Embedded Systems 1 - Dokumentation

Kevin Fritz

15. Juni 2017, Aalen

Inhaltsverzeichnis

Einleitung

Einleitung muss noch verfasst werden.

1 Getting Started

1.1 Aufgabenstellung

1. Nehmen Sie das Programm „HelloWorld2“ in Betrieb.
2. Entfernen Sie die Verzögerungsfunktion und messen Sie die Frequenz, mit der die LED angesteuert wird. Überprüfen Sie das Ergebnis durch Analyse des generierten Assembler-Codes. Wie groß ist die Rechenleistung in MIPS?
3. Erhöhen Sie die CPU-Frequenz auf den maximal möglichen Wert. Weisen Sie durch eine Messung nach, dass die CPU-Frequenz tatsächlich erhöht wurde. Wie groß ist die Rechenleistung in MIPS?

1.2 Lösung

1. Programm „HelloWorld2“ in Betrieb nehmen. Der Programmcode ist in Listing 1 zu sehen.
2. Die Verzögerungsfunktion wurde auskommentiert. Der Oszi-Aufnahme aus Abbildung 1 kann entnommen werden, dass die Zeitdauer um einen Port ein- bzw. auszuschalten jeweils ungefähr $5,75us$ beträgt. Daraus ergibt sich eine Frequenz von $f = \frac{1}{T} = \frac{1}{5,75us} \approx 174kHz$. Aus Listing 2 geht hervor das zum Toggeln der LED 16 Assemblerbefehle benötigt werden.
$$MIPS = \frac{16}{5,75us} * 10^{-6} \approx 2,783MIPS \quad (1)$$
Die Einheit MIPS gibt an, wie viele Maschinenbefehle (Instruktionen) ein Mikroprozessor pro Sekunde ausführen kann. 1 MIPS bedeutet, er kann eine Million Maschinenbefehle pro Sekunde ausführen. MIPS ist eine ungenaue Einheit, da verschiedene Assemblerbefehle verschieden viel Zeit benötigen.
3. Um die CPU-Frequenz auf den Maximalen Wert zu erhöhen wird die auf dem Board verbaute PLL verwendet. Die Parameter zur Konfiguration der PLL sind dem Datenblatt (Abbildung 2) zu entnehmen. Der Wertebereich der PLL Parameter kann Abbildung 3 entnommen werden. Die Parameter wurden (mit einem Excel Sheet, Abbildung 4) so ausgelegt, dass sich eine Taktfrequenz F_{OSC} von $140MHz$ ergibt. Aus dem Oszillator Modul (online zu finden bei mikrochip) kann eine Code-Sequenz entnommen werden wie die jeweiligen PLL-Parameter zu setzen sind. Der Ausschnitt aus dem Datenblatt wurde für unsere Zwecke angepasst (Listing 3).
Nach Konfigurieren der PLL wurde wieder die Zeitdauer zum toggeln der LED gemessen ($302ns$ für 16 Assembler-Befehle), hieraus ergibt sich eine Rechenleistung von:
$$MIPS = \frac{16}{302ns} * 10^{-6} \approx 52,980MIPS \quad (2)$$
Setzt man die ausgerechneten MIPS ins Verhältnis, kommt man zu dem Entschluss das die gemessenen Werte plausibel sind, da: $2,783 * \frac{140}{7,37} \approx 52,867$.

```
// Check for Project Settings
#ifndef __dsPIC33EP512MU810__
#error "Wrong Controller"
#endif
#include <xc.h>    //Include appropriate controller specific
    headers
#include <stdint.h>//Standard typedefs
// Oscillator Configuration
_FOSCSEL(FNOSC_FRC); //Initial Oscillator:  Internal Fast RC
_FOSC(POSCMD_NONE);  //Primary Oscillator disabled (not used)

/* Substitute for stdlib.h */
#define EXIT_SUCCESS  0
#define EXIT_FAILURE  1

/* Hardware  */
#define _LED200 LATBbits.LATB8

void delay_ms(uint16_t u16milliseconds){
  uint16_t ui16_i=0;
  while(u16milliseconds){
    for (ui16_i=0;ui16_i<331;ui16_i++){//1 ms delay
      __asm__ volatile("nop \n\t"
      "nop \n\t"
      "nop \n\t");
    }//for
  u16milliseconds--;
  }//while
}
int main() {
/* Port Configurations */ // DS70616G-page 209
// ODCB (open drain config) unimplemented (DS70616G, Table 4-56)
ANSELBbits.ANSB8=0;      //Digital I/O
CNENBbits.CNIEB8=0;      //Disable change notification interrupt
CNPUBbits.CNPUB8=0;      //Disable weak pullup
CNPDBbits.CNPDB8=0;      //Disable weak pulldown
TRISBbits.TRISB8=0;      //Pin B8: Digital Output
LATBbits.LATB8=0;        //Pin B8: Low
/* Endless Loop */
while(1){
  /* LATBbits.LATB8 = !(LATBbits.LATB8); //Toggle Pin B8 */
  _LED200=!_LED200; //Toggle LED
  delay_ms(500);
}//while
return (EXIT_SUCCESS);  //never reached
} //main()
```

Listing 1: Quellcode HelloWorld2

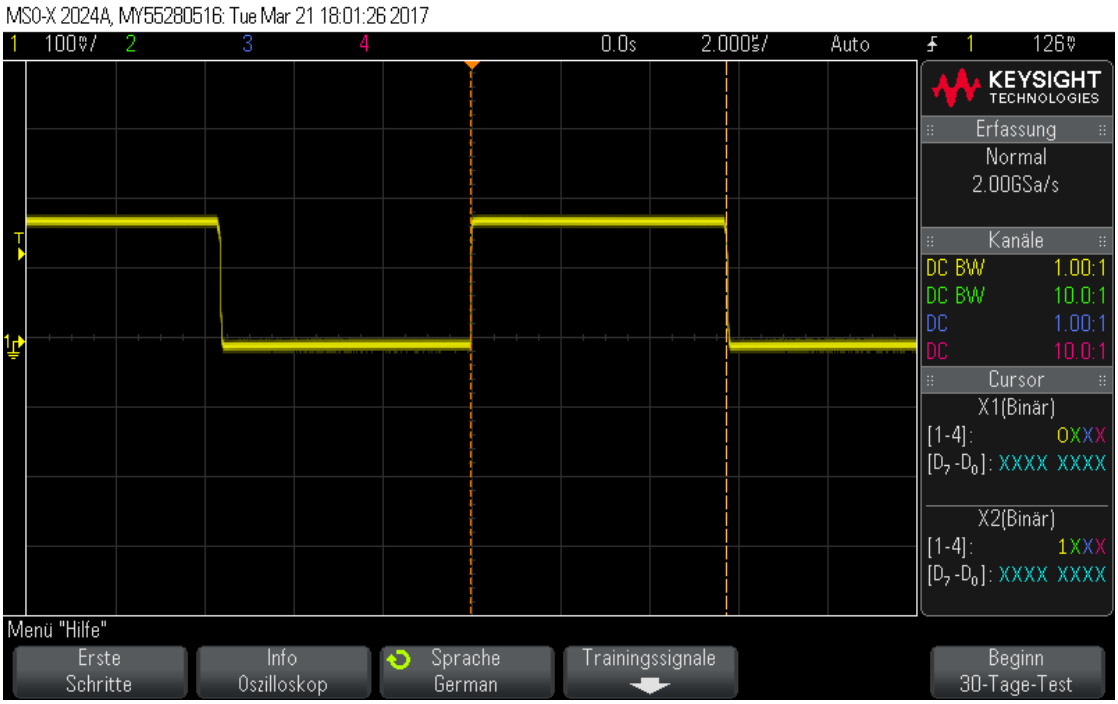


Abbildung 1: Ansteuerungsfrequenz der LED

```
//while(1){
// _LED200=!_LED200; //Toggle LED
00033E  8070A1      MOV LATB, W1
000340  201000      MOV #0x100, W0
000342  608000      AND W1, W0, W0
000344  A7F000      BTSC W0, #15
000346  EA0000      NEG W0, W0
000348  E90000      DEC W0, W0
00034A  DE004F      LSR W0, #15, W0
00034C  784000      MOV.B W0, W0
00034E  FB8000      ZE W0, W0
000350  600061      AND W0, #0x1, W0
000352  DD0048      SL W0, #8, W0
000354  8070A1      MOV LATB, W1
000356  A18001      BCLR W1, #8
000358  700001      IOR W0, W1, W0
00035A  8870A0      MOV W0, LATB
//}//while
00035C  37FFF0      BRA 0x33E
//return (EXIT_SUCCESS);  //never reached
//} //main()
```

Listing 2: Assembler Befehle zum toggeln

```
// Select Internal FRC at POR
_FOSCSEL(FNOSC_PRIPLL); //Initial Oscillator: Primary Oscillator
    (XT, HS, EC) with PLL
_FOSC(POSCMD_HS);  //HS Crystal Oscillator Mode

int main()
{
// Configure PLL prescaler, PLL postscaler, PLL divisor
PLLFBD=455; // PLLDIV
CLKDIVbits.PLLPOST=2;
CLKDIVbits.PLLPRE=2;

// Wait for PLL to lock
while (OSCCONbits.LOCK!= 1);


while(1)
{
//endless loop
}

return 1; //never reached
}
```

Listing 3: Code Example for Using PLL with 7.37 MHz Internal FRC

9.1 CPU Clocking System

The dsPIC33EPXXX(GP/MC/MU)806/810/814 and PIC24EPXXX(GP/GU)810/814 family of devices provides seven system clock options:

- Fast RC (FRC) Oscillator
- FRC Oscillator with Phase-Locked Loop (PLL)
- Primary (XT, HS or EC) Oscillator
- Primary Oscillator with PLL
- Secondary (LP) Oscillator
- Low-Power RC (LPRC) Oscillator
- FRC Oscillator with postscaler

Instruction execution speed or device operating frequency, FCY, is given by Equation 9-1.

EQUATION 9-1: DEVICE OPERATING FREQUENCY

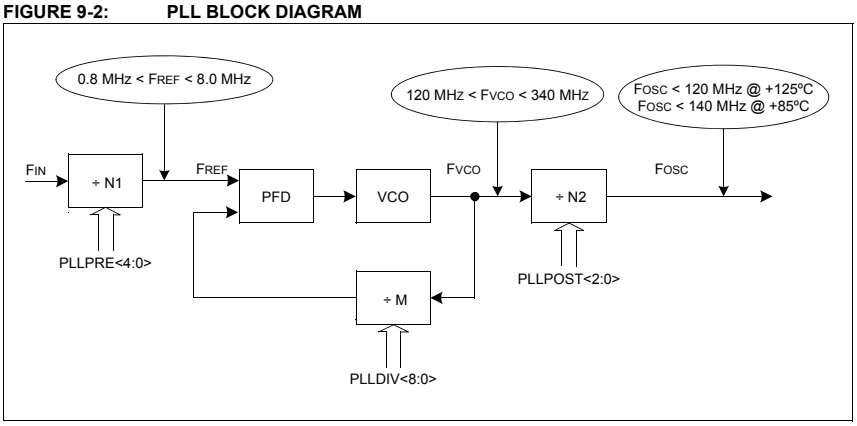
$$FCY = FOSC/2$$

Figure 9-2 is a block diagram of the PLL module.

Equation 9-2 provides the relation between input frequency (FIN) and output frequency (FOSC).

Equation 9-3 provides the relation between input frequency (FIN) and VCO frequency (FVCO).

FIGURE 9-2: PLL BLOCK DIAGRAM



EQUATION 9-2: FOSC CALCULATION

$$FOSC = FIN \times \left(\frac{M}{N1 \times N2}\right) = FIN \times \left(\frac{(PLLDIV + 2)}{(PLLPRE + 2) \times 2(PLLPOST + 1)}\right)$$

Where,

$N1 = PLLPRE + 2$

$N2 = 2 \times (PLLPOST + 1)$

$M = PLLDIV + 2$

EQUATION 9-3: FVCO CALCULATION

$$FVCO = FIN \times \left(\frac{M}{N1}\right) = FIN \times \left(\frac{(PLLDIV + 2)}{(PLLPRE + 2)}\right)$$

Abbildung 2: CPU Clocking System mit PLL Block Diagramm

PLLPOST<1:0>: PLL VCO Output Divider Select bits (also denoted as 'N2', PLL postscaler)
11 = Output divided by 8
10 = Reserved
01 = Output divided by 4 (default)
00 = Output divided by 2

Unimplemented: Read as '0'

PLLPRE<4:0>: PLL Phase Detector Input Divider Select bits (also denoted as 'N1', PLL prescaler)
11111 = Input divided by 33
•
•
•
00001 = Input divided by 3
00000 = Input divided by 2 (default)

PLLDIV<8:0>: PLL Feedback Divisor bits (also denoted as 'M', PLL multiplier)
111111111 = 513
•
•
•
000110000 = 50 (default)
•
•
•
000000010 = 4
000000001 = 3
000000000 = 2

Abbildung 3: Wertebereich der PLL Parameter

B8 =B2*(B4+2)/((B5+2)*2*(B6+1))

	A	B	C	D	E
1			Einheit	Wertebereich	
2	FIN	7,37	[MHz]		
3					
4	PLLDIV	455		2...513	
5	PLLPRE	2		2...33	
6	PLLPOST	2		2;4;8	
7					
8	FOSC	140,337083	[MHz]		

Abbildung 4: PLL Parameter Excel

2 Rechenleistung

2.1 Aufgabenstellung

1. Ermitteln Sie die durchschnittliche Laufzeit einschließlich Streuung arithmetischer Grundoperationen für verschiedene vom XC-16-Compiler unterstützten Datentypen. Wie erklären Sie die Unterschiede?

2. Berechnen Sie die ersten 10 Primzahlen, die größer als 1.000.000 (1E6) sind. Implementieren Sie denselben Algorithmus auf einem PC und vergleichen Sie die Rechenzeiten.

2.2 Lösung

1. Der Programmcode wird so umgeändert wie in Listening 4 zu sehen. Zuerst wird mit dem Oszi nur die Zeit gemessen wie lange eine LED aus ist (ohne Rechenoperation, $29,1ns$). Anschließend kann man mit dem Oszi messen wie lange eine Grundoperation (mit Zufallszahlen) inklusiv LED ein/ausschalten benötigt. Die folgende Auflistung beinhaltet nur die Zeitdauer für die jeweilige Rechenoperation (ohne LED ein/aus).

Datentyp	Operation	Zeitdauer		Datentyp	Operation	Zeitdauer
uint8_t	+	43,7 ns		int8_t	+	57,4 ns
	-	58,1 ns			-	57,4 ns
	*	72,1 ns			*	71,9 ns
	/	72,5 ns			/	343,9 ns
uint16_t	+	43,4 ns		int16_t	+	43,3 ns
	-	43,5 ns			-	43,4 ns
	*	86,2 ns			*	87,4 ns
	/	330,9 ns			/	329,9 ns
uint32_t	+	115,4 ns		int32_t	+	142,9 ns
	-	114,9 ns			-	112,9 ns
	*	245,9 ns			*	230,9 ns
	/	7,559 us			/	7,95 us
uint64_t	+			int64_t	+	254,9 ns
	-				-	226,9 ns
	*				*	1,5 us
	/				/	131 us
float	+			long double	+	
	-				-	
	*				*	
	/				/	

2. Der geforderte Algorithmus ist in Listening 5 abgebildet. Bei dem verfügbaren Computer (Intel Core i7 2.6GHz, 16GB RAM, 64Bit Windows 10) ergab sich eine Laufzeit von ungefähr $5us$. (Gemessen mit CodeBlocks, $50s$ für 10^7 Durchläufe)

   Die Laufzeit des selben Programms (angepasst auf die Hardware) benötigte auf dem Mikrocontroller Board $47,8ms$. Der Code hierzu ist in Listing 6 abgebildet.

   Damit ist der Computer ca 10.000 mal schneller als der μC. ($\frac{47,8ms}{5us} = 9560$, Abbildung 5).

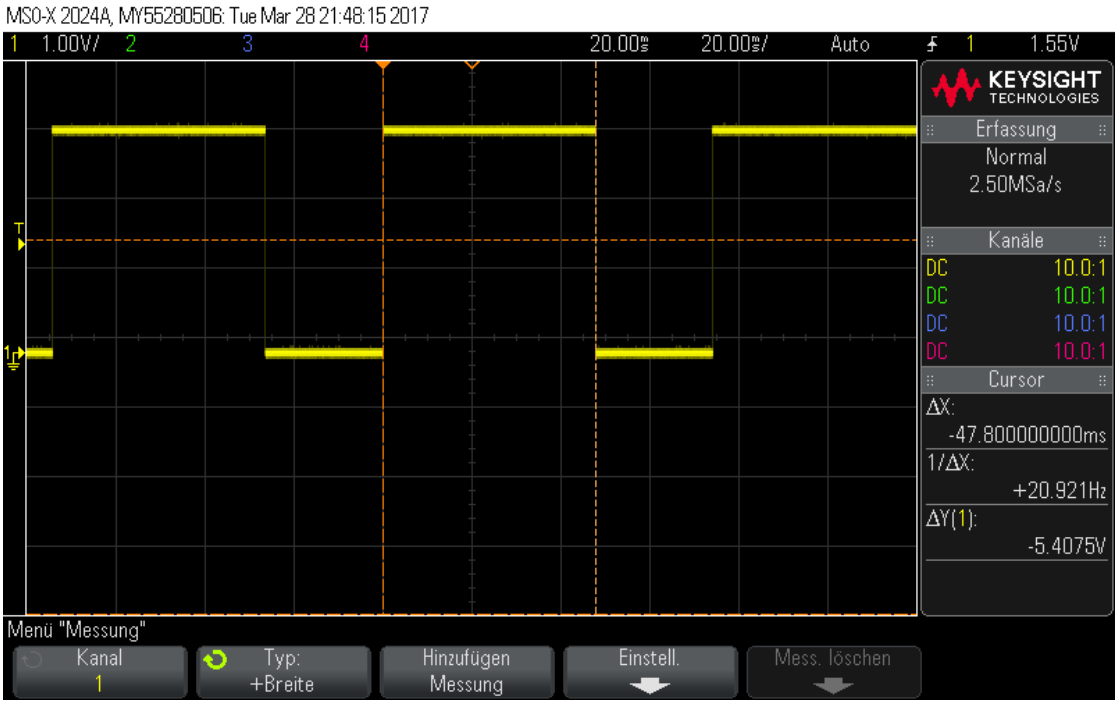


Abbildung 5: Laufzeit der Primzahlenberechnung

```
/* Endless Loop */
while(1){
_LED200=0;
ui8Var1 *= ui8Var2;
_LED200=1;
}//while
```

Listing 4: Bestimmen der Rechenleistung

```
#include<stdint.h>
#include<stdlib.h>
#include<math.h>

uint8_t isPrim(uint32_t ui32Number);

int main()
{
  uint32_t ui32Number= 1e6;  //start value
  uint16_t ui8PrimeCounter=0; //counts the number of calculated
     prime numbers
  const uint16_t ui8PrimMax=100;

  for(; ui8PrimeCounter<ui8PrimMax; ui32Number++)
    if(isPrim(ui32Number)) //check if the number is prime
    {
    //printf("%d\t%d\n",ui8PrimeCounter,ui32Number);
    ui8PrimeCounter++; //increase PrimeCounter, if the number is
       prime
    }
  return 0;
}
uint8_t isPrim(uint32_t ui32Number){
  uint32_t ui32Divider;
  uint32_t ui32SqrtNumber =((uint32_t)
     sqrt((double)(ui32Number)))+1;
  for(ui32Divider=2; ui32Divider<ui32SqrtNumber; ui32Divider++)
  {
    if((ui32Number%ui32Divider) == 0)
    {
      return 0; //uiNumber32 isn't a prime number
    }
  }
  return 1; //ui32Number is a Prime Number
}
```

Listing 5: Algorithmus zur Berechnung der ersten 100 Primzahlen größer als 1E6

```
int main() {    //scope_25 47,8ms

```

```
3   PLLFBD = 418;
4   CLKDIVbits.PLLPOST = 2;
5   CLKDIVbits.PLLPRE = 2;
6
7   /* Port Configurations */
8   // DS70616G-page 209
9   // ODCB (open drain config) unimplemented (DS70616G, Table 4-56)
10  ANSELBbits.ANSB8=0;      //Digital I/O
11  CNENBbits.CNIEB8=0;      //Disable change notification interrupt
12  CNPUBbits.CNPUB8=0;      //Disable weak pullup
13  CNPDBbits.CNPDB8=0;      //Disable weak pulldown
14  TRISBbits.TRISB8=0;      //Pin B8: Digital Output
15  LATBbits.LATB8=0;        //Pin B8: Low
16  _LED200 = 1;
17  //uint32_t    ui32Var1=2;
18  //uint32_t    ui32Var2=2;
19  //uint32_t    ui32Var3=2;
20  while (OSCCONbits.LOCK!= 1);
21  /* Endless Loop */
22
23  uint32_t ui32Number= 1000000;      //start value
24  uint16_t ui8PrimeCounter=0;        //counts the number of
        calculated prime numbers
25  const uint16_t ui8PrimMax=10;
26
27  while(1){
28
29    _LED200=1;       //hard on/off 28,5ns  scope_15
30
31    ui32Number= 1000000;      //start value
32    ui8PrimeCounter=0;        //counts the number of calculated
          prime numbers
33    //ui8PrimMax=10;
34
35    for(; ui8PrimeCounter<ui8PrimMax; ui32Number++)
36    if(isPrim(ui32Number)) //check if the number is prime
37    {
38      //printf("%d\t%d\n",ui8PrimeCounter,ui32Number);
39      ui8PrimeCounter++; //increase PrimeCounter, if the number
            is prime
40    }
41
42    _LED200=0;
43    delay_ms(500);
44
45  }//while
46
47  return (EXIT_SUCCESS);  //never reached
48 } //main()
```

```
49
50 uint8_t isPrim(uint32_t ui32Number){
51
52   if(ui32Number==0 || ui32Number==1)
53   return 0;
54
55
56   if((ui32Number%2)==0)
57   {
58     if(ui32Number==2)
59     {
60       return 1;
61     }
62     else
63     {
64       return 0;
65     }
66   }
67   uint32_t ui32Divider;
68   uint32_t ui32SqrtNumber =((uint32_t) sqrt((long
        double)(ui32Number)))+1;
69
70   for(ui32Divider=3; ui32Divider<ui32SqrtNumber; ui32Divider+=2)
71   {
72     if((ui32Number%ui32Divider) == 0) //check if ui32Divider is a
          in whole divider of ui32Number
73     {
74       return 0; //uiNumber32 isn't a prime number
75     }
76   }
77   return 1; //ui32Number is a Prime Number
78 }
```

Listing 6: Algorithmus zur Berechnung der ersten 100 Primzahlen größer als 1E6 auf dem uC

3 IO-Bibliothek

3.1 Aufgabenstellung

Die populäre Arduino-Plattform (https://www.arduino.cc/en/Reference/) kapselt die Pinkonfiguration und -ansteuerung mit folgenden Funktionen:

- pinMode()
- digitalRead()
- digitalWrite()

Übertragen Sie dieses Konzept auf das EDA-Board. Anwendungsbeispiele:

- pinMode(SW1, IPUT_PULLUP)

soll den Pin, an den SW1 angeschlossen ist, als digitalen Eingang konfigurieren und den Pullup-Widerstand einschalten.

- digitalWrite(LED203, HIGH)

soll an dem Pin, an den LED203 angeschlossen ist, einen High-Pegel ausgeben. Verwenden Sie die Bezeichner aus dem Schaltplan. Modularisieren Sie Ihre Software, verwenden Sie dazu die Dateinamen *edaPIC33Hardware.h* und *edaPIC33Hardware.c*.

Dokumentieren Sie die Funktionen mit Doxygen.

Messen Sie die Zeit, die zur Ansteuerung eines Ausgangspins mit den IO-Bibliotheksfunktionen notwendig ist und vergleichen Sie diese mit einem direkten Schreiben in die entsprechenden Hardwareregister.

3.2 Lösung

Alle Device-Pins (ausgenommen VDD, VSS, MCLR and OSC1/CLKI) sind aufgeteilt auf die Ports für Peripheriegeräte und parallel I/O Ports. Alle I/O Ports sind Schmitt-Trigger Input (verbesserte Störungsunempfindlichkeit / Rauschempfindlichkeit).
Alle Port Pins besitzen acht Register, durch diese Register lässt sich der I/O Port wie in Tabelle 1 dargestellt konfigurieren. Die Register Maps sind in Abbildung 13 dargestellt.

Ein Beispiel wie auf die einzelnen Register Bits zugegriffen werden kann und wie ein Port konfiguriert werden kann ist in Listing 7 zu sehen.

Die relevanten Auszüge aus dem Datenblatt sind in Abbildung 13 abgebildet.

Es wurde die Bibliothek mit den Dateien *edaPIC33Hardware.h* und *edaPIC33Hardware.c* erstellt. Die Funktion wurde mit Doxygen dokumentiert (TODO Anhang...).

Register Bit	Function
TRISx	determines whether the pin is an input or an output 0:=output, 1:=input default: all port pins are defined as inputs after a reset
PORTx	read reads the port, write writes the latch
LATx	read reads the latch, write writes the latch
ODCx	configures pin for digital or open-drain output 0:=digital output, 1:=open drain output
CNENx	enables change notification (CN) interrupts 0:=interrupts disabled, 1:=interrupts enabled
CNPUx	enables weak pullup 0:=pullup disabled, 1:=pullup enabled
CNPDx	enables weak pulldown 0:=pulldown disabled, 1:=pulldown enabled
ANSELx	controls the operation of the analog port pins 0:=port operates as digital I/O port 1:=port operates as analog I/O port
note	Any bit and its associated data and control registers that are not valid for a particular device is disabled. This means the correspon -ding LATx and TRISx registers and the port pin are read as zeros.
note	The open-drain feature allows the generation of outputs higher than VDD (e.g., 5V on a 5V tolerant pin) by using external pull-up resistors. The maximum open-drain voltage allowed is the same as the maximum VIH specification for that pin.
note	Pull-ups and pull-downs on change notification (CN) pins should be disabled when the port pin is configured as a digital output.

Tabelle 1: Konfigurationsmöglichkeiten der I/O Ports

```
1 //configure RB8 as digital input with pullup
2 TRISBbits.TRISB8=1;      //configure as input
3 ANSELBbits.ANSB8=0;      //configure as digital
4 CNENBbits.CNIEB8=0;      //disable change notification interrupt
5 CNPUBbits.CNPUB8=1;      //enable weak pullup
6 CNPDBbits.CNPDB8=0;      //disable weak pulldown
```

Listing 7: Konfigurieren des Ports RB8 als Digitaler Input mit Pullup Widerstand

3.3 Update der I/O Bibliothek

Im laufe der Vorlesung stellte es sich als sehr mühsam und fehleranfällig heraus jeden Port des PIC von Hand wie in Listing 7 zu konfigurieren.
Deshalb wurde ein Konzept entwickelt, mit welchem man möglichst allgemein und mit wenig Programmieraufwand jeden einzelnen I/O Port konfigurieren kann.

```
1  void pinMode(const uint8_t ui8Port,const  uint8_t ui8Mode);
2
3  void digitalWrite(const uint8_t ui8Port,const  uint8_t ui8Value);
4
5  void digitalToggle(const uint8_t ui8Port);
6
7  uint8_t digitalRead(const uint8_t ui8Port);
8
9  uint16_t* getpTRIS(uint8_t Port);
10
11 uint16_t* getpPORT(uint8_t Port);
12
13 uint16_t* getpLAT(uint8_t Port);
14
15 uint16_t* getpODC(uint8_t Port);
16
17 uint16_t* getpCNEN(uint8_t Port);
18
19 uint16_t* getpCNPU(uint8_t Port);
20
21 uint16_t* getpCNPD(uint8_t Port);
22
23 uint16_t* getpANSEL(uint8_t Port);
24
25 void setBit(uint16_t* pui16Var, uint8_t ui8Bit, uint8_t ui8Value);
26
27 uint8_t getBit(uint16_t ui16Var, uint8_t ui8Bit);
28
29 uint8_t getPortBitNumb(uint8_t Port);
```

Listing 8: Funktionsprototypen edaPIC33Hardware-Library

```
//Supported Modes:DIGITAL_INPUT, DIGITAL_INPUT_PULLDOWN,
    DIGITAL_INPUT_PULLUP, DIGITAL_OUTPUT, OPEN_DRAIN_OUTPUT,
    ANALOG_INPUT, ANALOG_OUTPUT, ANALOG_INPUT_PULLDOWN,
    ANALOG_INPUT_PULLUP
void pinMode(const uint8_t ui8Port,const uint8_t ui8Mode)
{
uint16_t* pTRIS =  getpTRIS(ui8Port);     //Output/Input select
uint16_t* pCNEN =  getpCNEN(ui8Port);     //CN Interrupt Enable Bit
uint16_t* pCNPU =  getpCNPU(ui8Port);     //Pull-Up enable Bit
uint16_t* pCNPD =  getpCNPD(ui8Port);     //Pull-Down enable Bit
uint16_t* pANSEL = getpANSEL(ui8Port);    //Analog select Bit
uint16_t* pODC =   getpODC(ui8Port);      //open drain select Bit
switch(ui8Mode)
{   //set ANSEL, TRIS, CNEN, CNPU, CNPD and ODC bit
  case DIGITAL_INPUT:
    setBit(pANSEL,getPortBitNumb(ui8Port), 0);
    setBit(pTRIS, getPortBitNumb(ui8Port), 1);
    setBit(pCNEN, getPortBitNumb(ui8Port), 0);
    setBit(pCNPU, getPortBitNumb(ui8Port), 0);
    setBit(pCNPD, getPortBitNumb(ui8Port), 0);
    setBit(pODC,  getPortBitNumb(ui8Port), 0);
  break;

  case DIGITAL_INPUT_PULLUP:
    setBit(pANSEL,getPortBitNumb(ui8Port), 0);
    setBit(pTRIS, getPortBitNumb(ui8Port), 1);
    setBit(pCNEN, getPortBitNumb(ui8Port), 0);
    setBit(pCNPU, getPortBitNumb(ui8Port), 1);
    setBit(pCNPD, getPortBitNumb(ui8Port), 0);
    setBit(pODC,  getPortBitNumb(ui8Port), 0);
  break;

  case DIGITAL_INPUT_PULLDOWN:
    setBit(pANSEL,getPortBitNumb(ui8Port), 0);
    setBit(pTRIS, getPortBitNumb(ui8Port), 1);
    setBit(pCNEN, getPortBitNumb(ui8Port), 0);
    setBit(pCNPU, getPortBitNumb(ui8Port), 0);
    setBit(pCNPD, getPortBitNumb(ui8Port), 1);
    setBit(pODC,  getPortBitNumb(ui8Port), 0);
  break;

  case DIGITAL_OUTPUT:
    setBit(pANSEL,getPortBitNumb(ui8Port), 0);
    setBit(pTRIS, getPortBitNumb(ui8Port), 0);
    setBit(pCNEN, getPortBitNumb(ui8Port), 0);
    setBit(pCNPU, getPortBitNumb(ui8Port), 0);
    setBit(pCNPD, getPortBitNumb(ui8Port), 0);
    setBit(pODC,  getPortBitNumb(ui8Port), 0);
  break;
```

```
47
48    case OPEN_DRAIN_OUTPUT:
49      setBit(pANSEL,getPortBitNumb(ui8Port), 0);
50      setBit(pTRIS, getPortBitNumb(ui8Port), 0);
51      setBit(pCNEN, getPortBitNumb(ui8Port), 0);
52      setBit(pCNPU, getPortBitNumb(ui8Port), 0);
53      setBit(pCNPD, getPortBitNumb(ui8Port), 0);
54      setBit(pODC,  getPortBitNumb(ui8Port), 1);
55    break;
56
57    case ANALOG_OUTPUT:
58      setBit(pANSEL,getPortBitNumb(ui8Port), 1);
59      setBit(pTRIS, getPortBitNumb(ui8Port), 0);
60      setBit(pCNEN, getPortBitNumb(ui8Port), 0);
61      setBit(pCNPU, getPortBitNumb(ui8Port), 0);
62      setBit(pCNPD, getPortBitNumb(ui8Port), 0);
63      setBit(pODC,  getPortBitNumb(ui8Port), 0);
64    break;
65
66    case ANALOG_INPUT:
67      setBit(pANSEL,getPortBitNumb(ui8Port), 1);
68      setBit(pTRIS, getPortBitNumb(ui8Port), 1);
69      setBit(pCNEN, getPortBitNumb(ui8Port), 0);
70      setBit(pCNPU, getPortBitNumb(ui8Port), 0);
71      setBit(pCNPD, getPortBitNumb(ui8Port), 0);
72      setBit(pODC,  getPortBitNumb(ui8Port), 0);
73    break;
74
75    case ANALOG_INPUT_PULLDOWN:
76      setBit(pANSEL,getPortBitNumb(ui8Port), 1);
77      setBit(pTRIS, getPortBitNumb(ui8Port), 1);
78      setBit(pCNEN, getPortBitNumb(ui8Port), 0);
79      setBit(pCNPU, getPortBitNumb(ui8Port), 0);
80      setBit(pCNPD, getPortBitNumb(ui8Port), 1);
81      setBit(pODC,  getPortBitNumb(ui8Port), 0);
82    break;
83
84    case ANALOG_INPUT_PULLUP:
85      setBit(pANSEL,getPortBitNumb(ui8Port), 1);
86      setBit(pTRIS, getPortBitNumb(ui8Port), 1);
87      setBit(pCNEN, getPortBitNumb(ui8Port), 0);
88      setBit(pCNPU, getPortBitNumb(ui8Port), 1);
89      setBit(pCNPD, getPortBitNumb(ui8Port), 0);
90      setBit(pODC,  getPortBitNumb(ui8Port), 0);
91    break;
92    }
93  }
```

Listing 9: pinMode() Funktion

```
void digitalWrite(const uint8_t ui8Port,const uint8_t ui8Value)
{
  if(ui8Value)
    setBit(getpLAT(ui8Port), getPortBitNumb(ui8Port), 1);
  else
    setBit(getpLAT(ui8Port), getPortBitNumb(ui8Port), 0);
}
```

Listing 10: digitalWrite() Funktion

```
void digitalToggle(const uint8_t ui8Port)
{
  digitalWrite(ui8Port, !digitalRead(ui8Port));
}
```

Listing 11: digitalToggle() Funktion

```
uint8_t digitalRead(const uint8_t ui8Port)
{
  return getBit(*getpPORT(ui8Port), getPortBitNumb(ui8Port));
}
```

Listing 12: digitalRead() Funktion

```
uint8_t getPortBitNumb(uint8_t Port)
{
  switch(Port)
  {
    case RA0:  return 0;
    case RA1:  return 1;
    case RA2:  return 2;
    //[...] fuer alle Pins definiert
    case RB0:  return 0;
    case RB1:  return 1;
    case RB2:  return 2;
    //[...]
    case RG13: return 13;
    case RG14: return 14;
    case RG15: return 15;
  }
  return 0;
}
```

Listing 13: getPortBitNumb() Funktion

```
void setBit(uint16_t* pui16Var, uint8_t ui8Bit, uint8_t ui8Value)
{
if(ui8Bit < 16 && ui8Value<2)
{
```

```
5   if(ui8Value)
6     *pui16Var |= (0x0001 << ui8Bit); //set to 1
7   else
8     *pui16Var &= (~(0x0001 << ui8Bit)); //set to 0
9   }
10 }
```

Listing 14: setBit() Funktion

```
1 uint8_t getBit(uint16_t ui16Var, uint8_t ui8Bit)
2 {
3 return (ui16Var & ( 1 << ui8Bit )) >> ui8Bit;
4 }
```

Listing 15: getBit() Funktion

```
1 uint16_t* getpTRIS(uint8_t Port)
2 {
3   if( Port == RA0 ||  Port == RA1 ||  Port == RA2 ||  Port == RA3
       ||  Port == RA4 ||  Port == RA5 ||  Port == RA6 ||  Port ==
       RA7 ||  Port == RA9 ||  Port == RA10 || Port == RA14 || Port
       == RA15)
4   return (uint16_t*)&TRISA;
5   //[...] fuer alle Pins definiert
6   if( Port == RG0 ||  Port == RG1 ||  Port == RG2 ||  Port == RG3
       ||  Port == RG6 ||  Port == RG7 ||  Port == RG8 ||  Port ==
       RG9 ||  Port == RG12 || Port == RG13 || Port == RG14 || Port
       == RG15)
7   return (uint16_t*)&TRISG;
8   return 0;
9 }
```

Listing 16: getpTRIS() Funktion

4 Timer-Konfiguration

4.1 Aufgabenstellung

Konfigurieren Sie Timer 1 als freilaufenden Timer. Implementieren Sie eine Software-PWM:

- Erzeugen Sie ein Ausgangssignal mit einer Frequenz von 1 kHz. Messen Sie das Signal mit dem Oszilloskop.
- Schreiben Sie ein Programm, das ein pulsweitenmoduliertes Signal mit einer Periodendauer von 1 ms (ggfs. 10 ms) und einem einstellbaren Tastverhältnis ausgibt.
- Erzeugen Sie damit ein PWM-Signal, dessen Gleichanteil einen dreieckförmigen Verlauf (Frequenz ca. 1 Hz) hat.
- Steuern Sie mit diesem Signal eine Leuchtdiode an

Optional: Ändern Sie das Programm "Hallo World" so ab, dass das Blinken der LED durch zyklisches Lesen des Timer1-Registers gesteuert wird.

4.2 Lösung

Das Timer1 Module ist ein 16-bit Timer (Zähler der mit einer konfigurierbaren Frequenz inkrementiert wird), welcher in folgenden Betriebsarten betrieben werden kann:

- Timer mode
- Gated Timer mode
- Synchronous Counter mode
- Asynchronous Counter mode

Die restlicher Timer (2-8) sind jeweils als 16-bit single Timer, oder als 32-bit Timer konfigurierbar. Bei den 32-bit Timern werden immer zwei Timer (2/3,4/5,6/7,8/9) zusammengefasst. Zur Konfiguration siehe Datenblatt.
Die Konfigurationsmöglichkeiten der Timer sind in Tabelle 2 abgebildet, das Blockdiagramm in Abbildung 6.

Mode	**TCS**	**TGATE**	**TSYNC**	Clock
Timer	0	0	X	$\frac{F_{OSC}}{2}$
Gated Timer	0	1	X	$\frac{F_{OSC}}{2}$
Synchronous Timer	1	X	1	T1CK pin
Asynchronous Timer	1	X	0	T1CK pin

Tabelle 2: Timer Mode Settings

Die Konfiguration von Timer1 ist in Listening 17 zu sehen. Hierbei wurde Timer1 als freilaufender Timer1 konfiguriert. Durch zyklisches Abfragen der Werte von TMR1 kann man verschiedene Zeitdifferenzen bestimmen. Die Anzahl der Inkrements pro Sekunde berechnet sich zu:

$$f_{inc} = \frac{F_{OSC}}{2 * Prescaler} \tag{3}$$

Die Anzahl der Inkrements die man für eine bestimmte Zeitdauer benötigt berechnet sich zu:

$$Inkrements = f_{inc} * t \tag{4}$$

Somit benötigt man beispielsweise für 0.5ms: $0.5ms * \frac{140MHz}{2*64} \approx 547$ Inkrements.

Bevor man die Differenzen der Timerwerte ausrechnet muss man einen Typecast auf int16_t vornehmen, durch die Interpretation als Zweierkomplemennt ist sichergestellt, das auch bei einem Überlauf des Timers die richtige Differenz berechnet wird. Ein Beispiel hierzu ist in Listening 18 zu sehen.
Der Tastgrad ist auch variabel eingestellt, wie in Listing 18 zu sehen toggelt der digitale Ausgang immer zwischen HIGH und LOW, abhängig von der Anzahl an "dT" Increments (siehe if-Abfragen).

```
T1CONbits.TON = 0; // Disable Timer
//set the multiplexer to timer mode
T1CONbits.TCS = 0;    // Select internal instruction cycle clock
T1CONbits.TGATE = 0; // Disable Gated Timer mode

T1CONbits.TCKPS = 0b10; // Select 1:64Prescaler

TMR1 = 0x00; // Clear timer register
PR1 = 0xFFFF; // Load the period value

//disable interrupt
IPC0bits.T1IP = 0x00;// Set Timer 1 Interrupt Priority Level
IFS0bits.T1IF = 0; // Clear Timer 1 Interrupt Flag
IEC0bits.T1IE = 0; // Enable Timer1 interrupt

T1CONbits.TON = 1; // Start Timer
```

Listing 17: Timer1 Configuration

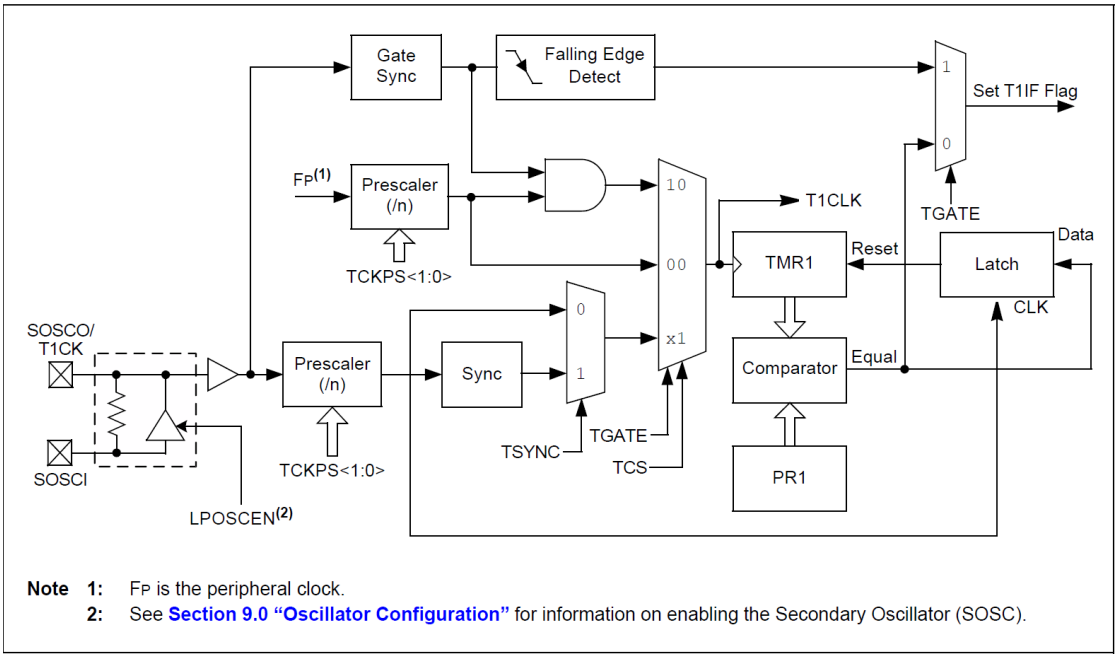
```
int16_t     _i16Timer1_1kHz_dTOn=0;
int16_t     _i16Timer1_1kHz_dTOff=0;
uint8_t     _ui8Timer1_1kHz_Port=0;

void onCycleTimer1PWM_1kHz()
{
  static uint8_t _ui8Timer1_1kHz_Mode=0;
  static int16_t _i16Timer1_1kHz_T0 = 0;
  int16_t i16_T1 = (int16_t)TMR1;
```

```
10    if(_ui8Timer1_1kHz_Mode == 0)
11    {
12      if((i16_T1 - _i16Timer1_1kHz_T0) >= _i16Timer1_1kHz_dTOff)
          //_i16Timer1_1kHz_dTOff
13      {
14        digitalWrite(_ui8Timer1_1kHz_Port,HIGH);
15        _i16Timer1_1kHz_T0 = (int16_t)TMR1;
16        _ui8Timer1_1kHz_Mode=1;
17      }
18    }
19    else
20    {
21      if((i16_T1 - _i16Timer1_1kHz_T0) >= _i16Timer1_1kHz_dTOn)
          //_i16Timer1_1kHz_dTOn
22      {
23        digitalWrite(_ui8Timer1_1kHz_Port,LOW);
24        _i16Timer1_1kHz_T0 = (int16_t)TMR1;
25        _ui8Timer1_1kHz_Mode=0;
26      }
27
28    }
29  }
```

Listing 18: Timer1 on Cycle

FIGURE 12-1: 16-BIT TIMER1 MODULE BLOCK DIAGRAM



Note 1: FP is the peripheral clock.
2: See **Section 9.0 "Oscillator Configuration"** for information on enabling the Secondary Oscillator (SOSC).

Abbildung 6: Timer1 Block Diagramm

5 SystemTime

5.1 Aufgabenstellung

Schreiben Sie ein Modul mit folgender Funktionalität:

- getSystemTimeMillis()

Die genannte get-Funktion soll den Wert eines Zählers liefern, welcher in der Timer-1-ISR jede Millisekunde inkrementiert wird. Der Zähler selbst soll außerhalb des Moduls nicht sichtbar sein. Innerhalb des Moduls soll auch der Timer konfiguriert werden.
Testen Sie das Modul mit einem Programm, das mittels getSystemTimeMillis() ein Rechtecksignal mit einer Frequenz von 50 Hz und einer Einschaltdauer von 25% generiert.

Hinweise zur Implementierung:
Beachten Sie, dass ein Interrupt auch eine 32-Bit-Leseoperation unterbrechen kann. Entwickeln Sie einen Mechanismus, mit dem getSystemTimeMillis() auch in diesem Fall einen konsistenten Zählerstand zurückliefert.

Ergänzung:
Konfigurieren Sie Timer 3 in Verbindung mit dem 32-kHz-Quarz so, dass jede Sekunde ein Interrupt ausgelöst wird. Schreiben Sie in gleicher Weise wie oben eine Funktion getSystemTimeSeconds() Testen Sie die Funktion, indem Sie damit eine LED im Zweisekundentakt blinken lassen.

5.2 Lösung

Die geforderte Funktion wurde durch die drei Funktionen aus Listing 19 realisiert. Die Funktion configSystemTimeMillis() konfiguriert Timer 1 als nicht freilaufenden Timer mit der Incrementfrequenz $F_P = \frac{F_{OSC}}{2}$ und erlaubt Interrupts.
In der Interrupt Service Routine (ISR) wird dann die globale Variable ui32SystemTimeMillis nach jedem ausgelöstet Interrupt erhöht.
Bevor diese Variable in der getSystemTimeMillis() Funktion gelesen wird, wird ein globales Access Flag gesetzt (ui8SystemTimeMillisAccesFlag). Versucht man während eines Interrupts die Variable zu lesen, dann erkennt dies die ISR und setzt ein Kollisions-Flag (ui8SystemTimeMillisConflictFlag). Wurde eine Kollision erkannt, dann liefert getSystemTimeMillis() den zuletzt gespeicherten Wert zurück, so wird eine falscher Rückgabewert beim Lesen der ui32SystemTimeMillis-Variable während einem Interrupt vermieden.

```
void configSystemTimeMillis()
{
  T1CONbits.TON = 0; //Disable Timer
  T1CONbits.TCS = 0; //internal instruction cycle clock
  T1CONbits.TGATE = 0; // Disable Gated Timer mode
  T1CONbits.TCKPS = 0b01; // Select 1:8 Prescaler
  TMR1 = 0x00; // Clear timer register
  PR1 = 8750-1;  //Period Value: 140Mhz: 8750-1 / 120Mhz: 7500-1
  IPC0bits.T1IP = 0x01; //Timer 1 Interrupt Priority Lvl
  IFS0bits.T1IF = 0; // Clear Timer 1 Interrupt Flag
  IEC0bits.T1IE = 1; // Enable Timer1 interrupt
  T1CONbits.TON = 1; // Start Timer
}

uint32_t getSystemTimeMillis()
{
  uint32_t ui32ReturnValue=0;
  static uint32_t ui32OldValue=0;

  ui8SystemTimeMillisAccesFlag = 1;
  ui32ReturnValue = ui32SystemTimeMillis;
  ui8SystemTimeMillisAccesFlag = 0;

  if(ui8SystemTimeMillisConflictFlag ==1)
  {
  ui8SystemTimeMillisConflictFlag=0;
  ui32ReturnValue =  ui32OldValue;
  }

  ui32OldValue = ui32ReturnValue;

  return ui32ReturnValue;
}

void __attribute__((__interrupt__, no_auto_psv))
    _T1Interrupt(void) //Timer1 ISR
{
  ui32SystemTimeMillis++;  //increase millis counter
  if (ui8SystemTimeMillisAccesFlag == 1)    //if acces flag=1,
     set config flag
  ui8SystemTimeMillisConflictFlag = 1;
  else
  ui8SystemTimeMillisConflictFlag = 0;

  IFS0bits.T1IF = 0;      //Clear Timer1 interrupt flag
}
```

Listing 19: System Time Funktionen

6 Non-blocking Code

6.1 Aufgabenstellung

Schreiben Sie ein Programm mit folgender Funktionalität:

- Die 4 LEDs auf dem Board sollen mit unterschiedlichen Periodendauern blinken: 2*300ms, 2*500ms, 2*700ms und 2*1100ms blinken.
- Zu wie viel Prozent ist die CPU ausgelastet?
- Die Leuchtdauer der LEDs soll per Tastendruck einstellbar sein. Genau die Zeitdauer, während der ein zugeordneter Taster gedrückt ist, definiert den Sollwert für die halbe Periodendauer.

Hinweise zur Implementierung:

- Wenden Sie das Konzept des kooperativen Multitaskings (non-blocking Code) an.
- Verwenden Sie getSystemTimeMillis().

6.2 Lösung

Bei der Implementierung der geforderten Funktionen wurde davon ausgegangen, dass die Funktionen zyklisch jede ms aufgerufen werden. Somit können die Funktionen losgelöst von Timern implementiert werden.
Die Funktion zum zyklischen blinken der LED sind in Listening 20 dargestellt. Die Funktion zum messen der ToggleTime sind in Listening 21 dargestellt.
Die Auslastung der CPU liegt bei ca. 1%. In Listening 22 ist die main Funktion dargestellt, diese stellt sicher, dass alle Funktionen zyklisch jede ms aufgerufen werden.

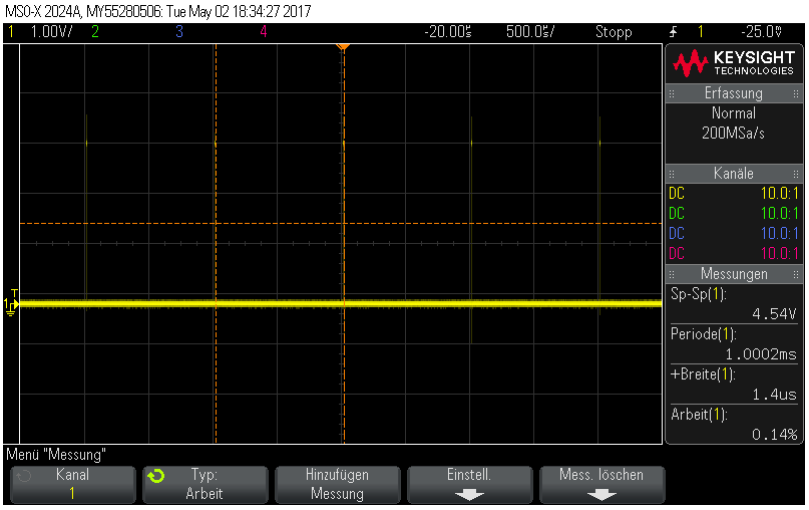


Abbildung 7: Auslastung der CPU

```
/**
* @brief Toggle LED0 (RB_8) with toggle Time
* @param uint16_t ui16ToggleTime Toggle Time in Functions Calls
* @details Toggles RB_8 with the number of functions calls.
* @details Example: If Toggle Time is 20, you have to call the
    function 20 times to toggle RB_8
* @attention toggle pin has to be defined as digital output first
*/
void blinkLed0(uint16_t ui16ToggleTime)
{
  static uint16_t ui16Counter = 0;

  if(ui16ToggleTime == 0)
  {
    ui16Counter=0;
    digitalWrite(LED0,LOW);
    return;
  }
  else
    ui16Counter++;

  if(ui16Counter >= ui16ToggleTime)
  {
    ui16Counter=0;
    digitalToggle(LED0);
  }
}
```

Listing 20: blinkLed0() Funktion

```
/**
* @brief Measure hold Time from RG_12
* @returns uint16_t ui16HoldTime HoldTime is measured in function calls
* @details Measures Hold Time in Functions calls, if the Pin is not hold (high level) the functions retuns the last measured hold time
* @details Example: If the Pin is hold for 40ms you get the return value 40 when the function is called each ms, or the return value 20 when the function is called each 2ms
* @attention Pin has to be configured as input first, the functions is low active
*/
uint16_t measureToggleTimeSW0()
{
  static uint16_t ui16ToggleTime=0;
  static uint8_t  ui8MeasureMode=0;

  if(digitalRead(SW0)==LOW && (ui8MeasureMode==0))
  {
    ui16ToggleTime=0;
    ui8MeasureMode=1;
  }

  if(ui8MeasureMode==1)
  {
    ui16ToggleTime++;
  if(digitalRead(SW0)==HIGH)
    ui8MeasureMode=0;
  }

  return ui16ToggleTime;
}
```

Listing 21: measureToggleTimeSW0() Funktion

```
int main() {
  configOscillator();

  //set LED pinmodes
  pinMode(LED0,OUTPUT);
  pinMode(LED1,OUTPUT);
  pinMode(LED2,OUTPUT);
  pinMode(LED3,OUTPUT);

  //set switch pinmodes
  pinMode(SW0, INPUT_PULLUP);
  pinMode(SW1, INPUT_PULLUP);
  pinMode(SW2, INPUT_PULLUP);
  pinMode(SW3, INPUT_PULLUP);

  //config timer 1 for getSystemTimeMillis();)
  configSystemTimeMillis();

  /* Endless Loop */
  while(1){
    static uint32_t ui32Time= 0;

    blinkLed(LED0, measureToggleTimeSW(SW0));
    blinkLed(LED1, measureToggleTimeSW(SW1));
    blinkLed(LED2, measureToggleTimeSW(SW2));
    blinkLed(LED3, measureToggleTimeSW(SW3));

    ui32Time++; //increase ms counter
    while(getSystemTimeMillis() < ui32Time) //wait rest of 1ms
    {
      ClrWdt();   //clear watchdog timer
    }
  }//while
  return (EXIT_SUCCESS);  //never reached
} //main()
```

Listing 22: NonBlockingCode Main-Funktion

7 Software-Entwurf mit FSM

7.1 Aufgabenstellung

1. Entprellen der Taster
   - Messen Sie die typische Prelldauer einer Taste mit dem Oszilloskop. Welche Entprellzeit schlagen Sie vor?
   - Schreibe Sie eine Funktion `isPressed()`, mit der Tasteneingaben entprellt werden.
2. LED mit einem Taster an- und ausschalten
   - Eine LED soll mit einem Tastendruck angeschaltet und mit einem weiteren Tastendruck ausgeschaltet werden.
   - Optional: Mit jedem Tastendruck soll eine LED aus- und die nächste angeschaltet werden.
3. Nachbildung eines Treppenlichtautomaten
   - Auf Tastendruck geht das Licht (LED) an
   - Das Licht bleibt für eine bestimmte (tbd.) Zeit an
   - Anschließend wird das Licht kontinuierlich bis auf Null gedimmt
   - Der Automat soll nachtriggerbar sein

Hinweise zur Implementierung:

- Modellieren Sie die Funktionalität als FSM (Moore-Automat).
- Beachten Sie, dass das Debouncen (Entprellen) bei beiden Flankenwechseln stattfinden muss

7.2 Lösung

1. Es wurden mehrere Prellvorgänge mit dem Oszilloskop gemessen. Je nach verwendetem Taster und der Art wie man den Taster drückt sind verschiedenen Prellvorgänge aufgenommen worden, diese sind in Abbildung 8 dargestellt.
   Die längste aufgenommene Prelldauer beträgt 250us. Ich wähle jedoch pauschal eine Entprelldauer ("Debounce-Time") von $1ms$ vor, da durch vorherige Programmierschritte festgelegt wurde, dass der Mikrocontroller die "Loop"zyklisch im Takt von $1ms$ durchläuft.
   Die Funktion *isPressedSW0()* (Listening 23) wurde als Zustandsautomat Implementiert. Die Funktion sollte zyklisch aufgerufen werden. Eine Skizze des Zustandsautomats ist in Abbildung 9 zu sehen.
2. Es wurde eine Funktion geschrieben, welche immer bei einem Tastendruck (Flanke von High auf Low) die LED toggled (Listening 24).

3. Der Treppenlichtautomat wurde ebenfalls als Zustandsautomat implementiert, von der Auflistung des Quellcodes wird aus Übersichtsgründen abgesehen.

```
#define STATE_STABLE_HIGH   0
#define STATE_INSTABLE_HIGH 1
#define STATE_STABLE_LOW    2
#define STATE_INSTABLE_LOW  3
const uint16_t cui16DebounceTime=1;


uint8_t isPressedSW0()
{
  static uint8_t ui8State = STATE_STABLE_HIGH; //default state
  static uint16_t ui16Counter=0;
  uint8_t ui8ReturnValue=1;

  switch(ui8State)
  {
  case STATE_STABLE_HIGH:
    if(digitalRead(SW0)==LOW)
    {
      ui8State = STATE_INSTABLE_HIGH;
      ui16Counter=0;
    }
    ui8ReturnValue= HIGH;
  break;

  case STATE_INSTABLE_HIGH:
    ui16Counter++;
    if(digitalRead(SW0)==LOW)
    {
      if( ui16Counter>=cui16DebounceTime )
      {
        ui8State = STATE_STABLE_LOW;
        ui8ReturnValue= LOW;
      }
      else
        ui8ReturnValue= HIGH;
    }
    else
    {
      ui8State = STATE_STABLE_HIGH;
      ui8ReturnValue= HIGH;
    }
  break;

  case STATE_STABLE_LOW:
    if(digitalRead(SW0)==HIGH)
    {
```

```
47        ui8State = STATE_INSTABLE_LOW;
48        ui16Counter=0;
49      }
50      ui8ReturnValue= LOW;
51    break;
52
53    case STATE_INSTABLE_LOW:
54      ui16Counter++;
55      if(digitalRead(SW0)==HIGH)
56      {
57        if(ui16Counter >= cui16DebounceTime)
58        {
59          ui8State = STATE_STABLE_HIGH;
60          ui8ReturnValue= HIGH;
61        }
62        else
63          ui8ReturnValue= LOW;
64      }
65      else
66      {
67        ui8State = STATE_STABLE_LOW;
68        ui8ReturnValue= LOW;
69      }
70    break;
71
72    default:
73      ui8State = STATE_STABLE_HIGH;
74      ui8ReturnValue= HIGH;
75    break;
76    }
77
78    return ui8ReturnValue;
79  }
```

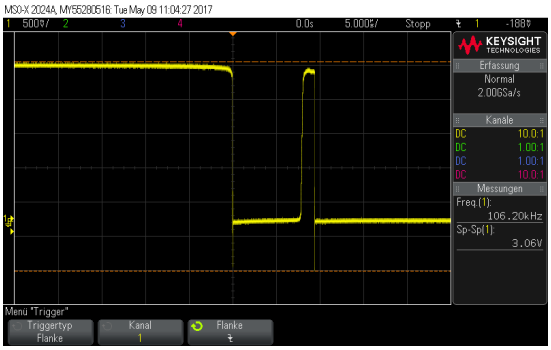
Listing 23: Funktion isPressedSW0()

```
1  void FlipFlopLED0(uint8_t ui8SwitchState)
2  {
3    static uint8_t ui8OldSwitchState = HIGH;
4
5    if(ui8OldSwitchState == HIGH && ui8SwitchState == LOW) //Flanke
       von HIGH auf LOW
6    {
7      digitalToggle(LED0);
8    }
9    ui8OldSwitchState=ui8SwitchState;
10 }
```

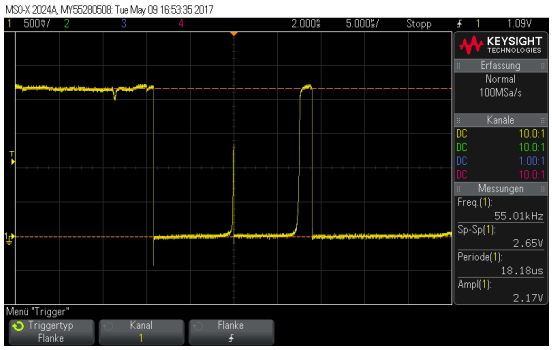
Listing 24: LED auf Tastendruck Toggeln

```
1  void digitalCountLED0to3(uint8_t ui8SwitchState)
2  {
3  static uint8_t ui8OldSwitchState = HIGH;
4  static uint8_t ui8CountValue=0;
5
6    if(ui8OldSwitchState == HIGH && ui8SwitchState == LOW) //Flanke
       von HIGH auf LOW
7    {
8      ui8CountValue++;
9      digitalWrite(LED0, ui8CountValue&0x01);
10     digitalWrite(LED1, ui8CountValue&0x02);
11     digitalWrite(LED2, ui8CountValue&0x04);
12     digitalWrite(LED3, ui8CountValue&0x08);
13   }
14   ui8OldSwitchState=ui8SwitchState;
15 }
```

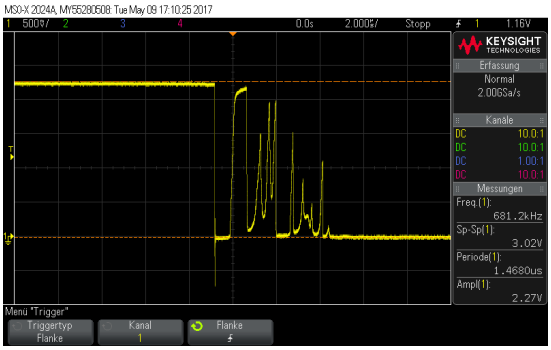
Listing 25: Funktion digitalCountLED0to3()



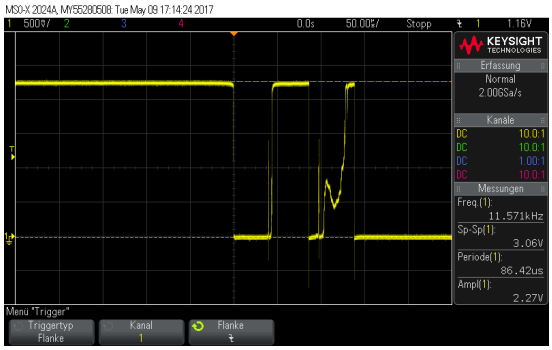
(a) Prellvorgang von ca $10us$



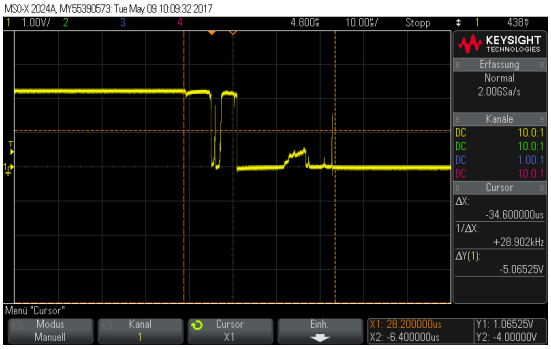
(b) Prellvorgang von ca $8ns$



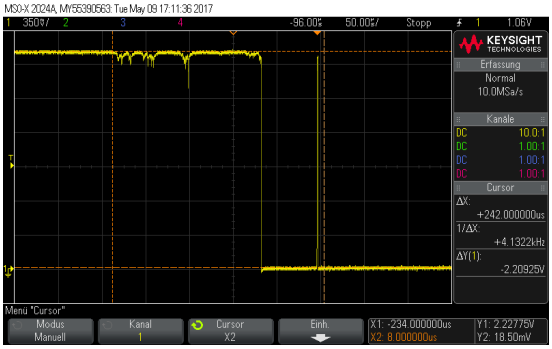
(c) Prellvorgang von ca $6us$



(d) Prellvorgang von ca $150us$

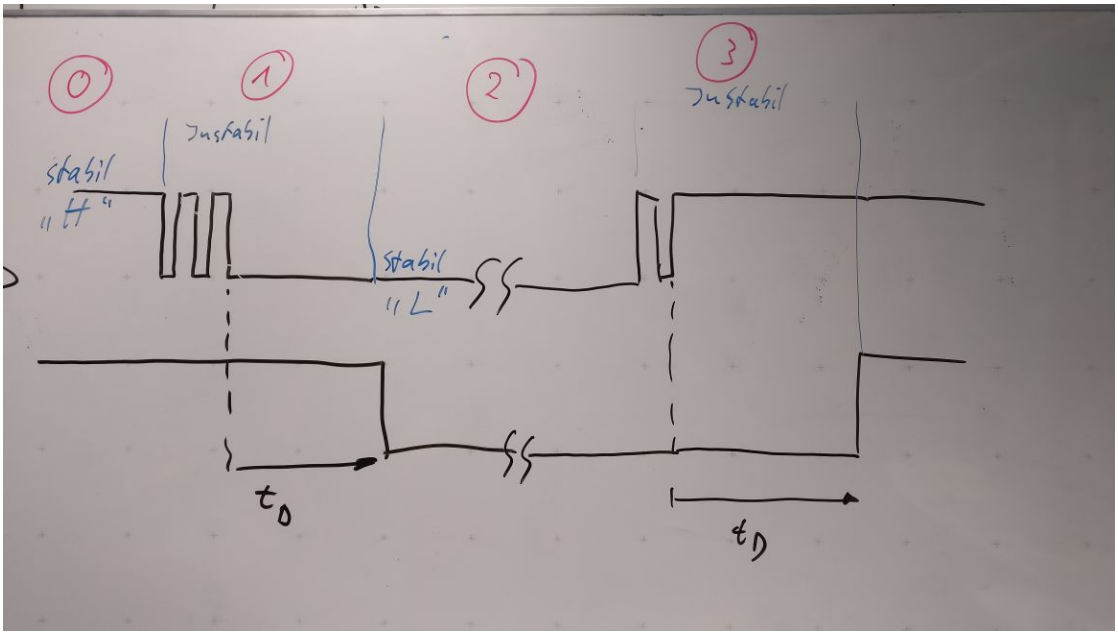


(e) Prellvorgang von ca $35us$

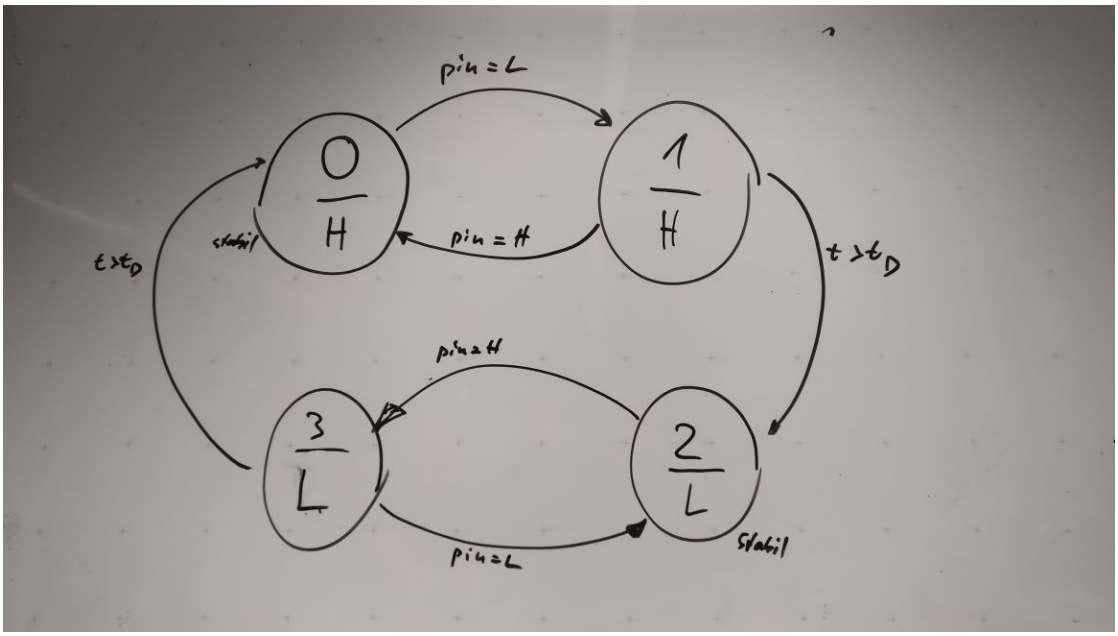


(f) Prellvorgang von ca $250us$

Abbildung 8: Mehrere Prellvorgänge (entnommen aus Moodle)



(a) Zeitliches Verhalten beim Prellen



(b) Zustandsautomat

Abbildung 9: Skizzen zu der Funktion isPressed()

8 LCD-Bibliothek

8.1 Aufgabenstellung

Schreiben Sie eine Bibliothek zur Ansteuerung des LC-Displays auf dem EDAdsPIC33-Board:

- Initialisierung (blocking Code erlaubt)
- Zyklisches Kopieren eines Schattenspeichers in den Display-Speicher (non-blocking code).

Hinweise zur Implementierung

Literatur:

- DiJasio(2012): Chapter 9
  https://www.dropbox.com/sh/4bwfkqux88a3j62/AAAhId8mi25vsYaToLJbrS1Ba?dl=0&preview=09_Glass_Bliss.pdf
- Datenblatt FDCC1602L-NSWBBW-91LE,
  http://farnell.com/datasheets/653662.pdf
- Datenblatt Displaycontroller
  https://sparkfun.com/datasheets/LCD/HD44780.pdf
  http://download.maritex.com.pl/pdfs/op/TC1602E-06H.pdf
- Simulator
  http://dinceraydin.com/djlcdsim/djlcdsim.html

Vorgehensweise:

- Schreiben Sie zunächst eine Funktion putLCD(), die ein Zeichen auf dem Display ausgibt (blocking code)
- Ändern Sie die Funktion so, dass sie kooperativ wird.
- Implementieren Sie anschließend das Kopieren des Schattenspeichers

8.2 Lösung

Die Initialisierung des LCD-Controllers wurde wie im Datenblatt beschrieben umgesetzt. Die im Datenblatt vorgeschlagene Initialisierungssequenz ist in Abbildung 17 (Seite 55) abgebildet. Die Funktion `initMyLCD()` ist in Listening 29 (Seite 49) zu finden.

Alle Funktionenprototypen der edaPIC33LCD-Library sind in Listening 30 (Seite 50) zu finden.
Eine Ausführliche Beschreibung und alternativer Quellcode ist in Programming 16-Bit PIC Microcontrollers in C (Di Jasio) - Chapter 9 zu finden.

Abstrahierte Verwendung des LCD

Um die Verwendung des LCD zu vereinfachen wurde eine Funktionalität entwickelt, welche einen Schattenspeicher zyklisch auf das LCD schreibt. Hierfür wurden die Funktionen in Listing 26 realisiert.

```
1 extern char ShadowString[32];
2 void setLCDLine(const char* pStr, uint8_t ui8Line);
3 void sendDataToLCD();
```

Listing 26: Funktionsprototypen für Schattenspeicher Funktionalität

Der Funktion `setLCDLine(pStr, Line)` wird ein String und die gewünschte Zeile des LCD übergeben. Der übergeben String (max. 16 Zeichen) wird in den `ShadowString` kopiert. Je nach übergebener Zeile werden die Daten ab `ShadowString[0]` (Zeile 0) bzw ab `ShadowString[16]` (Zeile 1) gespeichert.
Ist der übergebene String (`pStr`) kleiner als 16 Zeichen, so wird die restliche Zeile von `ShadowString` mit Leerzeichen gefüllt (dies vereinfacht die Programmierung der Funktion `sendDataToLCD()`). Ist der übergebene String größer als 16 Zeichen, so werden nur die ersten 16 Zeichen kopiert.
Der String `pStr` muss zwingend nullterminiert sein (Ende gekennzeichnet durch `'\0'`).
Die Funktion `sendDataToLCD()` ist so programmiert, dass bei einem Funktionsaufruf zuerst das Busy-Flag des LCD abgefragt. Anschließend wird, wenn der LCD nicht beschäftigt ist, ein Zeichen des `ShadowString` an den LCD gesendet. Im nächsten Funktionsaufruf wird dann das nächste Zeichen an den LCD gesendet, oder (wenn nötig) den LCD Cursor auf Zeile 2 setzen, bzw. auf Zeile 1.

In Abbildung 10 ist das Busy-Flag des LCD abgebildet. Das LCD benötigt zum schreiben eines Characters ca 30us.

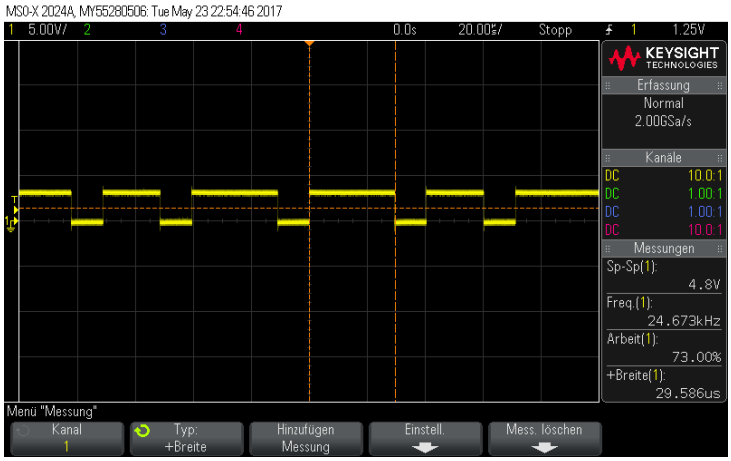


Abbildung 10: Busy-Flag des LCD Display

9 Rotary Encoder

9.1 Aufgabenstellung

Auswertung des Drehimpulsgebers

Funktionsprototyp

- `int8_t rotaryEncode()`

Vorgehensweise

1. Anzeige der Signale mit den auf dem Board vorhandenen LEDs
2. Entwicklung eines Konzepts zur Auswertung:
   - Drehrichtungserkennung
   - Entprellen?
   - Kommunikation mit dem aufrufenden Programm

Erweiterte Funktionalität (Schreibmaschine)

1. Schreiben und Bearbeiten eines Textes auf dem LC-Display

9.2 Lösung

Drehgeber (auch Inkrementaldrehgeber, Quadraturencoder, Drehencoder, Drehimpulsgeber genannt) dienen der dynamischen Erfassung von Winkeländerungen bei Achsen und Wellen. Sie werden sowohl für die manuelle Eingabe von Werten, als auch zur Ermittlung von Drehgeschwindigkeiten eingesetzt.
Inkrementalgeber können mit Schleifkontakten, photoelektrisch, oder magnetisch arbeiten. Sie liefern am Ausgang immer zwei um 90 Grad gegeneinander phasenverschobene Signale (siehe Abbildung 11), anhand deren sich Drehrichtung und -winkel bestimmen lassen.
Die Auswertung des Inkrementalgeber wurde nach dem Zustandsautomat aus Abbildung 12 umgesetzt. Die dazugehörigen Funktionsprototypen sind in Listing 27 aufgeführt.
Die Funktion muss zur Initialisierung des Startzustandes zweimal aufgerufen werden, bevor diese aktiv verwendet wird. Weiterhin ist es nötig die Funktion in regelmäßigen Zeitabständen aufzurufen.

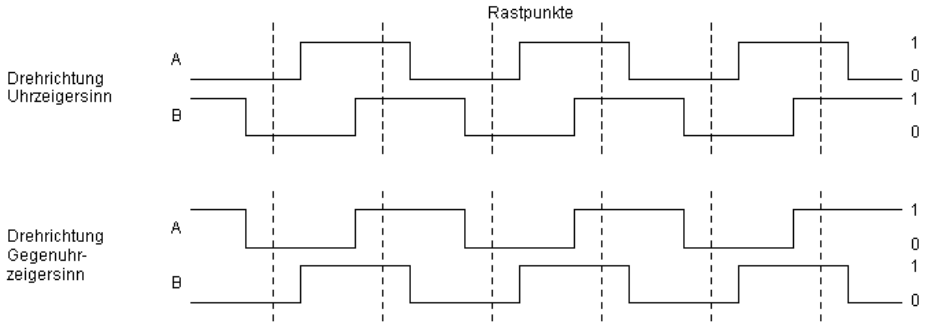


Abbildung 11: Signale des Drehimpulsgeber

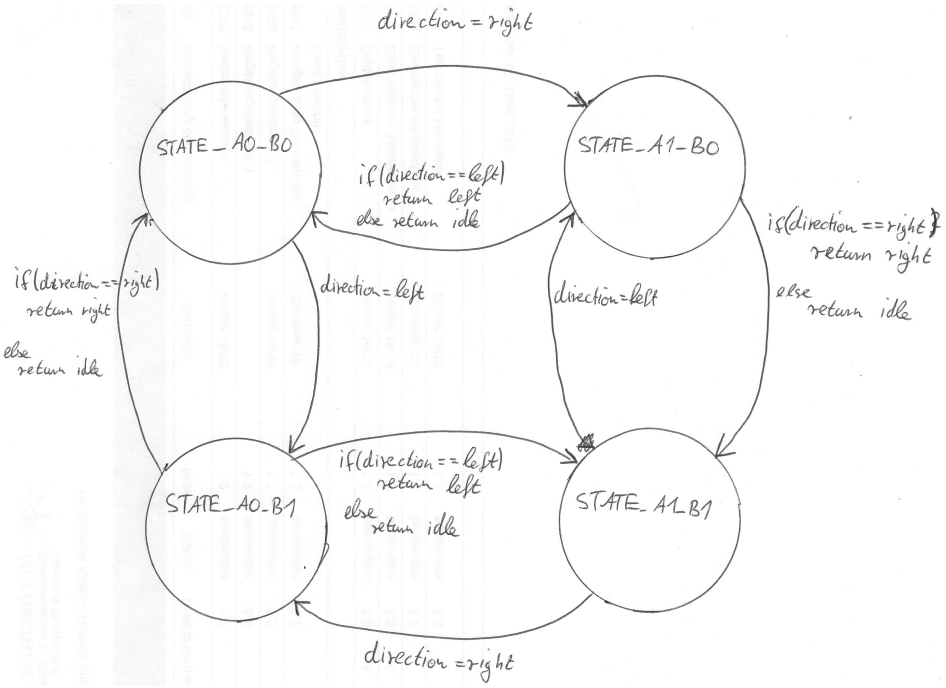


Abbildung 12: Statemachine rotaryEncode()

```
#define LEFT -1
#define RIGTH 1
#define IDLE 0
#define STATE_A0_B0 0
#define STATE_A1_B0 1
#define STATE_A0_B1 2
#define STATE_A1_B1 3
int8_t rotaryEncode();
```

Listing 27: Funktionsprototypen für Inkrementalgeber

10 AD-Wandler

10.1 Aufgabenstellung

- ADC Basics
  - Lesen Sie die auf dem Board vorhandenen zwei ADC-Kanäle ein
  - Zeigen Sie die Spannungswerte auf dem LC-Display an
- ADC Advanced
  - Implementieren Sie eine Quasi-Analoganzeige für ein Signal
    $\Rightarrow$ "Fortschrittsbalken"
  - Legen Sie jede Millisekunde einen Messwert in einem Ringspeicher ab
- Hinweise zur Implementierung:
  - diJasio 2012, S.179f. bzw. 351ff.
  - erweitern Sie die IO-Bibliothek entsprechend
  - Non-Blocking Code
- Schwellwertschalter mit Hysterese (Schmitt-Trigger)
  - Zeigen Sie das Ausgangssignal des Schmitt-Triggers mit einer Leuchtdiode an
  - Vergleichen Sie den Schmitt-Trigger mit einem Schwellwertschalter ohne Hysterese (Messung am Oszilloskop)

10.2 Lösung

```
1 //Quellcode
```

Listing 28: Listening Bezeichnung

11 Mitschrift aus Vorlesung

11.1 Vorlesung 04.04.2017 - IO-Bibliothek

- jeder uC is universell aufgebaut und muss für jeden Fall individuell konfiguriert werden
- kleine Bibliothek zum anpassen der IO-Ports ist praktisch
- Lese und Schreib Funktionalität soll realisiert werden
- Datenblatt Figure 11-1 -> Pins müssen über Treiber realisiert werden, Schmitt-Trigger ist enthalten
- Pegel um Treiber zu aktivieren? nachlesen Schaltbild könnte falsch sein
- Lesezugriff über Port, Schreibezugriff über Latch
- Erkenntnis: Lesen des Datenblattes liefert Aufschluss wie die Pins zu beschalten sind.

11.2 Vorlesung 11.04.2017 - Timer-Konfiguration

- An beliebigen Pin soll PWM ausgegeben werden-> Nachbildung durch Software
- Software PWM soll Leuchtdiode ansteuern
- Der dsPIC22EP810MU verfügt über 9 Zeitgeber Timer1 ... Timer9
- Unterschiedliche Struktur 16/32Bit
- Können Interrupts auslesen
- Übersicht/Kurzfassung in Datenblatt Abschnitt 12 / S271ff.
- Dataillierte Information Mikrochip dsPIC33E/PIC24E Familiy Reference Manual Abschnitt 11, mit Konfigurationscode, sehr hilfreich
- Timer1: Figure 12-1 16-Bit Timer1 Modul Block Diagramm
- Timer ist im Prinzip Zähler -> 16Bit Register das incrementiert werden kann
- Prescaler Teilerfaktor kann eingestellt werden
- Abbruch durch Vergleicher -> PR1.
- Flag kann gesetzt werden -> Interrupt erzeugen und/oder Zähler zurücksetzen
- es gibt viele Möglichkeiten, die durch äußere Parameter einstellbar sind
- 16-Bit-Timer-1 verwendbar als

  - Timer (Zeitgeber), auch in Verbindung mit dem low Power 32kHz Quarzoszillator -> kann Rechner z.B. schlafen legen
  - Gated Timer (Periodendauermessung eines externen Signals)
  - Counter (Zählen externer Impulse)
- Initialation Coder for 16-bit Timer Module Example 11-1, löst auch Interrupts aus, benötigen wir (noch) nicht. Abschnitt 11, S.11
- Alternativen zur Konfiguration mittels Handcodierung
  - OpenTimerX-Funktionen aus der plib-Bibliothek
  - MPLAB Code Configurator (grafisches Tool) ... leider nicht für alle μC verfügbar, für unseren auch nicht
- Timer3 Konfiguration
  - TypeC Timer kann auch A/D Wandlungen zyklisch abtasten
  - Timer sind kaskadierbar

11.3 Vorlesung 25.04.2017 - SystemTime

- Millis soll in 32Bit Variable gespeichert werden
- Interrupt unterbricht laufende Operationen und verzweigt an eine Stelle im Programmcode
- Interrupt soll ms Tigger incrementieren
- 16Bit rechner & 32Bit Variable kann Probleme beim lesen verursachen, da mehrere Operationen verwendet werden
- Timer 3 in Sekundentakt

Ein Interrupt ist die Unterbrechung eines Programm durch ein Ereignis, z.B.:

- Signale an I/O Pins
- Zeitgeber (Timer)
- ADC mit Konvertierung fertig
- Serielle Kommunikation
- EEPROM-Schreibzugriff beendet

Das Vorhandensein von Interruptquellen ist abhängig von im Baustein implementierten Peripheriekomponenten.

Interrupt:

- Ein Interrupt unterbricht das laufende Programm und verzweigt zur Interruptserviceroutine ISR
- Nach Abarbeiten der ISR wurd das unterbrochene Programm fortgesetzt

Interruptvektor:

- Sammlung aller Programmspeicherstellen bei denen Interrupts stehen
- ...

Interruptsserviceroutine

- Code, der nach einer Interruptanforderung abgearbeitet wird

Interruptlatenzzeit

- Zeitraum zwischen Interruptanforderung und Abarbeitung des Befehls in der ISR

Trap:

- Spezielle Interrupt, der bei schwerwiegenden Fehlern ausgelöst wird (z.B. Division durch Null, Oszillatorausfall etc.)
- führt Standartmäßig zu einem Reset

2Schritte:

- Konfigurieren der Hardware, die den Interrupt auslösen soll
- Konfiguration des dazugehörigen Interrupts

Interrupts konfigurieren:

- IECx: Interrupt enable
-

ISR vs. C-Funktion

- Da nicht vorhergesagt werden kann, an welcher Stelle ein Interrupt den Programmablauf unterbricht, müssen funktionskritische Daten z.B.: ALU-Stati, von der ISR gesichert sein und vor der Rückkehr zum unterbrochenen Programm wiederhergestellt werden
- Das dazugehörige Code wird vom Compiler eingefügt
- Der C-Compiler benötigt eine zusätzliche Information darüber, dass eine Codesequenz eine ISR und keine gewöhnliche Funktion ist (spezielle Syntax)

ISR:

- Unterscheidung von einer Funktion durch Schlüsselwörter oder pragma-Direktiven
- ist vom Typ void
- keine Parameterübergabe
- soll kurze Laufzeiten haben
- soll keine anderen Funktionen aufrufen
- Globale Variablen, die in der ISR Routine verändert werden als volatile deklarieren
- Nach Abarbeitung des Interrupts Interrupt Flag-löschen
- Flag von Timer zurücksetzen

11.4 Vorlesung 02.05.2017 - Non-blocking Code

- Implementierung mit State Machines...

11.5 Vorlesung 16.05.2017 - LCD-Bibliothek

- LC-Display
  - FDCC1602L-NSWBBW-91LE
  - alphanumerisches Display
  - 2Zeilen, 16Zeichen/Zeile
  - HD44780 Controller, steuer Display Hardware an
  - LC-Controller ist über einen 8bit Datenstrom angeschlossen
  - Die Kommunikation zwischen PIC und HD44780 erfolgt nach einem definierten Protokoll
  - Initialisierungssequenz für HD44780 Controller notwendig
  - Blockschaltbild von Display-Controller
- Aus Sicht des uC-Programmierers
  - HD44780 hat ein Instruktions-(IR) und ein Datenregister DR
  - beide Register sind 8Bit Register
  - Über diese beiden Register erfolgt die Kommunikation mit dem PIC uC
  - Interne Register
    - DDRAM: Display Data RAM
    - CGRAM: Character Generator RAM
  - 11 ansteuerbare Pins, D0-D7, E(nable), RS,RW
    - RS Register Select 0: Incstruction, 1: Data

        * R/notW 0:Write, 1:Read
        * E Enable
        * Dx D7...D0 (8-Bit-Modus) D7:BusyFlag
        * 8-Bit-Daten werden in einem Paket übertragen
        * mit fallender Flanke von E werden die Daten übernommen
        * Achtung bei Schreib/Lese Zugriff muss Pin Mode umgeschalten werden
        * Initialisierungssequenz: HD44780-Datenblatt S.45 mit Zeitangaben(!)
    - busy flag sollte abgefragt werden vor Schreibzugriff
    - Funktionen
        * configMyLCDport();
        * clockLCDenable();
        * initMyLCD();
        * sendCommandLCD(uint8_t u8Data);
        * writeDataLCD(uint8_t u8data);
        * setDDRAMAdressLCD(uint8_t u8adress)
        * uint8_t readBusyFlagAndAdressLCD();
        * void putsLCD(char *pData);
        * void putCharLCD(char c);
    - Anpassung an die Erfordernisse des kooperativen Multitasikings
        * Delays >10us durch State-Machine ersetzen
        * Ansatz: putc() schreibt ein Zeichen auf das LCD
        * isReadyLCD() fragt das Busy-Flag ab und gibt true zurück, wenn der nächste character geschrieben werden kann
    - Informationen
        * Datenblatt FDCC1602L-NSWBBW-91LE z.B. farnell
        * Datenblatt Dispaycontroller sparkfun, maritex
        * Beschreibungen wikipedia HD44780, sprut.de/electronic/lcd/index.htm
        * Simulator

11.6 Vorlesung 16.06.2017 - AD-Wandler

- Two Modules:
    - ADC1: Up to 32 analog input channels
    - ADC2: Up to 16 analog input channels

- Key Features
  - 10- or 12-bit ADC
  - Successive Approximation (SAR) Conversion (Dt.: Wägeverfahren)
    Kompromiss zw. Geschwindigkeit und Aufwand
  - Conversion Speed of up to 1,1Msps (10bit) or 500ksps(12Bit)
  - simultaneous sampling of up to 4 analog input pins (10bit)
- ADCx MODULE BLOCK DIAGRAM
  - 4 Sample and Hold -Glieder
  - Multiplexer
  - Referenzspannungen
    → Wir verwenden interne Referenzen
- AD-Umsetzer nach dem Wägeverfahren
  siehe Abb 17.41 Tietze, Schenk, Gamm
- Flash-Converter sind sehr schnell, jedoch sehr aufwändig zu realisieren
- EDA dsPICC33-Board
  - AN0, AN1
  - Potis mit SW500 zuschaltbar
  - Analoge Eingänge auf J501 herausgeführt
-
-
-
-

Anhang

```
void initMyLCD()
{
// 15mS delay after Vdd reaches nnVdc before proceeding with LCD initialization
// not always required and is based on system Vdd rise rate
uint16_t ui16I=0;
while(ui16I++<0xFFFF)Nop();

/* set initial states for the data and control pins */
DATA &= 0xFF00; //set RE0-RE7 low
RW = 0;                          // R/W state set low
RS = 0;                          // RS state set low
E = 0;                           // E state set low

/* set data and control pins to outputs */
TRISE &= 0xFF00;                 //set RE0-RE7 to output
RW_TRIS = 0;                     // RW pin set as output
RS_TRIS = 0;                     // RS pin set as output
E_TRIS = 0;                      // E pin set as output

/* 1st LCD initialization sequence */
DATA &= 0xFF00;                  //set RE0-RE7 low
DATA |= 0x0038;                  //set lcd type: 8-bit,2lines,5x7
clockLCDenable();                // toggle E signal
ui16I=0; while(ui16I++<23256)Nop(); //5ms Delay

// 2nd LCD initialization sequence
DATA &= 0xFF00;
DATA |= 0x0038;
clockLCDenable();          // toggle Enable signal
ui16I=0; while(ui16I++<4651)Nop(); //200us Delay

// 3rd LCD initialization sequence
DATA &= 0xFF00;
DATA |= 0x0038;
clockLCDenable();
ui16I=0; while(ui16I++<4651)Nop(); //200us Delay

sendCommandLCD( 0x38 );          // function set 8bit, 2lines
sendCommandLCD( 0x0C );          // Display on control, cursor blink off (0x0C), cursor off
sendCommandLCD( 0x06 );          // entry mode set (0x06), increment cursor, no shift
}
```

Listing 29: Initalisierungssequenz des LCD-Controllers

```
extern char ShadowString[32];

void initMyLCD();

void clockLCDenable();

void putcLCD(char c);

void putsLCD(char* pData);

void sendCommandLCD(uint8_t ui8data);

void sendCommandLCDNonBlocking(uint8_t ui8data);

void writeDataLCD(uint8_t ui8data);

void writeDataLCDNonBlocking(uint8_t ui8data);

void setDDRAMAddressLCD(uint8_t ui8address);

uint8_t readBusyFlagLCD();

void putncLCD(char* pData, uint8_t ui8n);

void SendDataToLCD();

void clearLCDStorage();

void SendDataToLCD();

void setLineLCD(const char* pStr, uint8_t ui8Line);

void setLCDLine1(const char* pString);

void setLCDLine2(const char* pString);
```

Listing 30: Funktionsprototypen edaPIC33LCD-Library

TABLE 4-55: PORTA REGISTER MAP FOR dsPIC33EPXXXMU810/814 AND PIC24EPXXXGU810/814 DEVICES ONLY

File Name	Addr.	Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	All Resets
TRISA	0E00	TRISA15	TRISA14	—	—	—	TRISA10	TRISA9	—	TRISA7	TRISA6	TRISA5	TRISA4	TRISA3	TRISA2	TRISA1	TRISA0	C6FF
PORTA	0E02	RA15	RA14	—	—	—	RA10	RA9	—	RA7	RA6	RA5	RA4	RA3	RA2	RA1	RA0	xxxx
LATA	0E04	LATA15	LATA14	—	—	—	LATA10	LATA9	—	LATA7	LATA6	LATA5	LATA4	LATA3	LATA2	LATA1	LATA0	xxxx
ODCA	0E06	ODCA15	ODCA14	—	—	—	—	—	—	—	—	ODCA5	ODCA4	ODCA3	ODCA2	ODCA1	ODCA0	0000
CNENA	0E08	CNIEA15	CNIEA14	—	—	—	CNIEA10	CNIEA9	—	CNIEA7	CNIEA6	CNIEA5	CNIEA4	CNIEA3	CNIEA2	CNIEA1	CNIEA0	0000
CNPUA	0E0A	CNPUA15	CNPUA14	—	—	—	CNPUA10	CNPUA9	—	CNPUA7	CNPUA6	CNPUA5	CNPUA4	CNPUA3	CNPUA2	CNPUA1	CNPUA0	0000
CNPDA	0E0C	CNPDA15	CNPDA14	—	—	—	CNPDA10	CNPDA9	—	CNPDA7	CNPDA6	CNPDA5	CNPDA4	CNPDA3	CNPDA2	CNPDA1	CNPDA0	0000
ANSELA	0E0E	—	—	—	—	—	ANSA10	ANSA9	—	ANSA7	ANSA6	—	—	—	—	—	—	06C0

Legend: x = unknown value on Reset, — = unimplemented, read as '0'. Reset values are shown in hexadecimal.

(a) Register A

TABLE 4-56: PORTB REGISTER MAP

File Name	Addr.	Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	All Resets
TRISB	0E10	TRISB15	TRISB14	TRISB13	TRISB12	TRISB11	TRISB10	TRISB9	TRISB8	TRISB7	TRISB6	TRISB5	TRISB4	TRISB3	TRISB2	TRISB1	TRISB0	FFFF
PORTB	0E12	RB15	RB14	RB13	RB12	RB11	RB10	RB9	RB8	RB7	RB6	RB5	RB4	RB3	RB2	RB1	RB0	xxxx
LATB	0E14	LATB15	LATB14	LATB13	LATB12	LATB11	LATB10	LATB9	LATB8	LATB7	LATB6	LATB5	LATB4	LATB3	LATB2	LATB1	LATB0	xxxx
ODCB	0E16	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	0000
CNENB	0E18	CNIEB15	CNIEB14	CNIEB13	CNIEB12	CNIEB11	CNIEB10	CNIEB9	CNIEB8	CNIEB7	CNIEB6	CNIEB5	CNIEB4	CNIEB3	CNIEB2	CNIEB1	CNIEB0	0000
CNPUB	0E1A	CNPUB15	CNPUB14	CNPUB13	CNPUB12	CNPUB11	CNPUB10	CNPUB9	CNPUB8	CNPUB7	CNPUB6	CNPUB5	CNPUB4	CNPUB3	CNPUB2	CNPUB1	CNPUB0	0000
CNPDB	0E1C	CNPDB15	CNPDB14	CNPDB13	CNPDB12	CNPDB11	CNPDB10	CNPDB9	CNPDB8	CNPDB7	CNPDB6	CNPDB5	CNPDB4	CNPDB3	CNPDB2	CNPDB1	CNPDB0	0000
ANSELB	0E1E	ANSB15	ANSB14	ANSB13	ANSB12	ANSB11	ANSB10	ANSB9	ANSB8	ANSB7	ANSB6	ANSB5	ANSB4	ANSB3	ANSB2	ANSB1	ANSB0	FFFF

Legend: x = unknown value on Reset, — = unimplemented, read as '0'. Reset values are shown in hexadecimal.

(b) Register B

TABLE 4-57: PORTC REGISTER MAP FOR dsPIC33EPXXXMU810/814 AND PIC24EPXXXGU810/814 DEVICES ONLY

File Name	Addr.	Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	All Resets
TRISC	0E20	TRISC15	TRISC14	TRISC13	TRISC12	—	—	—	—	—	—	—	TRISC4	TRISC3	TRISC2	TRISC1	—	F01E
PORTC	0E22	RC15	RC14	RC13	RC12	—	—	—	—	—	—	—	RC4	RC3	RC2	RC1	—	xxxx
LATC	0E24	LATC15	LATC14	LATC13	LATC12	—	—	—	—	—	—	—	LATC4	LATC3	LATC2	LATC1	—	xxxx
ODCC	0E26	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	0000
CNENC	0E28	CNIEC15	CNIEC14	CNIEC13	CNIEC12	—	—	—	—	—	—	—	CNIEC4	CNIEC3	CNIEC2	CNIEC1	—	0000
CNPUC	0E2A	CNPUC15	CNPUC14	CNPUC13	CNPUC12	—	—	—	—	—	—	—	CNPUC4	CNPUC3	CNPUC2	CNPUC1	—	0000
CNPDC	0E2C	CNPDC15	CNPDC14	CNPDC13	CNPDC12	—	—	—	—	—	—	—	CNPDC4	CNPDC3	CNPDC2	CNPDC1	—	0000
ANSELC	0E2E	—	ANSC14	ANSC13	—	—	—	—	—	—	—	—	ANSC4	ANSC3	ANSC2	ANSC1	—	601E

Legend: x = unknown value on Reset, — = unimplemented, read as '0'. Reset values are shown in hexadecimal.

(c) Register C

TABLE 4-59: PORTD REGISTER MAP FOR dsPIC33EPXXXMU810/814 AND PIC24EPXXXGU810/814 DEVICES ONLY

File Name	Addr.	Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	All Resets
TRISD	0E30	TRISD15	TRISD14	TRISD13	TRISD12	TRISD11	TRISD10	TRISD9	TRISD8	TRISD7	TRISD6	TRISD5	TRISD4	TRISD3	TRISD2	TRISD1	TRISD0	FFFF
PORTD	0E32	RD15	RD14	RD13	RD12	RD11	RD10	RD9	RD8	RD7	RD6	RD5	RD4	RD3	RD2	RD1	RD0	xxxx
LATD	0E34	LATD15	LATD14	LATD13	LATD12	LATD11	LATD10	LATD9	LATD8	LATD7	LATD6	LATD5	LATD4	LATD3	LATD2	LATD1	LATD0	xxxx
ODCD	0E36	ODCD15	ODCD14	ODCD13	ODCD12	ODCD11	ODCD10	ODCD9	ODCD8	—	—	ODCD5	ODCD4	ODCD3	ODCD2	ODCD1	ODCD0	0000
CNEND	0E38	CNIED15	CNIED14	CNIED13	CNIED12	CNIED11	CNIED10	CNIED9	CNIED8	CNIED7	CNIED6	CNIED5	CNIED4	CNIED3	CNIED2	CNIED1	CNIED0	0000
CNPUD	0E3A	CNPUD15	CNPUD14	CNPUD13	CNPUD12	CNPUD11	CNPUD10	CNPUD9	CNPUD8	CNPUD7	CNPUD6	CNPUD5	CNPUD4	CNPUD3	CNPUD2	CNPUD1	CNPUD0	0000
CNPDD	0E3C	CNPDD15	CNPDD14	CNPDD13	CNPDD12	CNPDD11	CNPDD10	CNPDD9	CNPDD8	CNPDD7	CNPDD6	CNPDD5	CNPDD4	CNPDD3	CNPDD2	CNPDD1	CNPDD0	0000
ANSELD	0E3E	—	—	—	—	—	—	—	—	ANSD7	ANSD6	—	—	—	—	—	—	00C0

Legend: x = unknown value on Reset, — = unimplemented, read as '0'. Reset values are shown in hexadecimal.

(d) Register D

TABLE 4-61: PORTE REGISTER MAP FOR dsPIC33EPXXXMU810/814 AND PIC24EPXXXGU810/814 DEVICES ONLY

File Name	Addr.	Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	All Resets
TRISE	0E40	—	—	—	—	—	—	TRISE9	TRISE8	TRISE7	TRISE6	TRISE5	TRISE4	TRISE3	TRISE2	TRISE1	TRISE0	03FF
PORTE	0E42	—	—	—	—	—	—	RE9	RE8	RE7	RE6	RE5	RE4	RE3	RE2	RE1	RE0	xxxx
LATE	0E44	—	—	—	—	—	—	LATE9	LATE8	LATE7	LATE6	LATE5	LATE4	LATE3	LATE2	LATE1	LATE0	xxxx
ODCE	0E46	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	0000
CNENE	0E48	—	—	—	—	—	—	CNIEE9	CNIEE8	CNIEE7	CNIEE6	CNIEE5	CNIEE4	CNIEE3	CNIEE2	CNIEE1	CNIEE0	0000
CNPUE	0E4A	—	—	—	—	—	—	CNPUE9	CNPUE8	CNPUE7	CNPUE6	CNPUE5	CNPUE4	CNPUE3	CNPUE2	CNPUE1	CNPUE0	0000
CNPDE	0E4C	—	—	—	—	—	—	CNPDE9	CNPDE8	CNPDE7	CNPDE6	CNPDE5	CNPDE4	CNPDE3	CNPDE2	CNPDE1	CNPDE0	0000
ANSELE	0E4E	—	—	—	—	—	—	ANSE9	ANSE8	ANSE7	ANSE6	ANSE5	ANSE4	ANSE3	ANSE2	ANSE1	ANSE0	03FF

Legend: x = unknown value on Reset, — = unimplemented, read as '0'. Reset values are shown in hexadecimal.

(e) Register E

TABLE 4-63: PORTF REGISTER MAP FOR dsPIC33EPXXXMU810/814 AND PIC24EPXXXGU810/814 DEVICES ONLY

File Name	Addr.	Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	All Resets
TRISF	0E50	—	—	TRISF13	TRISF12	—	—	—	TRISF8	—	—	TRISF5	TRISF4	TRISF3	TRISF2	TRISF1	TRISF0	313F
PORTF	0E52	—	—	RF13	RF12	—	—	—	RF8	—	—	RF5	RF4	RF3	RF2	RF1	RF0	xxxx
LATF	0E54	—	—	LATF13	LATF12	—	—	—	LATF8	—	—	LATF5	LATF4	LATF3	LATF2	LATF1	LATF0	xxxx
ODCF	0E56	—	—	ODCF13	ODCF12	—	—	—	ODCF8	—	—	ODCF5	ODCF4	ODCF3	ODCF2	ODCF1	ODCF0	0000
CNENF	0E58	—	—	CNIEF13	CNIEF12	—	—	—	CNIEF8	—	—	CNIEF5	CNIEF4	CNIEF3	CNIEF2	CNIEF1	CNIEF0	0000
CNPUF	0E5A	—	—	CNPUF13	CNPUF12	—	—	—	CNPUF8	—	—	CNPUF5	CNPUF4	CNPUF3	CNPUF2	CNPUF1	CNPUF0	0000
CNPDF	0E5C	—	—	CNPDF13	CNPDF12	—	—	—	CNPDF8	—	—	CNPDF5	CNPDF4	CNPDF3	CNPDF2	CNPDF1	CNPDF0	0000
ANSELF	0E5E	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	0000

Legend: x = unknown value on Reset, — = unimplemented, read as '0'. Reset values are shown in hexadecimal.

(f) Register F

TABLE 4-66: PORTG REGISTER MAP FOR dsPIC33EPXXXMU810/814 AND PIC24EPXXXGU810/814 DEVICES ONLY

File Name	Addr.	Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	All Resets
TRISG	0E60	TRISG15	TRISG14	TRISG13	TRISG12	—	—	TRISG9	TRISG8	TRISG7	TRISG6	—	—	—	—	TRISG1	TRISG0	F3C3
PORTG	0E62	RG15	RG14	RG13	RG12	—	—	RG9	RG8	RG7	RG6	—	—	RG3(1)	RG2(1)	RG1	RG0	xxxx
LATG	0E64	LATG15	LATG14	LATG13	LATG12	—	—	LATG9	LATG8	LATG7	LATG6	—	—	—	—	LATG1	LATG0	xxxx
ODCG	0E66	ODCG15	ODCG14	ODCG13	ODCG12	—	—	—	—	—	—	—	—	—	—	ODCG1	ODCG0	0000
CNENG	0E68	CNIEG15	CNIEG14	CNIEG13	CNIEG12	—	—	CNIEG9	CNIEG8	CNIEG7	CNIEG6	—	—	CNIEG3(1)	CNIEG2(1)	CNIEG1	CNIEG0	0000
CNPUG	0E6A	CNPUG15	CNPUG14	CNPUG13	CNPUG12	—	—	CNPUG9	CNPUG8	CNPUG7	CNPUG6	—	—	—	—	CNPUG1	CNPUG0	0000
CNPDG	0E6C	CNPDG15	CNPDG14	CNPDG13	CNPDG12	—	—	CNPDG9	CNPDG8	CNPDG7	CNPDG6	—	—	—	—	CNPDG1	CNPDG0	0000
ANSELG	0E6E	—	—	—	—	—	—	ANSG9	ANSG8	ANSG7	ANSG6	—	—	—	—	—	—	03C0

Legend: x = unknown value on Reset, — = unimplemented, read as '0'. Reset values are shown in hexadecimal.
Note 1: If RG2 and RG3 are used as general purpose inputs, the VUSB3V3 pin must be connected to VDD.

(g) Register G

Abbildung 13: PORTA-G Register Map

11.0 I/O PORTS

Note 1: This data sheet summarizes the features of the dsPIC33EPXXX(GP/MC/MU)806/810/814 and PIC24EPXXX(GP/GU)810/814 families of devices. It is not intended to be a comprehensive reference source. To complement the information in this data sheet, refer to **Section 10. "I/O Ports"** (DS70598) of the "*dsPIC33E/PIC24E Family Reference Manual*", which is available from the Microchip web site (www.microchip.com).

2: Some registers and associated bits described in this section may not be available on all devices. Refer to **Section 4.0 "Memory Organization"** in this data sheet for device-specific register and bit information.

All of the device pins (except VDD, VSS, $\overline{\text{MCLR}}$ and OSC1/CLKI) are shared among the peripherals and the parallel I/O ports. All I/O input ports feature Schmitt Trigger inputs for improved noise immunity.

11.1 Parallel I/O (PIO) Ports

Generally, a parallel I/O port that shares a pin with a peripheral is subservient to the peripheral. The peripheral's output buffer data and control signals are provided to a pair of multiplexers. The multiplexers select whether the peripheral or the associated port has ownership of the output data and control signals of the I/O pin. The logic also prevents "loop through," in which a port's digital output can drive the input of a peripheral that shares the same pin. Figure 11-1 illustrates how ports are shared with other peripherals and the associated I/O pin to which they are connected.

When a peripheral is enabled and the peripheral is actively driving an associated pin, the use of the pin as a general purpose output pin is disabled. The I/O pin can be read, but the output driver for the parallel port bit is disabled. If a peripheral is enabled, but the peripheral is not actively driving a pin, that pin can be driven by a port.

All port pins have eight registers directly associated with their operation as digital I/O. The Data Direction register (TRISx) determines whether the pin is an input or an output. If the data direction bit is a '1', then the pin is an input. All port pins are defined as inputs after a Reset. Reads from the latch (LATx) read the latch. Writes to the latch write the latch. Reads from the port (PORTx) read the port pins, while writes to the port pins write the latch.

Any bit and its associated data and control registers that are not valid for a particular device is disabled. This means the corresponding LATx and TRISx registers and the port pin are read as zeros.

When a pin is shared with another peripheral or function that is defined as an input only, it is nevertheless regarded as a dedicated port because there is no other competing source of outputs.

Abbildung 14: I/O Ports -Datenblatt S.207

FIGURE 11-1: BLOCK DIAGRAM OF A TYPICAL SHARED PORT STRUCTURE

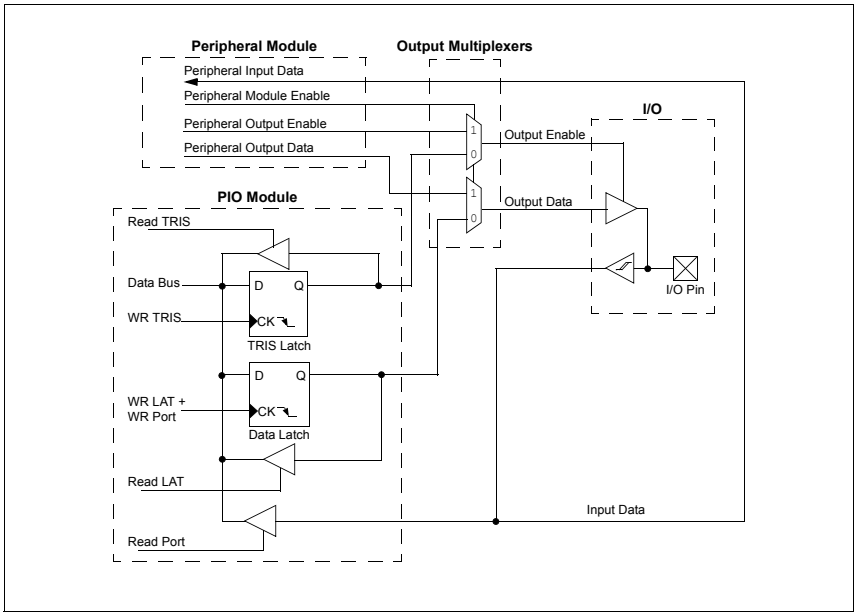


Abbildung 15: Block Diagramm I/O Ports -Datenblatt S.208

11.1.1 OPEN-DRAIN CONFIGURATION

In addition to the PORT, LAT and TRIS registers for data control, some port pins can also be individually configured for either digital or open-drain output. This is controlled by the Open-Drain Control register, ODCx, associated with each port. Setting any of the bits configures the corresponding pin to act as an open-drain output.

The open-drain feature allows the generation of outputs higher than VDD (e.g., 5V on a 5V tolerant pin) by using external pull-up resistors. The maximum open-drain voltage allowed is the same as the maximum VIH specification for that pin.

See the **"Pin Diagrams"** section for the available pins and their functionality.

11.2 Configuring Analog and Digital Port Pins

The ANSELx register controls the operation of the analog port pins. The port pins that are to function as analog inputs or outputs must have their corresponding ANSELx and TRISx bits set. In order to use port pins for I/O functionality with digital modules, such as Timers, UARTs, etc., the corresponding ANSELx bit must be cleared.

The ANSELx register has a default value of 0xFFFF; therefore, all pins that share analog functions are analog (not digital) by default. Refer to the Pinout I/O Descriptions (Table 1-1 in **Section 1.0 "Device Overview"**) for the complete list of analog pins.

If the TRISx bit is cleared (output) while the ANSELx bit is set, the digital output level (VOH or VOL) is converted by an analog peripheral, such as the ADC module or Comparator module.

When the PORT register is read, all pins configured as analog input channels are read as cleared (a low level).

Pins configured as digital inputs do not convert an analog input. Analog levels on any pin defined as a digital input (including the pins defined as Analog in Table 1-1 in **Section 1.0 "Device Overview"**) can cause the input buffer to consume current that exceeds the device specifications.

11.2.1 I/O PORT WRITE/READ TIMING

One instruction cycle is required between a port direction change or port write operation and a read operation of the same port. Typically this instruction would be an NOP, as shown in Example 11-1.

11.3 Input Change Notification

The input change notification function of the I/O ports allows the dsPIC33EPXXX(GP/MC/MU)806/810/814 and PIC24EPXXX(GP/GU)810/814 devices to generate interrupt requests to the processor in response to a Change-of-State (COS) on selected input pins. This feature can detect input Change-of-States even in Sleep mode, when the clocks are disabled. Every I/O port pin can be selected (enabled) for generating an interrupt request on a Change-of-State.

Three control registers are associated with the CN functionality of each I/O port. The CNENx registers contain the CN interrupt enable control bits for each of the input pins. Setting any of these bits enables a CN interrupt for the corresponding pins.

Each I/O pin also has a weak pull-up and a weak pull-down connected to it. The pull-ups act as a current source or sink source connected to the pin, and eliminate the need for external resistors when push-button or keypad devices are connected. The pull-ups and pull-downs are enabled separately using the CNPUx and the CNPDx registers, which contain the control bits for each of the pins. Setting any of the control bits enables the weak pull-ups and/or pull-downs for the corresponding pins.

Note:	Pull-ups and pull-downs on change notification pins should always be disabled when the port pin is configured as a digital output.

EXAMPLE 11-1: PORT WRITE/READ EXAMPLE

```
MOV   0xFF00, W0    ; Configure PORTB<15:8>
                    ; as inputs
MOV   W0, TRISB     ; and PORTB<7:0>
                    ; as outputs
NOP                 ; Delay 1 cycle
BTSS  PORTB, #13    ; Next Instruction
```


Abbildung 16: I/O Ports -Datenblatt S.209

15. INITIALIZATION

15.1 8-bit interface mode (Condition: fosc = 270KHZ)

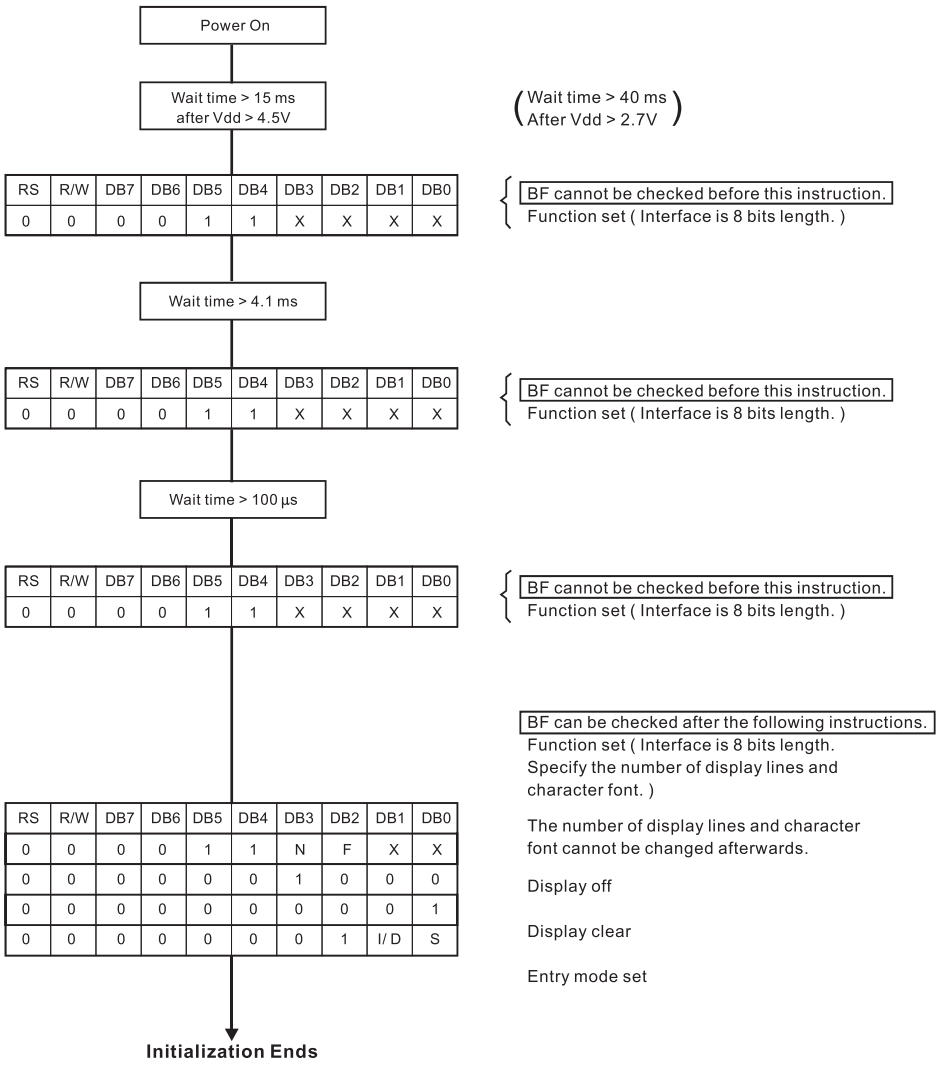


Abbildung 17: LCD Controller Initalization Sequenz