BERGISCHE UNIVERSITÄT WUPPERTAL
FACHBEREICH C - PHYSIK
ELEKTRONIKPRAKTIKUM

Version 4.2011, TEX: 8. April 2011

Versuch EP11 Digitalelektronik

Teil 3: Mikrocontroller

I. Zielsetzung des Versuches

Nachdem Sie in den letzten beiden Versuchen einfache Digital-ICs und programmierbare Logikbausteine kennengelernt haben, soll es im heutigen Versuch um Mikrocontroller und deren Programmierung gehen.

II. Vorkenntnisse

Grundlagen der Digitaltechnik (Versuch EP10)

Eigenschaften von und Unterschiede zwischen Logikschaltungen (CPLD, FPGA) und Mikrocontrollern.

Die Kenntnisse zum Umgang mit der Programmierumgebung finden Sie in dieser Beschreibung.

III. Theorie zum Versuch

1. Mikrocontroller

1.1. Aufbau

Zentraler Bestandteil jedes Computers ist der Mikroprozessor. Dieser Baustein übernimmt nur die eigentliche Rechenleistung. Die Kommunikation mit der Umgebung, das Abspeichern von Daten und das Programm usw. ist auf anderen Bausteinen untergebracht.

Bei einem **Mikrocontroller** (auch μController, μC, MCU) sind aber alle Funktionen in *einem* Baustein, auf einem Chip vereinigt. Auch der Arbeits- und Programmspeicher ist dort lokalisiert.

Ein Mikrocontroller ist daher praktisch ein Ein-Chip-Computersystem.

Moderne Mikrocontroller haben häufig auch komplexe Peripheriefunktionen wie z.B.

- Universal Serial Bus (USB)
- CAN-Bus (Controller Area Network)
- serielle Schnittstelle (RS232)
- Schnittstellen für andere Peripheriebausteine (SPI- oder IIC- bzw. TWI-Bus)
- Ethernet-Schnittstellen
- Pulsbreitenmodulierte Ausgänge (PWM) zur Erzeugung analoger Spannungen
- Analog-Digital-Wandler (ADC) zur Messung analoger Spannungen
- Schnittstellen für Flüssigkristall-Anzeigen (LCD-Controller und -treiber)

1.2. Unterschiede zwischen programmierbarer Logik und Mikrocontrollern

Programmierbare Logik, insbesondere die CPLDs, die Sie im letzten Versuch kennengelernt haben, arbeitet „in Echtzeit“. Die Eingangssignale werden sofort von den Gattern und Flipflops in der gewünschten Weise verknüpft und an die Ausgänge gegeben. Es können dabei sehr viele Bits gleichzeitig verarbeitet werden.

Daher sind diese Bausteine sehr schnell und können oft Signalfrequenzen von mehreren 100 MHz verarbeiten.

Komplexe Rechenfunktionen sind aber — vor allem bei den einfacheren Bausteinen — nicht oder nur mühsam zu realisieren. Bei den wesentlich komplexeren FPGAs sind auch komplexe Rechnungen möglich, sie sind aber oft teuer.

Mikrocontroller arbeiten nach dem Prinzip eines Computers. Sie arbeiten ein bestimmtes Programm *nacheinander* (sozusagen Zeile für Zeile) ab, nehmen Eingangssignale entgegen, verknüpfen und vergleichen sie, geben sie danach als Ausgangssignale aus oder speichern etwas ab.

Daher sind diese Bausteine relativ langsam und können nur Signalfrequenzen von einigen MHz verarbeiten. Das liegt auch an den geringen Taktfrequenzen, die auch aus Gründen geringer Stromaufnahme oft nur bei höchstens einigen 10 MHz liegen.

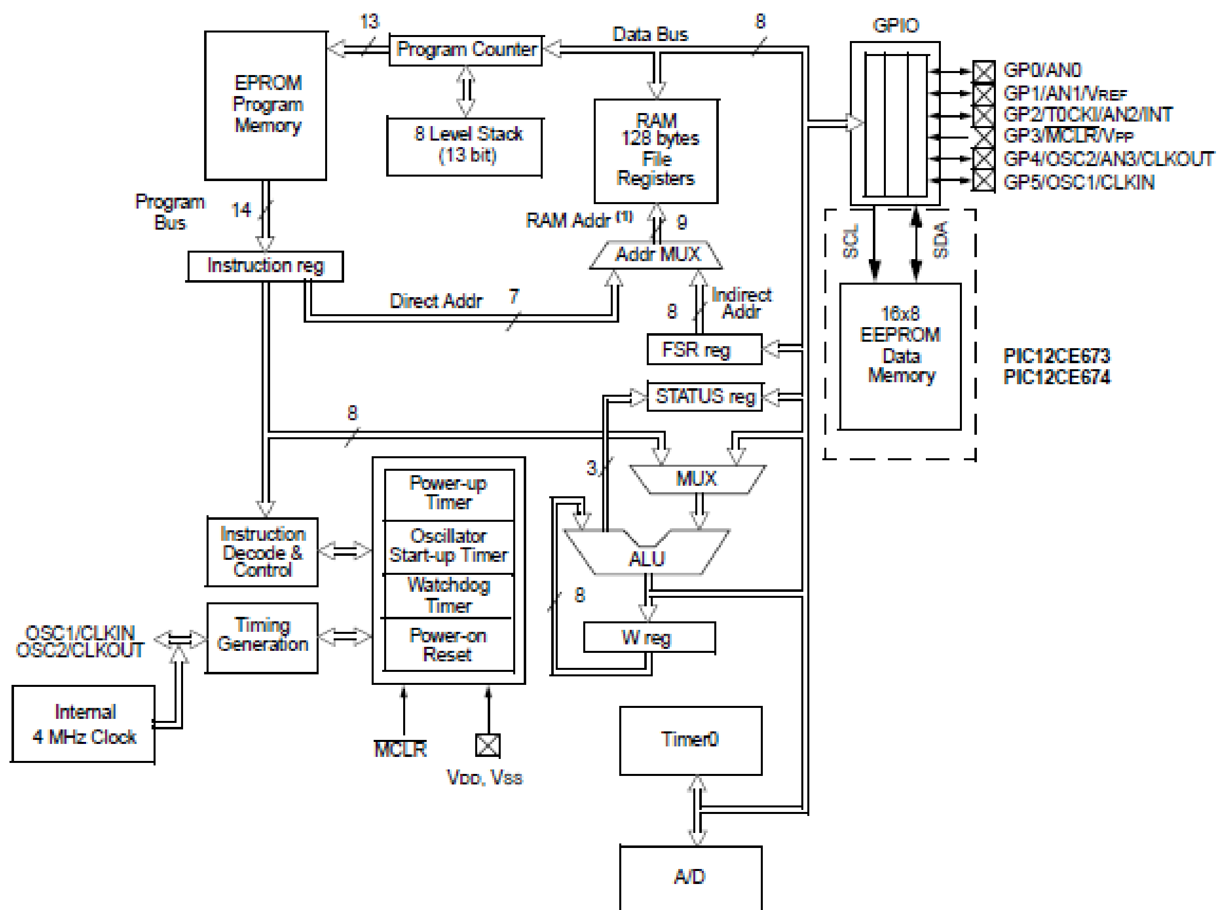
Komplexe Rechenfunktionen sind aber leicht zu realisieren.

Das Programm kann aus elementaren Programmierbefehlen (d.h. im Assemblercode) aufgebaut werden. Üblich weil wesentlich einfacher und bequemer ist aber die Verwendung einer Hochsprache wie BASIC oder C.

1.3. Mikrocontroller am Beispiel des PIC12

Als Mikrocontroller verwenden Sie im Versuch den Typ PIC18F4450 des Herstellers Microchip.

Da dessen Innenleben auf den ersten Blick sehr komplex ist (siehe Anhang), betrachten wir als Blockschaltbild den wesentlich einfacheren PIC12.



Zentraler Bestandteil ist die ALU (arithmetisch-logische Einheit, englisch arithmetic logic unit). Sie erzeugt alle möglichen logischen Verknüpfungen und Schiebe- oder Zählvorgänge. Dabei werden immer z.B. 8 Bit verarbeitet (bei größeren ALUs auch 16 oder 32).

Die Art der Verknüpfung und deren Reihenfolge ist im Programmspeicher abgelegt (Programm Memory, links oben).

Für den zeitlichen Ablauf sorgt ein interner oder externer Taktgenerator (links unten).

Die Verbindung mit der Außenwelt passiert über sogenannte Ports, das sind (oft 8 Bit breite) Gruppen von Anschlüssen (rechts oben).

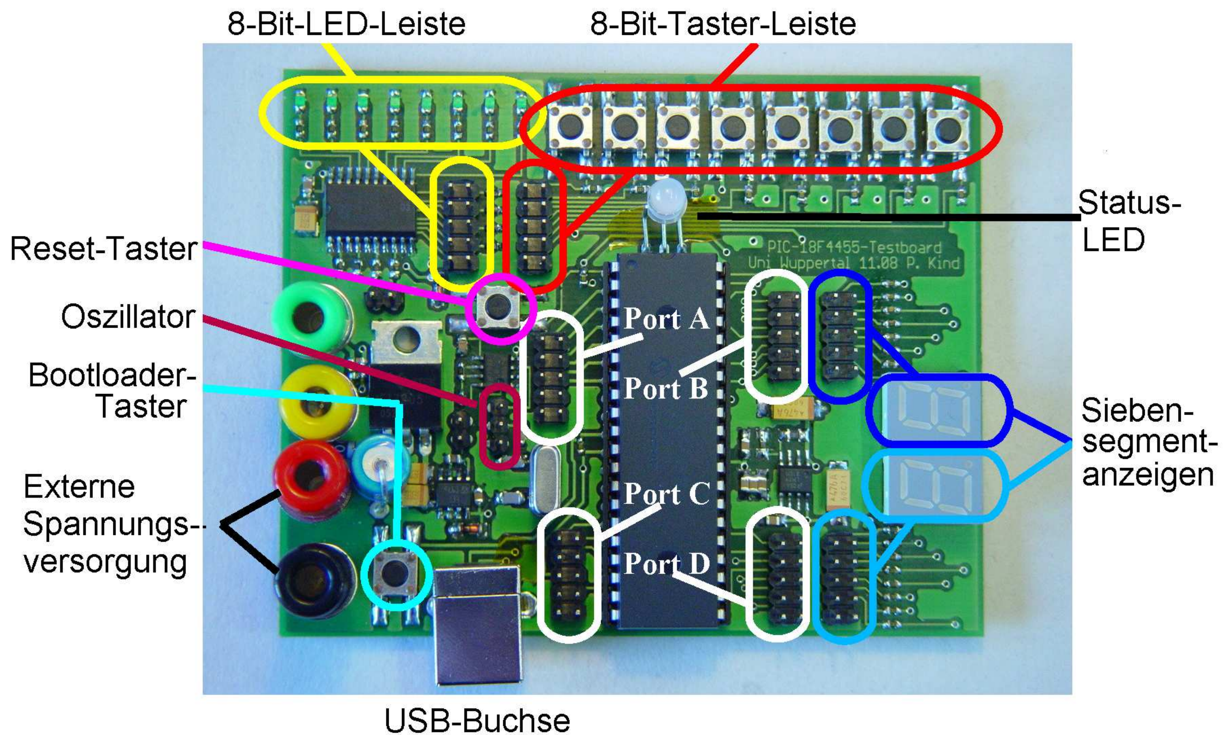
Zusätzliche Zähler (Timer) oder Funktionsgruppen zur Umwandlung von Analogsignalen (Spannungen) in Digitalwerte und zurück (A/D, rechts unten) vervollständigen den Mikrocontroller.

Oft sind noch weitere Funktionsblöcke vorhanden, mit denen standardisierte Datenübertragungsprotokolle realisiert werden, wie z.B. seriell (RS232) oder — wie auf Ihrem Versuchsboard — der USB-Anschluß.

IV. Hinweise zum Versuchsaufbau

Zur Programmierung eines Mikrocontrollers (PIC18F) haben wir eine kleine Leiterplatte vorbereitet. Dieses „Eval-Board“ ist dem CPLD-Board ähnlich, das Sie bereits aus dem Versuch zur programmierbaren Logik kennen.

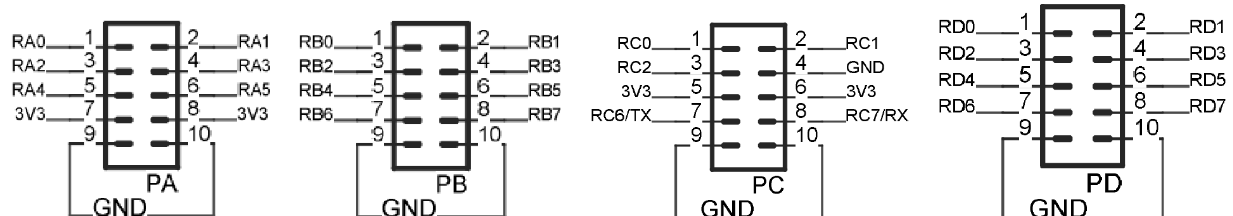
1. Das Eval-Board



Das PIC-Eval Board stellt diverse Peripherie zur Verfügung, darunter:

8 einzelne LEDs, 8 Taster nach 3,3V und Pull-Down-Widerstände, 2 Siebensegment-Anzeigen
1 Oszillator (ca. 6 kHz), 1 Referenz-Spannungsquelle (2,5 V)

Die Verbindung mit der Peripherie erfolgt über 10-adrige Flachbandkabel mit Doppelreihensteckern. Dabei sind die IO-Ports auf untereinander ähnliche Weise herausgeführt:



Bei den restlichen doppelreihigen Anschlüssen sind die beiden unteren Pins (9 und 10) ebenfalls mit Masse (GND) belegt. Achten Sie daher darauf, die Flachbandkabel nicht verdreht anzuschließen! Die farbige Ader sollte immer auf der gleichen Seite der Stecker sein, entweder oben oder unten.

Der Takt des Oszillators liegt am obersten Pin des 3poligen Oszillator-Anschlusses, an die beiden unteren kann ein zusätzlicher Kondensator angeschlossen werden, um die Taktfrequenz zu drosseln.

Der Mikrocontroller selbst und die Status-LED wird aus dem USB-Bus versorgt. Die beiden Siebensegmentanzeigen und die 8 LEDs brauchen aber zusammen viel Strom, der nicht immer aus dem USB-Bus bezogen werden kann. Daher gibt es wie beim CPLD-Board eine externe Spannungsversorgung. Schließen Sie daher an die rote und schwarze Banenenbuchse ca. 9 V an, wenn es mit der Versorgung aus dem USB-Bus Probleme gibt.

Die gelbe und grüne Bananenbuchse führt wie beim CPLD-Board an zwei Steckerpins (rechts neben der grünen Buchse). Der Pin nahe der grünen Buchse ist mit dieser verbunden, der andere Pin geht zur gelben Buchse. An die beiden Pins kann ein kleines Prüfkabel (für die Pins der 10poligen Port-Stecker) angeschlossen werden; die

Signale können dann über die gelbe bzw. grüne Bananenbuchse und Bananenkabel z.B. zum Oszilloskop gesendet werden. Dabei muß die schwarze Bananenbuchse (GND) mit der schwarzen Bananenbuchse des Oszilloskops (GND) verbunden werden.

2. Programmierung des Mikrocontrollers

2.1. Das Öffnen der Entwicklungsumgebung

Entweder: Desktop → MPLAB IDE v8.10

Oder: Start → Alle Programme → Microchip → MPLAB DIE v8.10 → MPALB IDE

2.2. Ein Projekt anlegen

Sie sehen nun die Entwicklungsumgebung vor sich. Um ein neues Projekt anzulegen, verwenden Sie am einfachsten unser vorgefertigtes Projekt:

Gehen Sie auf Projekt → Open und wählen Sie G:\EpraktTemplate\EpraktTemplate (die Datei heißt so wie der beinhaltende Ordner). Sollte diese Datei nicht vorhanden sein, müssen Sie sich einzelne Dateien zusammenstellen, siehe Anhang.

Sie sehen nun im Projektfenster links eine lange Liste mit verschiedenen Dateien, die jetzt automatisch zusammengestellt worden sind.

Das Projekt sollte sich nun kompilieren lassen. Drücken Sie dazu die Taste [F10], oder wählen Sie im Menü „Project“ die Option „Make“ um das Projekt zu kompilieren und ihre Einstellungen zu verifizieren. Die Entwicklungsumgebung zeigt Ihnen den Fortschritt und das Ergebnis des Kompilierens an, indem es das „Output“-Fenster öffnet. Wenn der Compiler seine Arbeit beendet hat, teilt er Ihnen in der letzten Zeile des „Output“-Fensters seinen Erfolg oder Misserfolg mit, indem er die Nachricht BUILD SUCCEEDED oder BUILD FAILED ausgibt.

Wenn der Code sich nicht kompilieren lässt und man keine Syntaxfehler findet, kann es helfen, alle Dateien des Projekts neu zu kompilieren. Dazu wählt man im „Project“-Menü „Build All“ oder drückt [Strg]+[F10].

2.3. Eigenen Programmcode einbinden

Wir haben an diesem Praktikumsnachmittag nicht genug Zeit, um alles selbst zu programmieren, was der Mikrocontroller braucht, um z.B. über die USB-Schnittstelle mit dem PC zu kommunizieren. Insbesondere benötigt der Mikrocontroller eine Software, damit er über die USB-Schnittstelle (d.h. auch ohne spezielles Programmiergerät) programmiert werden kann. Diese sogenannte Bootloader-Software ist bereits in den Mikrocontroller programmiert worden.

Einige weitere Programmteile sind bereits vorgefertigt und in den Projektdateien enthalten, z.B. für die Nutzung der USB-Schnittstelle, die Ansteuerung der zweifarbigen Status-LED und einige Routinen zur Konfiguration der ADC-Ports.

Sie müssen daher keine weiteren Vorbereitungen mehr treffen und können gleich damit beginnen, Ihr eigenes Programm zu schreiben. Einige kleine Beispielprogramme sind vorbereitet und nachfolgend beschrieben. Zum Verständnis sollten Sie sich mit Teilen der Mikrocontroller-Hardware vertraut machen (siehe Kapitel 3.).

Sie schreiben Ihren Programmcode in die Datei `user.c`, die sich im Projektverzeichnis befindet[1]. Diese Datei enthält bereits andere Programmteile, die Sie nicht verändern sollten. Wichtig für Sie ist der folgende Abschnitt etwa in der Mitte der user.c-Datei. Es handelt sich um die `ProcessIO()`-Routine.

```
void ProcessIO(void)
{
   BlinkUSBStatus();
    // User Application USB tasks
    if((usb_device_state < CONFIGURED_STATE)||(UCONbits.SUSPND==1)) return;

    // Fügen Sie hier Ihren Code ein ####################################

    // handleCommands();      // Wertet Kommandozeichen aus,
```

[1]Beim Start der User-Firmware wird immer die `main(void)`-Routine in der main.c-Datei aufgerufen, diese ruft dann die `user.c` auf

```
                                          // die über USB geschickt wurden
                                          // kann erstmal auskommentiert bleiben!

    // User-Code Ende #####################################################
}//end ProcessIO
```

Im Bereich zwischen den Zeilen `// Fügen Sie hier Ihren Code ein ###` und `// User-Code Ende ###` kann eigener Code in den Programmablauf eingefügt werden. Es bietet sich allerdings wegen der größeren Übersichtlichkeit an, eigenen Code zunächst in eine eigenen Routine zu schreiben, die unmittelbar vor der `ProcessIO()`-Routine stehen muß.

Ihre neue Routine rufen Sie dann in der `ProcessIO()`-Routine auf.

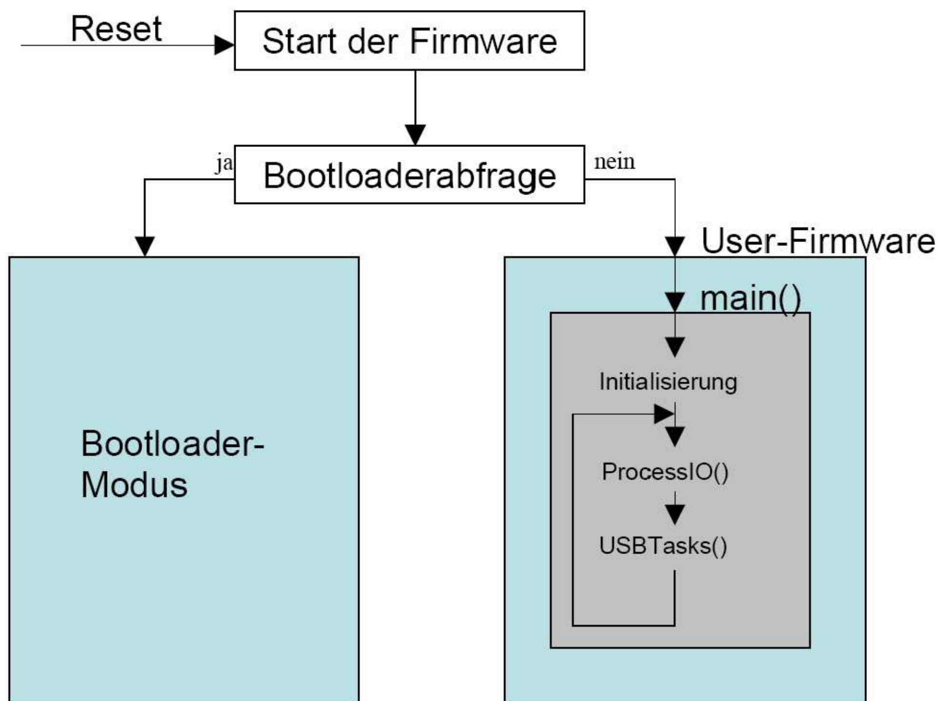
Code, welcher nur einmal abgearbeitet werden soll, lässt sich in die `UserInit()`-Routine einbauen, die sich ebenfalls vor der `ProcessIO()`-Routine befindet.

Für die ersten Teile der Versuchsdurchführung geben wir Ihnen Programmbeispiele. Weitere Informationen zum Mikrocontroller, insbesondere eine Liste der wichtigsten Befehle, finden Sie im Anhang ab Seite 17.

Der Mikrocontroller startet sein Programm nach dem Anlegen der Versorgungsspannung oder nach Drücken und Loslassen des Reset-Tasters (siehe Abbildung in Kap. 1.). Als erstes fragt der Mikrocontroller den Bootloader-Taster ab, denn es sind zwei Fälle zu unterscheiden:

1. Der Bootloader-Taster ist gedrückt. Dann erwartet der Mikrocontroller, mit dem von Ihnen geschriebenen Programm (auch User-Firmware genannt) programmiert zu werden.

2. Der Bootloader-Taster ist nicht gedrückt. Dann wird die Bootloader-Firmware übersprungen und der Prozessor startet die User-Firmware.

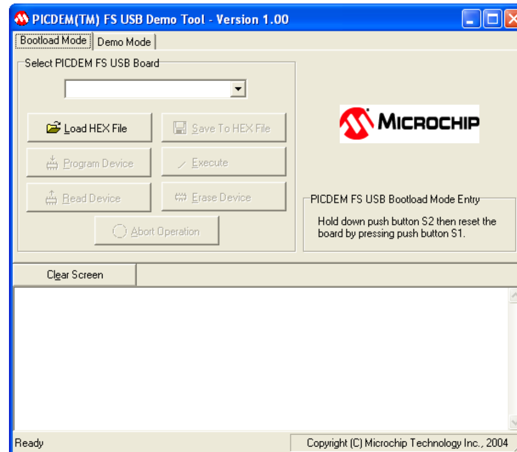


Flussdiagramm der Firmware

2.4. Programmieren des Mikrocontrollers (Brennen der User-Firmware)

Nach dem Kompilieren müssen Sie ihr Programm (die User-Firmware) zunächst in den Speicher des μC übertragen, um sie dort ausführen zu können. Dazu wird ein Demo-Tool der Firma Microchip genutzt, welches die Hex-Datei mit der kompilierten User-Firmware an den Mikrocontroler auf dem Eval-Board überträgt. Starten Sie dazu das „PICDEM FS USB Demo Tool“ auf dem Desktop oder unter

Start → Alle Programme → Microchip → MCHPFSUSB v2.3. → „PICDEM FS USB Demo Tool“.



Brennprogramm der Firma Microchip

Das Programmieren des Mikrocontrollers läuft nun so ab:

- Zuerst muss der μC in einen Zustand gebracht werden, indem er die Firmware über die USB-Schnittstelle entgegen nehmen kann (Bootloadermodus). Wie bereits erwähnt erreicht man das, indem man bei Start der Firmware die Bootloader-Taste gedrückt hält (siehe Kapitel 1.). Einen (Neu-)Start der Firmware erreicht man durch Betätigen des Reset-Tasters (siehe Kapitel 1.).
- Wenn sich das Board im Bootloadermodus befindet und vom Rechner korrekt erkannt wurde, sollte die Status-LED abwechselnd Rot und Grün blinken.
- Ist dies der Fall, kann man im Brenntool im Drop-Down-Menü (die Zeile unter Select PICDEM FS USB Board) einen Eintrag „*PICDEM FS USB 0 (Boot)*“ auswählen.
- Wenn Sie das getan haben, laden Sie mit der Schaltfläche [Load HEX File] ihre kompilierte HEX-Datei. Sie befindet sich im „output“-Ordner des Projektverzeichnisses, wenn das Projekt nach der obigen Anleitung erstellt wurde. Wenn eine Warnmeldung[2] kommt: *Configuration data contained in this hex file is different from the boards default setting.*, dann klicken Sie auf Ja/Yes.
- Nun können Sie über die Schaltfläche [Program Device] den μC programmieren. Die LED leuchtet währenddessen permanent rot.Warten Sie, bis der Vorgang abgeschlossen ist und starten Sie dann ihre Firmware über die [Execute]-Schaltfläche oder durch die Resettaste auf dem Board. Dabei sollten Sie die Bootloadertaste natürlich nicht gedrückt halten, es sei denn, Sie möchten erneut in den Bootloader-Modus.

[2] Einige Bits können aus dem Mikrocontroller nicht richtig ausgelesen werden; die Warnmeldung erscheint daher praktisch immer.

V. Versuchsdurchführung: Einfache Programme schreiben und auf dem PIC-Mikrocontroller testen

1. Hinweise zur Protokollierung

Geben Sie zu jedem Arbeitsschritt eine kurze Beobachtung und den Quellcode an!

Kommentieren Sie ihren Quellcode! Erstellen Sie für jeden Aufgabenteil eine neue Routine und ändern Sie entsprechend den Aufruf in `ProcessIO()`.

2. Messen der Arbeitsgeschwindigkeit des PIC18F4455

2.1. Schnellsmögliches Programm, d.h. ohne Verzögerungsbefehle

Um ein Gefühl für die Geschwindigkeit zu bekommen, mit der die PIC18-Prozessoren ihre Programme abarbeiten, soll eine Firmware programmiert werden, welche einen IO-Pin abwechselnd auf logisch 1 und logisch 0 schaltet. Öffnen Sie dazu die Datei `user.c` in Ihrer Programmierumgebung MPLAB. Fügen Sie oberhalb der `ProcessIO()`-Routine eine neue Routine ein, welche Sie `void test_pic_speed()` nennen. Diese soll den Code enthalten, um einen IO-Port als Ausgang zu schalten und ihn in einer Endlosschleife möglichst schnell zwischen seinen beiden Zuständen umschalten zu lassen[3].

Rufen Sie Ihre neu erstellte Routine in `ProcessIO()` oberhalb von `handleCommands()` auf. Da es nicht benötigt wird, kann `handleCommands()` vorerst auskommentiert bleiben.

Ihr Programm könnte für z.B. so aussehen, wenn Sie im Port A das Bit 0 schalten, also das Signal RA0. (Auf welchem Pin des Steckers PA liegt RA0?). Weitere Informationen zur Ansteuerung der Pins finden Sie in der Tabelle auf Seite 19.

```
void test_pic_speed(void){

TRISA = 0b000000;            // alle 6 Pins (Bits) von Port A sollen Ausgänge sein

while(1){                    // Eine Endlosschleife: while(1),
                             //   d.h. "solange es wahr (1) ist ..."
         LATA  = 0b000001;   // Schreibe eine 1 in das Bit 0, d.h. setze RA0 auf 1
         LATA  = 0b000000;   // Schreibe eine 0 in das Bit 0, d.h. setze RA0 auf 0
        }                    // Ende der while-Schleife, d.h. jetzt geht
                             //   es wieder von vorn los, also von hinter while(1)
                             //   mit dem Befehl "setze RA0 auf 1"
}                            // Ende der Routine   test_pic_speed

void ProcessIO(void)
{
    BlinkUSBStatus();
    // User Application USB tasks
    if((usb_device_state < CONFIGURED_STATE)||(UCONbits.SUSPND==1)) return;

      // Hier folgt Ihr Code ###############################################

    test_pic_speed();      // Hier wird die Routine test_pic_speed aufgerufen,
                           // die einige Zeilen weiter oben programmiert worden ist

    // handleCommands();   //Wertet Kommandozeichen aus, die über USB geschickt wurden
                           // kann erstmal auskommentiert bleiben!

    // User-Code Ende #######################################################
}//end ProcessIO
```

[3] Da USB-Funktionen nicht benötigt werden, ist es nicht erforderlich, in Ihrem Programm die Funktion `USBTasks()` aufzurufen

Verbinden Sie den benutzten Port über ein Flachbandkabel mit der LED-Leiste. Eine der LEDs wird nun schwach leuchten. Dies ist eine Täuschung Ihres trägen Auges, eigentlich blinkt die LED mit einigen kHz, evtl. sogar einigen MHz.

Entfernen Sie das LED-Flachbandkabel wieder und messen Sie die Spannung am entsprechenden IO-Pin mit dem Oszilloskop. Verwenden Sie dazu das rot-gelbe Prüfkabel, um den IO-Pin mit der gelben oder grünen Bananenbuchse zu verbinden. Machen Sie sich ein Bild davon, wie viel Zeit einzelne Befehle in Anspruch nehmen. Verwenden und untersuchen Sie, wenn möglich, mehrere Sorten von Endlosschleifen oder Wege, den IO-Pin umzuschalten.

2.2. Programm mit Verzögerungsbefehlen aus for-Schleifen

Bauen Sie zusätzlich eine Verzögerung in Ihre Routine ein, um einen Takt mit einer Periodendauer von genau einer Millisekunde zu erzeugen. Nutzen Sie dazu zunächst eine for-Schleife (siehe Tabelle im Anhang) mit entsprechender Abbruchbedingung.

```
void test_pic_delay_for(void){  //  Beginn der Routine
int i;                           // Die Variable i deklarieren

TRISA = 0b000000;               // alle 6 Pins (Bits) von Port A sollen Ausgänge sein

while(1){                        // Eine Endlosschleife: while(1),
                                 // d.h. "solange es wahr (1) ist ..."
         LATA  = 0b000001;      // Schreibe eine 1 in das Bit 0,
                                // d.h. setze RA0 auf 1
         for(i=0;i<10;i++){     // For-Schleife,  i++ bedeutet i = i+1
                    Nop();      // NOP = no operation = tue nichts
                          }     // Ende der for-Schleife
         LATA  = 0b000000;      // Schreibe eine 0 in das Bit 0,
                                // d.h. setze RA0 auf 0
         for(i=0;i<10;i++){     // For-Schleife,  i++ bedeutet i = i+1
                    Nop();      // NOP = no operation = tue nichts
                          }     // Ende der for-Schleife
        }                       // Ende der while-Schleife, jetzt geht es
                                // wieder von vorn los (setze RA0 auf 1)
}                               // Ende der Routine   test_pic_speed
```

2.3. Programm mit fertigen Verzögerungsbefehlen

Für Mikrosekunden-Verzögerungen können Sie auch den Befehl `delay_us()` verwenden (siehe Befehlsliste im Anhang), für Millisekunden den Befehl `delay_ms()`. Die while-Schleife in Ihrer Routine `test_pic_speed` könnte z.B. so aussehen:

```
while(1){                        // Eine Endlosschleife: while(1),
                                 // d.h. "solange es wahr (1) ist ..."

         LATA  = 0b000001;      // Schreibe eine 1 in das Bit 0,
                                // d.h. setze RA0 auf 1
         delay_us(25);          // Warte 25 Mikrosekunden
         LATA  = 0b000000;      // Schreibe eine 0 in das Bit 0,
                                // d.h. setze RA0 auf 0
         delay_us(25);          // Warte 25 Mikrosekunden
        }                       // Ende der while-Schleife, jetzt geht es
                                // wieder von vorn los (setze RA0 auf 1)
}                               // Ende der Routine   test_pic_speed
```

Welche Periodendauer erwarten Sie jetzt bei Ihrem Signal RA0? Messen Sie diese auch? (Falls nicht: Dann stimmt irgendwo in einem Unterprogramm eine Oszillatorfrequenz nicht und die Routine delay_us berechnet die Anzahl der Verzögerungsschritte falsch. Um welchen Faktor ist Oszillatorfrequenz falsch?)

3. Benutzung der Taster

3.1. Einfache Tasterabfrage

In dieser Aufgabe sollen Sie einen der 8 Taster dazu nutzen, eine der LEDs ein- und auszuschalten. Verwenden Sie dazu die Polling-Methode. Das heißt, fragen Sie den Pin des Tasters kontinuierlich auf Veränderung ab.

Beispiel: Wir verbinden den Taster mit Port B (Port C sollte es nicht sein[4]) und zwar mit Signal RB0. Die LED ist wie bisher an RA0. Dann wäre das einfachste Programm wie folgt:

```
void test_taster(void){
 byte i;                  // Zwischenspeicher für die Tasterzustände
 TRISA = 0b000000;        // alle 6 Pins (Bits) von Port A sollen Ausgänge sein
 TRISB = 0b11111111;      // alle 8 Pins (Bits) von Port B sollen Eingänge sein

while(1)
        {                 // Eine Endlosschleife: while(1) ...
         i = PORTB;       // Lese den ganzen Port B (über Eingangsregister PORTB)
                          // und speichere ihn auf i
         LATA  = i;       // Gebe den Inhalt von i  (über Ausgangsregister LATA)
                          // auf Port A aus
        }                 // Ende der while-Schleife, d.h. jetzt wieder von vorn
}                         // Ende der Routine test_taster
```

Vergessen Sie nicht, diese Routine auch einige Zeilen tiefer in der `ProcessIO` aufzurufen!

Weil wir den gesamten Port abfragen, fragen wir auch alle 8 Taster ab und können das Ergebnis an fast alle LEDs geben. Da der Port A aber nur 6 Bits hat und die Pins 7 und 8 immer auf 3,3 V liegen, leuchten die LEDs 7 und 8 immer. Welchen Port brauchen Sie für die LEDs, damit alle 8 Taster angezeigt werden?

3.2. Taster steuern Speicherfunktionen

Programmieren Sie nun eine Routine, welche wieder im normalen Firmwareverlauf aufgerufen wird und den Zustand eines Pins abfragt. Ist der Wert des Pins = 1, soll eine LED umgeschaltet werden, wenn der Pin den Wert 0 hat, soll nichts passieren. Das Setzen der TRIS-Register können Sie in der Routine `UserInit()` vornehmen. Vergessen Sie nicht, den Aufruf der Routine aus der ersten Aufgabe wieder zu entfernen! Verbinden Sie die benutzten Ports über Flachbandkabel mit der Taster-Zeile und der LED-Zeile.

Eine der LEDs sollte nun wieder schwach leuchten bzw. bei näherer Betrachtung schnell blinken, solange der Taster gedrückt ist.

Diese Aufgabe löst man üblicherweise mit einer if-Bedingung. Achten Sie auf die doppelten Gleichheitszeichen in der if-Klammer! Ein Beispiel:

```
void test_taster(void){
 byte i;                  // Zwischenspeicher für die Tasterzustände
 TRISA = 0b000000;        // alle 6 Pins (Bits) von Port A sollen Ausgänge sein
 TRISB = 0b11111111;      // alle 8 Pins (Bits) von Port B sollen Eingänge sein

while(1) {                     // Eine Endlosschleife: while(1), ...
   if(PORTB == 0b00000001){    // Wenn an Port B das Bitmuster 0000 0001 anliegt ...
      LATA  = 0b000001;        // Schreibe eine 1 in das Bit 0, d.h. setze RA0 auf 1
      }                        // (Ende des if-Blocks)
   delay_ms(25);               // Warte 25 Millisekunden
   if(PORTB == 0b00000001){    // Wenn an Port B das Bitmuster 0000 0001 anliegt ...
```

[4] Wenn Sie für die Taster den Port C verwenden, können Sie den Taster SW4 nicht verwenden. Wenn ein Taster gedrückt wird, liegt sein Pin auf 3,3 V. Am Pin 4 des Steckers PC ist aber GND, d.h. es gibt einen Kurzschluss!

```
      LATA  = 0b000000;       // Schreibe eine 0 in das Bit 0, d.h. setze RA0 auf 0
      }                       // (Ende des if-Blocks)
   delay_ms(25);              // Warte 25 Millisekunden
   }                          // Ende der while-Schleife, wieder von vorn ...
}                             // Ende der Routine test_taster
```

Diese Lösung hat noch einen kleinen Nachteil. Welchen? (Überlegen Sie was passiert, wenn man mehr als nur einen Taster drückt.)

Wenn man nur genau den einen Taster abfragen will, darf man auch nur dessen Bit prüfen. Dazu kann die PORTB-Abfrage verfeinert werden (siehe auch Tabelle auf Seite 19).

Ändern Sie Ihr Programm wie folgt ab:

```
if(PORTBbits.RB0 == 0b1)
```

3.3. Tasterabfrage und Verzögerungen

Um die LED wie gewünscht ein- und auszuschalten, müsste man den Taster wieder loslassen, bevor die Abfrage ein zweites Mal stattfindet. Da dies ein wahrscheinlich zu hohes Maß an Geschick erfordert, empfiehlt es sich, nach Umschalten der LED eine Zeit lang zu warten. Die Unterbrechung der Abfrage lässt sich am einfachsten mit der `delay_ms(byte)`-Funktion realisieren, welche eine Abarbeitungszeit des übergebenen Arguments in ms hat. Verzögern Sie um 500 ms.

Nun lässt sich die LED zwar komfortabel schalten, jedoch nur maximal mit einer Frequenz von 1 Hz.

3.4. Taster steuern einen Zähler

Wir wollen nun das Programm folgendermaßen ändern: Alle 8 LEDs sollen am Port D angeschlossen sein und den Inhalt eines 8-Bit-Binärzählers anzeigen. Der Zählerstand soll sich bei jedem Tastendruck um 1 erhöhen. Im Prinzip geht das mit folgender Struktur:

```
void test_taster(void){
   byte taster_alt;      // Zwischenspeicher für den alten Zustand des Tasters
   byte taster_neu;      // Zwischenspeicher für den aktuellen Tasterzustand
   byte counter;         // Ein 8-Bit-Zähler
   TRISD = 0b00000000;   // alle 8 Pins (Bits) von Port D sollen Ausgänge sein
   TRISB = 0b11111111;   // alle 8 Pins (Bits) von Port B sollen Eingänge sein

   while(1) {                            // Eine Endlosschleife (while):
      taster_neu = PORTB;                // Taster abfragen
      if (taster_neu != taster_alt){     // Wenn sich ein Tasterzustand geändert hat ...
         counter = counter + 1;          // Erhöhe Zähler um 1
         LATD = counter;                 // Der Port D mit den LEDs
                                         // bekommt den Zählerstand
         taster_alt = taster_neu;        // speichere den Tasterzustand
         }                               // Ende if-Block
   }                                     // Ende der while-Schleife
}                                        // Ende der Routine test_taster
```

Nun ändert sich der Zählerstand bei jedem Drücken *oder* Loslassen eines Tasters.

Wie müssen Sie das Programm ändern, damit der Zähler *nur beim Drücken* eines Tasters um 1 erhöht wird?

3.5. Kontaktprellen der Taster — eine Lösung

Wenn Sie genau hinschauen, werden Sie feststellen, dass der Zähler pro Tastendruck manchmal um mehr als 1 weitergelaufen ist. Das liegt am sogenannten Kontaktprellen, d.h. der Taster schließt nicht sauber, sondern springt einige Male auf und zu.

Effektiv ist es, die Ausführung der Firmware so lange zu unterbinden, bis der Benutzer die Taste wieder loslässt. Eine Möglichkeit dazu, bietet eine `while`-Schleife, welche solange ausgeführt wird, bis der Taster wieder zurückschaltet. Zur Sicherheit empfiehlt es sich auch hier, an diversen Stellen eine Verzögerung vorzusehen, um Kontaktprellen aus dem Weg zu gehen. Programmieren Sie die Tasterabfrage nach folgendem Schema:

```
wenn Taster gedrückt
     warte 10ms
     warte bis Taster nicht mehr gedrückt
     warte 10ms
```

An welcher Stelle im oben skizzierten Ablauf Sie die LED umschalten lassen wollen, bleibt ihnen überlassen, je nach dem, ob sie beim Drücken oder Loslassen schalten soll.

4. Zähler für Oszillatortakte

4.1. Zähler ohne Vorteiler

In dieser Aufgabe soll ein Zähler programmiert werden, dessen Takt liefert aber jetzt der auf dem Eval-Board befindliche Oszillator[5]. Da dieser allerdings mit mehreren kHz schwingt, muß der Takt mit einem Vorteiler heruntergeteilt werden, wenn man die Frequenz man mit dem menschlichen Auge wahrnehmen will.

Zunächst programmieren wir aber den Zähler selbst, ohne Vorteiler:

Da der Zähler selbst 8 Bit breit werden soll, brauchen wir auch einen so breiten LED-Port. Die Auswahl ist daher auf die Ports B oder D in beschränkt. Für den Zähler-Eingang, welcher mit dem Oszillator verbunden wird, können Sie einen beliebigen freien Pin wählen.

Achtung! Freie, also nicht angeschlossene Port-Pins fangen sich beliebige Störsignale ein und haben daher irgendeinen, nicht vorhersagbaren Zustand (0 oder 1). Sie können also nicht davon ausgehen, daß diese alle 0 oder 1 haben uns Sie dürfen daher nur den einen Pin abfragen/auswerten, der den Oszillatortakt bekommt bzw. weitere Pins, an denen z.B. ein Taster angeschlossen ist.

Programmieren Sie zunächst eine Routine, welche bei Statusänderung des Port-Pins eine Variable hochzählt (inkrementiert). Die Statusabfrage kann wie beim Taster erfolgen. Als Zähler-Variable können Sie direkt einen der Ports verwenden, also B oder D (d.h. PORTB oder PORTD). Fügen Sie ihren Code wieder in den normalen Firmwareablauf ein, also als Aufruf in `ProcessIO()`. Legen Sie außerdem, wie im ersten Versuch, eine Endlosschleife an.

Tipp: Um eine Änderung festzustellen, müssen Sie den aktuellen Wert mit einem alten vergleichen. Messen Sie die Frequenzen der 8 blinkenden LEDs. Was fällt auf?

4.2. Zähler mit Vorteiler

Da die LEDs bis jetzt nur „leuchtend“ wahrgenommen werden, muss die Zählerfrequenz noch heruntergeteilt werden. Fügen Sie dazu eine Variable ein, welche statt des LED-Ports inkrementiert wird. Immer wenn diese Variable einen bestimmten Wert übersteigt, soll der LED-Port inkrementiert werden.

Am einfachsten geht dies mit einer Byte-Variablen, welche mit jeder Taktflanke um eins erhöht wird. Nach 255 Flanken läuft diese über und fängt wieder bei 0 an zu zählen. Wenn man nun diese Variable auf einen bestimmten Wert prüft (z.B. 0) und beim Erreichen dieses Wertes den LED-Port inkrementiert, zählt der LED-Port mit einer Frequenz, welche 1/256 der Oszillatorfrequenz entspricht.

Messen Sie die Oszillatorfrequenz und programmieren Sie einen Teiler, der die LED-Zeile mit 5 Hz zählen lässt. Tipp: Der Variablentyp `word` umfasst einen Wertebereich vom 0 bis 65535.

[5]Gemeint ist der RC-Oszillator, nicht der 4-MHz-Quarzoszillator.)

5. Verwendung des Analog-Digital-Konverters (ADC)

Der Mikrocontroller besitzt auch einen ADC, mit Sie Spannungen von 0 bis etwa 4 Volt messen können[6]. Ein analoger Spannungswert wird dabei in einen digitalen Wert von 8 oder 10 Bits umgewandelt. Technische Einzelheiten finden Sie ab Seite 21.

Wir wollen den ADC in Betrieb nehmen und das Ergebnis auf verschiedene Weise anzeigen. So bauen wir uns ein einfaches Digitalvoltmeter.

5.1. Anzeige des ADC-Wertes auf 8 LEDs

Die einfachste Version eines Voltmeters besteht darin, die 8 Bit des ADC-Ergebnisses unverändert auf die 8 LEDs unseres Boards zu geben. Der ADC konvertiert in 10 Bits. Damit wir nur die obersten 8 Bits auf die LEDs bekommen, wird der 10-Bit-Wert in `read_adc(ADC_READ)` mit dem Befehl $\gg 2$ um zwei (Binär-)stellen nach rechts geschoben. Die unwichtigsten (unteren) beiden Bits fallen dadurch weg. Das Ergebnis wird dann auf den 8-Bit-Speicherplatz `value` übertragen.

Schreiben Sie die folgende ADC-Routine, und zwar wie die vorherigen oberhalb der `ProcessIO()`-Routine. Verschiedene Programmbefehle kennen Sie bereits, es ist alles kommentiert, die genaue Bedeutung der ADC-Befehle finden Sie ab Seite 21. Vergessen Sie nicht, diese neue Routine auch innerhalb der `ProcessIO()`-Routine aufzurufen und die alten Routinen zu entfernen oder zumindest auszukommentieren.

```
void adc_example (){           // Beginn der ADC-Routine
  char value;                  // Eine Speichervariable, 8 Bit
  TRISB = 0b00000000;          // Port B als Ausgang für die LEDs
  TRISAbits.TRISA0 = 1;        // Bit 0 von Port A als Eingang,
                               // das ist der ADC-Eingang

  setup_adc_ports(AN0_TO_AN1); // Die ADC-Pins 0 und (später) 1 werden
                               // gebraucht, d.h. Port A, Bits 0 und 1 können
                               // nicht mehr digital verwendet werden.
  setup_adc(ADC_CLOCK_DIV_32); // Konversionszeit ca. 30us
  set_adc_channel(0);          // Kanal auf AN0 setzten
  read_adc(ADC_START_ONLY);    // Konversion starten
  while (1) {                  // Endlosschleife
      delay_ms(10);            // Verzögerung von 10 ms
      value = read_adc(ADC_READ)>>2;  // Die obersten 8 Bit des ADC
                                      // nach  value  schreiben (siehe Text)
      LATB = value;            // value an Port B geben
            }                  // Ende von while
       }                       // Ende der ADC-Routine
```

Kompilieren Sie das Programm und laden Sie es in den Mikrocontroller. Verbinden Sie die LEDs über ein Flachbandkabel mit Port B. Achten Sie darauf, daß die rote Seite des Kabels an beiden Steckern in die gleiche Richtung weist.

Schließen Sie AN0 (= RA0), das ist Pin 1 vom Port-A-Stecker, über das zweiadrige (rot-gelbe) Meßkabel an den linken der beiden Stifte an, die sich rechts neben der grünen Bananenbuchse befinden. Dieser Stift ist (über 10 kΩ) mit der grünen Buchse verbunden, der danebenliegende Stift ist (über 10 kΩ) mit der gelben Bananenbuchse verbunden.

Schließen Sie einen der regelbaren Ausgänge Ihrer Spannungsquelle an die grüne (plus) und schwarze (minus) Bananenenbuchse an. Dadurch wird diese Spannung an Kanal AN0 gegeben[7].

[6] Je nach Einstellung kann bis 2,5 V oder bis zur Versorgungsspannung des Controllers, das sind etwa 4,3 V, gemessen werden.

[7] Wegen des 10-kΩ-Widerstandes besteht keine Gefahr für den ADC, falls die Spannung wesentlich höher als die Versorgungsspannung, also ca. 5 Volt, ist.

Machen Sie (mindestens) folgende Messungen. Beachten Sie dabei: LED1 hat das niederwertigste Bit (Binärwert 1), LED8 hat das höchstwertigste Bit (Binärwert 128)

- Stellen Sie 0 Volt ein. Was zeigen Ihre LEDs an? Welchem ADC-Wert (binär oder dezimal) entspricht das?
- Stellen Sie 5 Volt ein. Was zeigen Ihre LEDs an? Welchem ADC-Wert (binär oder dezimal) entspricht das?
- Bei welcher Spannung wird gerade eben der maximale Wert angezeigt?
- Bei welcher Spannung wird etwa der halbe maximale Wert angezeigt? Welchem ADC-Wert (binär oder dezimal) entspricht das?

5.2. Anzeige des ADC-Wertes auf der Siebensegmentanzeige

Die Anzeige auf 8 LEDs ist nicht besonders komfortabel. Aber wir haben ja die beiden Siebensegmentanzeigen. Außerdem gibt es bereits eine fertige Routine (siehe Seite 23), die 4 Bit in ein Siebensegmentmuster umwandelt. Die Bitmuster 0000 bis 1001, also 0 bis 9, werden als Ziffern 0 bis 9 dargestellt, die Bitmuster 1010 bis 1111, also 10 bis 15, werden — wie in Hexadezimalschreibweise üblich — als Buchstaben A,b,C,d,E,F dargestellt.

Verbinden Sie die beiden Anzeigen mit den Ports B und D und schreiben Sie ein Programm, das den ADC-Wert als zweistellige Hexadezimalzahl, also von 00 bis FF darstellt.

Tips:

- Mit dem Befehl `valH = value>>4;` holen Sie sich die oberen 4 Bit aus der Variable `value` heraus (warum?) und schreiben Sie in die variable valH.
- Was macht die folgende Programmzeile?: `LATD = dec_7seg[valH];`
- Sie können auch kleine Berechnungen aus Variablen und festen Zahlen programmieren, z.B. variable1 = variable2 - variable3 * 11.
- Wie kommen Sie damit an die unteren vier Bits von value?
- Sie können auch eine logische Verknüpfung machen. Z.B. erzeugt variable4 = (variable5 & 0b11000011) einen Wert, bei dem jedes der 8 Bits von variable5 mit dem entsprechenden Bit einer Binärzahl UND-verknüpft wird. Hier z.B. bleiben nur die beiden linken und die beiden rechten Bits übrig, die mittleren 4 Bits werden auf 0 gesetzt. Das geht natürlich mit beliebigen Binärzahlen, außerdem gibt es den Operator | für die OR-Verknüpfung, ! für die Invertierung.
- Vergessen Sie nicht, alle neuen Variablen (Speicherplätze) am Anfang Ihrer Routine zu definieren, so wie Sie `value` definiert haben. Es reicht oft der 8-Bit-Typ `char`, 4-Bit-Typen gibt es nicht

Wiederholen Sie mit dem so programmierten Mikrocontroller die Messungen des letzten Punktes.

- Stellen Sie 0 bzw. 5 Volt ein. Was zeigen Ihre Siebensegmentanzeigen für einen Hexadezimalwert?
- Bei welcher Spannung wird gerade eben der maximale Wert angezeigt?
- Bei welcher Spannung wird etwa der halbe maximale Wert angezeigt? Welcher Hexadezimalzahl entspricht das?

5.3. Anzeige von zwei ADC-Kanälen auf der Siebensegmentanzeige

Programmieren Sie den ADC so, dass die Kanäle AN0 und AN1 abwechselnd angezeigt werden können. Die Umschaltung zwischen den beiden Kanälen soll über einen Taster ausgelöst werden, der am Port C angeschlossen ist (Flachbandkabel)[8]. Beispiel: Wenn Taster SW1 gedrückt ist (eine 1 an Pin RC0), soll Kanal AN1 gemessen und angezeigt werden. Ist dieser Taster nicht gedrückt, soll AN0 gemessen und angezeigt werden.

Verbinden Sie Kanal AN1 (das ist der Pin von RA1) über das rot-gelbe zweiadrige Kabel mit dem Pin gelben Bananenbuchse. Schließen Sie dort eine zweite zu messende Spannung an, z.B. eine Spannung aus dem Funktionsgenerator, die sich sehr langsam (Periode von mind. 10 Sekunden) ändert.

[8]Achtung! Drücken Sie dann niemals auf Taster SW4, das würde einen Kurzschluß verursachen!

Testen Sie grob, ob Ihr Programm funktioniert.

Hinweis: Man muß nach *jedem* Neusetzen des ADC-Kanals (set_adc_channel(n)) neu die Konversion starten (read_adc(ADC_START_ONLY)).

5.4. Anzeige des ADC-Wertes auf einer Balkenanzeige

Ändern Sie ihr Programm so um, dass die LEDs nicht mehr das Bitmuster der 8-Bit Werte (von Kanal AN0) anzeigen sondern eine Balkenanzeige bilden. Also bei höheren Werten sollen mehr benachbarte LEDs leuchten, ausgehend von der linken/rechten LED.

Die Anzeige soll linear sein, d.h. beim Maximalwert leuchten alle 8 LEDs, bei der halben Spannung 4 LEDs, bei einem Viertel der Maximalspannung leuchten 2 LEDs.

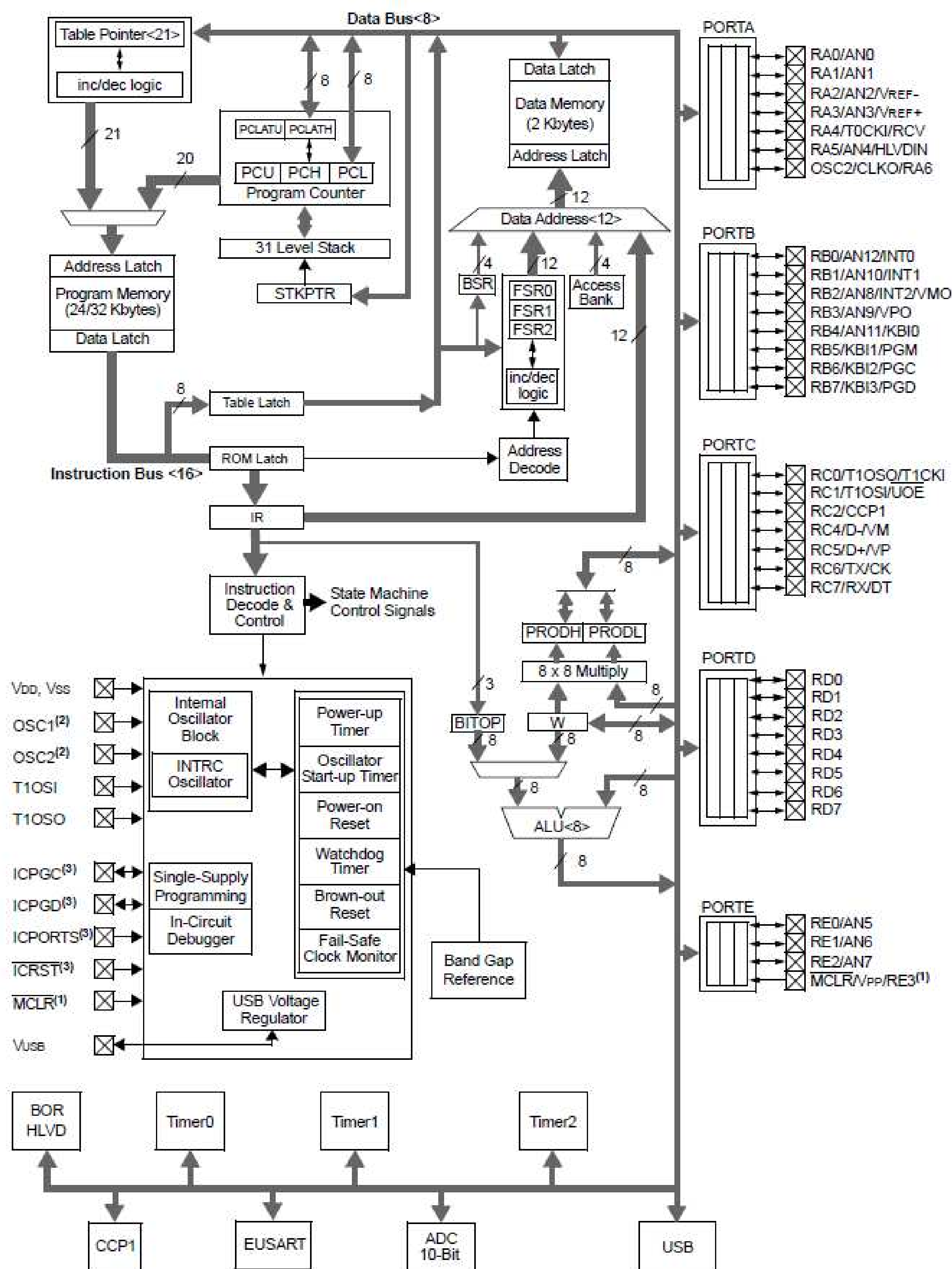
Tip: Falls Ihnen die case-Anweisung nicht vertraut ist, überlegen Sie, ob es mit if-Befehlen geht.

5.5. Für Fortgeschrittene: Anzeige des ADC-Wertes als zweistellige Dezimalzahl

Ändern Sie ihr Programm so um, dass nicht mehr die Hexadezimalwerte von AN0 von den Siebensegmentanzeigen dargestellt werden. Stattdessen sollen die Werte auf eine Skala von 0-99 linear und als Dezimalzahl abgebildet werden und diese dann auf den Siebensegmentanzeigen dargestellt werden.

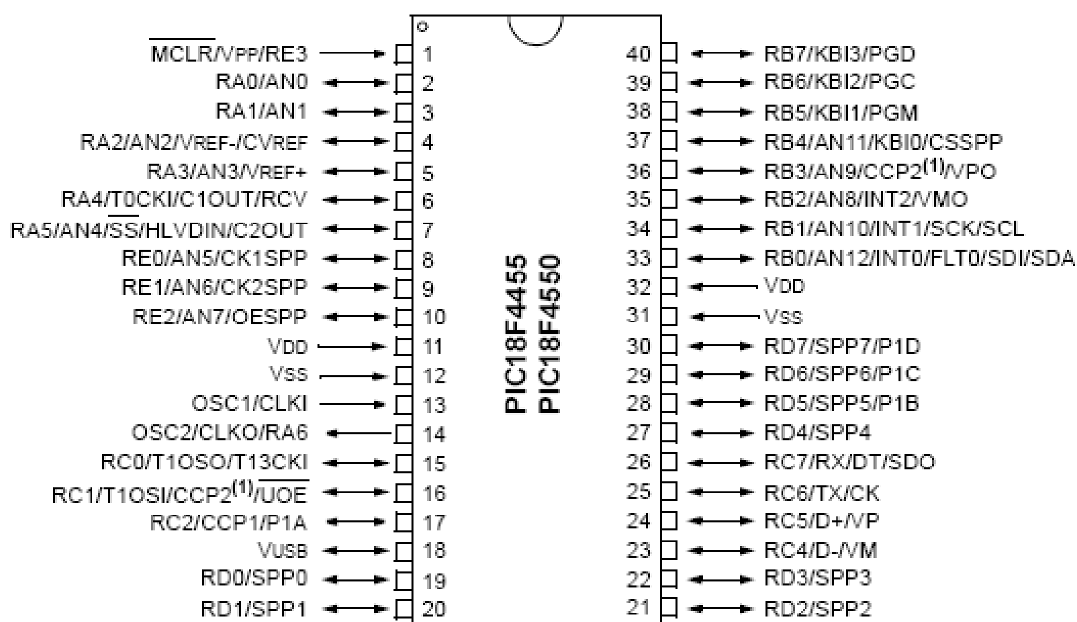
VI. Anhang

1. Blockschaltbild des PIC18F4450 (Nur für Interessierte!)



Ausführliche Datenblätter findet man auf den Internetseiten des Herstellers:
`www.microchip.com` (für den PIC)

2. Pinbelegung des PIC18F4450 (Nur für Interessierte!)



3. Mikrocontroller-Hardware — Peripherie des PIC18F4455

3.1. Ansteuerung der IO-Ports

Unter einem Port versteht man bei Mikrocontrollern Anschlüsse (IC-Beinchen), welche entweder eine Spannung messen können oder sich durch die Programmierung (Firmware) auf bestimmte Spannungen legen lassen. Dabei besteht ein Port meistens aus 8 Anschlüssen, welche im Folgenden als Pins bezeichnet werden. Ports können entweder als Einheit angesprochen werden, dabei behandelt man sie als Variable des Byte-Typs (8 Bit → [0...255]), oder über die einzelnen Pins. Diese werden als Variable vom Typ Boolean (Bit) behandelt.

Der PIC18F4455 verfügt über 5 IO-Ports welche aus bis zu 8 Pins bestehen. 4 dieser IO-Ports sind auf dem Eval-Board als zweireihige Stiftleisten herausgeführt (siehe Abbildung). Zwei dieser Ports sind 8 Bit breit, die anderen beiden sind nicht voll belegt und haben daher weniger Pins. Bei PIC-Mikrocontroller heißen diese Port A bis D.

Jeder Port bzw. jeder Pin hat in einem Mikrocontroller zwei Register (Einstellungen) welche unabhängig voneinander gesetzt und ausgelesen werden können. Dabei handelt es sich zum einen um das Latchregister, welches den Wert des Ports/Pins speichert, und zum anderen das Tri-State-Register, welches die Richtung bestimmt (Input/Output).

Wenn das Tri-State-Register bei PIC-Prozessoren eine 0 enthält, handelt es sich bei dem betreffenden Pin um einen Ausgang, enthält er eine 1, handelt es sich um einen Eingang. Als Eingang haben die Pins am Mikrocontroller einen hohen Innenwiderstand, als Ausgang einen geringen (ähnlich bei Spannungsquelle und Spannungsmessgerät). Das Latch-Register arbeitet in sehr naheliegender Weise: Enthält es eine 1, liegt am Pin eine (positive) Spannung; enthält es eine 0, liegt der Pin auf Masse.

Wird der Pin als Eingang betrieben, also Tri-State-Bit = 1, hat eine Änderung des Latch-Bits keine Wirkung. Liest man das Latch-Bit des zugehörigen Pins aus, so enthält man immer einen Wert entsprechend der am Pin anliegenden Spannung. Bei einem Tri-State-Bit = 0 (Output) liegt am zugehörigen Pin eine dem Latch-Bit entsprechende Spannung (High-Pegel, z.B. 4 V).

Diese Unterscheidung ist sehr wichtig, denn durch falsche Werte in den Tri-State-Registern kann man den Mikrocontroller zerstören! Will man einen IO-Pin beispielsweise als Eingang betreiben, legt also eine Spannung an den Pin (z.B. 0 V), setzt aber das Tri-State-Bit auf 0, kommt es zu einem Kurzschluss, wenn das Latch-Bit den falschen Wert (hier 1) enthält. Dies erklärt sich dadurch, dass der Mikrocontroller mit den Transistoren der Ausgangsstufe jenen Pin auf den Highpegel zieht (z.B. 5 V). Der Benutzer hat diesen Pin allerdings mit der Masse (GND/Logisch 0) verbunden. Der Ausgang ist also kurzgeschlossen und es können Ströme fließen, die die Transistoren zerstören (max. erlaubt pro Pin: 25 mA).

In der Firmware erreicht man die eben beschriebenen Register folgendermaßen: (x = 0 . . . 7)

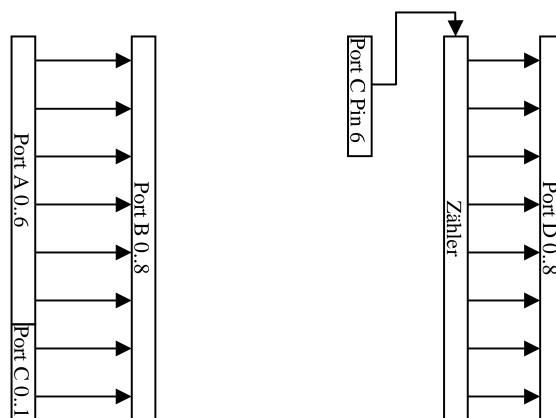
Register	Port A	Port B	Port C	Port D
Tris-Register (Byte)	TRISA	TRISB	TRISC	TRISD
Tris-Register (Bit)	TRISAbits.TRISAx	TRISBbits.TRISBx	TRISCbits.TRISCx	TRISDbits.TRISDx
Latch-Register (Byte) Ausgang	LATA	LATB	LATC	LATD
Latch-Register (Bit) Ausgang	LATAbits.LATAx	LATBbits.LATBx	LATCbits.LATCx	LATDbits.LATDx
Latch-Register (Byte) Eingang	PORTA	PORTB	PORTC	PORTD
Latch-Register (Bit) Eingang	PORTAbits.RAx	PORTBbits.RBx	PORTCbits.RCx	PORTBDbits.RDx

Wie bereits erwähnt, sind nicht alle Ports vollständig belegt. Folgende Pins können adressiert werden:

Port A:	Bit 0 . . . 5
Port B:	Bit 0 . . . 7
Port C:	Bit 0, 1, 2, 6, 7
Port D:	Bit 0 . . . 7

Zusätzlich verfügt der B-Port über interne Pull-Up-Widerstände, welche über das Register `INTCON2bits.RBPU` hinzugeschaltet werden können. Enthält das Register eine 0, sind die Pull-Ups aktiv.

Beispiel: Möchte man z.B. den Port A und die ersten beiden Bits des Port C auf Port B invertiert abbilden, außerdem gleichzeitig einen Zähler an Port D betreiben, welcher die Änderungen des Pin RC6 (Port C, Pin 6) zählt und an Pin RC7 einen Übertrag des Zählers ausgeben, wäre folgende Realisierung möglich:



Schematische Darstellung des Beispiels

```
void Port_Beispiel(){
     static byte RC6_alt=0; // Variable zum Speichern des
                            // Zählerpinzustands
     TRISA = 0b00111111;    // alle 6 Pins von Port A sind Eingänge
     TRISCbits.TRISC0 = 1; // RC0 als Eingang
     TRISCbits.TRISC1 = 1; // RC1 als Eingang
     TRISCbits.TRISC6 = 1; // RC6 als Eingang (Zähler)
     TRISCbits.TRISC7 = 0; // RC7 als Ausgang (Übertrag)
     TRISB = 0;            // Port B als Ausgang
     TRISD = 0;            // Port D als Ausgang Port A
     LATBbits.LATB0=!PORTAbits.RA0; // Setzten der
     LATBbits.LATB1=!PORTAbits.RA1; // Pins an Port B
     LATBbits.LATB2=!PORTAbits.RA2; //
     LATBbits.LATB3=!PORTAbits.RA3; //
     LATBbits.LATB4=!PORTAbits.RA4; //
     LATBbits.LATB5=!PORTAbits.RA5; //
     LATBbits.LATB6=!PORTCbits.RC0; //
     LATBbits.LATB7=!PORTCbits.RC1; //

     if(PORTCbits.RC6!=RC6_alt){ // Abfrage der
                                 // Pinänderung
          LATD++;                // Inkrementieren des
                                 // Zählers
          if(LATD==0)            // Prüfen auf Überlauf
               LATCbits.LATC7=1; // Überlauf-Pin setzen
          else                   // oder
               LATCbits.LATC7=0; // zurücksetzen
     }                           //

     RC6_alt= PORTCbits.RC6;     // Pinzustand speichern
}
```

Diese Routine könnte in `ProcessIO()` aufgerufen werden und würde so mit jedem Durchlauf der Firmware einmal abgearbeitet. In der Praxis ist diese Art der Realisierung aber nicht zu empfehlen, da der Zähler aufgrund von Kontaktprellen (unsauberes Schließen/Öffnen des Schalters) nicht vernünftig zu betreiben wäre. Außerdem könnte man, um einen übersichtlicheren Code zu produzieren, die Tri-State-Register in `UserInit()` setzen, wenn sich diese nicht in der Laufzeit ändern sollen.

3.2. Der AD-Wandler

Der PIC18F4455 verfügt über einen eingebauten 10-Bit-AD-Wandler, welcher Spannungen an bis zu 13 Pins in Zahlen wandeln kann. Dabei wird über einen Multiplexer jeweils der gewünschte Pin mit dem ADC (Analog-Digital-Converter) verbunden. Diese 13 Pins teilt sich der ADC mit anderen Hardwarefunktionen des Mikrocontrollers, beispielsweise den Byte-Ports. Es muss also vor Betrieb festgelegt werden, ob man einen Pin als ADC verwenden möchte. Dies ist standardmäßig nicht der Fall. Außerdem ist die Wahl der Pins, welche als ADC verwendet werden sollen, nicht völlig beliebig. Es können immer nur aufeinander folgende AD-Eingänge (Kanäle) benutzt werden, beginnend bei dem ersten (AN0). Also z.B. nur AN0, AN1 und AN2, nicht aber AN0, AN1 und AN4, wenn die Pins der AN2 und AN3 anderweitig eingesetzt werden sollen. Eine Zuordnung, welche Pins, welche Aufgaben besitzen, finden Sie in der Pinbelegung in der Abbildung auf Seite 18.

Den Wertebereich entnimmt sich der ADC aus der Versorgungsspannung des Mikrocontrollers. Liegt 0 V (GND) am ADC an, entspricht dies einer 0; liegt Vdd (ca. 4,3 V) an, einer 1023. Spannungen außerhalb dieses Bereichs liefern den näher gelegenen Grenzwert, sollten aber vermieden werden, da dies den Mikrocontroller zerstören kann[9].

Um Ihnen das manuelle Setzten der 3 Kontrollregister des ADCs zu ersparen, befinden sich in den beim Erstellen des Projekts eingebundenen Dateien bereits vorgefertigte Routinen zur Steuerung des ADC. Diese Routinen lassen allerdings nur eine maximale Anzahl von fünf ADC-Eingängen zu. Lediglich das TRIS-Register des jeweiligen Pins müssen Sie noch manuell auf 1 setzen, also den Pin zu einem Eingang machen.

Um den ADC zu nutzen, sind folgende Schritte notwendig:

- Einstellen des ADC: `setup_adc(mode)` Der ADC kann in verschiedenen Konversionsraten arbeiten, welche sich aus dem Prozessortakt ableiten. Die Konversionsrate legt man fest, indem man der `setup_adc(mode)`-Routine für `mode` eines der folgenden Schlüsselwörter übergibt:

  ADC_CLOCK_DIV_64
  ADC_CLOCK_DIV_32
  ADC_CLOCK_DIV_16
  ADC_CLOCK_DIV_8
  ADC_CLOCK_DIV_4
  ADC_CLOCK_DIV_2
  ADC_OFF

  Mit dem ADC_OFF Schlüsselwort lässt sich die ADC-Hardware im PIC abschalten.

  Hinweis: Eine Konversion mit 10 Bit Genauigkeit braucht elf Konversionstakte. Mit einem Teilungsfaktor des CPU-Takts von 8 (d.h. ADC_CLOCK_DIV_8) wären das also 88 CPU-Takte. Der PIC18f4455 ist intern mit 12MHz getaktet, ein Taktzyklus dauert also 83ns. Damit würde sich eine Konversionszeit von 7,3 μs ergeben.

- Festlegen der ADC-Ports: `setup_adc_ports(nports)` Sie müssen der Firmware angeben, welche Ports mit dem ADC verbunden werden sollen. Dies erledigt die `setup_adc_ports(nports)`-Routine. `nports` bestimmt dabei die Anzahl der als ADC genutzten Pins. Es können folgende Schlüsselwörter übergeben werden:

  NO_ANALOGS
  AN0
  AN0_TO_AN1
  AN0_TO_AN2
  AN0_TO_AN3
  AN0_TO_AN4

- Lesen eines ADC-Ports:

  Um einen bestimmten Kanal (Pin) mit dem ADC auszulesen, sind mehrere Schritte nötig. Zuerst muss mit `set_adc_channel(n)` der auszulesende Kanal festgelegt werden. *n* entspricht dabei der Nummer des Kanals, also ANn. Dann muss die Konversion mit `read_adc(ADC_START_ONLY)` gestartet werden. Das eigentliche Auslesen erfolgt daraufhin mit der Funktion `read_adc(ADC_READ_ONLY)`. Es wird ein Wert

[9] Erlaubt sind minimal -0,5 V und maximal Vdd + 0,5 V. Bei höheren Spannungen fließen höhere Ströme, wie bei einer (Zener)diode. Durch die 10-kΩ-Widerstände an den Bananenbuchsen (gelb, grün) werden diese Ströme hier aber auf für den Mikrocontroller ungefährliche Werte begrenzt und es können auch bei versehentlichem Überschreiten des zulässigen Spannungsbereichs in der Regel keine Beschädigungen auftreten.

vom Typ `unsigned int` (16Bit) zurückgegeben. Da der ADC nur 10 Bit Auflösung besitzt, sind die 6 MSB = 0.

Zwischen Start der Konversion und Auslese sollte logischerweise genug Zeit vergehen (siehe Einstellen des ADC). Wird vor Ende der Konversion gelesen, wird der Wert 0 zurückgegeben.

Bei kontinuierlichem Auslesen des ADC ist es möglich, Auslese und Neustart der Konversion mit `read_adc(ADC_READ)` gleichzeitig zu erledigen. Die Wahl des Kanals muss dabei vor dem (Neu-)Start der Konversion erfolgen.

Beispiel: Es sollen abwechselnd die ADC-Kanäle AN0 und AN1 im Abstand von 10 ms gelesen werden.

```
void adc_example(){
      byte n=0;unsigned int value;      //Hilfsvariablen
           TRISAbits.TRISA0=1;          //ADC-Kanäle als Eingänge
           TRISAbits.TRISA1=1;          //schalten
           setup_adc_ports(AN0_TO_AN1); //Kanal AN0 und AN1 wählen
           setup_adc(ADC_CLOCK_DIV_32); //Konversionszeit ca. 30us
           set_adc_channel(n);          //Kanal auf AN0 setzten
           read_adc(ADC_START_ONLY);    //Konversion starten
           while (1){                   //Endlosschleife
                     delay_ms(10);               //Verzögerung von 10 ms
                     set_adc_channel((n++)&0x1); //Kanal umschalten
                     value=read_adc(ADC_READ)>>2;//8-Bit-Wert auslesen
                     USBTasks();                 //USB-Komm. erhalten
                                                 //falls ADC-Werte über
                                                 //USB gesendet werden sollen
                     }
                }
```

3.3. USB-Kommunikation (NUR FÜR INTERESSIERTE!)

Die Firmware zu Steuerung der PIC-internen USB-Schnittstelle beherrscht 2 Möglichkeiten der Datenübertragung an den PC: Gepuffert und ungepuffert. Dabei unterscheiden sich die beiden Übertragungsmöglichkeiten hauptsächlich dadurch, dass bei gepuffertem Senden die Daten erst in einen Zwischenspeicher geschrieben werden und dann über einen Befehl manuell als ein Paket gesendet werden. Diese Pakete dürfen eine Größe von 128 Zeichen/Bytes nicht überschreiten!

In der Anwendung bedeutet das, dass nach den Routinen zur gepufferten Übertragung ein `flush_out()` aufgerufen werden muss, um die Daten abzuschicken.

Manche Routinen erwarten als Argument nicht die Daten selbst, sondern einen Pointer. Wenn Sie jedoch nur mit Variablen arbeiten, welche die Daten direkt enthalten, setzen Sie beim Funktionsaufruf den Adressoperator & vor ein Argument:

`n=48; usb_cdc_send(&n,1);` sendet den Wert von n, also 48 bzw. als ASCII-Zeichen eine Null.

Anmerkung: Die Lese-Routinen nehmen sich das erste Zeichen im Empfangsbuffer und und löschen dieses dort. Ist der Empfangsbuffer leer, warten die Routinen, bis Daten eintreffen. Da dies mitunter sehr lange dauern kann, empfiehlt es sich, vor Aufruf zu prüfen, ob Daten im Empfangspuffer vorliegen.

Eine Übersicht über die Befehle zum Senden und Empfangen finden Sie im Anhang bei der Liste der wichtigsten Befehle.

Beim Start der User-Firmware wird immer die `main(void)`-Routine in der main.c-Datei aufgerufen[10].

---

[10]Die `main()`- Routine initialisiert das USB-Protokoll, steuert die USB-Kommunikation währen der Laufzeit (`USBTasks()`) und ruft den Usercode auf, welcher sich in der `ProcessIO()`-Routine in der user.c-Datei befindet.

Die USB-Kommunikation wird über ein Polling-System geregelt, d.h. es muss regelmäßig eine Routine zur Abarbeitung der anfallenden Anfragen in der USB-Schnittstelle aufgerufen werden. Diese Anfragen können in ihrer Firmware entstehen, wenn Sie Zeichen zum Senden an den PC in einen Puffer schreiben, oder vom USB-Controller im PC ausgehen, welcher die Kommunikation aufrecht erhält bzw. Steuerzeichen aus dem Terminal an den μC sendet. In der Regel erledigt das die Firmware, indem sie nach Abarbeitung der User-Firmware in die Endlosschleife der `main()`-Routine zurückkehrt und `USBTasks()` aufruft. Dies sollte jede Millisekunde geschehen, wenn Daten übertragen werden. Wenn Sie also eine eigene Schleife in ihre Firmware einbauen, welche lange Iterationszeiten besitzt, oder Ihr Code generell sehr viel Zeit beansprucht, sollten Sie die `USBTasks()`-Routine manuell aufrufen. Geschieht dies nicht, wird entweder der Puffer in Ihrem μC überlaufen oder es kann vorkommen, dass Windows das USB-Gerät aufgrund von Inaktivität abmeldet.

4. Liste der wichtigsten Befehle

Befehl	Funktion	Beispiel/Anmerkung
delay_ms(byte)	Verzögert die Ausführung der Firmware um byte Millisekunden	delay_ms(250); wartet 250 ms
delay_us(byte)	Verzögert die Ausführung der Firmware um byte Mikrosekunden	delay_us(123); wartet 123 us
read_adc(mode)	Liest und/oder startet den ADC. Gibt einen int-Wert zurück (16Bit).	siehe Kapitel 3.2. Der AD-Wandler
setup_adc_ports(mode)	Bestimmt die Zahl der benutzten ADCs	siehe Kapitel 3.2. Der AD-Wandler
setup_adc(mode)	Konfiguriert den ADC-Takt	siehe Kapitel 3.2. Der AD-Wandler
set_adc_channel(byte)	Bestimmt den ADC-Kanal, der ausgelesen werden soll	siehe Kapitel 3.2. Der AD-Wandler
while(Bedingung) {Anweisungen}	While-Schleife, die ihre Anweisungen solange wiederholt, wie Bedingung erfüllt ist	while(RA0==1){ n++; } Inkrementiere n, solange Pin A0 logisch 1 ist.
for (Initialisierung; Bedingung; Schrittweite) {Anweisungen;}	Allgemeine for-Schleife	for(i=0;i<10;i++){ usb_cdc_putc(i); }
dec_7seg[0-15]	Array mit Bitmustern für Sieben-Segmentanzeigen	PORTB=dec_7seg[5]; stellt eine 5 dar
aschex(char)	Gibt den Wert eines Hex-Zeichens von 0-f/F zurück	byte=Aschex('B');
hexasc(byte)	Gibt den Wert von Byte (0- 15) als Hex-Zeichen zurück	char=hexasc(11);
usb_cdc_kbhit()	Prüft, ob Daten im Empfangspuffer stehen	if(usb_cdc_kbhit()== 1) usb_cdc_getc();
clr_input()	Leert den Empfangspuffer	-
usb_cdc_getc()	Gibt das erste Zeichen im Empfangsbuffer zurück	char=usb_cdc_getc(); Wartet bis min. ein Zeichen im Buffer liegt, siehe usb_cdc_kbhit()
usb_cdc_send(*char, byte)	Sendet die Zeichen des char- Arrays über die USB-Schnittstelle. Byte gibt die Länge des Arrays bzw. die Anzahl der zu übertragenden Zeichen an. Ungepuffert.	char='@'; usb_cdc_send(&char,1); siehe auch usb_cdc_send_int(), usb_cdc_putc()

usb_cdc_send_int(*int, byte)	Sendet die Werte des int-Arrays. Byte gibt die Länge des Arrays bzw. die Anzahl der zu übertragenden Zeichen an. Gepuffert.	int buf[5];
		usb_cdc_send_int(&buf,5);
		flush_out(); siehe auch usb_cdc_send(), flush_out();

usb_cdc_putc(char)	Sendet ein Zeichen char über USB. Ungepuffert.	usb_cdc_putc('@');
usb_cdc_gethex()	Wandelt die ersten beiden Zeichen im Empfangsbuffer, sofern diese den Hex-Zeichen entsprechen, in einen Byte-Wert um.	byte=usb_cdc_gethex( ); Wartet bis min. zwei Zeichen im Buffer liegen, siehe usb_cdc_kbhit()

usb_cdc_putc_hex(char)	Sendet den Hex-Wert des ASCII-Zeichens char in Form von zwei Zeichen über USB. Gepuffert.	usb_cdc_putc_hex('0');
		oder usb_cdc_putc_hex(48);
		siehe auch flush_out()

usb_cdc_putc_buf(char)	Sendet ein Zeichen char über USB. Gepuffert.	usb_cdc_putc('@'); flush_out();
usb_cdc_put_unsigned (int,byte)	Sendet den Wert aus int als Dezimalzahl mit byte Stellen.	usb_cdc_put_unsigned (1234,5); sendet 01234
flush_out()	Sendet alle im Puffer befindlichen Daten an den PC	siehe alle Befehle zur gepufferten Übertragung

5. Zusammenstellen der benötigten Dateien und Pfade

Falls kein vorgefertigtes Projekt vorhanden ist,wählen Sie im Menü „Project“ den ersten Punkt „Project Wizard“. Folgen Sie den Anweisungen, dazu wählen Sie:

1. Device: PIC18F4455
2. Toolsuite:
   a) Active Toolsuite: Microchip C18 Toolsuite
   b) Toolsuite Contents: MPLAB C18 Compiler (mcc18.exe)
   c) Location: C:\Program Files\MCC18\bin\mcc18.exe
3. Erstellen Sie einen Pfad (auf G:) für Ihr Projekt und geben Sie einen Namen an
4. Fügen Sie folgende Dateien in Ihr Projekt ein (in der linke Liste anklicken, dann auf Add) . Wählen Sie den Buchstaben vor der eingefügten Datei wie angegeben (ändern durch Anklicken). Diese Buchstaben bestimmen, ob die Dateien als Systemdateien, oder als Kopie eingebunden werden. **C** steht dabei für die Kopie, d.h. es wird eine Kopie der importierten Datei im Projektverzeichnis abgelegt, welche frei benannt werden kann. Dateien mit **S** (System) oder **A** (Auto, wird hier als Systemdatei interpretiert) werden als Systemdateien eingebunden, d.h. es wird die Originaldatei bearbeitet. Die Pfadangaben für kopierte bzw. Benutzerdateien (**U**) sind dabei relativ, Systemdateipfade absolut.

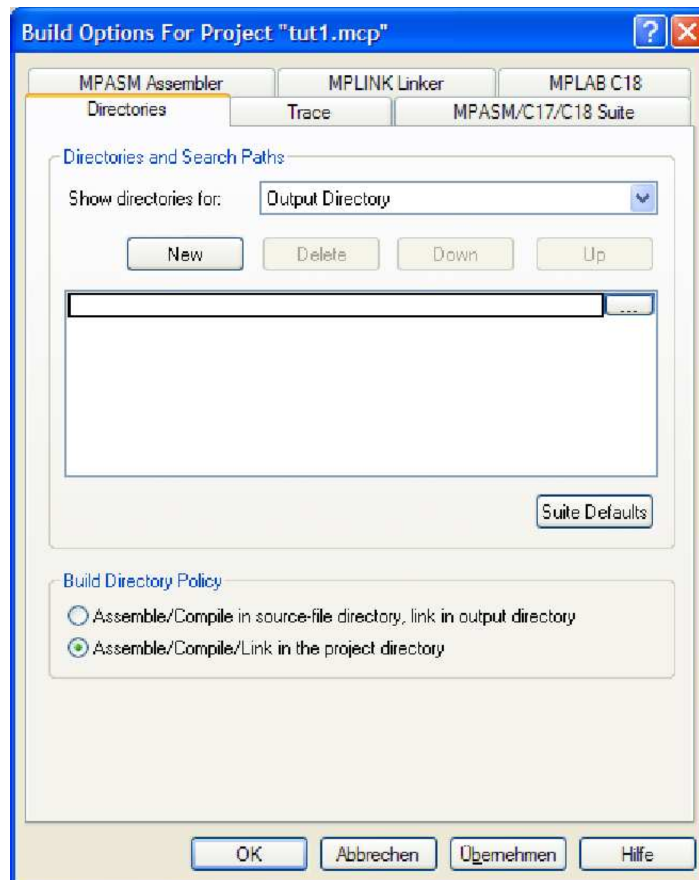
   Alle zu importierenden Dateien befinden sich im Ordner „C:\EP-PIC Dateien“, es werden nur relative Pfadangeben gemacht:

   a) **C** main.c
   b) **C** io_cfg.h
   c) **S** 18f4455.lkr
   d) **C** user\tools.c
   e) **C** user\tools.h
   f) **C** user\user.c
   g) **C** user\user.h
   h) **A** autofiles*alle Dateien*
   i) **A** system\usb\class\cdc*alle Dateien*
   j) **A** system\usb\usb9*alle Dateien*
   k) **A** system\usb\usbctrltrf*alle Dateien*
   l) **A** system\usb\usbdrv\usbdrv.c
   m) **A** system\usb\usbdrv\usbdrv.h
   n) **A** system\usb\usb.h
   o) **A** system\usb\usbmmap.h
   p) **A** system\usb\usbmmap.c
   q) **A** system\typedefs.h

   Wenn alle Dateien zusammengestellt sind, klicken Sie auf Weiter.
5. Schließen Sie den Assistenten mittels der Schaltfläche „Fertig stellen“.

Sie sollten nun das Projekt vor sich sehen, mit allen eingefügten Dateien. Sollte dies nicht der Fall sein, wählen Sie im Menü „View“ den Punkt „Project“.

Als nächstes legen Sie die Umgebungspfade des Projekts fest. Dazu öffnen Sie im „Project“Menü unter dem Punkt „Build-Options. . . “ den untersten (und wahrscheinlich noch einzigen) Punkt „Project“(Project → Build Opions → Project). Folgendes Fenster öffnet sich:



Umgebungspfade

Dort wählen Sie den Reiter „Directories“ und legen für die Verzeichnisse in der Drop-Down-Box verschiedene Pfade fest, die Sie wie folgt eingeben.

Wählen Sie neben „Show directories for:“ einen der fünf aufgezählten Punkte.

Im großen Feld darunter werden nun die für diesen Eintrag gespeicherten Pfade angezeigt. Wahrscheinlich ist ihn Ihrem neu erstellten Projekt noch kein Pfad angegeben. Um einen Pfad hinzuzufügen, benutzen Sie die Schaltfläche [New]. (Unter Umständen gibt es für manche oben genannten fünf Punkte mehrere Einträge, dann wiederholen Sie das.) Im unteren Fenster erscheint nun eine leere Zeile. In diese Zeile kann nun entweder von Hand das Verzeichnis eingetippt, oder aus einem Verzeichnisbaum ausgewählt werden (Schaltfläche am rechten Ende der neuen Zeile [. . .]). Suchen Sie im Fenster„Ordner suchen“ den richtigen Ordner, klicken Sie einmal auf den ordner, dann auf OK. Möglicherweise wird Ihnen vom Programm bereits der jeweils richtige Pfad im Verzeichnisbaum vorgeschlagen.

Folgende Pfade sind anzulegen:

- Output Directory: *Projektverzeichnis*\output
  - Verzeichnis muss erst angelegt werden
- Intermediary Directory: *Projektverzeichnis*\output
- Include Search Path: C:\EP-PIC Dateien und C:\Programme\MCC18\h
- Library Search Path: C:\Programme\MCC18\lib
- Linker-Script Search Path: C:\Programme\MCC18\lkr

6. Bei Problemen mit dem USB-Treiber ...

... muß man (ggf. mit Administratorrechten) den Treiber aktualisieren:

System, Geräte-Manager, Microchip custom USB-Device

Dort rechtsklicken: Treiber aktualisieren, (kein Windows), Software aus Liste, selbst wählen, Microchip USB ...