Schaltung, an der eine Betriebsspannung anliegt, angeschlossen ist, muss die Versorgungsspannung für die Programmierung von dem Programmiergerät zur Verfügung gestellt werden.

4.1 Programmierung mit dem ICD2

Im Folgenden wird die Programmierung des PIC über die Pins PGC, PGD und VPP mit dem ICD2 erklärt. Auf der CD-ROM findet man eine kurze Beschreibung, wie man den Mikrocontroller mit der kleinen Selbstbauschaltung programmieren kann. Die Selbstbauschaltung kostet mit allen Bauteilen weniger als 2 €. Leider kann man mit dieser Schaltung nicht die Controller in der Schaltung debuggen und so die Hardwareschnittstellen prüfen. Allerdings ist sie völlig ausreichend, wenn man z. B. ein Lauflicht erfolgreich programmiert und simuliert hat und dann die Software nur noch in den PIC laden möchte. Die Schaltung besteht nur aus einer Handvoll einfacher Bauteile und ist in kurzer Zeit auf einer Lochrasterplatine aufgebaut. Beim Aufbau der Schaltung sollte man auf möglichst kurze Leitungen zum Mikrocontroller achten, da es hier sehr leicht zu Störungen kommen kann und dann der Mikrocontroller fehlerhaft programmiert wird. Man muss sich im Klaren sein, dass man von dieser einfachen Schaltung nicht den Funktionsumfang eines professio-

nellen Programmier- und Debug-Geräts verlangen kann. Möchte man sich tiefer mit der Programmierung der unterschiedlichen Controllertypen auseinandersetzten, kann man Details der Bedienungsanleitung des ICD2 entnehmen. Falls man Genaueres über die Programmierung des Flash-Speichers lernen möchte, sollte man sich das Dokument *FLASH Memory Programming Specification_PIC16F87XA* durchlesen.

Um Ärger beim Programmieren und Debuggen aus dem Weg zu gehen, sollte man nach Möglichkeit die Pins *PGC* und *PGD* ausschließlich für Programmierzwecke verwenden. Ist es aus irgendwelchen Gründen trotzdem erforderlich, diese Pins für andere Zwecke zu benutzen, sollte man versuchen, weniger wichtige Features (z. B. Warnung, wenn die Batteriespannung einen bestimmten Wert unterschreitet) oder einfache passive Schaltungen (z. B. Taster) an diese Pins anzuschließen. Im schlimmsten Fall kann der Mikrocontroller in der Schaltung nicht debugged werden. Der Pin *PGM* ist zwar nicht für die Programmierung erforderlich, man sollte aber auch hier einige Punkte beachten. Über diesen Pin wird dem Mikrocontroller mitgeteilt, dass das Programmieren mit geringer Spannung, also nicht mit 11 bis 13 V an VPP, erfolgt. Um den Baustein mit lediglich 5 V zu programmieren, muss zuerst das Konfigurationsbit *Enable Low Voltage Programming* gesetzt werden. Liegt dann ein High-Pegel am Pin *PGM* an, kann der PIC auch ohne hohe Programmierspannung beschrieben werden. Dies kann z. B. sinnvoll sein, wenn der Programmcode im fertigen Gerät vom Kunden aktualisiert werden soll. Der Kunde verfügt in der Regel nicht über ein Programmiergerät, um einen Mikrocontroller zu programmieren. Um sich keinen Ärger bei der Programmierung einzuhandeln, sollte man versuchen, diesen Pin nur als Ausgang zu verwenden, der auf einen Eingang eines externen Bauteils geht. Zusätzlich kann man noch einen Pull-Down-Widerstand von ca. 10 kΩ anschließen, damit der Pin erst auf High-Pegel gezogen wird, wenn dies auch gewollt ist. Schließt man z. B. den Ausgang eines externen Schwellwertschalters an diesen Pin an, kann man nie sicher sein, welcher Pegel zum Programmierzeitpunkt an diesem Pin anliegt.

Über den Eingang *MCLR/VPP* wird die hohe Programmierspannung an den Mikrocontroller angelegt. Ist diese Spannung über 9 V, wird der internen Hardware mitgeteilt, dass der Flashspeicher programmiert werden soll. Nach dem Programmieren wird dieser Pin als Reset-Pin verwendet. Wird der Pin also während des Betriebs auf Low-Pegel gelegt, wird ein Reset ausgeführt und das Programm beginnt von vorn. Um nun einen unbeabsichtigten Reset zu vermeiden, muss dieser Pin immer auf High-Pegel liegen. Aber Vorsicht: Der Pin darf nicht direkt mit der Betriebsspannung VDD verbunden werden, da sonst auch die hohe Programmierspannung an dem Spannungsversorgungspin anliegen würde und den Baustein zerstören könnte. VPP muss daher über einen Widerstand mit VDD verbunden werden. Der Widerstand hat in der Regel einen Wert von 10 kΩ. Über einen Taster oder Jumper nach Masse kann so ein kontrollierter Reset ausgeführt werden. Die vollständige Beschaltung für eine sichere Programmierung kann *Abb. 4.1* entnommen werden.

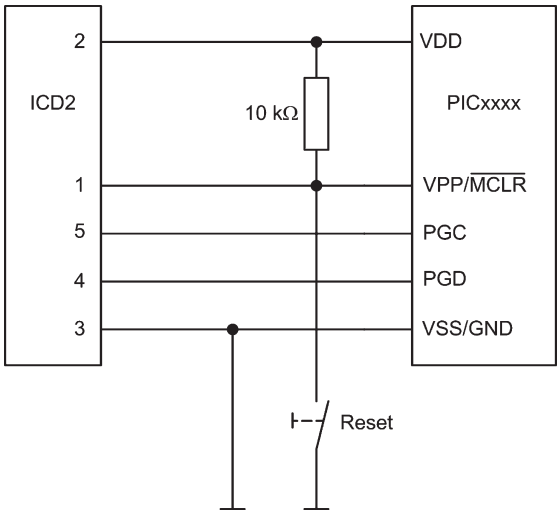


Abb. 4.1: Anschluss des ICD2 an den Mikrocontroller

4.2 Ablauf der Programmierung

Bei der Programmierung werden verschiedene Schritte durchlaufen. Um ein neues Programm in den Flashspeicher des Mikrocontrollers zu schreiben, muss dieser zuvor gelöscht sein. Nach dem Beschreiben möchte man auch sicher sein, dass die Daten richtig im Programmspeicher stehen. Daher werden die geschriebenen Daten nochmals ausgelesen und mit den Daten auf dem PC verglichen.

Selbstverständlich kann man jeden Schritt einzeln ausführen und so die Daten in den Mikrocontroller schreiben und prüfen. In der Regel läuft die Programmierung jedoch automatisiert ab und die nötigen Schritte werden nacheinander ausgeführt. Damit eine Datenübertragung vom PC in den PIC ermöglicht wird, muss zuerst der ICD2 von MPLAB initialisiert und eine Verbindung hergestellt werden. Nach dem Herstellen der Verbindung wird von MPLAB geprüft, ob der richtige Mikrocontroller angeschlossen ist. Wurde bei den Einstellungen ein falscher Typ ausgewählt, z. B. PIC16F876 statt PIC16F876A, wird an dieser Stelle eine Warnung ausgegeben. In den meisten Fällen ist dies nicht schlimm und der PIC kann trotzdem programmiert werden. Es kann aber in Einzelfällen zu Problemen bei der Programmausführung kommen. Bei MPLAB findet man die nötigen Befehle für das Programmieren im Menü *Programmer*. Mit den Befehlen können die folgenden Aktionen ausgeführt werden:

Erase Part: Bei diesem Befehl wird der interne Flash-Speicher des Mikrocontrollers gelöscht. Im Speicher stehen nach dem Löschvorgang nur Einsen.

Blank Check: Um zu prüfen, ob das Löschen erfolgreich war, führt man danach einen sogenannten *Blank Check* durch. Dabei wird der Inhalt des Flash-Speichers gelesen und geprüft, ob an jeder Speicherstelle eine Eins steht. Ist dies nicht der Fall, befindet sich noch ein altes Programm im Chip. Bei einer gestörten Übertragung, z. B. durch ein zu langes Kabel vom Programmiergerät zur Schaltung, kann es vorkommen, dass nicht alle Speicherzellen ordnungsgemäß gelöscht werden. Ist dies der Fall, sollte man die Verbindung prüfen und den Vorgang wiederholen.

Program: Nachdem keine Daten mehr im Programmspeicher vorhanden sind, kann man das Programm mit dem Befehl *Program* in den Flash-Speicher laden. Das Programm muss natürlich zuvor geladen und übersetzt werden. Bei der Programmierung werden die Konfigurationsbits erst ganz zum Schluss in den PIC programmiert. Nach der Programmierung wird automatisch die Überprüfung des Programmcodes und der Konfigurationsbits gestartet.

Read: Mit dem Befehl *Read* wird der gesamte Speicherinhalt gelesen, sofern der Chip nicht mit dem Konfigurationsbit *Code Protect* gegen Auslesen geschützt ist. Im Fall eines leeren Mikrocontrollers wird aus jeder Programmspeicherzelle der Wert 0x3FFF ausgelesen. Steht ein Programm im Speicher, sieht man den binär codierten Ausführungscode und die zurückgewandelten Assemblerbefehle (Disassembly). Das Ganze ist natürlich ohne den Kommentar und die Variablennamen, die im Quellcode angegeben wurden. Um den Programmcode zu sehen, muss man eventuell das Fenster über das Menü *View* und den Punkt *Program Memory* sichtbar schalten.

Verify: Um nicht jede Zeile einzeln mit dem programmierten Code zu vergleichen, wählt man den Befehl *Verify* aus und lässt MPLAB die Daten automatisch vergleichen und prüfen.

Nachdem der Programmspeicher erfolgreich beschrieben wurde, kann man die Betriebsspannung entfernen und wieder anlegen, ohne dass die Daten verloren gehen.

4.3 Die Konfigurationsbits

Auf das Register, in dem die Konfigurationsbits stehen, kann man nur während des Programmierens zugreifen. Nachdem der Chip programmiert wurde, können diese Bits nicht mehr verändert werden. Um eine Änderung an diesen Bits vorzunehmen, muss zuerst der gesamte Chip gelöscht und anschließend mit geänderten Einstellungen neu programmiert werden. Dies ist auch sinnvoll, da man sonst nachträglich das Bit für den Ausleseschutz zurücksetzten könnte, um so die Daten aus dem Speicher auszulesen.

Die Konfigurationsbits können entweder direkt im Code gesetzt werden oder über das Menü *Configure* im Untermenü *Configuration Bits…* . Es ist empfehlenswert, die Bits im Code zu setzen. So vergisst man auch zu einem späteren Zeitpunkt nicht, den Aus-

leseschutz zu aktivieren, und hat gleich die entsprechenden Einstellungen für eine Serienproduktion. Um die Konfigurationsbits im Code zu setzen, kann man die Voreinstellungen aus der *.inc*-Datei verwenden. Im folgenden Abschnitt sind die möglichen Einstellungen aus der Include-Datei ebenfalls mit angegeben. Man erkennt sie durch den Unterstrich vor dem Namen.

Das Register, in dem die Konfigurationsbits gesetzt werden, hat eine Breite von 14 Bits. Um die Konfigurationsbits im Code zu setzen, benutzt man das Schlüsselwort *__CONFIG*. Nach diesem Wort werden die Konfigurationsbits durch ein *&* verknüpft aufgelistet.

Beispiel: __CONFIG _CP_OFF & _WDT_OFF & _BODEN_OFF & ...

4.3.1 Oszillator

Da der Mikrocontroller ohne Takt nicht arbeiten kann, hat man verschiedene Möglichkeiten, das Rechenwerk mit einem Takt zu versorgen. Bei der Auswahl hängt es von den Anforderungen an die zeitliche Genauigkeit ab, welcher Oszillatortyp zu verwenden ist. Will man nur in Abhängigkeit von Eingangssignalen verschiedene Ausgänge schalten, reicht ein einfacher RC-Oszillator oft aus (RC = Widerstand-Kondensator-Oszillator). Soll aber mit dem Mikrocontroller eine Stoppuhr für Geschwindigkeitsmessungen aufgebaut werden, sind die Anforderungen an die zeitliche Auflösung deutlich höher und man sollte einen Quarzoszillator verwenden.

Man hat vier verschiedene Einstellmöglichkeiten:

```
RC: Resistor/Capacitor (_RC_OSC)
LP: Low-Power Crystal (_LP_OSC)
XT: Crystal/Resonator (_XT_OSC)
HS: High-Speed Crystal/Resonator (_HS_OSC)
```

Soll der Mikrocontroller mit einem Quarzresonator betrieben werden, verwendet man für Taktfrequenzen unter 4 MHz den XT-Mode. In der Regel ist diese Taktfrequenz für eine Vielzahl von Schaltungen ausreichend und man bekommt sehr günstige Quarze, die eine ausreichende Genauigkeit liefern. Für schnellere Anwendungen, in denen mehr Rechenleistung benötigt wird, stellt man für den Bereich zwischen 4 und 20 MHz den Hochgeschwindigkeitsmodus (HS-Mode) ein. Für die Verwendung eines Quarzes sind noch zwei zusätzliche Kondensatoren an den Pins OSC1 und OSC2 nötig. Bei einem 4-MHz-Quarz sollten die Werte der Kondensatoren zwischen 15 und 68 pF liegen.

Der Modus LP wird für Quarze mit geringer Frequenz unter 500 kHz verwendet. Durch die niedrige Taktfrequenz ist auch die Stromaufnahme des Mikrocontrollers gering. Oft wird der Mikrocontroller in einen Schlafmodus versetzt und zyklisch aufgeweckt, um zu prüfen, ob neue Daten vorhanden sind. Der Prozessor benötigt im

Schlafzustand nicht die volle Rechenleistung und kann daher mit einem geringen Takt arbeiten. Dies hat ebenfalls den Vorteil, dass interne Zähler nicht so schnell hochgezählt werden und man längere Zeiten mit kleineren Registern erreichen kann. In der Regel wird für den Low-Power-Modus ein sogenannter *Uhrenquarz* mit einer Frequenz von 32,768 kHz verwendet. Der Frequenzwert sieht auf den ersten Blick ungewöhnlich aus. Teilt man diesen Wert aber 15-mal durch 2, erhält man einen Takt von genau einer Sekunde – daher auch der Name *Uhrenquarz*.

Soll eine kostenoptimierte Schaltung aufgebaut werden, die keine hohen Anforderungen an die Oszillatorgenauigkeit stellt, kann ein externer RC-Oszillator verwendet werden. Leider kann die Frequenz nicht exakt berechnet werden und schwankt in Abhängigkeit von der Betriebsspannung und der eingesetzten Bauteile. Bei einem Widerstand von 5 kΩ und einem Kondensator von 100 pF ergibt sich eine Taktfrequenz von ca. 1,3 MHz bei 5 V Versorgungsspannung. Es gibt auch einige PIC-Mikrocontroller, die einen internen RC-Oszillator besitzen. Dieser ist vom Hersteller abgeglichen und kann ohne externe Bauteile verwendet werden. Bei diesen Mikrocontrollern gibt es ein spezielles Register, in dem der Abgleichwert steht. Über dieses Register kann auch die Taktfrequenz in gewissen Bereichen angepasst werden. Man muss allerdings darauf achten, dass man dieses Register vor dem Löschen des Mikrocontrollers sichert und nach der Programmierung wieder in den Mikrocontroller zurückschreibt, damit man wieder mit dem abgeglichenen Oszillator arbeiten kann.

Der Mikrocontroller muss nicht zwingend mit einem eigenen Oszillator versorgt werden. Der Takt kann auch von einem externen Baustein eingespeist werden. Der Takt wird dann an Pin OSC1 angelegt und Pin OSC2 bleibt offen.

4.3.2 Watchdog-Timer

Watchdog heißt übersetzt „Wachhund“ – und so verhält sich auch dieser Timer. Er „hält Wache“ über den Mikrocontroller. Ist dieser Timer aktiviert, läuft ein Countdown im Hintergrund. Ist dieser Countdown abgelaufen, wird der Prozessor zurückgesetzt und das Programm beginnt von vorn. Dies will man im normalen Betrieb natürlich verhindern und muss daher den Timer wieder zurücksetzen und von vorn starten lassen. Verfängt sich der Prozessor in einer unerwarteten Endlosschleife, wird der Mikrocontroller automatisch zurückgesetzt, ohne dass der Benutzer eine Reset-Taste drücken muss. Um den Watchdog-Timer zu aktivieren, setzt man das entsprechende Bit mit _WDT_ON. Deaktiviert wird der Timer durch _WDT_OFF.

4.3.3 Power-Up-Timer

Durch den Power-Up-Timer wird der Mikrocontroller nach dem Anlegen der Versorgungsspannung für die Dauer von 72 ms im Reset-Status gehalten. Der Prozessor beginnt demnach nicht sofort mit der Abarbeitung des Programms, sondern wartet noch

eine kurze Zeit. Das ist sinnvoll, wenn die Betriebsspannung an den Chip angelegt wird. Da an der Versorgungsspannung des Geräts oft recht große Kondensatoren anliegen, die erst vollständig geladen werden müssen, dauert es eine gewisse Zeit, bis die volle Betriebsspannung erreicht ist. Um sicherzugehen, dass der Prozessor mit der vollen Betriebsspannung versorgt wird, schaltet man den Power-Up-Timer ein. Dadurch ist auch sichergestellt, dass die richtige Referenzspannung für eine Analog-Digital-Wandlung anliegt. Dauert der Anstieg der Versorgungsspannung auf die volle Betriebsspannung 1 ms, würden bei einem 4-MHz-Takt bereits ca. 1.000 Befehle bearbeitet werden. Daher sollte man bei Problemen am Anfang des Programms zuerst prüfen, ob stabile Zustände am Mikrocontroller vorliegen und der Mikrocontroller nicht für eine kurze Zeit außerhalb der Spezifikation betrieben wird. Will man den Power-Up-Timer aktivieren, geht dies im Programmcode durch die Konstante _PWRTE_ON. Über _PWRTE_OFF wird der Timer ausgeschaltet.

4.3.4 Brown-Out Detect

Während des normalen Betriebs sollte immer eine ausreichend hohe und konstante Spannung an dem Mikrocontroller anliegen. Sollte die Versorgungsspannung durch irgendwelche äußeren Einflüsse unter einen bestimmten Wert fallen, würde man den Prozessor im nicht stabilen Bereich betreiben und es könnte zu Fehlern in der Bearbeitung der Befehle kommen. Dieses Ereignis kann z. B. durch einen kurzzeitigen Kurzschluss an einer nach außen geführten Schnittstelle hervorgerufen werden. Um solche Ereignisse zu erkennen, verwendet man den *Brown-Out Detect*. Fällt die Betriebsspannung für eine intern festgelegte Zeit (ca. 100 µs) unter eine Schwelle (ca. 4 V), wird ein Reset ausgelöst und das Programm beginnt nach einer Wartezeit von ca. 72 ms von vorn. Aktiviert wird dieses Feature über _BODEN_ON, deaktiviert durch _BODEN_OFF.

4.3.5 Low Voltage Program

Durch das Bit *Low Voltage Program* wird es dem Benutzer ermöglicht, den Mikrocontroller mit einer geringen Spannung in Höhe der Versorgungsspannung zu programmieren. Ist das Bit durch _LVP_ON aktiviert, hat der Pin RB3/PGM die Funktion PGM. Das heißt: Wird jetzt ein High-Pegel am Pin PGM angelegt, kann der Mikrocontroller programmiert werden. Wird das Bit durch _LVP_OFF deaktiviert, verhält sich Pin RB3 wie ein digitaler I/O-Pin und es muss zum Programmieren des Speichers die hohe Spannung von ca. 12 V an den Pin MCLR angelegt werden.

4.3.6 Data EE Read Protect

Stehen Daten im internen EEPROM, die von anderen Benutzern nicht gelesen werden sollen, setzt man das Bit 8 (CPD) im Konfigurationsregister auf *0*. Dies kann über die

Konstante _CPD_ON im Quellcode geschehen. Ist die Codeprotektion aktiviert, können die Daten nicht mehr von außen gelesen werden und stehen nur noch dem internen Programm zur Verfügung. Auf diese Weise kann z. B. ein Passwort gegen unbefugtes Auslesen geschützt werden. Wünscht man den Ausleseschutz nicht, gibt man das Lesen durch die Angabe von _CPD_OFF frei.

4.3.7 Flash Program Write

Der Flash-Speicher im PIC kann auch für das Speichern von Daten während der Programmausführung verwendet werden. Auf der einen Seite kann dies eine elegante Lösung sein, den internen Speicher besser auszunutzen. Auf der anderen Seite birgt diese Möglichkeit aber auch die Gefahr, dass der Programmcode durch einen Fehler im Programm überschrieben werden kann und das Programm dadurch nicht mehr ausführbar ist. Der Mikrocontroller müsste dann neu programmiert werden. Um das Überschreiben des Programmcodes zu verhindern, wird der Flash-Speicher in vier Teile unterteilt, die durch die Konfigurationsbits vor dem unbeabsichtigten Überschreiben geschützt werden. Über die Bits WRT0 und WRT1 wird der zu schützende Bereich festgelegt. Für den PIC16F876A gelten folgende Bereiche:

- _WRT_OFF: Der Schreibschutz ist deaktiviert und der gesamte Speicherbereich kann beschrieben werden.
- _WRT_256: Der Adressbereich zwischen 0x0000 und 0x00FF ist vor dem Überschreiben geschützt. Der Speicher zwischen 0x0100 und 0x1FFF kann als Datenspeicher genutzt werden.
- _WRT_1FOURTH: Das erste Viertel (0x0000 bis 0x07FF) kann nicht beschrieben werden. Der Bereich zwischen 0x0800 und 0x1FFF kann über das Kontrollregister EECON beschrieben werden.
- _WRT_HALF: Nimmt das Programm die Hälfte des Speichers (0x0000 bis 0x0FFF) in Anspruch, kann dieser über die Bits WRT1 = 0 und WRT0 = 0 geschützt werden. Die obere Hälfte des Speichers (0x1000 bis 0x1FFF) kann dann noch für Daten verwendet werden.

4.3.8 Code Protect

Hat man ein Gerät entwickelt, will man in der Regel die Software gegen unbefugtes Auslesen schützen. Dafür setzt man das Bit *Code Protection* im Konfigurationsregister auf *0*. Jetzt kann der im Chip gespeicherte Programmcode nicht mehr ausgelesen werden und es wird verhindert, dass eine Kopie des Programms erstellt werden kann. Sollen Änderungen am Code vorgenommen werden, muss der Originalcode bearbeitet und erneut in den Speicher geladen werden. Der Leseschutz wird im Quellcode durch _CP_ALL aktiviert und über _CP_OFF deaktiviert.

4.3.9 Konfigurationsbits im Überblick

Werden keine Konfigurationsbits im Quellcode angegeben oder wird eines vergessen, steht im Konfigurationsregister an dieser Stelle eine *1* und die Schutzeigenschaften sind dadurch ausgeschaltet. Der Schreib- und Leseschutz muss daher bei Bedarf explizit eingeschaltet werden.

- _RC_OSC / _HS_OSC / _XT_OSC / _LP_OSC: Auswahl des Oszillatortyps
- _WDT_ON / _WDT_OFF: Aktivieren/Deaktivieren des Watchdogtimers
- _PWRTE_ON / _PWRTE_OFF: Aktivieren/Deaktivieren des Power-Up-Timers
- _BODEN_ON / _BODEN_OFF: Aktivieren/Deaktivieren des Brown-Out-Resets
- _LVP_ON / _LVP_OFF: Aktivieren/Deaktivieren des Low-Voltage-Programiermodus
- _CPD_ON / _CPD_OFF: Aktivieren/Deaktivieren des Leseschutzes des EEPROM
- _WRT_OFF / _WRT_256 / _WRT_1FOURTH / _WRT_HALF: Auswahl des zu schützenden Flash-Speicherbereichs
- _CP_ALL / _CP_OFF: Aktivieren/Deaktivieren des Leseschutzes des Programmspeichers

Die beschriebenen Konstanten beziehen sich alle auf den PIC16F876A. Bei anderen Typen weichen die Namen teilweise ab oder sind nicht vorhanden. Die vordefinierten Konstanten können der *Include*-Datei des Mikrocontrollers entnommen werden.

4.4 OTP-Typ

Bei der bisherigen Beschreibung der Programmierung wurde immer davon ausgegangen, dass der Programmcode im internen Flash-Speicher des Mikrocontrollers gespeichert wird. Dieser Speicher ist aber im Vergleich zu einem Speicher, der nur einmal programmiert werden kann, relativ teuer. Daher gibt es von vielen Mikrocontrollern eine Variante mit einem Speicher, der nur ein einziges Mal beschrieben werden kann. Soll dann ein anderes Programm verwendet werden, muss man den Baustein auslöten und einen neuen mit dem aktuellen Programm einlöten. OTP bedeutet daher *One Time Programmable*. Bei Microchip erkennt man die OTP-Typen durch ein „C“ im Namen (z. B. PIC16C622A).

7 Die Timer

Ein wichtiger Bestandteil eines Mikrocontrollers sind die Timer. Timer sind getaktete Zähler, die durch einen Takt hoch- oder heruntergezählt werden. Timer können für die unterschiedlichsten Zwecke verwendet werden. So wird z. B. für den Watchdog ein Timer benötigt oder man kann einen Timer für eine Warteschleife verwenden. Es können Zeiten zwischen zwei Impulsen bestimmt oder es können Impulse mit einer vorgegebenen Dauer ausgegeben werden. Jeder Timer verfügt über ein Register, in dem ein Wert gespeichert wird, der dann aufwärts oder abwärts gezählt wird. Bei einem Über- oder Unterlauf dieses Registers kann dann ein Interrupt ausgelöst werden. Einen Timer könnte man sich auch selbst programmieren, indem man ein beliebiges Register mit einem Wert lädt und dann in einer Schleife so lange herunterzählt, bis der Wert 0 ist. Dies ist eine einfache Methode, kurze Verzögerungszeiten in einem Programm zu implementieren. Bei dem folgenden Beispiel wird Pin RA2 für die Dauer von ca. 100 µs auf einen High-Pegel gesetzt. Für den Fall, dass der Prozessor mit einem Takt von 4 MHz versorgt wird, muss man in das Register *COUNTER* den dezimalen Wert 32 laden. Das ist ungefähr 100/3, da der Befehl *decfsz* 1 Befehlszyklus (= 1 µs) und der Befehl *goto* 2 Befehlszyklen (= 2 µs) benötigt.

Beispiel:

```
bsf PORTA, 2        ;setze Pin RA2 auf High-Pegel
movlw D‘32‘         ;lade das Register Counter mit 32
movwf COUNTER
decfsz COUNTER, F   ;verringere den Wert um 1
goto $-1            ;zähle so lange herunter, bis der Wert 0 ist
bcf PORTA, 2        ;setze Pin RA2 auf Low-Pegel
```

Sehr kurze Verzögerungen von wenigen Mikrosekunden lassen sich sehr gut über mehrere *nop*-Befehle realisieren. Ein *nop* benötigt für die Bearbeitung einen Befehlszyklus und entspricht bei einem 4-MHz-Takt einer Bearbeitungszeit von einer Mikrosekunde. Mit diesem Beispiel kann eine Zeitverzögerung bis ca. 765 µs (= 3 * 255) ohne den Einsatz eines Timers problemlos verwirklicht werden. Wenn eine Zeit größer als 765 µs implementiert werden soll, müsste man auf diese Weise schon zwei Register verwenden.

7.1 Der 8-Bit-Timer (Timer0)

Um längere Zeiten zu verwirklichen, ist es sinnvoll, einen Timer zu verwenden. Der große Vorteil eines Timers ist, dass der Takt, mit dem das Timer-Register hoch- oder

heruntergezählt wird, über einen vorgeschalteten Teiler (Prescaler) läuft. Das Register wird nun nicht mehr im Rhythmus des Befehlstakts verändert, sondern mit einem geringeren Takt. Das Teilerverhältnis kann je nach verwendetem Teiler zwischen 1:2 und 1:256 gewählt werden. Wird das Prescaler-Verhältnis auf 1:256 eingestellt und der 8-Bit-Timer mit dem hexadezimalen Wert 0xFF (255) geladen, entspricht dies bereits einer Verzögerung von 65.280 µs = 65,280 ms. Im folgenden Beispiel wird der Timer0 für eine Zeitverzögerung von 10 ms verwendet.

Beispiel:

```
_BANK_1                ;umschalten auf Bank 1
movlw b‘11000111‘      ;Prescaler 1:256 für Timer0
movwf OPTION_REG
_BANK_0                ;umschalten auf Bank 0
movlw b‘00100000‘      ;Interrupts ausschalten und TMR0-Flag
movwf INTCON           ;zurücksetzen
bsf PORTA, 2           ;Pin RA2 auf High-Pegel setzen
_BANK_2
movlw D‘217‘
movwf TMR0             ;starten des Timers
_BANK_0
btfss INTCON, TMR0IF   ;prüfen, ob der Timer übergelaufen ist
goto $-1
bcf PORTA, 2           ;Pin RA2 auf Low-Pegel setzen
```

Es wird zuerst im Register *OPTION_REG* der Vorteiler (Prescaler) auf den Wert 1:256 gesetzt und dem Timer0 zugeordnet. Man kann den Timer0 auch für den Watchdog verwenden, allerdings ist eine gleichzeitige Verwendung als Timer im Programm und als Watchdogtimer ausgeschlossen. Als Nächstes werden die Interrupts ausgeschaltet. Das entsprechende Interruptflag wird trotzdem gesetzt, es wird aber nicht in die Interruptroutine gesprungen. Man muss dann in einer Schleife prüfen, wann das Flag TMR0IF gesetzt wird. In diesem Fall ist das Register von 0xFF auf 0x00 zurückgesprungen. Vor dem Beschreiben des *Timer*-Registers wird noch Pin RA2 auf high gesetzt. Um den Timer mit der richtigen Zeit zu initialisieren, muss man den entsprechenden Wert in das Register *TMR0* schreiben. Da der Timer0 bei jedem Takt inkrementiert (hochgezählt) wird, muss der Wert nach Formel 7.1 berechnet werden:

$$TMR0 = 256 - \frac{f_{OSZ} \cdot t}{4 \cdot PS} \qquad \textbf{Formel 7.1}$$

TMR0: Zählerwert für Timer0
f_{OSZ}: Oszillatorfrequenz
t: Zeitverzögerung
PS: Wert des Vorteilers (Prescale)

Soll eine Zeitverzögerung von 10 ms erreicht werden, ergibt sich folgender Wert:

$$TMR0 = 256 - \frac{4MHz \cdot 10ms}{4 \cdot 256} = 16{,}94 \approx 217$$

Dies ist der Wert, der auch im Beispielprogramm in das Register eingetragen wurde. Nachdem der Wert nun im Register *TMR0* steht, läuft der Timer und man wartet, bis das Interruptflag *TMR0IF* gesetzt wurde. Danach wird Pin RA2 wieder auf Low-Pegel zurückgesetzt und man hat einen High-Impuls mit einer Dauer von 10 ms generiert. Da der Prescaler auf 1:256 gesetzt ist, ist die kleinste zeitliche Auflösung 256 µs, da man nur ganze Werte in das Register *TRM0* schreiben kann.

7.2 Der 16-Bit-Timer (Timer1)

Das Timer1-Modul verfügt über zwei 8-Bit-Register (TMR1L und TMR1H). Auch dieser Timer zählt, wie der Timer0, aufwärts und löst einen Interrupt aus, wenn der Registerwert von 0xFF in TMR1L und 0xFF in TMR1H auf den Wert 0x0000 springt. Durch die beiden 8-Bit-Register (= 16 Bit) kann dieser Timer bis 65535 (= 0xFFFF) zählen, bevor er überläuft. Allerdings kann der Wert für den Vorteiler nicht so hoch eingestellt werden wie bei Timer0. Bei dem Timer1 ist ein maximaler Teilerfaktor von 1:8 möglich. Das bedeutet, dass bei einem 4-MHz-Takt eine maximale Zeitverzögerung von ca. 0,524 Sekunden möglich ist. Auch die zeitliche Auflösung ist mit 8 µs relativ genau. Das Timer1-Modul kann in zwei verschiedenen Betriebsarten betrieben werden. Die erste Betriebsart ist die Verwendung als Timer, mit dem man Wartezeiten generieren oder die Zeit zwischen zwei Ereignissen (z. B. zwei Impulsen) messen kann. In der zweiten Betriebsart kann man die beiden Register als Counter verwenden und so die externen Impulsen zählen. Haben diese Impulse eine definierte Dauer, ist es auch möglich, eine Wartezeit über diesen externen Takt zu generieren. Der Inhalt der Register wird bei jeder steigenden Flanke um 1 erhöht.

Das folgende Beispiel zeigt den Programmcode für die Programmierung eines Lauflichts. Dazu werden die LEDs nacheinander für die Dauer von 0,4 Sekunden eingeschaltet.

Beispiel:

```
start                   ;Beginn des Programms
   ;Initialisierungen
   _BANK_0
   clrf PORTA           ;lösche alle Ausgangsports
   clrf PORTB
   clrf PORTC
   _BANK_1
   movlw b'00000110'    ;alle Pins von Port A sind digitale Ein-
   movwf ADCON1         ;oder Ausgänge
   movlw b'11110011'    ;setze RA3 und RA2 als Ausgang und den
   movwf TRISA          ;Rest als Eingang
   movlw b'11000000'    ;setze RB7 und RB6 als Eingang und den
   movwf TRISB          ;Rest als Ausgang
   movlw b'11111111'    ;alle Pins von Port C sind Eingänge
   movwf TRISC
   clrf INTCON          ;Interrups ausschalten
```

```
   clrf PIE1
   _BANK_0
   clrf PIR1
main                    ;Beginn der Hauptschleife
   ;Programmcode
   ;LED 1 für 0,4 Sekunden einschalten
   movlw 0x3C           ;lade den Wert 15536=0x3CB0 in die beiden
   movwf TMR1H          ;Timer1-Register
   movlw 0xB0
   movwf TMR1L
   bsf LED_1            ;LED 1 anschalten
   movlw b‘00110001‘    ;Prescale 1:8, Timer1 anschalten
   movwf T1CON
   btfss PIR1, TMR1IF
   goto $-1             ;wartet, bis der Timer überläuft
   bcf LED_1            ;LED 1 ausschalten
   clrf PIR1            ;Überlauf-Bit zurücksetzen
   ;LED 2 für 0,4 Sekunden einschalten
   movlw 0x3C           ;lade den Wert 15536=0x3CB0 in die beiden
   movwf TMR1H          ;Timer1-Register
   movlw 0xB0
   movwf TMR1L
   bsf LED_2            ;LED 2 anschalten
   movlw b‘00110001‘    ;Prescale 1:8, Timer1 anschalten
   movwf T1CON
   btfss PIR1, TMR1IF
   goto $-1             ;wartet, bis der Timer überläuft
   bcf LED_2            ;LED 2 ausschalten
   clrf PIR1            ;Überlauf-Bit zurücksetzen
   ;LED 3 für 0,4 Sekunden einschalten
   movlw 0x3C           ;lade den Wert 15536=0x3CB0 in die beiden
   movwf TMR1H          ;Timer1-Register
   movlw 0xB0
   movwf TMR1L
   bsf LED_3            ;LED 3 anschalten
   movlw b‘00110001‘    ;Prescale 1:8, Timer1 anschalten
   movwf T1CON
   btfss PIR1, TMR1IF
   goto $-1             ;wartet, bis der Timer überläuft
   bcf LED_3            ;LED 3 ausschalten
   clrf PIR1            ;Überlauf-Bit zurücksetzen
   ;LED 4 für 0,4 Sekunden einschalten
   movlw 0x3C           ;lade den Wert 15536=0x3CB0 in die beiden
   movwf TMR1H          ;Timer1-Register
   movlw 0xB0
   movwf TMR1L
   bsf LED_4            ;LED 4 anschalten
   movlw b‘00110001‘    ;Prescale 1:8, Timer1 anschalten
   movwf T1CON
   btfss PIR1, TMR1IF
   goto $-1             ;wartet, bis der Timer überläuft
   bcf LED_4            ;LED 4 ausschalten
   clrf PIR1            ;Überlauf-Bit zurücksetzen
   goto main
```

Am Anfang des Programms werden die Ein- und Ausgänge entsprechend der Pinbelegung des Entwicklungs-Boards festgelegt. Es werden auch die Interrupts ausgeschaltet, damit bei einem Zählerüberlauf nicht in die Interruptroutine gesprungen wird. Das Interrupt-Bit wird im Programmcode zyklisch abgefragt. Bei einem Oszillatortakt von 4 MHz muss der Wert 15536 in das Timer-Register (TMR1L und TMR1H) geladen werden, um eine Zeitverzögerung von 0,4 Sekunden zu erreichen. Welche Werte in die beiden Timer-Register geladen werden müssen, kann nach den Formeln 7.2, 7.3 und 7.4 berechnet werden.

$$TMR = 65536 - \frac{f_{OSZ} \cdot t}{4 \cdot PS} \qquad \textbf{Formel 7.2}$$

$$TMR1H = \frac{TMR}{256} \qquad \textbf{Formel 7.3}$$

$$TMR1L = TMR - TMR1H \cdot 256 \qquad \textbf{Formel 7.4}$$

TMR:	Wert des 16-Bit-Timer-Registers
f_{OSZ}:	Oszillatorfrequenz
t:	Zeitverzögerung
PS:	Wert des Vorteilers (Prescale)
TMR1H:	Highbyte des 16-Bit-Timers
TMR1L:	Lowbyte des 16-Bit-Timers

Für eine Zeitverzögerung von 0,4 Sekunden berechnen sich die Registerwerte wie folgt:

$$TMR = 65536 - \frac{4\ MHz \cdot 0{,}4\ s}{4 \cdot 8} = 15536 = 0x3CB0$$

$$TMR1H = \frac{15536}{256} = 60{,}6875 = 60 = 0x3C$$

$$TMR1L = 15536 - 60 \cdot 256 = 176 = 0xB0$$

Bevor die nächste Leuchtdiode angeschaltet wird, muss auch in den Timer-Registern der Wert für die Zeitverzögerung eingetragen werden, da das Register sonst von dem Wert 0 aufwärts zählen würde. Nachdem die LED gesetzt wurde und der Timer das Zählen gestartet hat, wird in einer Schleife geprüft, ob das Überlaufflag gesetzt wurde. Mit dem Befehl *goto $-1* springt der Programmzähler zu dem vorherigen Befehl. Das Zeichen *$* steht für den aktuellen Programmzähler. Wurde ein Zählerüberlauf festgestellt, wird der *goto*-Befehl übersprungen und die LED ausgeschaltet. Es muss ebenfalls das Überlaufflag von Timer1 zurückgesetzt werden, da sonst in der nächsten Schleife sofort wieder ein Überlauf angezeigt würde, egal, welcher Zählerstand im Timer-Register ist.

Makro mit Timer1

Zeitverzögerungen werden häufig in Programmen benutzt. Daher ist es sinnvoll, sich ein Makro zu programmieren, das eine Zeitverzögerung ermöglicht, deren Dauer über

einen Parameter eingestellt werden kann. Im folgenden Beispiel wird ein Makro vorgestellt, mit dem man eine Zeitverzögerung zwischen 20 µs und 524 ms realisieren kann. Die Berechnung der richtigen Registerwerte für TMR1L und TMR1H wird von dem Makro übernommen.

Beispiel:

```
_DELAY_TMR1_US macro usek
   variable timer_HL=0
   variable timer_H=0
   variable timer_L=0
   if usek > d'524000'
      error "MACRO: Wert für Makro _DELAY_TMR1_US zu groß!"
   endif
   if usek < d'20'
      error "MACRO: Wert für Makro _DELAY_TMR1_US zu klein!"
   endif

   ;Berechnung der Registerinhalte
   timer_HL = d'65536'-(OSC_FREQ/d'1000000'*usek/d'4'/d'8')

   ;Berechnung des Highbytes
   timer_H = (timer_HL >> d'8')

   ;Berechnung des Lowbytes
   ;Es wird 1 addiert, da das Makro aus mehreren Befehlen besteht
   timer_L = (timer_HL & 0x00FF)+d'1'

   bcf PIE1, TMR1IE     ;Interrupt von Timer1 ausschalten
   movlw timer_H        ;lade das Highbyte in Register TMR1H
   movwf TMR1H
   movlw timer_L        ;lade das Lowbyte in Register TMR1L
   movwf TMR1L
   movlw b'00110001'    ;schaltet Timer1 an, Prescaler 1:8
   movwf T1CON
   btfss PIR1, TMR1IF   ;prüft das Überlaufflag
   goto $-d'1'
   bcf PIR1, TMR1IF     ;Timer-Überlauf ist aufgetreten
                        ;Überlauf-Bit zurücksetzen
   endm
```

Da das Makro den Timer1 mit einem Vorteiler von 1:8 verwendet, ist die maximale Genauigkeit 1/8 des Befehlstakts. Am Anfang des Makros wird geprüft, ob die angegebenen Werte in einem Bereich liegen, der mit dem Timer1 realisiert werden kann. Ist der angegebene Wert größer als 524.000 µs oder kleiner als 20 µs, wird beim Übersetzen eine Fehlermeldung ausgegeben. Anschließend werden die Inhalte für die Register *TMR1L* und *TMR1H* berechnet. Die nachfolgenden Assemblerbefehle sind die gleichen wie im vorherigen Beispiel. Durch das Makro wird der Programmcode deutlich vereinfacht und verständlicher. Auf diese Weise kann die Wartezeit sehr einfach verändert werden, ohne die Registerwerte von Hand zu berechnen. Leider funktioniert eine

Änderung der Wartezeit nicht während der Programmausführung, da das Makro vor dem eigentlichen Übersetzungsvorgang interpretiert und bearbeitet wird. Ein Makro ist nur ein Textersatz, der das Programmieren vereinfachen kann.

Mithilfe des Makros kann das vorherige Beispiel deutlich kürzer geschrieben werden, obwohl der benötigte Programmspeicher genau gleich ist.

```
main
   ;LED 1 für 0,4 Sekunden einschalten
   bsf LED_1
   _DELAY_TMR1_US d‘400000‘
   bcf LED_1
   ;LED 2 für 0,4 Sekunden einschalten
   bsf LED_2
   _DELAY_TMR1_US d‘400000‘
   bcf LED_2
   ;LED 3 für 0,4 Sekunden einschalten
   bsf LED_3
   _DELAY_TMR1_US d‘400000‘
   bcf LED_3
   ;LED 4 für 0,4 Sekunden einschalten
   bsf LED_4
   _DELAY_TMR1_US d‘400000‘
   bcf LED_4
   goto main
```

7.3 Das Timer2-Modul

Bei dem Timer2-Modul handelt es sich, wie bei Timer0, um einen 8-Bit-Timer. Allerdings verfügt Timer2 über einen Vorteiler (Prescaler) und einen Nachteiler (Postscaler). Mit dem Vorteiler kann der Befehlstakt um 1, 4, oder 16 geteilt werden. Danach folgt ein 8-Bit-Register für die Zählung und im Anschluss folgt der Postscaler, der den Takt nochmals um einen Faktor zwischen 1:1 und 1:16 teilt. Der Nachteiler kann mit 16 Schritten sehr fein eingestellt werden und es sind so auch Teilerverhältnisse von 1:3 oder 1:13 möglich. Das Timer2-Modul wird auch für die Erzeugung von pulsweitenmodulierten Signalen (PWM) mit dem CCP-Modul benötigt. Die Generierung des Interrupts erfolgt nicht wie bei den anderen Timer-Modulen bei einem Überlauf des Registers, sondern nach dem Vergleich von Register *PR2* mit *TMR2*. Das Register *PR2* ist ein sogenanntes *Periodenregister,* mit dem eine zeitliche Periode vorgegeben werden kann. Der Wert in Register *TMR2* wird so lange erhöht, bis der Wert dem Inhalt von PR2 entspricht, und dann wieder auf 0 zurückgesetzt. Der Ausgang des Vergleichers ist an den Postscaler-Eingang angeschlossen und wird dann nochmals um den eingestellten Faktor geteilt. Zur Verdeutlichung der Funktionalität kann man sich das Ablaufdiagramm in *Abb. 7.1* ansehen.

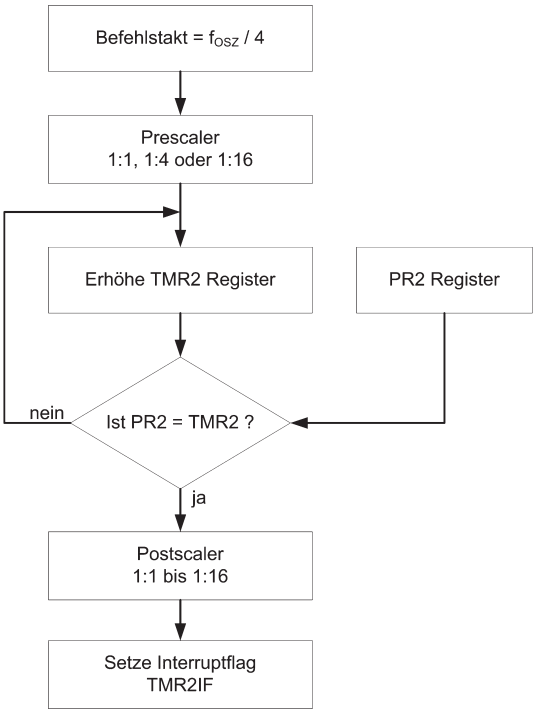


Abb. 7.1: Funktionsweise Timer2

Für die Berechnung einer Zeitverzögerung kann man Formel 7.5 verwenden. Aufgrund des 8-Bit-Registers und dem maximalen Teilerverhältnis von zweimal 1:16 ist mit diesem Timer eine maximale Zeitverzögerung von .

$$PR2 = \frac{f_{OSZ} \cdot t}{4 \cdot PRE \cdot POST} - 1 \qquad \textbf{Formel 7.5}$$

PR2: Periodenregister
f_{OSZ}: Oszillatorfrequenz
t: Zeitverzögerung
PRE: Teilerverhältnis des Prescalers
POST: Teilerverhältnis des Postscalers

Verwendet man einen 4-MHz-Quarz und möchte eine Zeitverzögerung von 1 ms einstellen, sollte man den Wert für den Postscaler auf 1:10 setzen, da man so die genauesten Ergebnisse erzielt. Der Prescaler kann dann auf den Wert 1:1 eingestellt werden. Setzt man die Werte in Formel 7.5 ein, erhält man für PR2 folgenden Wert:

$$PR2 = \frac{4MHz \cdot 1ms}{4 \cdot 1 \cdot 10} - 1 = 99$$

Durch den Teilerfaktor 1:10 ergeben sich ganzzahlige Werte und es entstehen keine Rundungsfehler. Im folgenden Beispiel wird die Zeitverzögerung von 1 ms mit dem Timer2-Modul umgesetzt.

Beispiel:

```
    _BANK_1             ;umschalten auf Bank 1
    movlw d‘99‘         ;Periodenreg. PR2 mit dem Wert 99 laden
    movwf PR2
    _BANK_0             ;umschalten auf Bank 0
    clrf TMR2
    movlw b’01001100’   ;Prescaler 1:1, Postscaler 1:10, Timer2 An
    movwf T2CON
    bsf PORTA, 2        ;setze Pin RA2 auf high
    btfss PIR1, TMR2IF  ;prüfe, wann der Interrupt ausgelöst wird
    goto $-1
    bcf PIR1, TMR2IF    ;Interruptflag zurücksetzen
    bcf PORTA, 2        ;setze Pin RA2 auf low
```

Sachverzeichnis

FRANZIS
PC & ELEKTRONIK

Michael Hofmann

Mikrocontroller für Einsteiger

Das Buch bietet eine Einführung in die Programmierung von Mikrocontrollern und gibt viele Tipps, wie die entsprechende Hardware aufgebaut werden muss. Da Mikrocontroller sehr hardwarenah programmiert werden, finden Sie auf der beiliegenden CD die Layoutdaten für ein eigenes Entwicklungsboard, mit dem viele Standardprobleme aus der Praxis untersucht werden können.

Nach der detaillierten Erklärung der Assemblerbefehle folgt eine ausführliche Erläuterung der Einstellungen und Funktionen der Entwicklungsumgebung MPLAB. Die Beispielprogramme beginnen bei der Ansteuerung von LEDs und der Abfrage von Tastern. Im Verlaufe des Buchs erfahren Sie, wie ein Display angesteuert wird, analoge Signale ausgewertet sowie Daten in einem externen EEPROM gespeichert und ausgelesen werden. Zum Abschluss wird die Kommunikation mit einem PC erläutert und wie man mit einer Infrarotfernbedienung die Ausgänge des Mikrocontrollers schalten kann.

In diesem Buch finden Sie alle Antworten zu den häufigsten Fragen rund um die Programmierung und Schaltungsentwicklung eines Mikrocontrollers

Aus dem Inhalt:

- Überblick über Mikrocontroller
- Die Programmierung mit MPLAB
- Verarbeitung analoger Signale
- Anzeige von Daten auf einem Display
- Messungen von Spannung und Leistung
- Datenübertragung über die serielle Schnittstelle
- Datenübertragung über den I²C-Bus
- und vieles mehr

Auf CD-ROM:

- Beispielprogramme
- Layoutdaten
- Schaltpläne
- Datenblätter

ISBN 978-3-7723-4318-6

Euro 39,95 [D]