FRANZIS
PC+ELEKTRONIK

Archiem Mueller-Wolkensteinn

Daten und Signale kabellos mit rfPICS übertragen

Tipps und Tricks rund um das rfPIC Development Kit 1

Auf CD-ROM:

- Alle Beispielprogramme
- Chip-Datenblätter
- MPLAB Tools V 7.61

Vorwort

Dies ist mein zweites Buch über PIC-Mikrocontroller. Es gibt viele Gründe, sich für den einen oder anderen Hersteller zu entscheiden. Bei mir ist die Entscheidung, wie sicher bei manch anderem Entwickler auch, im Studium im Rahmen eines Praktikums gefallen. Es sollte eine kleine Steuerung entwickelt werden und ein Kommilitone drückte mir das PICSTART Plus in die Hand ...

Das erste Buch war für Anfänger gedacht, um einen Einstieg in die Welt der PIC-Mikrocontroller zu finden. Dieses Buch wird einen Schritt weiter gehen, hat aber auch wieder den Anspruch, nicht nur die dicken Datenblätter zu übersetzen, sondern mit kurzen Erklärungen zu einem funktionierenden Ergebnis zu kommen. Wem dennoch Fragen zu Details offenbleiben, der wird sie in den Datenblättern des Herstellers beantwortet finden.

Dieses Buch wendet sich auch diesmal an Elektronikbastler, die auf der Suche nach einer Einführung in die Welt von rf-Mikrocontrollern sind. Thema ist die kabellose Daten- und Signalübertragung. Zudem sind auch kleine Schaltungserweiterungen rund um das PICkit 1 und die rf-Entwicklungsumgebung von Microchip zu finden.

Hilfreich beim Durcharbeiten des Buchs sind Grundkenntnisse im Programmieren mit dem PIC-Assembler. Die Grundlagen aus dem ersten Buch sind hierzu ausreichend. Dieses Buch setzt Kenntnisse in der Elektrotechnik voraus, es kann in diesem Bereich nicht alle Details erklären.

Ich wünsche Ihnen viel Spaß bei der Lektüre des Buchs.

Archiem Mueller-Wolkensteinn

Inhaltsverzeichnis

1 Grundlagen 11
1.1 Signale und Daten kabellos übertragen 11
1.2 Was ist ein rfPIC? 12
1.3 Reichweiten und Verschlüsselung 14
1.4 Abkürzungen und Begriffe 16

2 rfPICs 16
2.1 Die Unterschiede zum Standart PIC 12F675 16
2.2 Die Configbits der rfPICs 17

3 Aufbau des rf-Moduls 20
3.1 Die Senderplatine 21
3.2 Die Empfängerplatine 25

4 Das PICkit1 27

5 Die Demoanwendung 29

6 Kurze Einführung in Flussdiagramme 31

7 Die Entwicklungsumgebung MPLAB 33
7.1 Arbeiten mit MPLAB 34
7.2 Der Editor von MPLAB 39

8 Das Beispielprogramm im Sender 41
8.1 Konfiguration interrupt on change 43
8.2 IRS (Interrupt Service Routine) des Senders 44
8.3 Das Hauptprogramm des Senders 46
8.4 Lesen der Analogwerte 50

9 Das Beispielprogramm im Empfänger . . . 53

9.1 Die Struktur des Empfänger-Programms . . . 55
9.1.1 Die Datenerfassung . . . 57

10 Der PIC 16F684 . . . 61

10.1 Allgemeines zu den Ein- und Ausgängen . . . 61

11 Ein Bit senden . . . 65

11.1 Allgemeines . . . 65
11.2 Die Schaltung für den Sender . . . 65
11.3 Programmänderung für den Sender . . . 71
11.4 Die Schaltung für den Empfänger . . . 77
11.5 Programmänderung am Empfänger . . . 79

12 Grundlagen zur PWM . . . 82

13 Das PWM-Modul als Hilfsmittel . . . 85

14 Einen Wert mit Vorzeichen übertragen . . . 89

14.1 Das Empfängerprogramm . . . 91

15 Das rf-Kit in einer einfachen Anwendung . . . 95

15.1 Der A/D-Wandler im 12F675x . . . 95

16 Das Projekt Gartenbahn . . . 101

17 Daten kabellos übertragen – das Fazit . . . 107

18 Der PIC Assembler . . . 109

18.1 Das W-Register . . . 109
18.2 Das Carry-Bit . . . 109
18.3 Verhalten des Zero-Bits . . . 110
18.4 Lesen von Registern, wo ist das Bit 0 (Null)? . . . 111
18.5 Die Schreibweise von Befehlen . . . 112
18.6 Legende zum Assembler . . . 112
18.7 Die verschiedenen Zahlensysteme im Vergleich . . . 113
18.8 Die Befehle . . . 114

19 Platinen zum Buch ... **144**
19.1 Ein Bit übertragen ... 144
20 Bezugsquellen ... **147**
21 Befehlsliste ... **148**
22 Stichwortverzeichnis ... **152**

8 Das Beispielprogramm im Sender

Das Programm für den Sender ist – wie sollte es auch anders sein – für den rfPIC12F675x geschrieben. Wie bereits beschrieben, wählt man den Controller unter *Select Devices* aus. Lässt sich nun das PICkit1 nicht mehr als Programmer auswählen, hat man etwas vergessen. Es funktioniert nur, wenn man den PIC12F675 als Device auswählt.

Wer auch unterwegs lesen möchte, sollte sich den Quellecode einmal ausdrucken. Dann kann man ihn auch gleich um die eigenen Notizen erweitern. Alternativ kann man das Programm auch mit MPLAB auf dem PC betrachten und dabei um seine eigenen Kommentare erweitern.

Das Programm *xmit_demo.asm* ist in der Lage, 4 x 8 Bit (4 Byte) zu übertragen. In diesem Programm kann man schnell die angesprochene Werkzeugtechnik wiederfinden. Wenn man das Programm in einzelne Teile zerlegt, lässt sich gut eine Struktur erkennen, die man grafisch vereinfacht wie in dem folgenden Bild darstellen kann:

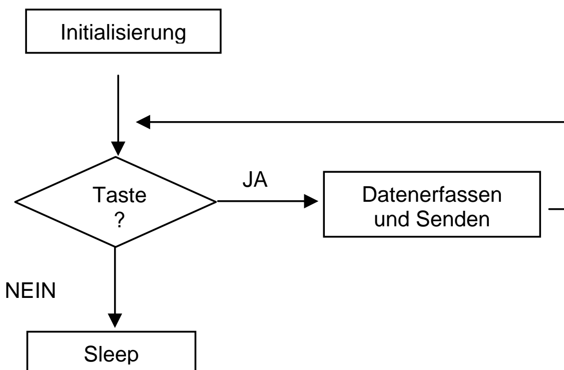


Abb. 8.1: Vereinfachter Ablauf des Senderprogramms

Als Erstes kommt die Initialisierung des PICs. Sie ist unabhängig vom Hersteller bei jedem Mikrocontroller erforderlich. Die Eigenschaften des Mikrocontrollers werden dabei festgelegt. Hier wird unter anderem definiert, welcher Pin ein Eingang oder Ausgang sein soll, mit welchem Pin ein analoges Signal verarbeitet werden kann und so manches mehr. Ein Ändern dieser Einstellungen ist auch im Betrieb jederzeit möglich, was in vielen Anwendungen erforderlich ist. In diesem Beispiel wird der aktive Analogeingang, je nach Tastenbetätigung, umgeschaltet.

Bank 0	File Address	Bank 1	File Address
Indirect addr.[1]	00h	Indirect addr.[1]	80h
TMR0	01h	OPTION_REG	81h
PCL	02h	PCL	82h
STATUS	03h	STATUS	83h
FSR	04h	FSR	84h
GPIO	05h	TRISIO	85h
	06h		86h
	07h		87h
	08h		88h
	09h		89h
PCLATH	0Ah	PCLATH	8Ah
INTCON	0Bh	INTCON	8Bh
PIR1	0Ch	PIE1	8Ch
	0Dh		8Dh
TMR1L	0Eh	PCON	8Eh
TMR1H	0Fh		8Fh
T1CON	10h	OSCCAL	90h
	11h		91h
	12h		92h
	13h		93h
	14h		94h
	15h	WPU	95h
	16h	IOC	96h
	17h		97h
	18h		98h
CMCON	19h	VRCON	99h
	1Ah	EEDATA	9Ah
	1Bh	EEADR	9Bh
	1Ch	EECON1	9Ch
	1Dh	EECON2[1]	9Dh
ADRESH	1Eh	ADRESL	9Eh
ADCON0	1Fh	ANSEL	9Fh
General Purpose Registers 64 Bytes	20h–5Fh	accesses 20h-5Fh	A0h–DFh
	60h–7Fh		E0h–FFh
Bank 0		Bank 1	

☐ Unimplemented data memory locations, read as '0'.

1: Not a physical register.

Abb. 8.2: Bänke im PIC

Ein weiteres Thema, das eng mit der Konfiguration der PIC-Controller verbunden ist, ist die Bankauswahl. Nicht jedes Register kann in den PIC-Controllern direkt angesprochen werden. Es muss erst der Bereich gewählt werden, in dem sich das Register befindet. Dazu ist es notwendig, im Statusregister das bank-selection-Bit entsprechend zu setzen. Dieses Bit findet man im Statusregister auf der Position 5.

Um Änderungen in den Registern aus der Liste im Bild 8.2 auf der vorigen Seite vornehmen zu können, muss das Bit gesetzt sein. Dies erreicht man am einfachsten mit dem Befehl:

```
bsf STATUS, RP0 ; ---- Auswahl Bank 1 -----
```

Ein Zurücksetzen des Bits ist ebenso leicht:

```
bcf STATUS, RP0 ; ---- Auswahl Bank 0 -----
```

Bei größeren Controllern mit mehr Speicher gibt es auch mehr als ein Bit dieser Art und man kann zwischen mehreren Banken wählen.

Weitere Beispiele zur Konfiguration eines PIC12F675 findet man im ersten Buch *PICs für Einsteiger*. Das Demoprogramm ist in diesem Bereich sehr ausführlich dokumentiert. Zu jeder Definition einer Konfiguration findet man eine kurze Erklärung und den Verweis auf das Kapitel des Datenblatts, in dem das Thema genauer erklärt wird. Spezielle Einstellungen und Lösungen, die für die Programme von Bedeutung sind, werden auch in den folgenden Beispielen angesprochen und gezielt vorgestellt.

8.1 Konfiguration interrupt on change

Das eigentliche Programm ist, wie man in dem Ablaufbild sieht, eine kleine Schleife, die der Controller nur zum Schlafen verlassen darf. Anders herum betrachtet: Der Controller schläft ständig und wird nur geweckt, um einmal kurz die betätigte Taste und den dazugehörenden Analogwert zu erfassen und anschließend zu senden. Auf den Alltag übertragen, entspricht es einem Arbeiter, der jeden Tag zur Arbeit und abends schlafen geht. Um morgens aus dem Bett zu kommen, benötigt er eine Hilfe, z. B. einen Wecker. In unserem Fall also einen *Interrupt*.

Bei unserem Controller ist es nicht anders. Auch er wacht nur wieder auf, wenn der „Wecker klingelt“. In diesem Fall ist der Wecker ein *interrupt on change*.

```
; Interrupt-on-Change Register (IOCB) (Kapitel 3.2.2)

;Jeder Eingangspin kann individuell mit der Möglichkeit, einen Interrupt
;on change auszulösen belegt werden. Eine „1“ im Register IOCB an der Stelle
```

```
;des Pins schaltet die Funktion für den entsprechenden Eingang aktiv.
;Zum Beschreiben des Registers IOCB muss in die Bank 1 gewechselt werden.
;Wichtig: Für den Betrieb müssen Globale Interrupts erlaubt sein!

  bsf STATUS, RP0       ; ---- Auswahl Bank 1 -----

; GPIO Pins = xx543210  ; mögliche Eingänge
  movlw  b'00011000'    ; erlaube Interrupt-on-change für: GPIO3 & 4
  movwf  IOCB           ; schiebe die Definition ins Register IOCB

  bcf STATUS, RP0       ; ---- Auswahl Bank 0 -----
```

Abb. 8.3: Konfiguration interrupt on change

Diese Möglichkeit, einen Controller zu wecken, wird im Konfigurationsbereich definiert. Wenn man also einen Controller mit dem SLEEP-Befehl schlafen schickt, darf man nicht vergessen, auch einen Interrupt (Wecker) zu definieren, der ihn irgendwann wieder weckt. Sonst weckt ihn nur ein Reset wieder aus dem Schlaf, z. B. durch das Aus- und wieder Einschalten der Betriebsspannung.

Der Controller geht im Sleep-Modus schlafen. Der kleine PIC 12F675 stellt im Sleep-Modus fast jegliche Aktivität ein. Bei komplexeren Controllern besteht hier auch eine Möglichkeit der Konfiguration, welche Aktivitäten im Sleep-Modus noch ausgeführt werden sollen. Dadurch kann vor allem der Energieverbrauch des Controllers auf ein absolutes Minimum heruntergefahren werden, wodurch sich der Energieverbrauch insgesamt meist erheblich verringern lässt. Dies ist bei Batterieanwendungen besonders wichtig, wie etwa bei Fernbedienungen oder dem vorliegenden rf-Board.

In vielen Anwendungen ist es dabei auch meist nicht erforderlich, dass der Controller Daten an die Zentrale überträgt, wenn nichts passiert. Es muss ja nicht die ganze Zeit die letzte Programmauswahl zum TV übertragen werden. Es reicht die Zeit aus, bis die Zentrale – z. B. das TV-Gerät – reagiert hat, wie es der Anwender wünscht.

Wie funktioniert nun das „Wecken" eines Controllers mit einem *Interrupt on change*? Die Definition für den PIC12F675 und die meisten kleinen anderen Controller ist so, dass bei jeder Art von Interrupt an die Adresse *ORG 0x004* gesprungen wird. An dieser Adresse hat dann die sogenannte IRS (Interrupt Service Routine) zu beginnen.

8.2 IRS (Interrupt Service Routine) des Senders

Im prinzipiellen Aufbau einer ISR wird zuerst der aktuelle Zustand des Controllers gerettet. Dazu werden das *STATUS*- und das *W*-Register in zwei Hilfsvariablen zwischengespeichert. Im mittleren Abschnitt, der dem eigentlichen ISR-Teil entspricht, wird die Quelle ermittelt, durch die der Interrupt ausgelöst wurde. Danach werden die Register zurückgeschrieben und der Controller springt wieder an den Punkt zurück,

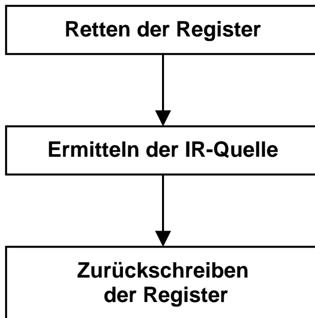


Abb. 8.4: Struktur der ISR

an dem er sich vor dem Interrupt befand. Im aktuellen Fall ist es das Bit *GPIF*, das für den *Interrupt on change* steht. Dies wird gesetzt und löst den Interrupt im Controller aus, wenn eine Eingangsänderung an den Pins 3 oder 4 erfolgt ist. Um dieses Bit *GPIF* zurücksetzen zu können, muss vorher ein Lese- oder Schreibbefehl auf den dazu gehörenden Port erfolgen. Das würde in diesem Fall folgendermaßen aussehen:

```
                         ; Teil der ISR
movfw  GPIO              ; lese GPIO
bcf    INTCON, GPIF      ; lösche Bit GPIF
```

Da in diesem Programm der Controller nur durch den Interrupt geweckt werden soll, erfolgt hier keine weitere Auswertung der IRQ-Quelle. Welche Taste den Controller geweckt hat, wird in dem Beispiel erst im Hauptprogramm ermittelt. Sonst müsste eventuell noch ein Merker gesetzt werden, was geschehen ist.

Hier nun die gesamte IRS des Senders:

```
  ; Beginn der Interrupt Service Routine (ISR)

                        ; Retten der Register
  ORG 0x004             ; Interrupt Startadresse
  movwf  w_temp         ; Sichern des W-Registers
  swapf  STATUS, W      ; swap status to be saved into W
  bcf STATUS, RP0       ; ---- Select Bank 0 -----
  movwf  status_temp    ; Sichern des STATUS-Registers
;---------------------------------------------------------------
; Eigentlicher Teil der ISR

  movfw  GPIO           ; Lese GPIO
  bcf INTCON, GPIF      ; Lösche Bit GPIF

;---------------------------------------------------------------
                        ; zurückschreiben der Register
  swapf  status_temp, W  ; swap status_temp into W, sets bank to original state
  movwf  STATUS         ; Zurückschreiben des STATUS-Registers
```

```
swapf  w_temp, F
swapf  w_temp, W      ; Zurückschreiben des W-Registers

retfie                ; Rücksprung aus der Interrupt Service Routine
```

Abb. 8.5: ISR-Interrupt-Service-Routine des Sendeprogramms

Als Grundsatz sollte gelten: Eine ISR sollte immer so kurz wie möglich gehalten sein. Dort sollte nur das Notwendigste abgearbeitet werden.

Soll der Interrupt nicht nur den Controller wecken, wird die ISR natürlich automatisch länger. Hier sollte man aber auch nur eine kurze Auswertung der Situation programmieren und diese mithilfe von Merkern an das Hauptprogramm übergeben. Nur zeitkritische Dinge handelt man sofort in der Interrupt-Service-Routine ab. Hierzu muss man die Aufgabenstellung einbeziehen.

Bei einer Fernbedienung soll der Tastendruck selbstverständlich auch sofort übertragen werden. In Anbetracht der menschlichen Wahrnehmung kann der Controller vorher durchaus noch 100 andere Befehle abarbeiten, bis er die Taste abfragt und das Ergebnis sendet. Bei einem Maschinenalarm hingegen, oder einer automatischen Regelung, kann es mit der benötigten Reaktionszeit schon ganz anders aussehen.

Vor allem die neueren und größeren Controller kennen in der Zwischenzeit verschiedene Interrupt-Level und eine Priorisierung von Interrupts. Das bedeutet, dass, je nach Interruptquelle, an eine andere Adresse gesprungen wird. Auch während eines Interrupts kann ein weiterer auftreten, der dem anderen vorgezogen wird, weil er vom Programmierer als wichtiger eingestuft wurde. Hier muss eine *Interrupt Service Routine* selbstverständlich ganz anders aufgebaut werden.

8.3 Das Hauptprogramm des Senders

Es wurde im Beispielprogramm bis jetzt die Initialisierung betrachtet, die für das „Wecken“ des Controllers erforderlich ist, also um den Controller wieder aus dem Sleep-Modus herauszubekommen. Ferner gibt es im Demoprogramm noch zwei kleine Unterprogramme, die sich mit dem Lesen und Schreiben des EEProms im PIC12F675 befassen. Diese Unterprogramme gehören mit zum Baukastenprinzip. Man kann auf sie zurückgreifen, wenn es in einer anderen Anwendung einmal erforderlich sein sollte, aber sie werden in den Beispielen zum Buch nicht benutzt.

Alle anderen Programmteile werden hingegen für einen funktionierenden Betrieb des Demoprogramms benötigt. Die eigentlichen Aufgaben des Controllers – das Erfassen und Senden der Eingaben auf der kleinen Platine – sind ebenfalls in kleine Programmteile gegliedert.

Liest man das Demoprogramm einmal von oben nach unten, wird man erkennen, dass in der ersten Zeile nur die Programmstartadresse *ORG 0x000* definiert ist und sofort mit einem GOTO-Befehl zu einem Label mit dem Namen *INITALIZE* gesprungen wird. Es folgt die Adresse für die ISR *ORG 0x004* und dann kommen einige Unterprogramme. Diese Struktur sollte man eigentlich in jedem Programm in irgendeiner Art wiederfinden können. Jedes Assembler-Programm beginnt mit einem Sprung über den *Interrupt-Service-Routine*-Bereich.

Am Label *INITALIZE* beginnt dann in diesem Fall das eigentliche Hauptprogramm. Die Initialisierung wird meist nicht in einem Unterprogramm angelegt, da die Initialisierung nur einmal, zu Beginn des Betriebs, durchlaufen werden muss. Wer es aber als übersichtlicher empfindet, kann dies natürlich nach eigenem Geschmack organisieren.

Am Label *MAIN* beginnt dann das eigentliche Programm. Zwei Zeilen später, am Label *SCANPB,* beginnt die in der Programmstruktur vorgestellte Schleife.

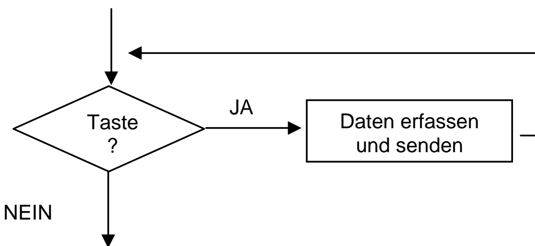


Abb. 8.6: Struktur des Programms SCANPB

Damit wird es für den Anwender interessant, denn ab dem Label *SCANPB* beginnt die Tastenabfrage und somit der eigentliche anwenderspezifische Programmteil des Senderprogramms. Möchte man durch kleine Änderungen eine andere Funktionalität erreichen, ist das der Programmteil, wo dies erfolgen muss.

Folgend der Programmteil der Tastenabfrage im Detail:

```
SCANPB

  movlw   0x00                          ; lade Null ins W-Register
  movwf   FuncBits                      ; Lösche das Function Bits register

  btfsc   PB3                           ; ist der Taster GP3 gedrückt?
   goto   SPB1
  movlw   0x23                          ; B'00100011' Funktion S0 gewählt
  iorwf   FuncBits, F
  call    READ_ANALOG_AN0               ; lese analog Kanal AN0
```

```
SPB1
  btfsc   PB4                        ; Taste GP4 gedrückt?
   goto   SPB2                       ; Nein, überspringe
  movlw   0x43                       ; Function S1 gewählt
  iorwf   FuncBits, F
  call    READ_ANALOG_AN1            ; lese analog Kanal AN1

SPB2
  movlw   0xFF
  andwf   FuncBits, W                ; War irgendeine Taste gedrückt?
  btfss   STATUS, Z
   goto   XMIT                       ; Ja, Übertrage den Buffer

  bcf     RFENA                      ; Nein, abschalten des Transmitters
  sleep                              ; schicke den PIC schlafen
  goto    SCANPB                     ; Upon wake-up on pin change, lese
                                       die Tasten
```

Abb. 8.7: Programmteil SCANPB

Die Beschreibung:

Es wird damit begonnen, das Register *FuncBits* auf den Wert *0* (Null) zu setzen. Danach wird getestet, ob die Taste am Eingang PB3 betätigt wurde. Diese Auswertung erfolgt über eine inverse Abfrage des Eingangspins, da ein Tastendruck am Eingang eine *0* (Null) liefert. Ist die Taste betätigt, wird sie über eine ODER-Verknüpfung in das Register *FuncBits* übertragen. Dieses Register fungiert als Merker, welche Taste betätigt wurde. Dies wird gemacht, da diese Information auch an die Zentrale übertragen werden soll und später noch einmal zur Verfügung stehen muss.

Das Gleiche gilt für die Auswertung der zweiten Taste. Die „3" im Hex-Wert der ODER-Verknüpfung ist Bestandteil der Seriennummer und wird auf diese Weise nur mit an die entsprechende Stelle ins Register geschrieben. Insgesamt können 4 x 8 Bits an Nutzdaten übertragen werden. Die Zusammensetzung der Daten wird später im Kapitel *Das Beispielprogramm im Empfänger* noch einmal genauer beschrieben. Vorab der schematische Aufbau des Daten-Streams.

```
;******************** Bedeutung der übertragenen Bits ********************
;
;bytes |  DATA1                      |           DATA0        |
;bits  | 7 | 6 | 5 | 4 | 3 | 2| 1| 0| 7| 6| 5| 4| 3| 2| 1| 0|
;desc. |S2 |S1 |S0 |S3 | 0 | 0|  SERIAL NUMBER 3| 2| 1| 0| 7| SERIAL...
;
;bytes |  DATA3                |            DATA2       |
;bits  | 7| 6| 5| 4| 3| 2| 1| 0| 7| 6| 5| 4| 3| 2| 1| 0|
;desc. |  COUNTER                                       |
;
```

Abb. 8.8: Bedeutung der übertragenen Bits

Welche Taste betätigt wurde, wird im Byte *Data1* an den Empfänger übertragen. Wie man aber sieht, sind auch die unteren zwei Bits noch Bestandteil einer möglichen Seriennummer.

Nach der Tastenauswertung wird entschieden, ob die ermittelten Daten gesendet werden sollen, oder ob der Controller wieder schlafen gehen kann. Diese Auswertung erfolgt ebenfalls über eine Maske. Dazu wird zuerst das W-Register mit 0xFF geladen. Das bedeutet, dass jetzt jedes Bit in dem Register gesetzt ist. Dieses wird im nächsten Schritt mit dem Merker *FuncBits* in einer UND-Verküpfung ausgewertet. Ist im Register *FuncBits* kein Bit gesetzt, ergibt die Verknüpfung *0* (Null). Ist das Ergebnis einer Operation *0* (Null), wird immer im Statusregister das sogenannte *ZERO-Bit* gesetzt. Mithilfe dieses Bits können die Ergebnisse von Maskenauswertungen oder Vergleichsoperationen erkannt werden.

Wurde eine Taste betätigt, wird das *ZERO Bit* nicht gesetzt. Es wird als Nächstes das Unterprogramm *XMIT* aufgerufen, da die Anweisung *btfss* ungültig ist und der nächste Befehl nicht übersprungen wird.

In der Funktion *XMIT* werden dann die gesammelten Daten zusammengestellt und an den Empfänger gesendet. Danach wird zur Tastenauswertung zurückgesprungen. Wurde keine Taste mehr erkannt, wird der Sender abgeschaltet und der Controller Schlafen geschickt, bis ihn eine erneut betätigte Taste durch einen Interrupt wieder weckt.

Das ist vom Grundgedanken her das eigentliche Anwenderprogramm, das die zu übermittelnden Daten erfasst. Möchte man nun andere Daten erfassen oder die Art und Weise der Erfassung ändern, sind nur in diesem Teil Programmänderungen zu machen.

Dieser Programmteil ist für sich betrachtet recht kurz, einfach gehalten und sicher auch für Einsteiger mit wenigen Grundkenntnissen noch verständlich. Wer beim Arbeiten mit Masken zum Erkennen einer betätigten Taste noch Probleme hat, sollte sich die im Programm benutzten Abkürzungen mit den dazugehörenden Werten in die binäre Form umrechnen und übereinander aufzeichnen. Das erleichtert das Verständnis.

Hier als Beispiel:

```
movlw 0b11111111              ; 0xFF für FuncBits
andwf 0b00100011              ; 0x23 Taste PB3 war gedrückt
 btfss   STATUS, Z
 goto    XMIT
```

Da das Ergebnis nicht *0* (Null) ist, wird das *Zero-Bit* nicht gesetzt und der nächste Befehl auch nicht übersprungen. Natürlich kann man diese Auswertung auch auf andere Art lösen. Dies hier ist nur ein möglicher Weg, die Tasten zu erkennen.

Das gleiche Prinzip wird angewendet, um das zur Taste gehörenden Poti mit dem dazugehörenden Analogwert auszulesen. Dazu wird immer das entsprechende Unterprogramm *READ_ANALOG_ANx* aufgerufen.

8.4 Lesen der Analogwerte

Ein Teil der Datenerfassung besteht natürlich auch aus der Analogwerterfassung, die in einem eigenen Unterprogramm abgearbeitet wird. Es kann aufgrund der Tastenauswertung an zwei verschieden Positionen gestartet werden.

Die Struktur zur Auswertung der A/D-Kanäle ist assemblertypisch gelöst. Es gibt ein Unterprogramm *READ_ANALOG_AN0*, das noch eine zweite Einsprungstelle mit dem Label *READ_ANALOG_AN1* hat. Der Unterschied bei den Einsprungstellen ist lediglich der dadurch gewählte und aktivierte Analogkanal. Auf diese Weise kann das gleiche Unterprogramm für zwei verschiedene Analogkanäle benutzt werden, ohne dass ein Wert übergeben werden muss. Programmiert man solch eine Struktur in einer Hochsprache, muss man in der Regel mit einer Variablenübergabe arbeiten, in der der auszuwertende Analogkanal als Zahl in einer Variablen übergeben wird.

Hier nun das Unterprogramm für die Auswertung der A/D-Eingänge im Detail:

```
READ_ANALOG_AN0

  bcf    ADCON0, CHS1   ; Wähle analog channel AN0
  bcf    ADCON0, CHS0

  goto   READ_ANALOG

READ_ANALOG_AN1

  bcf    ADCON0, CHS1   ; Wähle analog channel AN1
  bsf    ADCON0, CHS0

READ_ANALOG

  bsf    ADCON0, ADON   ; Einschalten des ADC-Moduls

  ; Nach der Auswahl eines neuen A/D-Kanals muss auf eine ausreichend große
  ; Erholzeit geachtet werden - die Länge der Erholzeit hängt von der internen
  ; Kapazität ab. Mehr dazu im Kapitel 7.2.

  movlw  D‘6‘           ; Bei 4 MHz, eine 21-us-Pause
  movwf  TEMP           ; (21 us = 2 us + 6 * 3 us + 1 us)
  decfsz TEMP, F
  goto   $-1
```

```
bsf     ADCON0, GO      ; Start A/D-Wandlung

btfsc   ADCON0, GO      ; Ist die A/D-Wandlung fertig?
goto    $-1

bcfADCON0, ADON         ; Abschalten des ADC-Moduls
                        ;Dies spart Batterieenergie

return                  ; Rücksprung aus dem Unterprogramm
```

Abb. 8.9: Die Unterprogramme READ_ANALOG_AN0 und READ_ANALOG_AN1

Was unbedingt beachtet werden muss ist, dass nach einem Wechsel des A/D-Kanals zwingend eine kleine Pause erfolgen muss, bevor eine erneute A/D-Wandlung gestartet wird. Für diese einzuhaltende Pause gibt es keine einheitliche Zeit. Sie ist für jeden Controllertyp verschieden und dabei auch noch temperaturabhängig. Ist es für die zu erstellende Anwendung wichtig, möglichst schnell zwischen den verschiedenen Analogkanälen hin und her zu schalten, findet man die Berechnung der Zeit in dem Kapitel *ACQUISITION TIME*. Bei dem PIC 12F675 ist es das Kapitel 7.2. Nach der dort vorgestellten Rechnung erhält man als kleinste Zeit den Wert 19,72 µSekunden.

$$
\begin{aligned}
T_{ACQ} &= \text{Amplifier Settling Time} + \text{Hold Capacitor Charging Time} + \text{Temperature Coefficient} \\
&= T_{AMP} + T_C + T_{COFF} \\
&= 2\mu s + T_C + [(\text{Temperature} - 25°C)(0{,}05\mu s/°C)] \\
T_C &= C_{HOLD}(R_{IC} + R_{SS} + R_S)\ln(1/2047) \\
&= -120pF(1k\Omega + 7k\Omega + 10k\Omega)\ln(0.0004885) \\
&= 16{,}47\mu s \\
T_{ACQ} &= 2\mu s + 16{,}47\mu s + [(50°C - 25°C)(0.05\mu s/°C) \\
&= 19.72\mu s
\end{aligned}
$$

Abb. 8.10: Berechnung der ACQUISITION TIME beim rfPIC 12F675

Im Unterschied dazu beträgt diese Zeit für den PIC 16F684 laut Datenblatt im Minimum nur noch 7,67 µSekunden.

Bleiben wir aber beim PIC 12F675. Diese errechnete Zeit lässt sich nun leider nicht mit einem rfPIC12F675 und einer Betriebsfrequenz von 4 MHz erzeugen. Um aber bei der Kanalumschaltung immer auf der sicheren Seite zu sein, wählt man mindestens den nächstgrößeren möglichen Wert. Im Demoprogramm zum Sender wird von 22 µSekunden gesprochen. Wenn man die dazu vorgestellten Zahlenwerte aber, wie in der Formel vorgegeben, berechnet, kommt man nur auf 21 µSekunden. Dies ist aber für eine geforderte Zeit von mindestens 19,72 µSekunden noch ausreichend.

Um eine kleine Pause von 21 µSekunden zu erzeugen, kann die folgende Warteschleife

benutzt werden:

```
        movlw D‘6‘              ; Bei 4 MHz, eine 21-µs-Pause
        movwf TEMP              ; (21 µs = 2 µs + 6 * 3 µs + 1 µs)
        decfsz TEMP, F          ; Dekrementiere TEMP um 1
        goto $-1                ; Springe eine Zeile zurück
```

Hierbei handelt es sich um eine kleine Zählschleife, die bis zur dezimalen Zahl 6 zählt.

21 µSekunden für die Schleife errechnen sich aus der Anzahl der Befehle und der Zeit, die der Controller für die Ausführung der Befehle benötigt. Dies errechnet sich wie folgt:

21 µs = 2 µs + 6 * 3 µs + 1 µs

Die ersten 2 µs ergeben sich aus dem Laden der Konstanten und der Variablen TEMP. Danach wird diese Variable 6 * dekrementiert, wobei der GOTO-Befehl immer zwei Zeiteinheiten benötigt. Die letzte 1 µSekunde wird für den Sprung über den GOTO-Befehl benötigt.

Benutzt man eine andere Betriebsfrequenz im Controller, erreicht man mit der Zählschleife einen anderen Zeitwert.

Interessant ist die Schreibweise hinter dem GOTO-Befehl: *$-1*. Hier handelt es sich um eine Compiler-Anweisung. Unabhängig von der Position im Speicher wird hier der Controller angewiesen, genau um eine Stelle im Programm zurückzuspringen. Alternativ kann statt der „Eins“ (1) dort auch eine andere Zahl stehen, um die zurückgesprungen werden soll. Durch ein Pluszeichen anstelle des Minuszeichens besteht dann auch die Möglichkeit, um eine bestimmte Anzahl von Befehlszeilen vorwärtszuspringen.

Aber Vorsicht mit solchen absoluten Sprüngen! Man sollte schon genau wissen, wo man dann landet.

Hiermit sollte das Wesentliche des Demo-Sendeprogramms, bezogen auf den Anwenderteil, erklärt worden sein, um die grundlegenden Funktionen zu verstehen. Einige Erweiterungen und Details werden noch in den Anwendungsbeispielen folgen.

9 Das Beispielprogramm im Empfänger

Hier gilt das Gleiche wie bei dem Programm für den Sender: Auch wenn das Programm im MPLAB-Editor etwa 1.500 Zeilen lang ist, ist es nicht so kompliziert, wie es aussieht. Vieles sind nur Kommentare zum Programm oder Hinweise zur Konfiguration des Controllers.

Das Demoprogramm des Empfängers ist im Original für einen PIC 16F676 geschrieben worden, der dem rf-Kit beiliegt.

Folgend wird etwas weiter ausgeholt und mit der Definition der Variablen angefangen. Die Definitionen erleichtern vor allem das Lesen und Verstehen eines Assembler-Programms. Hinter einer Definition kann sich ein Register oder auch ein fester Zahlenwert verbergen, schlimmstenfalls aber auch eine Definition.

Beispiele:

Definition von Registern:

Der Befehl *cblock* weist den Compiler an, der folgenden „Liste von Namen" – man kann sie auch als „Zeichenkette" beschreiben (DATA0,1,2,3 ...) – Adressen mit der angebenden Startadresse zuzuweisen. Die Liste endet mit dem Befehl *endc*.
Hierbei handelt es sich um einen Compilerbefehl.

```
cblock 0x20
            DATA0                ; 1st Byte der empfangenen Daten
            DATA1                ; 2nd Byte der empfangenen Daten
            DATA2                ; 3rd Byte der empfangenen Daten
            DATA3                ; 4th Byte der empfangenen Daten
            .
            .
            LEDREG  ; LED Array Register
endc
```

Eine weitere Möglichkeit einer Definition ist die Anweisung:

```
#define LED0ON b'00010000'
#define LED1ON b'00100000'
```

Hier wird mit *#define* der Zeichenfolge *LED0ON* der Wert *B*'00010000' zugewiesen (er wird gleichgesetzt). Natürlich kann man an jeder Stelle, an der man die Definition *LED0ON* benutzt, auch den Wert *B*'00010000' schreiben. Verwendet man, wie hier, einen beschreibenden Namen für den Zahlenwert, erhöht sich die Lesbarkeit eines Programms. In diesem Fall beschreibt *LED0ON* das erforderliche Bitmuster, das am Port des PIC anliegen muss, um die LED 0 einzuschalten.

In welcher Schreibweise (HEX, binär oder dezimal) die Zuweisung erfolgt, spielt keine Rolle. Bei diesen Zuweisungen kann man die Schreibweise wählen, die einem für den Zweck am sinnvollsten erscheinent.

Möchte man z. B. einzelne Bits an den Ports setzen, bietet sich die binäre Schreibweise an. So erkennt man gleich den gewählten Zustand an den Ausgängen. Wird der definierte Wert zum Rechnen benutzt und man arbeitet im Dezimalsystem, kann die dezimale Darstellung geeigneter sein.

Die Steigerung einer Definition ist deren Kombination. Ein Beispiel für eine Teildefinition zu einer Definition:

```
; LEDs
#define LED0 LEDREG, 0
#define LED1 LEDREG, 1
#define LED2 LEDREG, 2
```

Hier wird dem Begriff *LED0* gleich dem Registerbit *0* des Registers *LEDREG* gesetzt, wobei *LEDREG* in der ersten Definitionszuweisung eine Speicheradresse zugewiesen bekommen hat. Hier fragen sich jetzt vermutlich vor allem die Einsteiger, ob sich dadurch die Lesbarkeit eines Programms wirklich erhöht. Häufig vereinfacht man sich mit solchen Definitionen aber die Schreibarbeit und erleichtert sich auch eventuelle Anpassungen eines Assembler-Programms von einem Controllertypen auf einen anderen. Vor allem aber ist es eine Frage der Gewohnheit, mit solchen definierten Werten und Variablen zu arbeiten.

In dem Programm des Empfängers spielt die Definition *Status Counter* eine entscheidende Rolle für das Verständnis des Programms. Hier wird jedem Begriff eine Zahl von 1 bis 9 zugewiesen. Dabei steht jeder Name für einen Schritt in einer State- bzw. Zustandsmaschine. Eine ähnlich aufgebaute Statemachine findet man auch noch einmal bei der Ansteuerung der Leuchtdioden. Ihr Name lautet *LEDStateMachine*. Hier gibt es für jeden State ein Sprung-Label, in dem eine der acht Leuchtdioden geschaltet wird.

```
  ; Status Counter

#define        BEGN            0x00
#define        BEGN1           0x01
```

```
#define        HEADR          0x02
#define        HEADR1         0x03
#define        HIGHP          0x04
#define        LOWP           0x05
#define        RECRD          0x06
#define        WAIT           0x07
#define        VALID          0x08
#define        IMPLMNT        0x09
```

Abb. 9.1: Definition des Zählers

Controller-spezifische Abkürzungen und Definitionen wie *ANSEL* oder *Port*, die zu jedem PIC-Controller dazugehören, sind in einer eigenen Datei, die für jeden Controller existiert und von Microchip zu MPLAB mitgeliefert wird, hinterlegt. Diese Datei mit ihren Definitionen bindet man ganz zu Anfang eines Programms nach dem Befehl *List* mit *#include* und dem Dateinamen ein.

Beispiel:

```
list p=16f676 ; list directive to define processor
#include <p16f676.inc>; processor specific variable definitions
```

In diesem Beispiel ist es die Datei *p16f676.inc*. Diese zwei Zeilen müssen immer an den im Projekt verwendeten Controller angepasst werden. Stimmen diese Angaben nicht mit dem ausgewählten Controller in der Konfiguration von MPLAB überein, bekommt man sofort eine Fehlermeldung beim Übersetzen des Programms.

Fehlermeldung:

```
MESSAGE: (Processor-header file mismatch. Verify selected processor.)
```

Allerdings muss es durch eine falsch eingebundene Datei nicht zwangsläufig zu Fehlfunktionen im Controller kommen.

9.1 Die Struktur des Empfänger-Programms

Auch dieses Programm lässt sich stark vereinfacht in einer „Zwei-Task"-Struktur darstellen. Das Empfängerprogramm besteht aus einer Schleife, die in einer großen Schleife liegt. Nach der Initialisierung, die einmal durchlaufen wird, wartet das Programm in

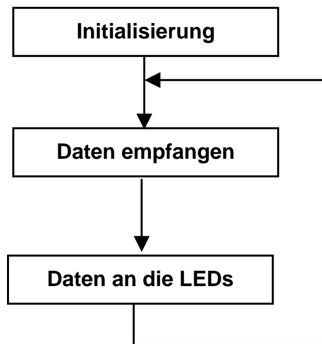


Abb. 9.2: Struktur Empfänger

einer Schleife auf aktuelle Daten vom Sender. Sobald eine gültige Datensequenz empfangen wurde, wird dies dem Benutzer über die Leuchtdioden dargestellt.

Der Programmteil, der die LEDs ansteuert, wird hier nicht näher betrachtet. Er kann als Anwendungsteil gesehen werden. Benutzt man das gegebene Programmgerüst für eigene Entwicklungen, kann man alles rund um die Ansteuerung der Leuchtdioden weglassen und schreibt am Label seine eigene Applikation zum Steuern von Relais, LEDs oder was auch immer betrieben werden soll.

Die Lernfunktion, die mit der Taste *learn* gesteuert wird, ist für den normalen Betrieb nicht erforderlich. Sie lässt aber bei Bedarf die Möglichkeit offen, eine Art Codierung des Senders auf den Empfänger zu integrieren. So besteht die Möglichkeit, in den Datenfeldern *Data0* und *Data1* eine Seriennummer in der Übertragung zu hinterlegen. Damit nun der Empfänger nur auf einen speziellen Sender hört, kann man den Sender die Seriennummer, mit der er zusammenarbeiten soll, mit der Lerntaste „lernen“ lassen. Betätigt man die Lerntaste, speichert der Empfänger die empfangenen Datenfelder *Data0* und *Data1* in seinem EEprom ab. Jeder nun empfangene Daten-Stream muss dann diese Seriennummer beinhalten, sonst werden die Daten als ungültig für diesen Empfänger verworfen. Standardmäßig ist diese Seriennummernauswertung aber nicht aktiv, sodass jeder Sender mit jedem Empfänger zusammenarbeitet.

Nun zum Empfänger-Programm: Die Initialisierung entspricht weitgehend der des Senders, da die PICs aus der gleichen Familie stammen und ungefähr die gleichen Möglichkeiten bieten. Auch wenn der Analogeingang in der Initialisierung definiert wird, wird er in diesem Programm selbst nicht benutzt.

Eine entscheidende Änderung gibt es allerdings doch: Die Bezeichnung der Eingänge ändert sich von GPIOx, wie sie beim PIC12F675 heißen, zu PORTAx oder PORTBx, wie die I/Os der PIC16F6xx-Familie heißen. Dies ist etwas gewöhnungsbedürftig, denn dadurch lassen sich Programme nicht so leicht von einem kleinen auf einen größeren Controller übertragen. Hier ist es dann hilfreich, wenn man mit Definitionen gearbeitet hat. So braucht man nichts im eigentlichen Programm, sondern nur die Zuweisung in der Definition von GPIOx nach PORTx ändern.

9.1.1 Die Datenerfassung

Kommen wir nun zum entscheidenden Teil des Programms: der Datenerfassung.
Der Ablauf der Datenerfassung ist in einer Statemachine-Struktur programmiert. Eine Statemachine (*Zustandsmaschine* oder *endlicher Automat*) ist in diesem Fall auch als eine Art Schrittkette zu erklären. Es kann immer erst in den nächsten State (Schritt) gesprungen werden, wenn der aktuelle Schritt erfolgreich abgearbeitet ist und alle Bedingungen für einen Sprung in den nächsten Schritt erfüllt sind. Eine Besonderheit ist, dass das Ziel nicht unbedingt der nächste Schritt sein muss. Es kann auch aus verschiedenen Positionen bei einer bestimmten Bedingung wieder an den Anfang gesprungen werden.
Theoretisch kann sogar auch aus jedem Schritt in jeden beliebigen anderen Schritt gesprungen werden.

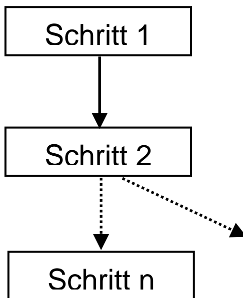


Abb. 9.3: Struktur einer Statemachine

Der Aufbau solch einer Statemachine ist recht einfach realisiert: Alle Schritte der Statemachine sind als kleine Unterprogramme angelegt, hier z. B. die Funktionen *BEGIN* und *BEGIN1*, die den Anfang einer Datenübertragung suchen.

```
; BEGIN
;      Diese Funktion sucht den Anfang eines Datenstreams.
BEGIN
  btfsc  RXDATA                  ; Daten Anfang?
   incf  STATECNTR, F            ; Ja, erhöhe den State-Zähler um 1
  goto   MAIN                    ; Nein, springe zurück zu MAIN

BEGIN1
  btfsc  RXDATA                  ; Wirklich Daten Anfang ?
   goto  MAIN                    ; Nein, zurück

  call   SETWATCH                ; Ja, starte Stoppuhr
  incf   STATECNTR, F            ; Erhöhe den State-Zähler um 1

  goto   MAIN                    ; Springe zurück zu MAIN
```

Abb. 9.4: Unterprogramme Begin und Begin1

Hier sieht man, dass in beiden Unterprogrammen der Zähler mit dem Namen *STATECNTR* bei Erreichen einer bestimmten Bedingung um eins erhöht wird. Wird die geforderte Bedingung nicht erfüllt, wird zu dem Label *Main* gesprungen. Das kommt einem *RESET* gleich, da dort dann der *Statecounter* gelöscht und somit die Schrittkette wieder von vorne abgearbeitet wird.

Würde man nun in einem State (Schritt) den Counter nicht nur um eins erhöhen, sondern eventuell auch gezielt mit einer bestimmten Zahl laden, wäre auch ein Sprung in jeden anderen Schritt möglich.

Wenn man sich dazu jetzt noch den Programmteil *Main* ansieht, erkennt man, wie das gelöst ist: In *Main* wird der Inhalt des Statecounters nicht direkt ausgewertet. Der Inhalt des Registers *STATECNTR* wird auf das Register *PCLATH* aufaddiert, wodurch ein definierter Sprung um den Wert im Register *STATECNTR* ausgelöst wird.

Was verbirgt sich hinter dem Register PCLATH und PCL? Wenn man Informationen zu Fehlermeldungen oder besonderen Begriffen sucht, ist es hilfreich, die Stichwörter in der Hilfe von MPASM-Assembler, oder einer der anderen Rubriken, einzugeben. Gibt man die beiden genannten Begriffe ein, erhält man folgende Antworten:

PCLATH Program Counter High Byte Latch
PCL Program Counter Low Byte

Da der Programmcounter der PICs aber im Gegensatz zu allen anderen Registern 13 Bit breit ist, besteht er organisatorisch aus diesen zwei Registern. Da eine direkte Adressierung nicht möglich ist, muss er besonders behandelt werden.

Mit dem Programmcounter wird bestimmt, welcher Befehl als Nächstes abgearbeitet wird. Beeinflusst man den Programmcounter gezielt, durch Addition, Subtraktion oder Zuweisen eines bestimmten Werts, kann man im Programm gezielt hin- und herspringen.

Der Aufbau des Programmcounters wird im Datenblatt beschrieben.

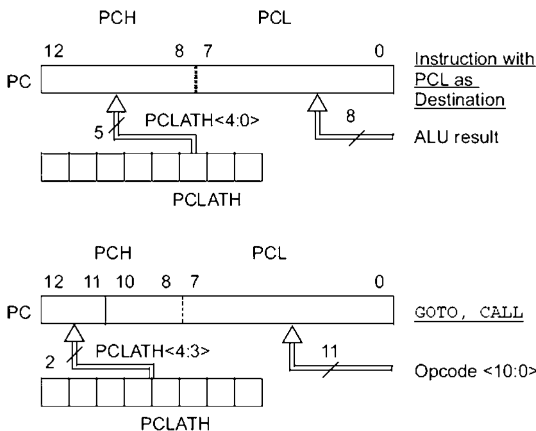


Abb. 9.5: Aufbau des Programmcounters

Alle Operationen, die auf diesem Register ausgeführt werden, unterliegen aber weiterhin den Restriktionen einer 8-Bit-Operation. Führt man mit diesem Register Rechenoperationen aus, muss man bedenken, dass es hier keinen automatischen Übertrag in die oberen Bits gibt. Auch gibt es keine Statusbits, die im Zusammenhang mit dem Register ausgewertet werden können. Hier muss ein Übertrag vom Programmierer gesondert erkannt werden.

Weitere Informationen zum Umgang mit dem Programmcounter findet man auch in der Technical Note AN556 von Microchip.

Da das Programm des Empfängers nicht für einen bestimmten Controller geschrieben ist, wird zuerst mit dem Operator *high*, oder in diesem speziellen Fall *HIGH STATEM*, das High-Byte des angegebenen Operators ausgelesen. Dieser Wert wird dann in den oberen Teil des Programmcounters geladen. In den nächsten Befehlen wird auf den unteren Teil des Programmcounters der Wert der Statemachine aufaddiert. Um Fehler zu vermeiden, werden die oberen vier Bits des Zählers aus Sicherheitsgründen schon vor der Addition ausmaskiert. Durch diese Addition zeigt nun der Programmcounter auf eine der folgenden GOTO-Zeilen. Ist der Wert größer als beabsichtigt, wird durch die GOTO-RESET-Zeilen der Controller automatisch neu gestartet.

Fertig ist die Statemachine bzw. eine Schrittketten-Programmierung. Länge und Umfang sind natürlich beliebig und müssen nicht, wie hier, immer 10 Schritte betragen.

Hier das Listing zur Statemachine aus dem Demoprogramm:

```
movlw  HIGH STATEM    ; high Return high byte of operand  movwf  PCLATH
movf   STATECNTR, W   ; Mask out the high order bits of
andlw  B‘00001111‘    ; STATECNTR (a noise guard)

addwf  PCL, F         ; The program clock (PCL) is incre STATEM
goto  BEGIN           ; mented by STATECNTR in order
goto  BEGIN1          ; to go to the appropiate routine
goto  HEADER
goto  HEADER1
goto  HIGHPLSE
goto  LOWPULSE
goto  RECORD
goto  WAIT4END
goto  VALIDATE
goto  IMPLEMNT
goto  RESET           ; These RESET commands correct
goto  RESET           ; erroneous values of STATECNTR
goto  RESET           ; not caught by the mask above.
goto  RESET
goto  RESET
goto  RESET
```

Abb. 9.6: Statemachine im Hauptprogramm

Dieser Programmaufbau kann natürlich ohne Weiteres auch in andere Programme übertragen und für andere Programmstrukturen mit anderen Aufgabenstellungen angepasst werden.

Weitere Details und Informationen erhalten Sie in den folgenden Anwendungsbeispielen.

22 Stichwortverzeichnis

Symbole

#define 54
#include 55

A

ACQUISITION TIME 51
ADCON0 73
ADRESH 96
ADRESL 96
ASK 15, 20, 21
Ausschaltzeit 88

B

BREAK 102
BTFFC 91
BTFFS 91

C

Carry-Bit 109
cblock 53
CCPR1H 84
CCPR1L 84, 93
Code Module Library 72, 76
COMF 92

E

Einschaltzeit 88
endc 53
ERROR[173] 34

F

FSK 15, 20

G

GPIF 45
GPIOx 56

H

hochohmig 28, 90

K

Konfiguration 18
Konfigurationsbits 18

L

Line Numbers 40

M

MCLR 62

N

no.asm file 38

O

ORG 0x000 47
ORG 0x004 47

P

PCL 58
PCLATH 58
PORTAx 56
PORTBx 56
PS 14

R

RESET 19
RESET-Eingang 17

S

Sendeleistung 14, 15
Statemachine 57, 59

T

tristate 28, 90

U

UAD 68, 69

V

VCFG 73, 97

W

W-Register 109

FRANZIS
PC+ELEKTRONIK

Archiem Mueller-Wolkensteinn

Daten und Signale kabellos mit rfPICS übertragen

Das Buch wendet sich an Elektroniker und Praktiker, die digitale Daten oder Signale kabellos übertragen möchten, ohne dabei in die Materie der Nachrichtentechnik einsteigen zu wollen.

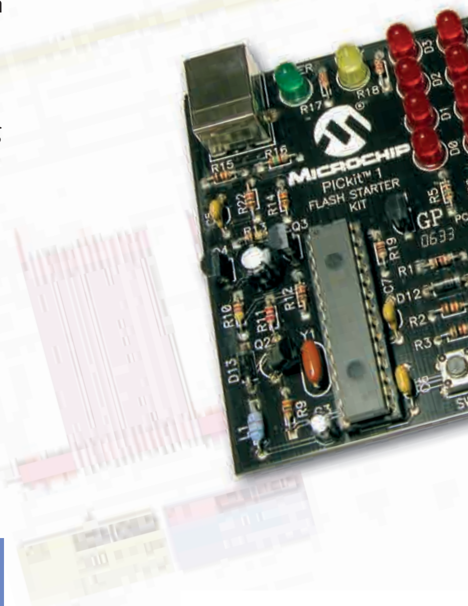
Dabei wird auf ein preiswertes Entwicklungstool mit einem Mikrocontroller der Firma Microchip, das „rfPIC Development Kit1", Bezug genommen. Enthalten sind ein Sender, Empfänger, Brenner und alle Programmiertools, die nötig sind, um einfach und schnell eine Lösung zur kabellosen Datenübertragung zu finden.

Zu Beginn des Buchs werden die benutzte Hardware und die zusätzlich verwendeten Controller näher erklärt. Dabei wird auf die wesentlichen Punkte, die für eine erste Inbetriebnahme des Kits erforderlich sind, eingegangen.

Anhand von Beispielen wird die Funktionsweise der Datenübertragung betrachtet. Diese beginnt bei der Übertragung eines einzelnen Bits und steigert sich in kleinen Schritten bis zur Übertragung eines vorzeichenbehafteten Analogwerts zum Steuern eines Fahrzeugs.

Das Demoprogramm wird dabei als Werkzeug benutzt und für jedes Beispiel mit kleinen Erweiterungen an die Aufgabenstellung angepasst. Zu allen Beispielen findet sich eine kleine Anwendung mit einem dazu passenden Platinenvorschlag im Buch.

Selbstverständlich werden auch alle 35 Assembler-Befehle der 8-Bit-PIC-Mikrocontroller ausführlich erörtert.



Aus dem Inhalt:

- Grundlagen
- Schaltungen
- Praktische Projekte
- Beispiele

ISBN 978-3-7723-4340-7

Euro 19,95 [D]

Besuchen Sie uns im Internet **www.franzis.de**