

Einstieg in die Programmierung von PIC18Fxxxx Mikrocontrollern.

ASSEMBLER & „C“

UNDER CONSTRUCTION

mit



MPLAB - X

Vorbemerkungen

Dieses Dokument ist im Moment noch in Bearbeitung und wird möglicherweise niemals „fertig“ sein. Die jeweils aktuellste Version sollte auf [GitHub](#) zu finden sein.

Ziel dieses Dokumentes ist es, eine einigermaßen leicht verständliche Anleitung für Studierende zur Verfügung zu stellen. Diese sollten sich damit „selbstständig“ in die Programmierung von PIC18 Controllern einarbeiten können.

Die in diesem Dokument beschriebenen Vorgehensweisen funktionierten zum Zeitpunkt der Erstellung mit einem bestimmten Equipment:

(MPLAB X ab 4.xy; C18 v3.4x oder XC8 Compiler ab v1.4x, uC-Quick Platine v2013/18, PIC18F2xK22, PICKIT3).

Erfahrungsgemäß ist das verwendete Equipment einem rasend schnellen Wandel unterworfen. Deshalb ist wichtig zu lernen wie man sich die nötigen Informationen beschafft um etwas tun zu können, und nicht nur zu versuchen exakt die vorgegebenen Schritte nachzuvollziehen die hier beschrieben sind. Das einfache „Nachmachen“ der hier vorgestellten Beispiele führt unter Umständen eher zur Frustration als zum Erfolg.

Achtung: Noch vorhandene Links auf die Hochschulseite funktionieren nicht mehr. Die Seiten der HS Mitarbeiter sind mit der Neustrukturierung der Homepage weggefallen. Auch alle Übungsbeispiele und Demoprogramme werden nach [GitHub](#) transferiert.

Warum Mikrochip PIC18 ?

Die aktuelle Verwendung eines 8-Bit PIC Controllers im Mikrocontroller Labor, des Instituts für Mechatronik und Medizintechnik an der Hochschule Ulm, ist eher zufällig bzw. historisch bedingt. Es hätte genau so gut auch ein Controller eines anderen Herstellers mit einer anderen Architektur verwendet werden können.

Durch die Erfahrung mehrerer Labor-Ingenieure, sowie vieler vorhandener Beispiele und Demo-Boards sind die 8-Bit Controller des Herstellers "Microchip Technology Inc." in der Fakultät Mechatronik und Medizintechnik einfach die am besten unterstützten Mikrocontroller.

Die Entwicklungsumgebung **MPLAB X** wird in vielen Pools über den Software-Shop Service verfügbar sein. Für Studenten stehen in der Bibliothek des Campus Albert-Einstein-Allee **µC-Quick Sets** bereit, die eine Übungsplatine und/oder ein Programmiertool enthalten.

Bei Bedarf kann ein eigenes Board erworben und mit den benötigten Bauteilen bestückt werden. Zudem sind in den Laboren viele weitere Demo-Boards und PICs (*auch 16 und 32 Bit*) vorhanden.

Die Auswahl des für ein bestimmtes Projekt optimal geeigneten PICs ist aufgrund der Vielzahl der angebotenen Varianten für einen Neueinsteiger bedingt durch die Unmenge an Varianten sehr schwierig und soll in diesem Dokument vorerst nicht behandelt werden.

Alle Einstiegsbeispiele werden anhand des im µC Labor verwendeten PIC18F2xK22 bearbeitet. Wurde die grundsätzliche Vorgehensweise verstanden, sollte die Übertragung auf andere PICs keine Probleme bereiten.

Hervorhebungen

kursiv: Zusätzliche Informationen werden oft kursiv dargestellt

rot: Wichtige, lesenswerte Informationen!

fett/kursiv: Hierbei handelt es sich meistens um Menüpunkte einer Software

grün/kursiv: Für interessierte Leser ;-)

Bla, bla ... (...): → muss ggf. noch ergänzt werden...

[13.2 C Templates](#) Verknüpfungen innerhalb dieses Dokumentes (*auch ohne Kapitelnummern*)

[Fundamentals of the C Programming Language \(www\)](#) Internetverknüpfungen. (*Oft nur Imgfu Verknüpfungen, damit es auch noch einigermaßen funktioniert, wenn sich Adressen ändern...*)

```
void main(void) {           blau, eingerahmt → Source-code  
(Fast alle auch in uCO\_THU\_Base.zip enthalten)
```

Anregungen und Verbesserungsvorschläge:

Vorschläge, wie diese Anleitung erweitert und verbessert werden könnte, sind höchst willkommen. Am besten entsprechend dem hier vorliegenden Layout ausarbeiten und per e-mail an:

Volker.Schilling-Kaestle@thu.de

Table of Contents

1 Die Basics (Überblick PIC18F2xK22).....	13
1.1 Informationsquellen.....	13
1.1.1 Data Sheet.....	13
1.1.2 „User Guide“ des Assemblers / Compilers / Linkers.....	13
1.1.3 User Guide / Hilfe der IDE.....	13
1.1.4 User Guide der Programmier- / Debug- Tools.....	13
1.1.5 Sonstige Informationsquellen (Tutorials, Wiki, Tutorials ...).....	13
1.2 Übersicht über den PIC18F2xK22 Mikrocontroller.....	14
1.2.1 µC Architektur.....	15
1.2.2 PIC18 Kernmodule.....	16
1.2.2.1 CPU (central processing unit) und ALU (arithmetic logic unit).....	16
1.2.2.1.1 8 x 8 Multiplier.....	16
1.2.2.1.2 Status Register.....	16
1.2.2.2 Oszillatormodul / Arbeitstakt.....	17
1.2.2.2.1 Externer Oszillator.....	17
1.2.2.2.2 Oszillator mit externen frequenzbestimmenden Bauteilen.....	17
1.2.2.2.3 Interner Oszillator.....	17
1.2.2.3 Programmspeicher (Flash).....	18
1.2.2.3.1 Program-Counter.....	18
1.2.2.3.2 Stack (Hardware Stack).....	18
1.2.2.3.3 Reset-Vektor (0000h).....	18
1.2.2.3.4 Interrupt Vektoren (0008h, 0018h).....	18
1.2.2.3.5 On-Chip Program Memory (0019h...).....	18
1.2.2.4 Datenspeicher (SRAM).....	19
1.2.2.4.1 Data Memory Banks.....	19
1.2.2.4.2 Access BANK.....	19
1.2.2.4.3 General-Purpose-Registers → GPR.....	19
1.2.2.5 Daten EEPROM.....	19
1.2.3 PIC18 Peripheriemodule.....	20
1.2.3.1 IO Ports.....	20
1.2.3.2 IO Pins.....	20
1.2.3.3 Timer / Counter.....	21
1.2.3.4 Capture/Compare/PWM Modul (CCP).....	22
1.2.3.5 Serielle Kommunikation.....	23
1.2.3.5.1 EUSART: <i>Enhanced Universal Synchronous Asynchronous Receiver Transmitter</i>	23
1.2.3.5.2 SPI: <i>Serial Peripheral Interface</i>	23
1.2.3.5.3 I ² C: <i>Inter-Integrated Circuit Bus</i>	23
1.2.3.6 10 Bit Analog-to-Digital-Converter (ADC).....	24
1.2.3.7 Comparator.....	24
1.2.3.8 Digital Analog Converter (DAC).....	24
1.2.3.9 Fixed Voltage Reference (FVR).....	24
1.2.3.10 Charge Time Measurement Unit (CTMU).....	25
1.2.3.11 Watchdog Timer (WDT).....	25
1.2.3.12 Programmier- und Debug-Schnittstelle (ICSP/ICD).....	25
1.2.4 Die Configuration Bits.....	26
1.2.5 Interrupt-System.....	27
1.2.5.1 Interrupt 0, INT0 (einfachstes Beispiel).....	27
1.2.5.2 Priority Mode vs. Compatibility Mode.....	28
1.2.5.2.1 Priority Mode.....	28

1.2.5.2.2 Compatibility Mode.....	28
1.2.6 Befehlssatz.....	29
1.2.6.1 Byte-orientierte Befehle.....	30
1.2.6.2 Bit-orientierte Befehle.....	30
1.2.6.3 Bedingte Sprünge / Verzweigungen.....	30
1.2.6.4 Kontroll Operationen ohne Bedingung.....	31
1.2.6.5 Literal Befehle (Konstanten als Operatoren).....	31
1.2.6.6 Datenaustausch zwischen Programm und Datenspeicherbereich.....	31
1.3 Integrierte Entwicklungsumgebung MPLAB-X.....	32
1.3.1 Assembler.....	32
1.3.2 C Compiler.....	32
1.3.2.1 <i>Mögliche C Compiler für PIC18 Controller</i>	32
1.3.2.1.1 XC8.....	32
1.3.2.1.2 Microchip C18.....	32
1.3.2.1.3 SDCC.....	32
1.3.3 Linker.....	33
1.3.4 Installation der Entwicklungsumgebung.....	33
1.3.4.1 Installationshinweise.....	33
1.3.4.1.1 Installationshinweis Windows.....	33
2 Grundkonfiguration neuer Projekte.....	34
2.1 Neues Projekt anlegen.....	34
2.1.1 Select Device / Tool.....	34
2.1.2 Select Header (nicht mehr automatisch angezeigt ab MPLAB X v3.x).....	35
2.1.3 Select Compiler.....	35
2.1.4 Projektnamen und Speicherort vergeben.....	36
2.2 Erster Überblick über IDE.....	37
2.3 Grundgerüst eines µC Programms in Assembler.....	38
2.3.1 Assembler File von der IDE erstellen lassen.....	38
2.3.2 Kommentare (; , //, /* */).....	39
2.3.3 Assembler Direktiven.....	39
2.3.3.1 #include Direktive.....	39
2.3.3.1.1 Die Funktion von Include-Dateien.....	39
2.3.3.1.2 Include Guards.....	40
2.3.3.2 Psect Direktive.....	40
2.3.3.2.1 Benutzer definierte Sections.....	41
2.3.3.2.2 Psect Flags.....	41
2.3.3.2.3 Vordefinierte Sections.....	41
2.3.3.3 End Direktive.....	42
2.3.3.4 Variablen.....	43
2.3.4 Assembler Direktiven.....	43
2.3.4.1 list p= (list – LISTING OPTIONS).....	43
2.3.5 Aufsplittung des Assembler Templates in mehrere Dateien.....	43
2.3.5.1 Configuration Template.....	43
2.3.5.2 Main Template.....	43
2.3.6 Aufteilung der Hauptschleife in einmalige Initialisierung und Main-Loop.....	44
2.3.7 CONFIG Direktive.....	44
2.3.7.1.1 code.....	44
2.3.7.1.2 res.....	45
2.4 Grundgerüst eines C Programms.....	46
2.4.1 Runtime Startup Code.....	46
2.4.2 Erste Sourcecode Datei mit der Funktion main().....	46
2.4.2.1 void main(void) { return; } ???.....	47

2.4.3 Kommentare in C „//“ und /* ... */.....	47
2.4.4 Die benötigten Include-Dateien.....	47
2.4.4.1 C-Namen von Registern und Bits des Mikrocontrollers.....	48
2.4.5 Funktionen (am Beispiel <code>__init()</code>).....	48
2.4.6 Ein einfaches C Programm.....	48
2.4.7 Variablen in C.....	49
2.4.8 Prototypen von Funktionen.....	49
2.4.9 Dateiabschluss mit „newline“.....	49
2.5 Vorhandene Dateien / Templates hinzufügen.....	50
2.5.1 Dokumentation hinzufügen.....	50
2.6 Configuration Bits (Assembler und C).....	51
2.7 Oszillator Konfiguration (Assembler und C).....	52
2.7.1 Oszillator Kontroll-Register – OSCON / OSCON2 / OSCTUNE.....	53
2.7.2 Configuration Bits FOSC<3:0>.....	54
2.7.3 Oszillator Einstellungen im Quelltext eintragen.....	54
2.8 Basisinitialisierung aller Pins, Module und Variablen.....	55
2.8.1 Konfiguration für alle zu verwendenden Pins ermitteln.....	55
2.8.2 Peripheriemodule initialisieren.....	55
2.9 Text Substitution Labels.....	55
3 Digitale Ein- und Ausgänge (<i>Hello World</i>).....	56
3.1 „Hello World“ für µC.....	56
3.1.1 Neues C Projekt.....	56
3.1.2 Pins für Taster und LED bestimmen.....	57
3.1.3 Config Settings überprüfen (einstellen).....	57
3.1.4 Pins initialisieren.....	58
3.1.4.1 Anlegen von Text Substitution Labels für LED und Taster.....	59
3.1.4.1.1 Text Substitution Labels für C-Projekt.....	59
3.1.4.2 Initialisierungscode einfügen.....	59
3.1.5 Taster abfragen und LED einschalten.....	60
3.1.6 Assemblieren / Compilieren des Programms.....	61
3.1.6.1 Build Configuration / Release oder Debug ?.....	61
3.1.6.2 Fehlermeldungen.....	61
3.1.7 Der erzeugte Maschinencode (<i>XC8 Compiler</i>).....	62
3.1.7.1 Disassembly Listing File.....	62
3.1.8 Erstes Testen / Debuggen des Programms.....	63
3.1.8.1 ICD System.....	63
3.1.8.2 Einrichten des Debuggers (Power).....	63
3.1.8.3 Programmieren der Hardware mit dem Debug-Code.....	64
3.1.8.4 Ausführen, Anhalten und Reset des Programms.....	65
3.1.8.5 Anzeigen von Registern.....	66
3.1.8.6 Program Breakpoint.....	67
3.1.8.7 Data Breakpoint.....	67
3.1.8.8 Breakpoint Pass-Count.....	67
3.1.9 Programmieren der endgültigen Software.....	68
3.2 Hello World II / LEDs umschalten.....	69
3.3 Pins „gleichzeitig“ als Ein- und Ausgang benutzen.....	70
3.4 Hello World III / IPO Pattern (<i>input-process-output</i>).....	71
3.5 Eigenschaften von I/O Pins.....	73
3.5.1 Maximale Ausgangsströme der I/O Pins (25mA?).....	73
3.5.2 Logic Level von TLL und Schmitt-Trigger Eingängen.....	74
3.5.3 Interne WEAK PULL-UPS.....	75
3.6 I/O Übung 1: Taster lassen LED-Anzeige rotieren.....	75

4 Zeitgesteuerte Abläufe.....	76
4.1 Blink Variante 1 – Zählen (delay).....	76
4.2 Blink 1 in Assembler Sprache.....	79
4.3 Vertiefung: Zufallsgenerator / Reaktionstest (Prof. Groß ...)	80
4.3.1 LED-Anzeige und Taste.....	80
4.3.2 Der „Pseudo“-Zufallsgenerator.....	80
4.3.2.1 Rotieren.....	81
4.3.2.2 XOR Verarbeitung.....	81
4.3.3 Zeitsteuerung und Tastenabfrage.....	82
4.3.4 Kontrolle der Reaktion.....	82
4.3.4.1 Maskierung.....	82
4.3.4.2 Mustererkennung.....	82
4.3.4.3 Kontrolle.....	83
4.4 Blink Variante 2 – Timer Flags.....	84
4.4.1 Die „ungenaue“ Methode.....	85
4.4.1.1 Übung: Blink Variante 2 in C.....	85
4.4.2 Die exakte Methode (nur für Freaks).....	86
4.5 Übung: Hello World IV, IPO mit Time-Slot.....	86
5 Interrupts (siehe auch 1.2.4 Interrupt-System).....	87
5.1 Timer_0 Interrupt (Blink Variante 3).....	87
5.1.1 Timer_0 Interrupt Initialisierung.....	88
5.1.2 Interrupt Programmcode.....	88
5.1.2.1 Interrupt Vektor Adressen.....	88
5.1.2.2 Context Saving / Restore.....	89
5.1.2.3 Bestimmung der IR-Quelle und Löschen des IR-Flags.....	90
5.1.3 Das Hauptprogramm main() der Blink Variante 3.....	91
5.2 Interrupt mit Special-Event-Trigger (CCP + Timer).....	92
5.2.1 Capture Compare Modul mit Special-Event-Trigger.....	92
5.2.2 Timer 1.....	93
5.2.3 C-Code Compare mode __init() und main().....	94
5.2.4 CCP Interrupt Initialisierung.....	94
5.2.4.1 IR-Quelle bestimmen, IR-Flag löschen und Code einarbeiten.....	96
5.2.5 Übung: Blink Variante 4 mit CCP_2 & Timer_3.....	96
5.3 Pin Interrupts INTx (Drehgeber rotiert LEDs).....	97
5.3.1 Drehgeber (Inkrementalgeber).....	97
5.3.2 INTx Interrupt.....	98
5.3.3 Flags - Kommunikation zwischen Interrupt und main().....	99
5.3.3.1 Variablen (Flags) in mehreren Source Dateien verwenden.....	99
5.3.4 Setzen der Flags im Interrupt.....	100
5.3.5 Abfrage der Flags in main().....	100
5.3.6 Mögliche Probleme bei der Encoder Auswertung über Pin Interrupts.....	100
6 Bibliotheken, Funktionen, Plib/MCC.....	101
6.1 Überblick.....	101
6.1.1 System-Bibliotheken.....	101
6.1.1.1 Start-Up Code.....	101
6.1.1.2 Prozessor unabhängige Bibliotheken.....	101
6.1.1.3 Prozessor-spezifische Bibliotheken.....	101
6.1.1.4 MCC - MPLAB Code Configurator.....	101
6.1.2 Eigene (nicht zum System gehörende) Bibliotheken.....	101
6.2 Eine eigene Sourcecode-Bibliothek erstellen / verwenden.....	102
6.3 Sourcecode und Header Dateien.....	104
6.4 Compiler Include Directories der IDE.....	105

6.5 LCD Bibliothek (Character Displays).....	106
6.5.1 Die XLCD Bibliothek der Microchip Compiler.....	106
6.5.2 LCD_lib_busy der Hochschule Ulm.....	106
6.5.2.1 lcd_config.h.....	107
6.5.2.2 LCD_lib_busy (.h / .c).....	108
6.5.3 LCD_lib_busy Beispiel.....	109
6.6 GLCD Bibliothek (Graphik Display).....	110
6.6.1 GLCD Zeichensätze.....	110
6.6.2 GLCDnokia (.h / .c).....	111
6.6.3 glcdNokia_config.h.....	111
6.7 General Software Library.....	112
6.7.1 Delay Funktionen.....	112
6.7.2 Debug Funktionen.....	112
6.7.3 Reset Funktionen.....	112
6.7.4 Datentyp Umwandlungen.....	112
6.7.5 Speicher und Zeichenketten Verarbeitung.....	112
6.7.6 Ausgabe Funktionen für Zeichenketten.....	112
6.7.7 Funktionen zur Zeichen Klassifizierung.....	112
6.8 Peripheral Library (PLIB).....	112
6.9 Software Peripheral Library.....	112
6.10 Microchip Code Creator Beispiel.....	112
7 Puls-Weiten-Modulation (<i>fast analog</i>).....	113
7.1 LED - Ein/Aus oder doch unterschiedlich hell ?.....	113
7.1.1 PWM durch manuelles Zählen und Vergleichen.....	113
7.1.2 PWM mit CCP-Modul im PWM Modus (10 Bit).....	114
7.1.3 PWM mit CCP-Modul im Compare Modus (16 Bit).....	116
7.2 Ton am Signalgeber generieren über PWM.....	117
7.3 Mehrere PWM Signale über einen Timer (special Event).....	118
7.4 PWM Vertiefungsübungen.....	118
7.4.1 PWM Übung 1: Helligkeit von LED_4 einstellen (Port B5).....	118
7.4.2 PWM Übung 2: Tonhöhe (Periode) und Lautstärke (Pulsdauer).....	118
7.4.3 PWM Übung 3: Automatischer Durchlauf.....	118
7.4.4 PWM Übung 4: Vier Signale an den vier LEDs (ein Timer).....	118
8 Analoge Signale.....	119
8.1 Analoge Signale erfassen (ADC).....	119
8.1.1 10-Bit ADC im PIC18FxxK22.....	119
8.1.1.1 ADC Channel Auswahl – Start einer Messung.....	119
8.1.1.2 ADC Referenz-Spannungen.....	119
8.1.1.3 ADC Result und Format.....	119
8.1.2 SAR Wandler Prinzip (Sukzessive Approximation Register).....	120
8.1.2.1 Analog Digital Conversion Clock.....	120
8.1.3 Acquisition Requirements / Automatic Acquisition Time.....	121
8.1.4 ADC Initialisierung und Einfache Messung.....	122
8.1.5 Timer -> ADC (Special-Event-Trigger 2).....	123
8.1.6 ADC Vertiefungsübungen.....	124
8.1.6.1 ADC Übung 1: Poti → PWM Sound Frequenz.....	124
8.1.6.2 ADC Übung 2: Zweiter ADC Channel → Lautstärke.....	124
8.2 Analoge Signale ausgeben (DAC).....	125
8.2.1 DAC Initialisierung und einfache Ausgabe.....	125
8.2.2 Ausgabe eines Sägezahnsignals.....	126
8.2.3 Ausgabe einer beliebigen Signalform.....	126
8.2.4 Erzeugung über PWM und RC-Filter.....	126

8.3 CTMU ???????.....	126
8.3.1 Kapazitiver Schalter.....	126
9 Zeiten und Frequenzen messen.....	127
9.1 Periodendauer eines Signals / Zeit Puls zu Puls.....	127
9.1.1 Übung Capture 1: Automatische Bereichsumschaltung.....	128
9.1.2 Übung Capture 2: Keine falschen Werte bei der ersten detektierten Flanke.....	128
9.2 Pulsdauer-Messungen mit dem Capture Modul (Ultraschall Entfernungsmesser SRF04 / SRF05).....	129
9.2.1 Übung Capture 3: Sensor Defekt Erkennung.....	131
9.2.2 Übung Capture 4: Sicherheitsabstand.....	131
10 Kommunikation.....	132
10.1 RS232 / COM-Port / UART (auch virtuell über USB oder BT).....	132
10.1.1 Die wichtigsten Grundlagen (tiefere Einblicke -> Wikipedia).....	132
10.1.2 Hardwarevoraussetzungen.....	133
10.1.3 EUSART Modul Grundkonfiguration.....	133
10.1.3.1 EUSART Register und Pins.....	133
10.1.3.2 Die Baud Rate.....	134
10.1.3.3 USART initialisieren.....	135
10.1.4 Senden von Daten.....	137
10.1.5 Empfangen von Daten / Befehlen.....	139
10.1.6 MiniRS232 (bidirektional).....	140
10.1.7 Erweiterte Funktionen des USART Moduls.....	141
10.1.7.1 Clock/Transmit Polarity.....	141
10.1.7.2 Data/Receive Polarity.....	141
10.1.7.3 Auto-Baud Detect mode.....	141
10.1.7.4 Wake-up Enable.....	141
10.1.7.5 Send Break.....	141
10.1.7.6 Address Detect.....	141
10.1.7.7 Clock Source Select.....	141
10.1.8 Fehlerbehandlung (Overflow und Framing ERRORs).....	142
10.1.8.1 Overflow Error - OERR.....	142
10.1.8.2 Framing Error FERR.....	142
10.1.9 Receiver Interrupt.....	143
10.1.10 Receiver Buffer.....	144
10.1.11 Transmitter Interrupt und Buffer.....	145
10.1.12 Kommunikation mit zusätzlichem Übertragungsprotokollen.....	146
10.1.12.1 Consistent Overhead Byte Stuffing (COBS).....	146
10.1.12.1.1 COBS Beispielcode.....	147
10.1.12.2 Checksumme (Prüfung der Fehlerfreiheit eines Datenpaketes).....	148
10.1.12.3 Maßnahmen zur Fehlerbehandlung.....	148
10.1.13 Kommunikation mit einem GPS Modul.....	148
10.2 SPI.....	149
10.2.1 Demoprojekte SPI.....	149
10.3 I2C.....	149
10.3.1 Demoprojekte I2C.....	149
10.4 1-Wire.....	149
10.4.1 Demoprojekt DS18x20 1-Wire Temperatur-Sensor.....	150
10.5 Wire Transmission Channel RGB WS2812 und APA102.....	151
10.5.1 RGB WS2812 (1 Wire; nur Daten).....	151
10.5.1.1 WS2812 Timing.....	152
10.5.1.2 Vorsicht vor Compiler Optimierungen.....	153
10.5.1.3 WS2812 Demo Projekt.....	153

10.5.2 RGB APA102 (2 Wire; Daten und Clock).....	154
10.6 USB.....	155
10.6.1 Basis CDC Projekt.....	155
10.6.2 Basis HID Projekt.....	155
10.7 CAN (CANopen).....	156
10.8 Kommunikation mit speziellen Sensoren.....	156
10.8.1 Ultraschalllsensor SR04 /SR05.....	156
10.8.2 DCF77.....	156
11 Verschiedene Grundtechniken.....	157
11.1 IPO (EVA) Pattern / Input-Process-Output.....	157
11.2 Time Slots.....	157
11.3 State Machines – Endliche Automaten.....	158
11.3.1 Einfache ON/OFF oder START/STOP State Machine.....	158
11.3.2 Einfache State Machine mit IPO Modell.....	160
11.3.3 Komplexere (reale) Zustandsautomaten.....	161
11.3.4 Zustandsmaschine mit Funktionszeigern.....	163
11.3.5 Übung Menü → Uhr mit Einstelfunktion.....	164
11.4 State Machine zur Auswertung von Drehgebern.....	167
11.4.1 Ermittlung der erforderlichen Abtastrate.....	167
11.4.2 Signalauswertung und Fehlererkennung.....	168
11.4.2.1 Quadratursignale - Gray-Code.....	168
11.4.2.2 Fehlererkennung und Richtungsauswertung beim Graycode.....	168
11.4.2.3 Verschiedene Encodertypen.....	169
11.4.3 Beispielcode.....	169
11.4.3.1 States.....	169
11.4.3.2 Interrupt Code.....	169
11.4.3.3 Hauptprogramm.....	170
11.4.4 Entprellung per Software.....	170
11.4.5 Bisher vernachlässigt	170
11.5 Speicherkarten.....	170
11.6	170
11.7 Bootloader (MCHP AN1310).....	171
11.7.1 Bootloader Firmware.....	171
11.7.1.1 Verwendeter Speicherbereich.....	171
11.7.1.2 Wie kommt man in den Bootloader-Modus.....	171
11.7.2 Application Firmware.....	172
11.7.2.1 Configuration Bits.....	172
11.7.2.2 Reservierung des Bootloader Speicherbereichs.....	172
11.7.2.3 Erzeugung des Hex-Files der Application.....	172
11.7.3 Bootloader GUI.....	172
11.7.4 Bootloader konfigurieren für neue PICs.....	173
12 Weiterführende Informationen zu.....	174
12.1 Empfehlenswerte Microchip Hardware / Einsteiger Kits.....	174
12.1.1 MPLAB Snap (PICkit 4 light).....	174
12.1.2 PICkit 4.....	174
12.1.3 Curiosity Development Boards.....	174
12.1.4 Xplained Boards.....	174
12.1.5 MPLAB Xpress Boards.....	174
12.1.6 PICkit 3.....	174
12.1.6.1 PICKIT 3 Starter Kit.....	174
12.1.6.2 PICkit 3 Debug Express.....	174
12.2 Debuggen (Hardware).....	175

12.2.1 Wie funktioniert ICD ??? (programmieren / debuggen).....	175
12.2.2 Debuggen FAQ – was tun?.....	175
12.2.3 TRAP - undokumentierter Programmhaltepunkt.....	176
12.2.4 __builtin_software_breakpoint(void).....	176
12.2.5 __debug_break(void).....	176
12.2.6 __conditional_software_breakpoint(expression).....	176
12.2.7 Daten über den Debugger editieren.....	176
12.3 Simulator MPSIM.....	177
12.3.1 Genaue Zeitmessungen mit dem Simulator MPSIM.....	177
12.4 Alternative 8-Bit PICs.....	178
12.4.1 Standard Anwendungen.....	178
12.4.2 CAN.....	178
12.4.3 USB.....	178
12.5 PIC Hardware.....	178
12.5.1 Befehlstakt.....	178
12.5.2 LAT Register & R-M-W.....	178
12.6 MPLAB-X Projekte.....	178
12.6.1 Konfigurationen.....	178
12.6.2 Kopieren.....	178
12.6.3 Packen.....	178
12.6.4 µC-Projekt Dateien (TODO überarbeiten !!!!!!!!!!!!!!!).....	179
12.6.4.1 Source Files.....	179
12.6.4.1.1 Assembler (*.asm).....	179
12.6.4.1.2 C (*.c).....	179
12.6.4.2 Header / Include - Files (*.h, *.inc).....	179
12.6.4.2.1 Prozessorspezifische Header/Include -Dateien.....	179
12.6.4.2.2 Include Guard für Header Files.....	180
12.6.4.3 Linker Script (*.lkr).....	180
12.7 Eigenheiten der uC-Quick Hardware.....	181
12.7.1 Allgemein.....	181
12.7.1.1 Gemeinsame Nutzung von Pins durch LEDs, Display und Taster.....	181
12.7.2 Version 2018.....	181
12.7.2.1 Neue Displayvariante.....	181
12.7.3 Version 2013.....	181
12.7.4 Version 2012.....	181
12.7.4.1 Probleme durch unbenutztes LCD display.....	181
13 Anhang.....	182
13.1 Fehlermeldungen MPLAB / Linker / C18 / PICKit	182
13.1.1 Target was not found. You must connect to a target device	182
13.1.2 The target device is not ready for debugging	182
13.1.3 Target Device ID (0x0) does not match expected Device ID (0x5540).....	182
13.1.4 PICkit 3 is trying to supply x,yz volts from the USB port, but the target VDD is measured to be ??? volts.....	182
13.2 Assembler Templates.....	183
13.2.1 Assembler Configuration Bits.....	183
13.2.2 Assembler Main.....	184
13.2.3 Assembler Interrupt.....	185
13.2.4 Assembler EEPROM.....	187
13.2.5 Assembler Include.....	188
13.3 C Templates.....	189
13.3.1 C Config.....	189
13.3.2 C Main.....	190

13.3.3 C Interrupt.....	191
13.3.4 C Header Board Version 2013.....	192
13.3.5 C Header Board Version 2018.....	195
13.4 Microchip C18 Compiler / MPASM Assembler (alt).....	198
13.4.1 C18-Compiler → XC8-Compiler migration.....	198
13.4.1.1 Delay Funktionen.....	198
13.4.1.2 Context saving.....	198
13.4.1.3 Interrupt – Vektoren – Prioritäten	198
13.4.2 C Bibliotheken und Dokumentation.....	198
13.4.3 Der Startup Code – erste Funktionen.....	198
13.4.3.1 Wie kommt der Startup Code in mein Projekt ?.....	199
13.4.4 Die benötigten Include-Dateien.....	200
13.4.5 __init() Funktion.....	200
13.4.6 main() Funktion.....	200
13.4.7 C18 Interrupts.....	201
13.4.8 MPASM.....	202
13.4.8.1 MPASM Templates.....	202
13.4.9 Assembler File von der IDE erstellen lassen.....	203
13.4.10 Grundgerüst eines µC Programms in MPASM.....	203
13.4.10.1 Kommentare (;).....	203
13.4.10.2 MPASM Assembler Direktiven.....	204
13.4.10.2.1 list p= (list – LISTING OPTIONS).....	204
13.4.10.2.2 #include.....	204
13.4.10.2.3 config.....	205
13.4.10.2.4 udata / udata_acs (udata_ovr, udata_shr, idata).....	205
13.4.10.2.5 res.....	205
13.4.10.2.6 code.....	205
13.4.10.2.7 end.....	206
13.4.10.3 Aufsplittung des Assembler Templates in mehrere Dateien.....	207
13.4.10.3.1 Configuration Template.....	207
13.4.10.3.2 Main Template.....	207
13.4.10.4 Linker C18/MPASM.....	208
13.4.10.4.1 Linker Script.....	208
13.5 ANSI C mini-Guide.....	209
13.5.1 Kommentare.....	209
13.5.2 #include Direktive.....	209
13.5.3 Variablen.....	209
13.5.3.1 Data Type.....	209
13.5.3.2 Identifier.....	209
13.5.4 Konstanten.....	209
13.5.5 Operatoren.....	209
13.5.6 Expressions / Statements.....	209
13.5.7 Bedingungen (Verzweigungen).....	209
13.5.8 Loops (Schleifen).....	209
13.5.9 Functions.....	209
13.5.10 Arrays and Strings (Felder und Zeichenketten).....	209
13.5.11 Pointers (Zeiger).....	209
13.5.12 Structures, Bit fields, Unions.....	209
13.5.13 Enumerations.....	209
13.5.14 Macros (#define).....	209
14 Notizen.....	210
14.1 Dringend erforderliche Korrekturen & Ergänzungen.....	210

1 Die Basics (Überblick PIC18F2xK22)

Die „Basics“ in Kapitel 1 geben einen **Überblick** über das, was ein Mikrocontroller eigentlich ist. Im Grunde umfasst er ein vollständiges, kleines System und besteht aus einer Ansammlung von Modulen für verschiedene Einsatzzwecke, die mehr oder weniger autark arbeiten können.

Zur Konfiguration der Module und zur Organisation des Zusammenspiels verschiedener Module, braucht man dann noch „etwas“ **Software**, also Programmcode. Um einen Mikrocontroller auch nur halbwegs sinnvoll programmieren zu können, muss man sich vorher mit der Hardware beschäftigen.

1.1 Informationsquellen

1.1.1 Data Sheet

Die Programmierung eines Mikrocontrollers ist ohne Verwendung des Data-Sheet praktisch unmöglich. Es ist die komplette, detaillierte Beschreibung des verwendeten Mikrocontrollers und das wichtigste Informationsmedium bei der Programmierung von Mikrocontrollern.

Das für den verwendeten Controller relevante Handbuch muss immer griffbereit sein.

Die meisten Abbildungen in diesem Dokument sind dem Datenblatt entnommen und oft stark verkleinert dargestellt.

Diese Abbildungen gelten immer nur beispielhaft.

Es sollten deshalb immer die des aktuellen Datenblattes verwendet werden!

Alle Punkte gelten nicht nur bei der Programmierung in Assembler, sondern auch bei einer Hochsprache wie C.

1.1.2 „User Guide“ des Assemblers / Compilers / Linkers

Die Installationen sämtlicher Tools enthalten eigentlich immer auch Benutzerhandbücher. Diese findet man beispielsweise:

XC8 Compiler: [.../Microchip/xc8/v2.46/docs/MPLAB_XC8_C_Compiler_User_Guide.pdf](#)

Assembler: [.../xc8/v2.46/docs/MPLAB_XC8_PIC_Assembler_User_Guide.pdf](#)

für Beide jeweils: ... [Users_Guide_for_EMBEDDED_Engineers_PIC.pdf](#)

1.1.3 User Guide / Hilfe der IDE

Auch ein Handbuch für die IDE ist im Installationsverzeichnis enthalten. z.B.:

[.../Microchip/MPLABX/v6.15/docs/MPLAB_X_IDE_Users_Guide.pdf](#)

1.1.4 User Guide der Programmier- / Debug- Tools

Mit der Installation der IDE werden auch die Tools wie das PICkit3 installiert und auch dafür werden zusätzliche Informationen mitgeliefert:

[.../Microchip/MPLABX/v6.15/docs/Readme_for_PICkit_3.htm](#)

1.1.5 Sonstige Informationsquellen (Tutorials, Wiki, Tutorials ...)

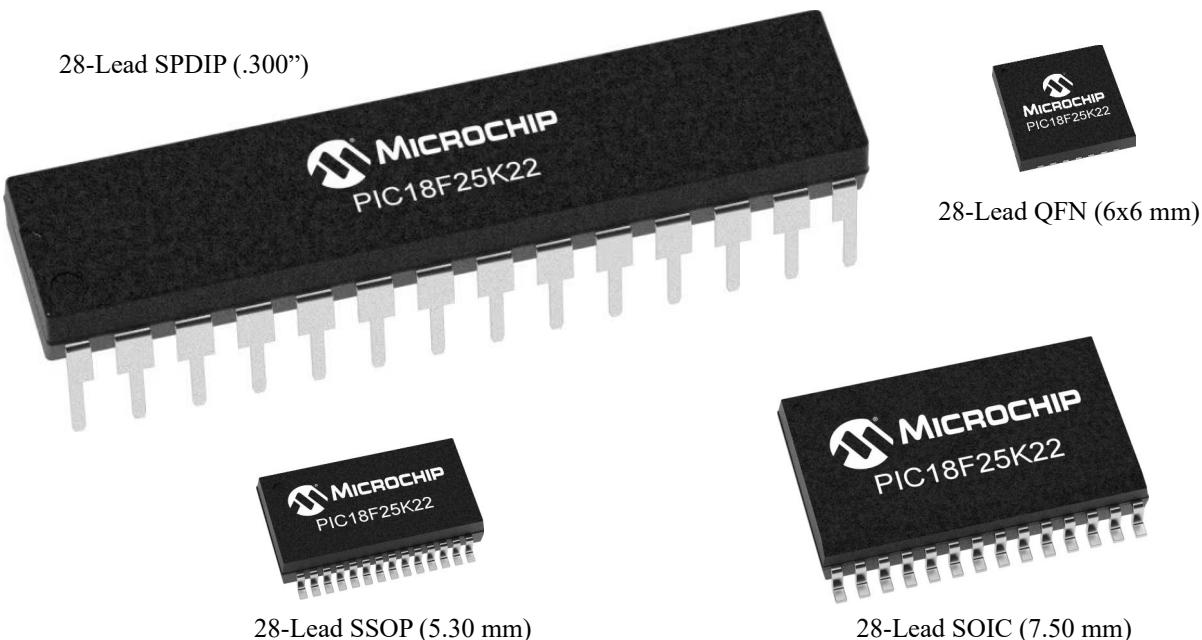
Auch diese Quellen können nützlich sein. Im besten Fall erläutern sie die Verwendung der oben genannten Hilfsmittel. Sehr viele Informationen findet man z.B. auf:

<https://developerhelp.microchip.com>

1.2 Übersicht über den PIC18F2xK22 Mikrocontroller

Den gleichen Microcontroller gibt es oft in unterschiedlichen Gehäusen, welche sich erheblich in Größe und Montageart unterscheiden können. Die bei den Gehäuseformen angegeben Maße beziehen sich auf die Breite des Plastikgehäuses (ohne Anschlüsse).

Gehäuseformen:



Eine Übersicht des Funktionsumfangs eines speziellen PIC Mikrocontrollers bieten die tabellarische Zusammenfassung auf den ersten Seiten des Data-Sheet und das (*Family*) Block-Diagramm.

Tabelle:

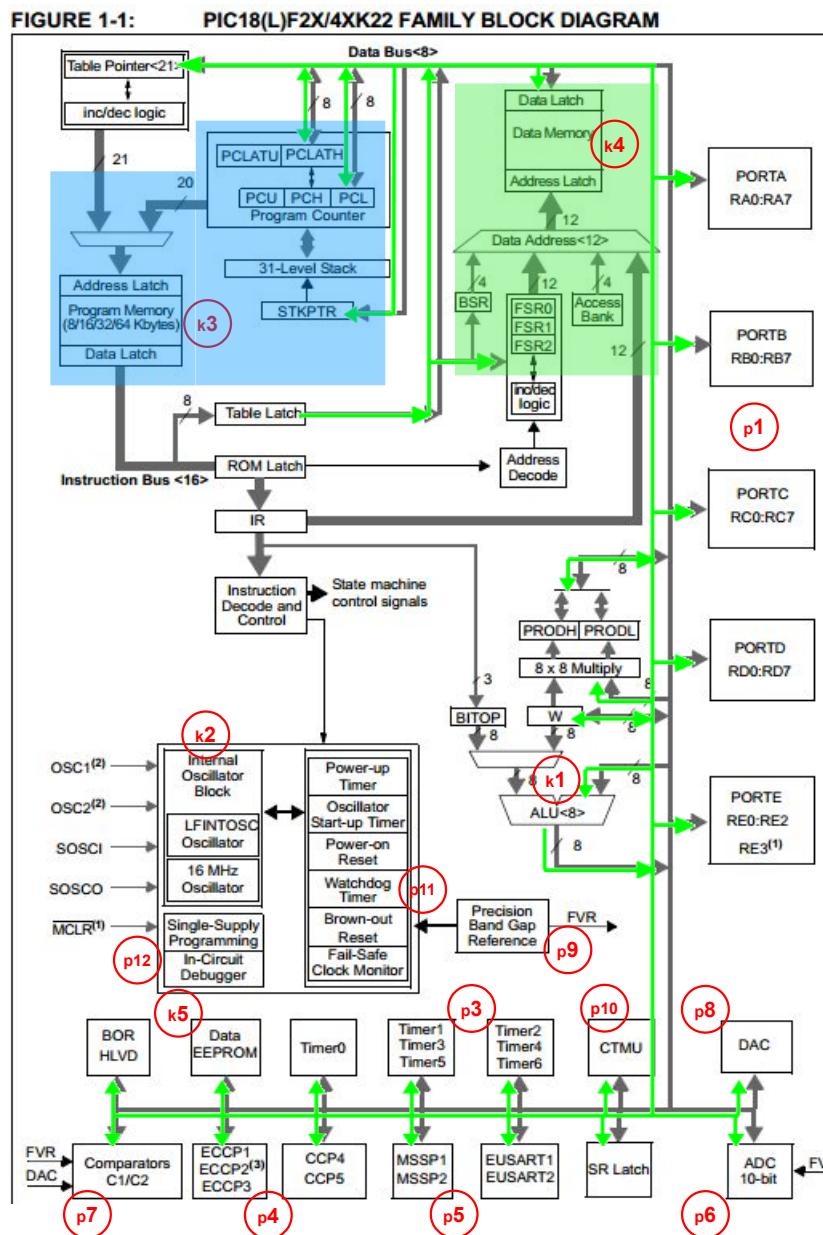
Device	Program Memory		Data Memory		I/O(1)	10-bit A/D Channels(2)	CCP	ECCP (Full-Bridge)	ECCP (Half-Bridge)	MSSP		SPI	I ² C™	EUUSART	Comparator	CTMU	BOR/LVD	SR Latch	8-bit Timer	16-bit Timer
	Flash (Bytes)	# Single-Word Instructions	SRAM (Bytes)	EEPROM (Bytes)						MSSP	I ² C™									
PIC18(L)F23K22	8K	4096	512	256	25	19	2	1	2	2	2	2	2	2	Y	Y	Y	3	4	
PIC18(L)F24K22	16K	8192	768	256	25	19	2	1	2	2	2	2	2	2	Y	Y	Y	3	4	
PIC18(L)F25K22	32K	16384	1536	256	25	19	2	1	2	2	2	2	2	2	Y	Y	Y	3	4	
PIC18(L)F26K22	64k	32768	3896	1024	25	19	2	1	2	2	2	2	2	2	Y	Y	Y	3	4	
PIC18(L)F43K22	8K	4096	512	256	36	30	2	2	1	2	2	2	2	2	Y	Y	Y	3	4	
PIC18(L)F44K22	16K	8192	768	256	36	30	2	2	1	2	2	2	2	2	Y	Y	Y	3	4	
PIC18(L)F45K22	32K	16384	1536	256	36	30	2	2	1	2	2	2	2	2	Y	Y	Y	3	4	
PIC18(L)F46K22	64k	32768	3896	1024	36	30	2	2	1	2	2	2	2	2	Y	Y	Y	3	4	

In der tabellarischen Zusammenfassung kann man sehr gut die Unterschiede der verschiedenen Mitglieder der PIC18FxxK22 Familie erkennen.

Die erste Ziffer hinter dem „F“ im Namen (2/4) gibt Aufschluss darüber, über wie viele Pins der PIC verfügt und somit auch eine unterschiedliche Anzahl von Peripheriekomponenten aufweist. Die nächste Ziffer (3/4/5/6) kennzeichnet den Speicherumfang. Je höher umso mehr Flash / SRAM

Blockdiagramm:

Das Blockdiagramm ist eine graphische Darstellung der verschiedenen Komponenten des PIC.



1.2.1 µC Architektur

Der grundsätzliche Aufbau eines Mikrocontrollers basiert auf der **Harvard Architektur**, bei der **Programmspeicher** und **Datenspeicher** vollkommen getrennt sind und auch unterschiedliche Bitbreiten aufweisen können. (*hier 8bit Daten, 16bit Programm*)

Das Programm wird direkt im **Programmspeicher** ausgeführt.

Der Zugriff auf Daten und auf sämtliche Peripherie-Module wie Ports, Timer usw. geschieht über den **Datenbus**.

Die Peripherie-Module sind als Register (*Adressen*) im Datenspeicher angelegt und somit Teil des RAM

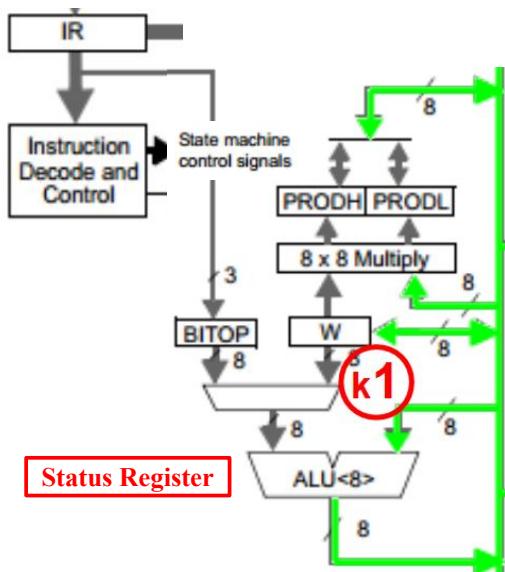
Ein Personal Computer hingegen basiert auf der „von Neumann“ Architektur; bei der das Programm zuerst (von der Festplatte) in das RAM geladen wird um dort ausgeführt zu werden. Programm und Daten sind dann beide im RAM enthalten.

1.2.2 PIC18 Kernmodule

1.2.2.1 CPU (central processing unit) und ALU (arithmetic logic unit)

Die Einheit welche für die Umsetzung der im Programmcode enthaltenen Befehle verantwortlich ist, nennt sich **Central Processing Unit**. Sie decodiert den momentan im Instruction Register (IR) liegenden Befehl und steuert die weiteren Kontroll-Signale.

Mit Hilfe der **Arithmetic logic unit** können einfache arithmetische Berechnungen (+-) sowie logische Operationen (*AND*, *OR*, *XOR*) durchgeführt werden. Bei diesen Operationen ist immer das Arbeitsregister W (*work*) beteiligt, welches mit einem Register aus dem Datenbereich wie in der Instruction bestimmt verrechnet wird. Das Ergebnis der Operation kann dann wieder im Arbeitsregister W oder im Datenbereich abgelegt werden.



1.2.2.1.1 8 x 8 Multiplier

Neben der einfachen ALU verfügen PIC18 noch über einen Hardware-Multiplier der zwei 8-Bit Werte miteinander multiplizieren und das Resultat in einem 16-Bit Register (*PROD*) ablegen kann.

1.2.2.1.2 Status Register

REGISTER 5-2: STATUS: STATUS REGISTER

U-0	U-0	U-0	R/W-x	R/W-x	R/W-x	R/W-x
—	—	—	N	OV	Z	DC ⁽¹⁾
bit 7						bit 0

Zusätzlich zum reinen Ergebnis der ALU Operationen, welches entweder im Arbeitsregister W oder im beteiligten Register aus dem RAM gespeichert wird, werden in einem Statusregister noch verschiedene Flags gesetzt, welche die letzte Operation beschreiben.

Diese Flags bilden die Basis für viele Kontroll Befehle

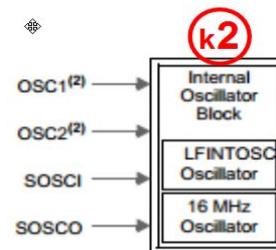
STATUS-FLAGS

- N: wenn das Ergebnis einer vorzeichenbehafteten arithmetischen Operation negativ wäre*
- OV: wenn vorzeichenbehaftet ein Überlauf (Overflow) stattgefunden hätte*
- Z : das Ergebnis war Null (Zero)
- C: beim Ergebnis einer arithmetischen Addition muss ein Übertrag (Carry-out) in das- oder von dem- nächst höherwertigen Byte berücksichtigt werden. z.B. $0b1111111 + 0b00000101 = 0b\text{00000001}00000100$
Ein Byte verhält sich hier genau wie eine Stelle im Dezimalsystem. Bei der Addition von $7+5 = 12$ erfolgt auch ein Übertrag von der Einer- in die Zehner-Stelle.
- /C: bei der Subtraktion wird aus dem Carry-Bit ein Borrow. Das heißt, falls der Subtrahend größer ist als der Minuend, wird ein erforderlicher Stellenübertrag (das Borgen) durch eine „0“ im Carry-Bit angezeigt!
 $z.B. 0b\text{00000010}00000101 - 0b1111111 = 0b00000000\text{1}00000110$
Auch hier passiert genau das gleiche wie bei einer Subtraktion im Dezimalsystem (z.B. $12-5 = 07$)
- DC: Digit Carry (Übertrag von bit 4 nach bit 5)

*Die ALU weiß nicht ob es sich um Vorzeichen behaftete Zahlen verarbeitet wurden, bzw. ob überhaupt „Zahlen“ im eigentlichen Sinn verarbeitet wurden. Das kann nur der Programmierer wissen!

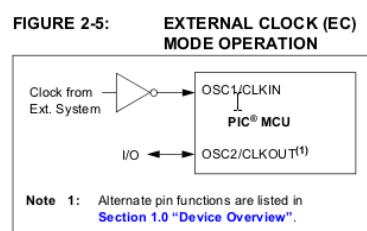
1.2.2.2 Oszillator-Modul / Arbeitstakt

Eines der wichtigsten Elemente eines Controllers ist das Oszillator-Modul, welches meist vielfältige Möglichkeiten zur Bereitstellung des Arbeitstaktes umfasst. Ein PIC18 bietet mehrere externe und auch interne Möglichkeiten den Arbeitstakt bereit zu stellen, die sich insbesondere hinsichtlich Stabilität, Kosten und Energieverbrauch unterscheiden.



Die vollständige Schaltung des Taktgebers kann dabei komplett aus externen, internen, oder auch aus externen und internen Komponenten bestehen.

1.2.2.1 Externer Oszillator



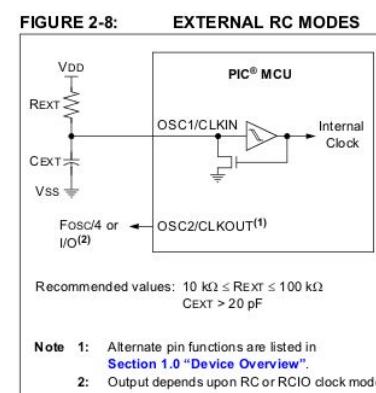
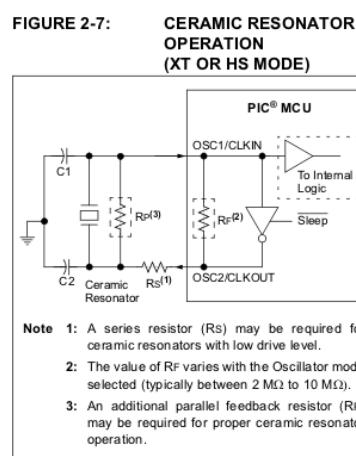
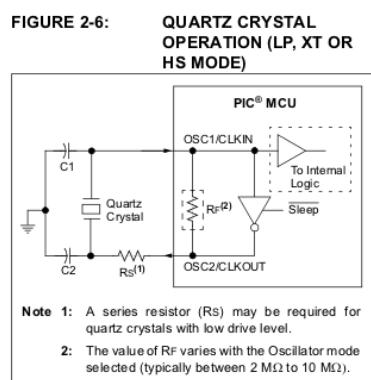
Im External Clock Mode wird nur ein Pin des Controllers benötigt, an welcher das Signal einer externen Taktquelle eingespeist wird. Genauigkeit und Stabilität des Arbeitstaktes sind so vom Controller unabhängig und können bei Auswahl eines entsprechenden externen Oszillators sehr hoch sein. Dies wirkt sich natürlich auch auf die Kosten des Systems aus, die dann ebenfalls steigen.

1.2.2.2 Oszillator mit externen frequenzbestimmenden Bauteilen

Bei den folgenden Varianten bestimmen die externen Komponenten Genauigkeit, Stabilität und Kosten. Alle drei aufgezählten Faktoren fallen dabei in der folgenden Aufzählung von links nach rechts.

Quarze und Keramische Resonatoren bilden zusammen mit der im Controller integrierten Rückkopplungsschaltung einen Schwingkreis und bestimmen dessen Frequenz.

Beim RC-Oszillator wird ein Kondensator über einen Widerstand aufgeladen. Bei einer gewissen Spannung löst dann ein Schmidt-Trigger die Entladung über einen internen Transistor aus.



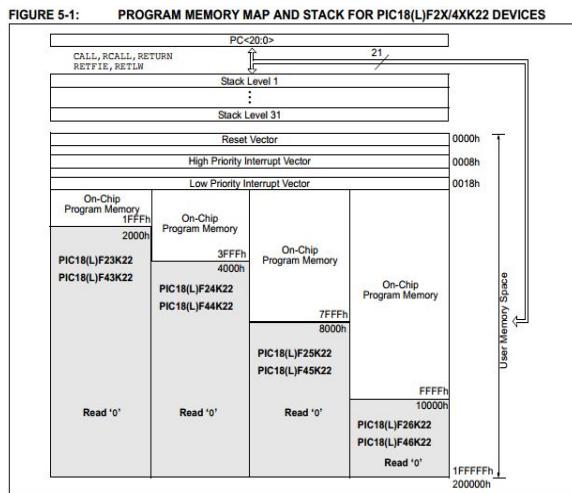
1.2.2.3 Interner Oszillator

Vom Internal Oscillator Mode spricht man, wenn keine externe Komponenten erforderlich sind. Somit werden auch keine Pins benötigt und diese stehen komplett für andere Anwendungen zur Verfügung. Beim internen Oszillator handelt es sich um einen relativ präzisen und kalibrierten RC-Oszillator ($\pm 1\%$), welcher für sehr viele Anwendungen ausreichend genau und stabil ist. Unter Laborbedingungen mit relativ konstanten Umgebungstemperaturen kann man damit auch asynchrone serielle Kommunikation mit anderen Systemen betreiben.

Speziellere Erläuterungen zum internen Oszillator Modul folgen [2.7 Oszillator Konfiguration](#)

1.2.2.3 Programmspeicher (Flash)

Der Programmspeicher eines PIC18F24K22 umfasst 16Kb. Da ein einfacher Befehl zwei Bytes (16 Bits) Speicherplatz belegt entspricht dies 8192 „Single instruction words“.



1.2.2.3.1 Program-Counter

Der Program-Counter ist eigentlich kein Teil des Programmspeichers, sondern ein Zeiger auf die aktuelle Adresse im Flash, d.h. auf den Befehl der gerade ausgeführt wird.

Im „normalen“ Programmablauf, ohne Verzweigung, wird der PC nach dem Abarbeiten eines Befehls automatisch um eins erhöht und zeigt somit auf den nächsten Befehl.

Bei Verzweigungen des Programms oder dem Aufruf von Unterprogrammen wird der PC über die CPU mit den im momentan aktiven Befehl enthaltenen Zieladressen geladen.

1.2.2.3.2 Stack (Hardware Stack)

Auch der Stack ist kein Teil des Flash-Speichers. Immer wenn in einem Programm Unterprogramme (*oder Funktionen*) aufgerufen werden, muss der momentane PC, bzw. die darauf folgende Adresse, zwischengespeichert werden damit das Programm nach Ausführung der Unterroutine wieder an dieser Stelle weitergeführt werden kann. Dieses Abspeichern wird auf dem (*Hardware*) Stack realisiert.

Die 31 Level des Hardware Stacks bedeuten, dass bis zu 31 Funktionsaufrufe „ineinander“ stehen können. Das Hauptprogramm ruft dabei die erste Funktion auf, die erste Funktion die zweite ... und schließlich die dreißigste die einunddreißigste.

Nach Beendigung der einunddreißigsten Funktion kehrt das Programm zurück zur dreißigsten, von da zur neunundzwanzigsten ... von der zweiten in die erste und von der ersten in das Hauptprogramm.

1.2.2.3.3 Reset-Vektor (0000h)

Der Reset-Vektor ist die erste Adresse im Flash-Speicher und der Startpunkt des Programms. Nach dem Einschalten der Spannungsversorgung zeigt der PC auf die Adresse 0x0000 und der Befehl an dieser Adresse (*meist nur ein Sprung über die Interruptvektoren*) wird als erstes ausgeführt.

Auch während des Betriebs kann man einen Rücksprung des PC auf Null erzwingen. Dieser kann durch Hardware (*Reset Taster an /MCLR Pin*) oder Software (*Programmbefehl*) erfolgen. Das Programm startet dann neu, ohne dass man dafür die Spannungsversorgung unterbrechen muss.

1.2.2.3.4 Interrupt Vektoren (0008h, 0018h)

Soll das Programm durch ein nicht durch den sequenziellen Ablauf vorgegebenes Ereignis unterbrochen werden, dann kann man dafür Interrupts nutzen. Wenn das Programm durch ein solches „paralleles“ Ereignis unterbrochen wird, dann springt der PC auf einen der Interrupt Vektoren. Näher zu Interrupts folgt im Kapitel [1.2.4 Interrupt-System](#)

1.2.2.3.5 On-Chip Program Memory (0019h...)

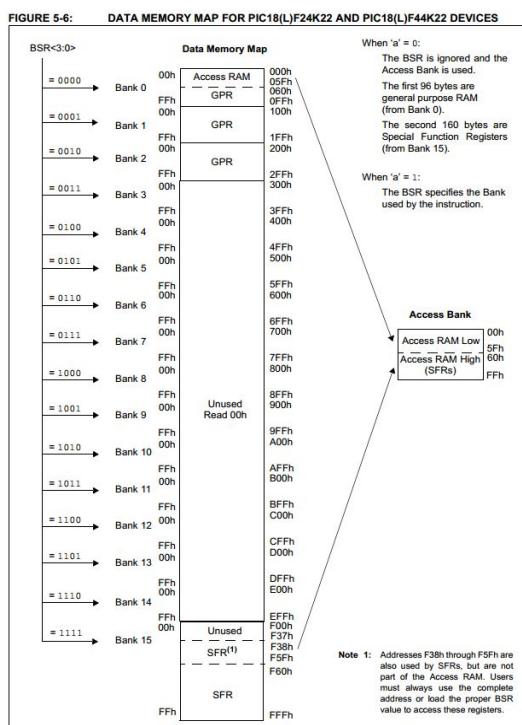
Dieser Speicher enthält im Normalfall das Programm, kann aber auch für Daten verwendet werden.

1.2.2.4 Datenspeicher (SRAM)

Die 768 (24K22) Bytes des Datenspeichers enthalten nicht nur den für das Programm frei verfügbaren Bereich, sondern auch sämtliche Register der Hardwaremodule. Beispielsweise sind die Ein-Ausgabe Ports auch nur Adressen im Datenspeicherbereich.

Die Daten im SRAM sind flüchtig, das heißt, sie bleiben beim Abschalten der Versorgungsspannung nicht erhalten, bzw. werden beim Wiedereinschalten auf definierte Werte eingestellt.

Die 200 Register der Hardwaremodule werden „*Special function register*“ (SFR) genannt, die restlichen frei verfügbaren 568 Bytes werden mit „*General purpose register*“ (GPR) bezeichnet.



1.2.2.4.1 Data Memory Banks

Der Datenspeicher eines PIC18 ist in einzelne Bereiche (*Banks*) aufgeteilt, welche jeweils einen Umfang von 256 Bytes haben. Der Grund für die Aufteilung ist einfach der, dass nicht die gesamte Adresse, welche 12 Bit umfassen würde, in einem Befehl (*16 Bit*) untergebracht werden kann. Die oberen 4 Bit werden in einem separaten Bank-Select-Register (*BSR*) abgelegt, das bei einem Wechsel der Bank über zusätzliche Befehle eingestellt werden muss. (siehe auch [1.2.6 Befehlssatz](#))

1.2.2.4.2 Access BANK

Ein Teil des Datenspeichers kann auch unter Ignorieren des BSR angesprochen werden. Dieser Bereich wird Access Bank genannt. Die Auswahl, ob das BSR genutzt wird oder das Access RAM wird über ein Bit ('a') im Befehl getroffen.

Die Access Bank setzt sich aus zwei Regionen zusammen die eigentlich in unterschiedlichen Bereichen des Data Memory liegen.

Access RAM High (160 Bytes) → SFR

Hier liegen die meisten Special-Function-Register (SFR) der Peripheriemodule.
(Weniger oft benutzte SFRs liegen in Bank 15)

Access RAM Low (96 Bytes)

Die 96 Bytes des Access RAM welche nicht durch SFR belegt sind, können im Programm für Variablen benutzt werden, auf die häufig zugegriffen werden muss. Durch das Wegfallen der Bankeinstellungsbefehle kann man so Rechenzeit (*und Programmspeicher*) einsparen.

1.2.2.4.3 General-Purpose-Registers → GPR

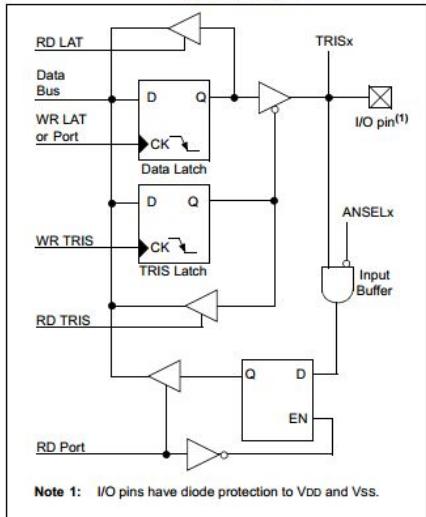
Auf Daten im GPR Bereich wird unter Berücksichtigung des BSR zugegriffen. D.h. das BSR muss vor dem Zugriff entsprechend eingestellt sein und der Zugriff ist unter Umständen etwas langsamer als auf Daten im Access RAM. (falls das BSR umgestellt werden muss)

1.2.2.5 Daten EEPROM

Im Daten EEPROM Speicher der 256 Bytes umfasst, können Daten dauerhaft gespeichert werden. Diese bleiben auch beim Aus- und Wiedereinschalten erhalten. Der Zugriff auf Daten im EEPROM ist allerdings wesentlich langsamer als auf Daten im SRAM. (vor allem beim Schreiben)

1.2.3 PIC18 Peripheriemodule

FIGURE 10-1: GENERIC I/O PORT OPERATION



1.2.3.1 IO Ports

Als IO Ports werden **digitale Ein- und Ausgänge** bezeichnet welche zu 8er Gruppen zusammengefasst sind und die über ein Register (*8 Bit*) angesprochen werden können. Nicht alle Mitglieder der 8er Gruppe müssen dabei für die gleiche Signalrichtung oder überhaupt als digitaler IO konfiguriert sein.
→ [3 Digitale Ein- und Ausgänge](#)

1.2.3.2 IO Pins

Die nebenstehende Abbildung zeigt die grundsätzliche Logik hinter einen Pin, falls dieser als digitaler IO konfiguriert ist. Die vielfältigen zusätzlichen Konfigurationsmöglichkeiten der einzelnen Pins und die Zuordnung zu verschiedenen Peripheriemodulen kann man unter anderem der Tabelle „PIN SUMMARY“ entnehmen.

TABLE 1: PIC18(L)F2XK22 PIN SUMMARY

28-SSOP, SOIC 28-SPDIP	28-QFN, UQFN	I/O	Analog	Comparator	CTMU	SR Latch	Reference	(E)CCP	EUSART	MSSP	Timers	Interrupts	Pull-up	Basic
2	27	RA0	AN0	C12IN0-										
3	28	RA1	AN1	C12IN1-										
4	1	RA2	AN2	C2IN+			VREF-DACOUT							
5	2	RA3	AN3	C1IN+			VREF+							
6	3	RA4		C1OUT		SRQ		CCP5			T0CKI			
7	4	RA5	AN4	C2OUT		SRNQ	HLVDIN				SS1			
10	7	RA6												OSC2 CLK0
9	6	RA7												OSC1 CLK1
21	18	RB0	AN12			SRI		CCP4 FLT0		SS2		INT0 Y		
22	19	RB1	AN10	C12IN3-				P1C		SCK2 SCL2		INT1 Y		

Die ersten beiden Spalten enthalten die Zuordnung eines Pins zu einer Gehäuseform. Die restlichen Spalten listen verschiedene Peripherie-Module auf und die Funktion die ein bestimmter Pin in diesem Modul haben kann.

(...)

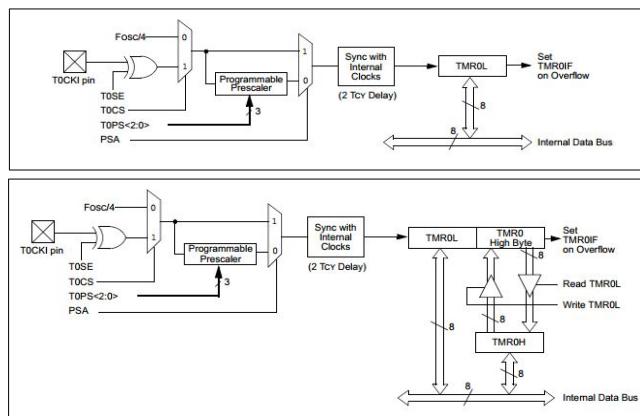
1.2.3.3 Timer / Counter

Eines der elementarsten Hardwaremodule neben den IOs sind die Timer, welche für alle Aufgaben verwendet werden können, die einen festen zeitliche Ablauf erfordern. Zudem können damit auch Signale im Zeitbereich wie Pulsdauer, Periodendauer, Frequenz usw. gemessen werden.

Ein PIC18FxxK22 verfügt über drei 8-Bit und vier 16-Bit Timer Module. Einige Module können über ihre externen Eingänge sowohl als Timer als auch als Counter (*Zähler*) verwendet werden. (*Timer sind Counter, die in einer konstanten Geschwindigkeit zählen*)

Alle Timermodule können Interrupts auslösen und so das Hauptprogramm in bestimmten Zeitabständen unterbrechen, um dann vordefinierte Aktionen durchzuführen.

TIMER0 BLOCK DIAGRAM 8/16 BIT MODE

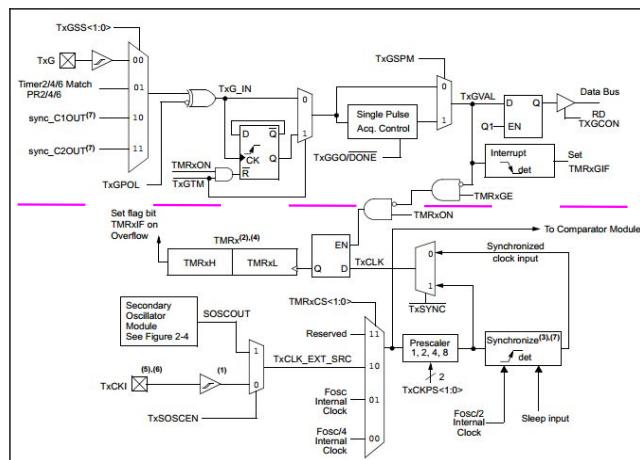


Timer 0 kann als 8-Bit oder 16-Bit Timer verwendet werden. D.h ein kompletter Durchlauf umfasst maximal 256 (2^8) oder 65536 (2^{16}) Schritte.

Durch Manipulation des Timer Registers TMR0 kann man auch kürzere Perioden erreichen.

Der Takteingang für Timer 0 kann vom Oszillator ($1/4$) abgeleitet werden oder auch ein externes Signal von einem Pin sein.

TIMER1/3/5 BLOCK DIAGRAM



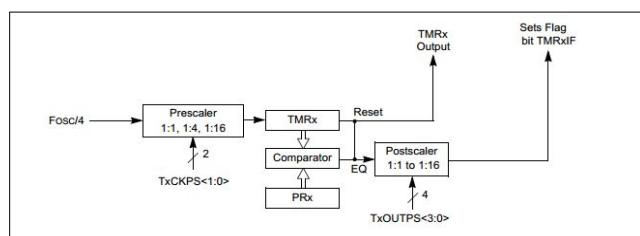
Die Timer 1, 3 und 5 sind identisch aufgebaut und etwas komplexer als Timer 0.

Die Bitbreite beträgt generell 16 Bit und kann nicht auf 8 umgestellt werden.

Der Takteingang kann wie beim Timer 0 extern oder intern sein, bietet aber zusätzlich die Option den Oszillatortakt direkt zu nutzen und nicht nur ein Viertel des Oszillatortaktes.

Im oberen Teil der Abbildung ist eine „Gating-Logic“ dargestellt, mit der man das Zählen zusätzlich zum Takt nochmals beeinflussen kann.

TIMER2/4/6 BLOCK DIAGRAM

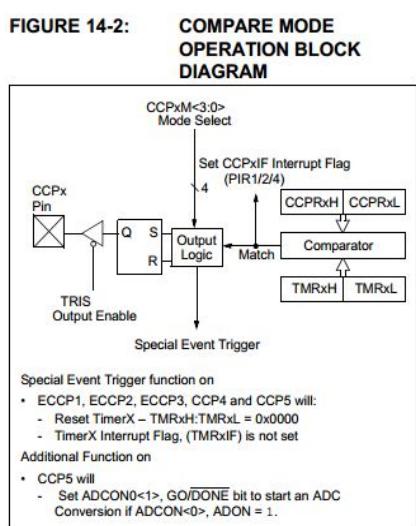
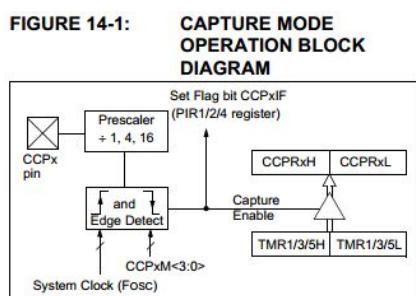


Timer 2, 4 und 6 sind wieder identische, aber jetzt 8-Bit Timer, die nur intern getaktet werden können.

Die Timer verfügen über ein zusätzliches Periodenregister (*PR*), in dem ein Endwert festgelegt werden kann, bis zu dem gezählt werden soll. (*optional zum Überlauf*)

1.2.3.4 Capture/Compare/PWM Modul (CCP)

CCP Module werden immer im Verbund mit Timern benutzt. Die Bezeichnungen „Capture“, „Compare“ und „PWM“ stehen für drei verschiedene Einsatzzwecke:



- Capture: (mit Timer 1/3/5)

Bei einem bestimmten Ereignis an einem Eingangspin wird der aktuelle Wert eines Timers automatisch abgespeichert und kann dann später verarbeitet werden. (*Prinzip Stoppuhr*)

Aus zwei in Folge gespeicherten Werten kann so beispielsweise die Periodendauer eines Signals bestimmt werden.

- Compare: (mit Timer 1/3/5)

Dieser Modus wird für die Steuerung von periodischen Vorgängen verwendet. Dabei wird der Timerwert mit einem Wert im Compare Register verglichen. Bei Übereinstimmung wird ein Interrupt-Flag gesetzt und weitere Aktionen automatisch ausgeführt (*konfigurierbar*)

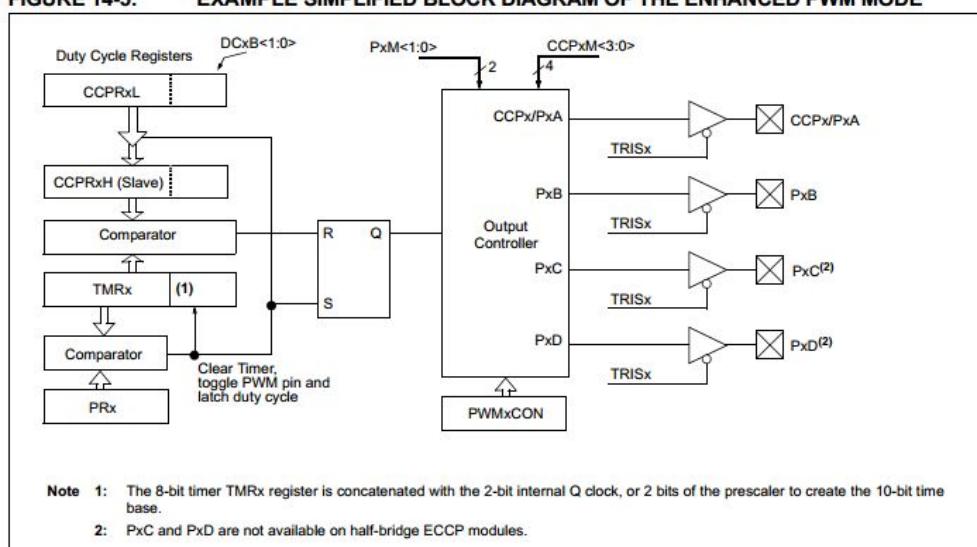
Zusätzlich kann der Timer automatisch auf Null zurück gestellt und eine Analog-Digital Wandlung gestartet werden.

Durch diese Art der Verwendung kann sehr einfach eine feste Abtastrate für Signale eingestellt werden. (*Special-event-trigger Modus*)

- PWM und Enhanced PWM: (mit Timer 2/4/6)

Erzeugung eines Puls-Weiten modulierten Signals durch den Vergleich eines Timerwertes mit zwei Werten für die Pulsdauer und die Periode. Mit pulsweitenmodulierten Signalen kann man sehr effektiv die Helligkeit von Lichtquellen, die Leistung von Heizelementen oder die Drehzahl von Motoren steuern.

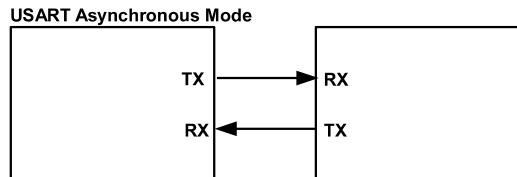
FIGURE 14-5: EXAMPLE SIMPLIFIED BLOCK DIAGRAM OF THE ENHANCED PWM MODE



1.2.3.5 Serielle Kommunikation

Für die Kommunikation mit anderen Systemen (*PC, Sensoren, Speicher, µC*) stehen verschiedene serielle Schnittstellen zur Verfügung.

1.2.3.5.1 EUSART: Enhanced Universal Synchronous Asynchronous Receiver Transmitter



RS232 Schnittstelle (*TTL Pegel*) für die Kommunikation mit anderen Controllern oder einem PC. Für die Kommunikation mit PCs sind zusätzliche Pegelwandler ICs erforderlich, welche die TTL Pegel verstärken und invertieren.

Durch die höheren Pegel sind dann auch längere Kabelverbindungen möglich.

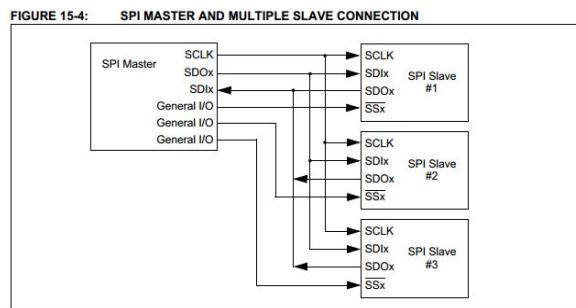
Auch Verbindungen mit virtuellen Com Ports (*VCP*) am PC über USB-Seriell Wandler oder Funkverbindungen über Bluetooth sind über diese Schnittstelle sehr einfach realisierbar.

Die **asynchrone** Datenübertragung erfolgt über separate Signalleitungen für jede Richtung ohne ein Clock Signal. Es ist nur eine Kommunikation zwischen zwei Partnern möglich.

Zusätzlich zu den Sende (*TX*) und Empfangsleitungen (*RX*) können noch diverse Handshake-Verbindungen implementiert werden, von denen die wichtigsten „Request To Send“ (*RTS*) und „Clear To Send“ (*CTS*) wären

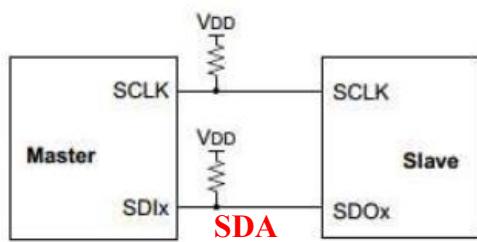
Da eine **UART** Verbindung über keine Synchronisation (*Clock*) verfügt müssen die Partner ihre Daten mit genau der gleichen Geschwindigkeit (*Baudrate*) übertragen.

1.2.3.5.2 SPI: Serial Peripheral Interface



Über den SPI-Bus kann ein Master mit mehreren Slaves (*Sensoren, Speicherbausteine ...*) kommunizieren. Die Kommunikation erfolgt Voll-Duplex und synchron. D.h. es stehen für Senden (*SDO*) und Empfangen (*SDI*) getrennte Signale zur Verfügung. Diese werden synchron mit einem Takt (*SCLK*) geschrieben bzw. gelesen. Ein zusätzliches Signal (*SSx*) selektiert dabei den jeweiligen Slave.

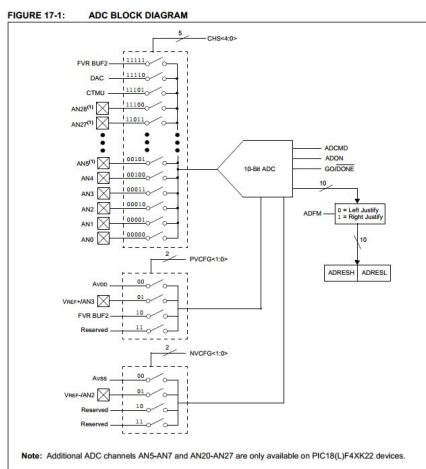
1.2.3.5.3 I²C: Inter-Integrated Circuit Bus



Auch der I2C Bus dient zur Anbindung von Slaves an einen Master. Meistens handelt es sich dabei um Verbindungen zwischen ICs auf der selben Platine oder über kurze Steck- bzw. Kabel-Verbindungen. Die Kommunikation erfolgt synchron mit einem Takt (*SCLK*). Die Datenleitung (*SDA*) ist immer bidirektional.

Im Gegensatz zu SPI geschieht die Adressierung der Slaves durch Adressen welche über die Datenleitung gesendet werden. Dadurch ist die Kommunikation mit mehreren Slaves ohne zusätzliche Chip-Select (*Slave-Select*) Leitungen möglich. Verglichen mit dem SPI-Bus werden also weniger Leitungen benötigt, dafür ist das Übertragungsprotokoll wesentlich aufwendiger.

1.2.3.6 10 Bit Analog-to-Digital-Converter (ADC)

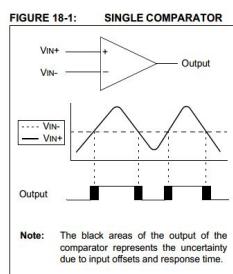


Bei den PIC18F2xK22 können 19 Eingangspins mit einem ADC Modul verbunden werden. Falls mehrere analoge Signale erfasst werden sollen, müssen die Eingangssignale allerdings sequentiell gemessen werden, da es nur ein ADC-Modul gibt.

Die 10 Bit des ADC-Moduls ermöglichen eine Auflösung von 1024 Schritten über den gesamten Bereich der Referenzspannung. Für die Referenzspannung gibt es verschiedene Einstellungsmöglichkeiten, welche die Versorgungsspannung, interne Referenzquellen und extern angelegte Spannungen für untere und obere Werte umfassen.

Die maximal erreichbare Abtastrate liegt bei 100.000 Messungen pro Sekunde.

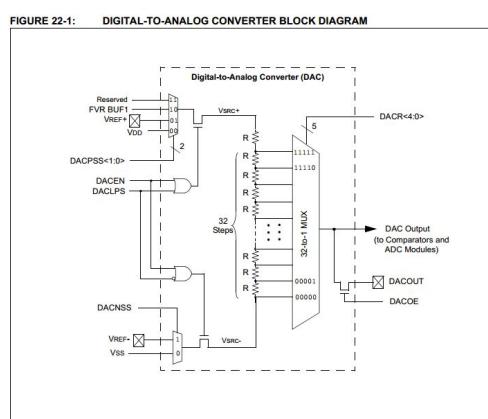
1.2.3.7 Comparator



Komparatoren werden für analoge Signale verwendet, die nicht wie bei der Analog-Digital-Wandlung komplett digitalisiert werden müssen.

Oft reicht ein einfacher Vergleich mit einem anderen analogen Signal oder einem Schwellwert, um einen Aussage zu bekommen ob ein Signal größer oder kleiner als das Vergleichssignal ist. Das Resultat des Vergleichs (*entweder „1“ oder „0“*) kann dann als Eingangsvariable für einen Steuer- oder Regelalgorithmus verwendet werden.

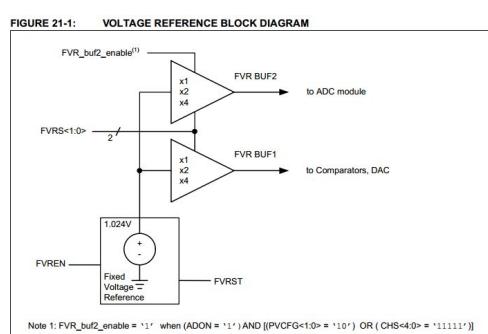
1.2.3.8 Digital Analog Converter (DAC)



Das DAC Modul dient vornehmlich der internen Erzeugung von Vergleichsspannungen für das Komparatormodul. Es besteht aber auch die Möglichkeit der Ausgabe des DAC Signals an einem Pin des μ C. Die Ausgangsspannung sollte dann aber vor weiterer Verwendung gepuffert werden!

Die Auflösung des DAC Moduls umfasst nur 5 Bit (32 Werte). Als untere und obere Referenzspannungen können wie beim ADC die Versorgungsspannung, intern erzeugte Referenzspannungen oder die extern an den Vref+ und Vref- Pins angelegten Spannungen verwendet werden.

1.2.3.9 Fixed Voltage Reference (FVR)

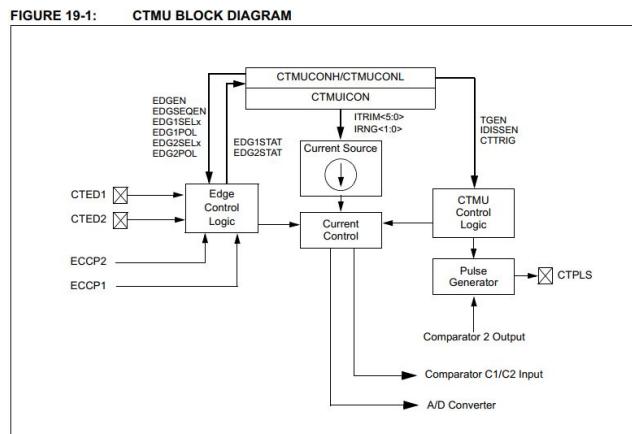


Der PIC18FxxK22 verfügt über eine interne Festspannungsreferenz von 1.024 V die mit den Faktoren 1, 2 und 4 verstärkt und für die ADC Wandlung und den DAC als Referenz verwendet werden kann.

Die 10-Bit (1024 Werte) ADC Auflösung entspricht dann genau einem, zwei oder vier Millivolt.

1.2.3.10 Charge Time Measurement Unit (CTMU)

Das CTMU Modul kann für genaue Zeit- oder Kapazitätsmessungen verwendet werden.



Bla...

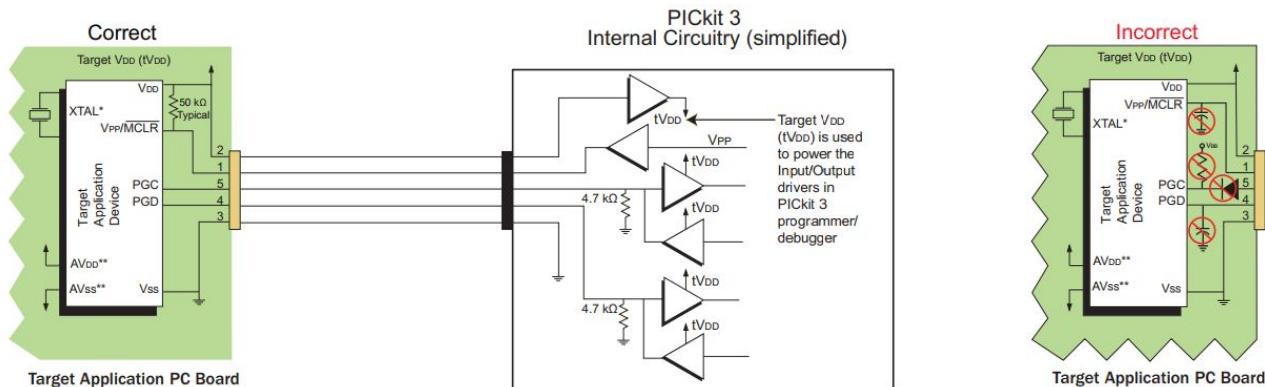
1.2.3.11 Watchdog Timer (WDT)

Der Watchdog Timer dient dazu, den Ablauf eines sicherheitskritischen Programms zu überwachen. Dies geschieht dadurch, dass das Programm den WDT regelmäßig zurücksetzen muss, bevor dieser überläuft. Geschieht dies nicht, kann angenommen werden, dass der vorgesehene Programmablauf gestört wurde, und das Programm wird durch einen Reset neu gestartet.

Bla...

1.2.3.12 Programmier- und Debug-Schnittstelle (ICSP/ICD)

Die Informationen zur Programmierschnittstelle in den PIC-Datenblättern sind eher spärlich. Ausführlicher sind die Infos, die man in den User-Guides der Tools wie z.B. [PICkit3](#), ICDx, ... findet. Sehr übersichtlich sind die benötigten Verbindungen in den Postern zu den Tools dargestellt.



Im grün hinterlegten Bereich, auf den Postern der Tools, wird die Verbindung zum PIC dargestellt. Das Programmier-/Debug-Tool kommuniziert mit dem **im PIC integrierten Debugger** über die Pins **PGC** (*ProGram Clock*) und **PGD** (*ProGram Data*). Als Bezugspotential für diese Signale wird natürlich auch noch **V_{SS}** benötigt. Die Versorgungsspannung **V_{DD}** wird verwendet, um den Signalpegel des Tools automatisch an die vorliegende Schaltung anzupassen. Zum Schreiben in den Programmspeicher des PICs ist eventuell noch eine spezielle Spannung **V_{PP}** nötig (höher als V_{DD}).

Die am PIC benötigten Pins dürfen nicht noch beliebig anderweitig beschaltet sein. (Correct links; Incorrect rechts) Genauere Auskünfte dazu findet man im User-Guide oder den Postern.

1.2.4 Die Configuration Bits

In den Configuration Bits sind Einstellungen zusammengefasst, die nicht zur Laufzeit des Programms vorgenommen werden können. Die Konfigurations-Einstellungen werden bei der Erstellung des Programms bestimmt und bei der Übertragung des Programms auf den Controller gesetzt. Während des normalen Programmablaufs ist der Zugriff auf diese Bits nicht vorgesehen.

TABLE 24-1: CONFIGURATION BITS AND DEVICE IDs

Address	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Default/ Unprogrammed Value
300000h	CONFIG1L	—	—	—	—	—	—	—	—	0000 0000
300001h	CONFIG1H	IESO	FCMEN	PRICLKEN	PLLCFG	FOSC<3:0>			—	0010 0101
300002h	CONFIG2L	—	—	—	BORV<1:0>	BOREN<1:0>		PWRDEN	—	0001 1111
300003h	CONFIG2H	—	—	WDPS<3:0>			WDTEN<1:0>		—	0011 1111
300004h	CONFIG3L	—	—	—	—	—	—	—	—	0000 0000
300005h	CONFIG3H	MCLRE	—	P2BMX	T3CMX	HFOFST	CCP3MX	PBADEN	CCP2MX	1011 1111
300006h	CONFIG4L	DEBUG	XINST	—	—	—	LVP ⁽¹⁾	—	STRVEN	1000 0101
300007h	CONFIG4H	—	—	—	—	—	—	—	—	1111 1111
300008h	CONFIG5L	—	—	—	—	CP3 ⁽²⁾	CP2 ⁽²⁾	CP1	CP0	0000 1111
300009h	CONFIG5H	CPD	CPB	—	—	—	—	—	—	1100 0000
30000Ah	CONFIG6L	—	—	—	—	WRT3 ⁽²⁾	WRT2 ⁽²⁾	WRT1	WRT0	0000 1111
30000Bh	CONFIG6H	WRTD	WRTB	WRTC ⁽³⁾	—	—	—	—	—	1110 0000
30000Ch	CONFIG7L	—	—	—	—	EBTR3 ⁽²⁾	EBTR2 ⁽²⁾	EBTR1	EBTR0	0000 1111
30000Dh	CONFIG7H	—	EBTRB	—	—	—	—	—	—	0100 0000
3FFFFFFh	DEVID1 ⁽⁴⁾	DEV<2:0>			REV<4:0>				qqqq qqqq	—
3FFFFFh	DEVID2 ⁽⁴⁾	DEV<10:3>							0101 qqqq	—

Legend: — = unimplemented, q = value depends on condition. Shaded bits are unimplemented, read as '0'.

Note 1: Can only be changed when in high voltage programming mode.

2: Available on PIC18(L)FX5K22 and PIC18(L)FX6K22 devices only.

3: In user mode, this bit is read-only and cannot be self-programmed.

4: See Register 24-12 and Register 24-13 for DEVID values. DEVID registers are read-only and cannot be programmed by the user.

Bestimmte Einstellungen wie z.B. die Konfiguration der Quelle für den Befehlstakt, können nicht erst zur Laufzeit des Programms eingestellt werden, weil ansonsten das Programm unter Umständen gar nicht laufen könnte, um diese Einstellungen vorzunehmen.

Auch die Bits zum Schützen des Programm- und EEPROM Speichers gegen unerlaubtes Kopieren, müssen schon wirksam sein, wenn das Programm noch gar nicht läuft.

REGISTER 24-1: CONFIG1H: CONFIGURATION REGISTER 1 HIGH

R/P-0	R/P-0	R/P-1	R/P-0	R/P-0	R/P-1	R/P-0	R/P-1
IESO	FCMEN	PRICLKEN	PLLCFG	FOSC<3:0>			
bit 7							bit 0

Legend:
R = Readable bit P = Programmable bit U = Unimplemented bit, read as '0'
-n = Value when device is unprogrammed x = Bit is unknown

bit 7 **IESO⁽¹⁾:** Internal/External Oscillator Switchover bit
1 = Oscillator Switchover mode enabled
0 = Oscillator Switchover mode disabled

bit 6 **FCMEN⁽¹⁾:** Fail-Safe Clock Monitor Enable bit
1 = Fail-Safe Clock Monitor enabled
0 = Fail-Safe Clock Monitor disabled

bit 5 **PRICLKEN:** Primary Clock Enable bit
1 = Primary Clock is always enabled
0 = Primary Clock can be disabled by software

bit 4 **PLLCFG:** 4 x PLL Enable bit
1 = 4 x PLL always enabled, Oscillator multiplied by 4
0 = 4 x PLL is under software control, PLLEN (OSCTUNE<6>)

bit 3-0 **FOSC<3:0>:** Oscillator Selection bits
1111 = External RC oscillator, CLKOUT function on RA6
1110 = External RC oscillator, CLKOUT function on RA6
1101 = EC oscillator (**low power, <500 kHz**)
1100 = EC oscillator, CLKOUT function on OSC2 (**low power, <500 kHz**)
1011 = EC oscillator (**medium power, 500 kHz-16 MHz**)
1010 = EC oscillator, CLKOUT function on OSC2 (**medium power, 500 kHz-16 MHz**)
1001 = Internal oscillator block, CLKOUT function on OSC2
1000 = Internal oscillator block
0111 = External RC oscillator, CLKOUT function on OSC2
0101 = EC oscillator (**high power, >16 MHz**)
0100 = EC oscillator, CLKOUT function on OSC2 (**high power, >16 MHz**)
0011 = HS oscillator (**medium power, 4 MHz-16 MHz**)
0010 = HS oscillator (**high power, >16 MHz**)
0001 = XT oscillator
0000 = LP oscillator

Note 1: When FOSC<3:0> is configured for HS, XT, or LP oscillator and FCMEN bit is set, then the IESO bit should also be set to prevent a false failed clock indication and to enable automatic clock switch over from the internal oscillator block to the external oscillator when the OST times out.

REGISTER 24-6: CONFIG5L: CONFIGURATION REGISTER 5 LOW

U-0	U-0	U-0	U-0	R/C-1	R/C-1	R/C-1	R/C-1
—	—	—	—	CP3 ⁽¹⁾	CP2 ⁽¹⁾	CP1	CP0
bit 7							bit 0

Legend:
R = Readable bit U = Unimplemented bit, read as '0'
-n = Value when device is unprogrammed C = Clearable only bit

bit 7-4 **Unimplemented:** Read as '0'
bit 3 **CP3:** Code Protection bit⁽¹⁾
1 = Block 3 not code-protected
0 = Block 3 code-protected

bit 2 **CP2:** Code Protection bit⁽¹⁾
1 = Block 2 not code-protected
0 = Block 2 code-protected

bit 1 **CP1:** Code Protection bit
1 = Block 1 not code-protected
0 = Block 1 code-protected

bit 0 **CP0:** Code Protection bit
1 = Block 0 not code-protected
0 = Block 0 code-protected

Note 1: Available on PIC18(L)FX5K22 and PIC18(L)FX6K22 devices.

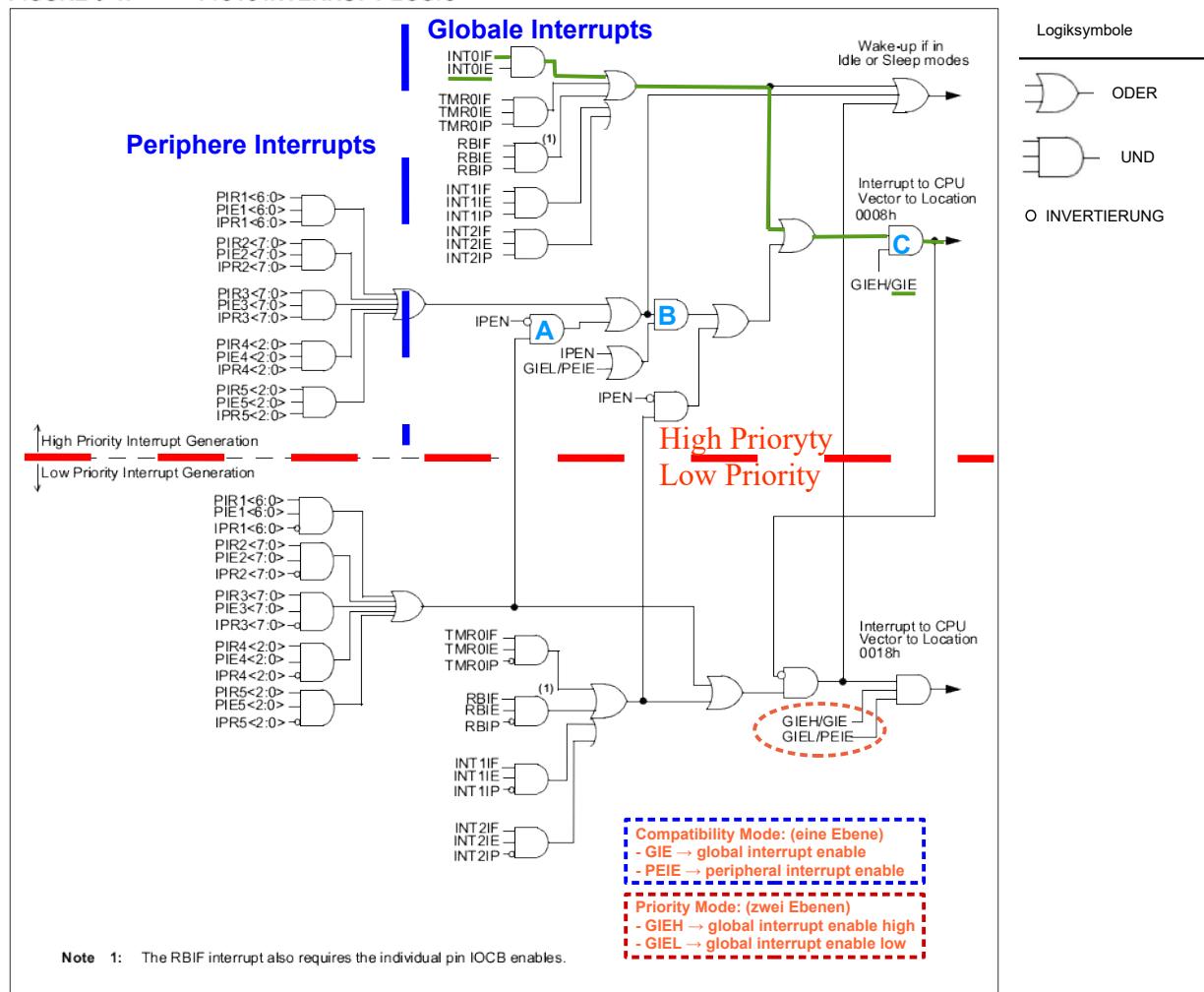
1.2.5 Interrupt-System

Interrupts dienen dazu, den normalen Ablauf des Hauptprogramms bei bestimmten Ereignissen zu unterbrechen. Es gibt sehr viele Quellen die einen Interrupt auslösen können. Dazu gehören Signale an den äußeren Pins des Mikrocontrollers oder von internen Peripheriemodulen wie den Timern oder Kommunikations-Schnittstellen.

Wird ein Interrupt ausgelöst, muss der Kontext des Hauptprogramms gesichert werden und der [Programmzeiger](#) springt auf eine bestimmte Adresse, an der die Befehle stehen müssen, die dann ausgeführt werden sollen. Nach Abarbeitung der für das jeweilige Ereignis notwendigen Befehle erfolgt wieder der Rücksprung in das Hauptprogramm.

Damit ein IR ausgelöst werden kann, muss der Pfad von der Quelle links bis zum Ausgang auf der rechten Seite durch die komplette Interrupt Logik erfolgen. Bei den UND-Verknüpfungen müssen dabei alle Eingänge eine logische Eins aufweisen, die ODER-Verknüpfungen können unabhängig von den anderen Eingängen passiert werden.

FIGURE 9-1: PIC18 INTERRUPT LOGIC



1.2.5.1 Interrupt 0, INT0 (einfachstes Beispiel)

Der einfachste IR ist **INT0** (**grün**) am entsprechenden Pin des Controllers. Ein Signal am **INT0 Pin** setzt zunächst das Interrupt-Flag **INT0IF**. An der ersten UND-Verknüpfung muss auch das Interrupt-Enable **INT0IE** gesetzt sein, damit das Signal passieren kann. Danach folgen zwei ODER-Verknüpfungen die das Signal durchlassen. Am Ende folgt eine UND-Verknüpfung mit dem globalen IR Enable **GIE**, welches auch gesetzt sein muss, damit ein Interrupt stattfindet.
(Auf die Bedeutung der Doppelnamen GIEH/GIE und GIEL/PEIE wird weiter unten eingegangen)

1.2.5.2 Priority Mode vs. Compatibility Mode

Bei PIC18 Controllern kann das Interrupt-System in zwei verschiedenen Modi betrieben werden. In der obigen Abbildung werden die verschiedenen Modi durch die Teilung mittels der farbigen Linien dargestellt. Je nachdem welcher Mode verwendet wird, benutzt man dann am Besten auch die alternativen Bezeichnungen für die Bits **GIEH/GIE** und **GIEL/PEIE** um im Programmcode zu verdeutlichen, in was für einem Mode sich der Controller befindet.

Die Mode Auswahl geschieht über das hier nicht dargestellte Interrupt-Priority-Enable Bit **IPEN** im Register **RCON**.

1.2.5.2.1 Priority Mode

Das PIC18 Interrupt System kann bei Bedarf aus zwei Ebenen (*Prioritäten*) bestehen. Das bedeutet, dass ein niedrig priorisierte Interrupt der das Hauptprogramm unterbrochen hat, nochmals von einem höher priorisierten IR unterbrochen werden kann.

Im Priority Mode werden die Interrupt-Priority Eingänge **xxxIP** wichtig, die sich an den ersten UND-Verknüpfungen (*außer der von INT0*) befinden. In der oberen Bildhälfte des IR-Logic Diagramms sind diese „normal“ angeschlossen, in der unteren „invertiert“ (kleiner Kreis).

Low Priority IR (Vektor 0x0018)

Ein Low Priority Interrupt kann das Hauptprogramm unterbrechen. Wie man am ausgangsseitigen UND-Gatter der unteren Diagrammhälfte erkennen kann, müssen dafür **GIEL** und **GIEH** gesetzt sein. Am vorgesetzten UND-Gatter kann man erkennen, dass der Weg blockiert ist, wenn ein High Priority IR aktiv ist und somit solange kein Low Priority IR ausgelöst werden kann, bis der High Priority IR abgearbeitet ist.

High Priority IR (Vektor 0x0008)

Ein High Priority IR kann das Hauptprogramm und gegebenenfalls auch einen Low Priority IR unterbrechen. Nur **GIEH**, und die jeweiligen **IE** und **IP** Bits müssen gesetzt sein. Der High Priority IR kann nicht mehr unterbrochen werden. (*Auch nicht von einem anderen High Priority IR*)

1.2.5.2.2 Compatibility Mode

Das „Compatibility“ im Namen kommt hier von der Kompatibilität mit den „kleineren“ 8-Bit Controllern der PIC10/12/16 Familien. Diese haben nur einen IR-Vektor und nur eine Ebene. Auch in diesem Mode gibt es eine Unterteilung der Interrupts in „globale“ und „periphere“. Das Bit **GIEL/PEIE** hat in diesem Mode die Funktion alle peripheren IR zu kontrollieren und wird deshalb Peripheral Interrupt Enable PEIE genannt. Sind beide Interrupt-Gruppen (globale und periphere) aktiv, dann sind auch beide gleichberechtigt. Die IP Bits sind in diesem Mode bedeutungslos. Wie am ersten Beispiel in Kapitel [5 Interrupts](#) noch zu sehen ist, beginnt zwar der Pfad des Interrupts in Abhängigkeit der IP Bits, das Ende liegt aber immer in der oberen Diagrammhälfte beim IR-Vektor 0x0008.

Global IR (Vektor 0x0008)

Zur Gruppe der globalen Interrupts zählen die Pin-Interrupts und der IR von Timer0. (...)

Peripheral IR (Vektor 0x0008)

Die Vielzahl der anderen von den internen Peripheriemodulen verfügbaren Interrupts wird im Diagramm nicht einzeln aufgeführt. Diese sind in den zusammengefassten Peripheral-Interrupt-Enable **PIEx** und Peripheral-Interrupt-Flag **PIFx** Registern dargestellt. Die zugehörigen Interrupt Priority Register **IPRx** sind, wie schon oben angesprochen, nur im Priority Mode von Bedeutung.

1.2.6 Befehlssatz

Der PIC18 ist ein sogenannter **RISC**-Prozessor. RISC steht für Reduced-Instruction-Set-Controller, also für einen Controller mit einem reduziertem (*möglichst geringen*) Umfang an Befehlen. Die meisten Befehle sollen innerhalb eines Zyklus abgearbeitet werden können.

Ein PIC18 verfügt laut Datenblatt über 75 Standardbefehle und 8 Befehle aus einem erweiterten Befehlssatz. Diese geringe Anzahl an Befehlen ermöglicht es den Programmspeicher mit einer Breite von 16 Bit anzulegen.

In den 16 Bit sind der **OPCODE** (*die eigentliche Befehlsanweisung*) und die **Operanden** enthalten, auf die der Befehl angewendet werden soll. Damit die 16 Bit Befehlsbreite ausreichen, ist die Aufteilung der Bits in Opcode und Operanden nicht immer gleich, sondern von der Art des Befehls abhängig. Eine Übersicht dazu gibt eine Abbildung im Data Sheet.

FIGURE 25-1 : GENERAL FORMAT FOR INSTRUCTIONS

Byte-oriented file register operations	Example Instruction																	
<table border="1"> <tr> <td>15</td><td>10</td><td>9</td><td>8</td><td>7</td><td>0</td></tr> <tr> <td>OPCODE</td><td>d</td><td>a</td><td colspan="3">f (FILE #)</td></tr> </table> <p>d = 0 for result destination to be WREG register d = 1 for result destination to be file register (f) a = 0 to force Access Bank a = 1 for BSR to select bank f = 8-bit file register address</p>	15	10	9	8	7	0	OPCODE	d	a	f (FILE #)			ADDWF MYREG, W, B					
15	10	9	8	7	0													
OPCODE	d	a	f (FILE #)															
Byte to Byte move operations (2-word)																		
<table border="1"> <tr> <td>15</td><td>12</td><td>11</td><td>0</td></tr> <tr> <td>OPCODE</td><td colspan="3">f (Source FILE #)</td></tr> </table> <table border="1"> <tr> <td>15</td><td>12</td><td>11</td><td>0</td></tr> <tr> <td>1111</td><td colspan="3">f (Destination FILE #)</td></tr> </table> <p>f = 12-bit file register address</p>	15	12	11	0	OPCODE	f (Source FILE #)			15	12	11	0	1111	f (Destination FILE #)			MOVFF MYREG1, MYREG2	
15	12	11	0															
OPCODE	f (Source FILE #)																	
15	12	11	0															
1111	f (Destination FILE #)																	
Bit-oriented file register operations																		
<table border="1"> <tr> <td>15</td><td>12</td><td>11</td><td>9</td><td>8</td><td>7</td><td>0</td></tr> <tr> <td>OPCODE</td><td>b (BIT #)</td><td>a</td><td colspan="3">f (FILE #)</td><td></td></tr> </table> <p>b = 3-bit position of bit in file register (f) a = 0 to force Access Bank a = 1 for BSR to select bank f = 8-bit file register address</p>	15	12	11	9	8	7	0	OPCODE	b (BIT #)	a	f (FILE #)				BSF MYREG, bit, B			
15	12	11	9	8	7	0												
OPCODE	b (BIT #)	a	f (FILE #)															
Literal operations																		
<table border="1"> <tr> <td>15</td><td>8</td><td>7</td><td>0</td></tr> <tr> <td>OPCODE</td><td colspan="3">k (literal)</td></tr> </table> <p>k = 8-bit immediate value</p>	15	8	7	0	OPCODE	k (literal)			MOVLW 7Fh									
15	8	7	0															
OPCODE	k (literal)																	
Control operations																		
CALL, GOTO and Branch operations																		
<table border="1"> <tr> <td>15</td><td>8</td><td>7</td><td>0</td></tr> <tr> <td>OPCODE</td><td colspan="3">n<7:0> (literal)</td></tr> </table> <table border="1"> <tr> <td>15</td><td>12</td><td>11</td><td>0</td></tr> <tr> <td>1111</td><td colspan="3">n<19:8> (literal)</td></tr> </table> <p>n = 20-bit immediate value</p>	15	8	7	0	OPCODE	n<7:0> (literal)			15	12	11	0	1111	n<19:8> (literal)			GOTO Label	
15	8	7	0															
OPCODE	n<7:0> (literal)																	
15	12	11	0															
1111	n<19:8> (literal)																	
<table border="1"> <tr> <td>15</td><td>8</td><td>7</td><td>0</td></tr> <tr> <td>OPCODE</td><td>S</td><td colspan="2">n<7:0> (literal)</td><td></td></tr> </table> <table border="1"> <tr> <td>15</td><td>12</td><td>11</td><td>0</td></tr> <tr> <td>1111</td><td colspan="3">n<19:8> (literal)</td></tr> </table> <p>S = Fast bit</p>	15	8	7	0	OPCODE	S	n<7:0> (literal)			15	12	11	0	1111	n<19:8> (literal)			CALL MYFUNC
15	8	7	0															
OPCODE	S	n<7:0> (literal)																
15	12	11	0															
1111	n<19:8> (literal)																	
<table border="1"> <tr> <td>15</td><td>11</td><td>10</td><td>0</td></tr> <tr> <td>OPCODE</td><td colspan="3">n<10:0> (literal)</td></tr> </table>	15	11	10	0	OPCODE	n<10:0> (literal)			BRA MYFUNC									
15	11	10	0															
OPCODE	n<10:0> (literal)																	
<table border="1"> <tr> <td>15</td><td>8</td><td>7</td><td>0</td></tr> <tr> <td>OPCODE</td><td colspan="3">n<7:0> (literal)</td></tr> </table>	15	8	7	0	OPCODE	n<7:0> (literal)			BC MYFUNC									
15	8	7	0															
OPCODE	n<7:0> (literal)																	

Die meisten der Befehle, welche das Arbeitsregister W und ein Byte aus dem RAM als Operanden enthalten, benötigen 8 Bits für die Adresse des Bytes im RAM (*f*), 1 Bit für die Angabe ob das Byte im Access RAM liegt (*a*) und 1 Bit als Angabe wohin das Ergebnis des Befehls geschrieben werden soll (*d*). Damit bleiben 6 Bit für den Opcode.

Bit-orientierte Befehle, die sich nur auf ein Bit in einem Byte des RAMs beziehen, brauchen zusätzlich noch die genaue Position des Bits im Byte. Um die 8 möglichen Positionen zu adressieren werden drei ($2^3 = 8$) zusätzliche Bits benötigt. Weil die Richtungsangabe für das Ergebnis weg fällt, bleiben 4 Bit für den Opcode.

Für Befehle die unmöglich in 16 Bit passen, werden zwei Programm Adressen verwendet. Dies betrifft Sprungbefehle und Funktionsaufrufe im 20 Bit großen Adressraum des Programmspeichers.

Auch der Kopierbefehl MOVFF weiter oben, welcher ein direktes Kopieren von Daten im RAM ohne den Umweg über das Arbeitsregister W erlaubt, benötigt zwei Programm Wörter.

1.2.6.1 Byte-orientierte Befehle

Byte-orientierte Befehle verarbeiten immer ein komplettes Register. Die Verarbeitung kann nur auf das Register selbst bezogen sein, oder auch im Verbund mit dem Arbeitsregister WREG geschehen.

TABLE 25-2: PIC18(L)F2X/4XK22 INSTRUCTION SET

Mnemonic, Operands	Description	Cycles	16-Bit Instruction Word				Status Affected	Notes	
			MSb			Lsb			
BYTE-ORIENTED OPERATIONS									
ADDWF f, d, a	Add WREG and f	1	0010	01da	ffff	ffff	C, DC, Z, OV, N	1, 2	
ADDWFC f, d, a	Add WREG and CARRY bit to f	1	0010	00da	ffff	ffff	C, DC, Z, OV, N	1, 2	
ANDWF f, d, a	AND WREG with f	1	0001	01da	ffff	ffff	Z, N	1, 2	
CLRF f, a	Clear f	1	0110	101a	ffff	ffff	Z	2	
COMF f, d, a	Complement f	1	0001	11da	ffff	ffff	Z, N	1, 2	
CPFSEQ f, a	Compare f with WREG, skip =	1 (2 or 3)	0110	001a	ffff	ffff	None	4	
CPFGT f, a	Compare f with WREG, skip >	1 (2 or 3)	0110	010a	ffff	ffff	None	4	
CPFSLT f, a	Compare f with WREG, skip <	1 (2 or 3)	0110	000a	ffff	ffff	None	4	
DECf f, d, a	Decrement f	1	0000	01da	ffff	ffff	C, DC, Z, OV, N	1, 2, 3, 4	
DECFSZ f, d, a	Decrement f, Skip if 0	1 (2 or 3)	0010	11da	ffff	ffff	None	1, 2, 3, 4	
DCFSNZ f, d, a	Decrement f, Skip if Not 0	1 (2 or 3)	0100	11da	ffff	ffff	None	1, 2	

Auch einige Verzweigungsbedingungen, die eigentlich zu den Control-Operations gehören, werden in der Übersichtstabelle den Byte-orientierten Befehlen zugeordnet, wenn die Sprungbedingung direkt aus der Verarbeitung eines Registers abgeleitet wird (*CPFSEQ*, *DECFSZ*, *DCFSNZ*, ...).

Der Tabelle kann man sehr schnell entnehmen, wie viele Zyklen die Verarbeitung jedes Befehls dauert, und welche Bits des Statusregister durch die Operation beeinflusst werden.

Zu jedem Befehl gibt es natürlich noch eine detaillierte Beschreibung weiter hinten im Data-Sheet.

1.2.6.2 Bit-orientierte Befehle

Bit-orientierte Befehle beeinflussen im Allgemeinen nur ein einzelnes Bit in einem Register.

Wendet man einen solchen Befehl fälschlicherweise auf ein Bit in einem PORT-Register (*am PIN*) an, kann es zu einem ungewollten **R-M-W Effekt** (*read-modify-write*) kommen, weil das Register, welches das betreffende Bit enthält, zuerst komplett gelesen werden muss um modifiziert werden zu können. ([12.1.2 LAT Register & R-M-W](#))

BIT-ORIENTED OPERATIONS								
BCF f, b, a	Bit Clear f	1	1001	bbba	ffff	ffff	None	1, 2
BSF f, b, a	Bit Set f	1	1000	bbba	ffff	ffff	None	1, 2
BTFSC f, b, a	Bit Test f, Skip if Clear	1 (2 or 3)	1011	bbba	ffff	ffff	None	3, 4
BTFSS f, b, a	Bit Test f, Skip if Set	1 (2 or 3)	1010	bbba	ffff	ffff	None	3, 4
BTG f, b, a	Bit Toggle f	1	0111	bbba	ffff	ffff	None	1, 2

Wie bei den Byte-orientierten, gibt es auch hier zwei **Control-Operations** (*BTFSC*, *BTFSS*)

1.2.6.3 Bedingte Sprünge / Verzweigungen

Bei den **Control Operations** gibt es welche mit Bedingungen, die sich auf vorher durchgeföhrte Operationen bezieht. Die vorherige Operation hat dabei ein Bit im Statusregister (*Carry, Negative, Overflow...*) beeinflusst, welches man jetzt prüft, um eine Verzweigungsmöglichkeit im Programm zu realisieren.

CONTROL OPERATIONS								
BC n	Branch if Carry	1 (2)	1110	0010	nnnn	nnnn	None	
BN n	Branch if Negative	1 (2)	1110	0110	nnnn	nnnn	None	
BNC n	Branch if Not Carry	1 (2)	1110	0011	nnnn	nnnn	None	
BNN n	Branch if Not Negative	1 (2)	1110	0111	nnnn	nnnn	None	
BNOV n	Branch if Not Overflow	1 (2)	1110	0101	nnnn	nnnn	None	
BNZ n	Branch if Not Zero	1 (2)	1110	0001	nnnn	nnnn	None	

Branch steht hier für „Sprung“ (*des Programmzeigers*). Ein Sprung hat allerdings eine rel. kleine Reichweite, die durch die 8 verfügbaren (*n*) Bits begrenzt wird. (*8 Bit → max. 255 Programmwoerte*)

1.2.6.4 Kontroll Operationen ohne Bedingung

Der Rest der **Control Operations** wird bedingungslos ausgeführt.

CALL	k, s	Call subroutine 1st word 2nd word	2	1110	110s	kkkk	kkkk	None	
CLRWDT	—	Clear Watchdog Timer	1	0000	0000	0000	0100	TO, PD	
DAW	—	Decimal Adjust WREG	1	0000	0000	0000	0111	C	
GOTO	k	Go to address 1st word 2nd word	2	1110	1111	kkkk	kkkk	None	
NOP	—	No Operation	1	0000	0000	0000	0000	None	
NOP	—	No Operation	1	1111	xxxx	xxxx	xxxx	None	4
POP	—	Pop top of return stack (TOS)	1	0000	0000	0000	0110	None	
PUSH	—	Push top of return stack (TOS)	1	0000	0000	0000	0101	None	
RCALL	n	Relative Call	2	1101	lnnn	nnnn	nnnn	None	
RESET	—	Software device Reset	1	0000	0000	1111	1111	All	
RETFIE	s	Return from interrupt enable	2	0000	0000	0001	000s	GIE/GIEH, PEIE/GIEL	
RETLW	k	Return with literal in WREG	2	0000	1100	kkkk	kkkk	None	
RETURN	s	Return from Subroutine	2	0000	0000	0001	001s	None	
SLEEP	—	Go into Standby mode	1	0000	0000	0000	0011	TO, PD	

Unter diese Operationen fallen z.B. der Aufruf und eines Unterprogramms (Subroutine), die Rücksprünge, RESET, Sleep

1.2.6.5 Literal Befehle (Konstanten als Operatoren)

Für Operationen mit konstanten Werten werden die Literal Operations verwendet. Die Befehle beziehen sich dabei immer auf das Arbeitsregister **WREG**, das Bank-Select-Register **BSR** (*Auswahl der Bank im RAM*) oder das File-Select-Register **FSR** (*indirekte Adressierung des RAM*).

LITERAL OPERATIONS								
ADDLW	k	Add literal and WREG	1	0000	1111	kkkk	kkkk	C, DC, Z, OV, N
ANDLW	k	AND literal with WREG	1	0000	1011	kkkk	kkkk	Z, N
IORLW	k	Inclusive OR literal with WREG	1	0000	1001	kkkk	kkkk	Z, N
LFSR	f, k	Move literal (12-bit) 2nd word to FSR(f) 1st word	2	1110	1110	00ff	kkkk	None
				1111	0000	kkkk	kkkk	
MOVLB	k	Move literal to BSR<3:0>	1	0000	0001	0000	kkkk	None
MOVLW	k	Move literal to WREG	1	0000	1110	kkkk	kkkk	None
MULLW	k	Multiply literal with WREG	1	0000	1101	kkkk	kkkk	None
RETLW	k	Return with literal in WREG	2	0000	1100	kkkk	kkkk	None
SUBLW	k	Subtract WREG from literal	1	0000	1000	kkkk	kkkk	C, DC, Z, OV, N
XORLW	k	Exclusive OR literal with WREG	1	0000	1010	kkkk	kkkk	Z, N

Beispiel: Arbeitsregister um 10 erhöhen → ADDLW 10;

1.2.6.6 Datenaustausch zwischen Programm und Datenspeicherbereich

Zu diesem Zweck stehen die **DATA MEMORY ↔ PROGRAM MEMORY OPERATIONS** zur Verfügung. Sie finden beispielsweise bei der Programmierung eines Bootloaders Verwendung, bei dem ein zukünftiger Inhalt des Programmspeichers von der Anwendung, über eine meist serielle Schnittstelle, sequenziell empfangen und im RAM zwischengespeichert wird.

So empfangene Programmblöcke werden anschließend auf Übertragungsfehler überprüft und in den Programmspeicher übertragen.

DATA MEMORY ↔ PROGRAM MEMORY OPERATIONS								
TBLRD*		Table Read	2	0000	0000	0000	1000	None
TBLRD+*		Table Read with post-increment		0000	0000	0000	1001	None
TBLRD-*		Table Read with post-decrement		0000	0000	0000	1010	None
TBLRD+*		Table Read with pre-increment		0000	0000	0000	1011	None
TBLWT*		Table Write	2	0000	0000	0000	1100	None
TBLWT+*		Table Write with post-increment		0000	0000	0000	1101	None
TBLWT-*		Table Write with post-decrement		0000	0000	0000	1110	None
TBLWT+*		Table Write with pre-increment		0000	0000	0000	1111	None

1.3 Integrierte Entwicklungsumgebung MPLAB-X

Die vom Hersteller für die Entwicklung von PIC - Programmen kostenlos zur Verfügung gestellte Entwicklungsumgebung nennt sich MPLAB-X IDE.

Sie basiert auf der NETBEANS Umgebung und kann als CrossOver (*X*) Software auf den gängigsten Betriebssystemen eingesetzt werden.

Achtung: Das "Integrierte-Develeopment-Environment" enthält keine Assembler oder Compiler.

1.3.1 Assembler

Ab der MPLAB-X IDE v5.40 ist kein Assembler mehr integriert. Für 8-Bit PICs wird der Assembler **pic-as** verwendet, welcher beim XC8 Compiler dabei ist.

Ein Assembler ist kein "Compiler" im eigentlichen Sinne. Der Assembler kann meist nur für eine Familie von Controllern verwendet werden und "ersetzt" die für den Programmierer verständlicheren Assembler-Befehle durch für den Controller verständliche Maschinensprache. Von Erweiterungen wie z.B. Makros abgesehen, erfolgt dieses "Ersetzen" 1:1. Das heißt, aus einer Assembler-Anweisung wird exakt eine Anweisung in Maschinensprache. Alle Assembler Befehle und der zugehörige Maschinencode können dem Datenbuch des Controllers entnommen werden.

1.3.2 C Compiler

Im Gegensatz zu einem Assembler muss der C Compiler die in seiner "Hoch-Sprache" vorgegebenen Strukturen in Maschinensprache "übersetzen". Aus einer einzigen Anweisung in einer Hochsprache können sehr viele Anweisungen in Maschinensprache werden. Zusätzlich werden durch den Compiler meist Ressourcen des Controllers belegt. (*für den Programmierer nicht immer leicht ersichtlich*)

Der C Compiler erzeugt aus jeder Sourcecode-Datei (*.c) eines Projektes eine Object-Datei (*.o)

1.3.2.1 Mögliche C Compiler für PIC18 Controller

1.3.2.1.1 XC8

Microchip Technology Inc. bietet die XC-Compilerfamilie für 8-, 16- und 32bit Controller an. (*XC8, XC16, XC32*) Alle diese C-Compiler müssen bei Bedarf zusätzlich zur MPLAB X IDE installiert werden. Wie die IDE sind alle Compiler für Windows, Linux und MAC OS erhältlich.

In diesem Dokument wird hauptsächlich auf die Verwendung des XC8 Compilers eingegangen. War der durch die ersten **XC8 „FREE“** Versionen generierte Maschinencode aufgrund der in der FREE Version total abgeschalteten Optimierung nicht wirklich akzeptabel, so hat sich das ab der Version 1.20 stark verbessert. Der XC8 Compiler ist für alle 8-Bit PICs (PIC10, 12, 16, 18) verwendbar.

1.3.2.1.2 Microchip C18

Der ältere Microchip **C18** Compiler, der für den 18FxxK22 noch sehr gut verwendet werden kann, erzeugt unter Umständen besseren/schnelleren Code, wird aber nicht mehr gepflegt, und ist somit auch nicht mehr für die aktuellsten PIC18 verwendbar. Die Einschränkungen der „Evaluation“ oder „FREE“ Versionen waren hier praktisch nicht relevant. Einige C18 spezifische Dinge sind im Anhang unter [13.4 Microchip C18 Compiler](#) zu finden.

1.3.2.1.3 SDCC

Eine weitere Alternative wäre noch der OpenSource Compiler **SDCC** (*Small Device C Compiler*). Durch die geringe Integration in die IDE ist das Debuggen hier etwas schwierig.

1.3.3 Linker

Der im XC8 Compiler enthaltene Linker ist dafür zuständig, aus den einzelnen Komponenten des Projekts, bestehend aus verschiedenen Asssembler Files, bzw. den vom Compiler erzeugten Object Files der C Dateien und den Bibliotheken eine einzige, ausführbare Datei zu machen.

Die Doku für den Linker findet man in den Handbüchern für XC8 C-Compiler und Assembler.

1.3.4 Installation der Entwicklungsumgebung

Wie schon bei den Vorbemerkungen erwähnt, ändern sich die Komponenten der Entwicklungsumgebung recht rasant. Während der Erstellung dieses Dokumentes wurden einige verschiedene Versionen der IDE und der Compiler eingesetzt, die immer wieder „Überraschungen“ enthielten. Inzwischen scheint sich das System einigermaßen stabilisiert zu haben und es müsste möglich sein die später folgenden Übungen mit den jeweils aktuellen Tools durch zu arbeiten, auch wenn dann immer mal wieder kleine Unstimmigkeiten zu dieser Dokumentation auftreten werden.

Aktuelle Versionen findet man auf der Herstellerseite, wenn man deren Namen einfach mit dem Namen des gesuchten Tools ergänzt. Die Anfrage wird dann entsprechend umgeleitet.

MPLABX IDE → www.microchip.com/MPLABX

XC8 Compiler → www.microchip.com/XC8 (*enthält auch den Assembler pic-as*)

Hier findet man dann die neuesten Versionen der Tools für das jeweilige Betriebssystem. Sollte man ältere Versionen benötigen, kann man diese möglicherweise im Archiv finden.

<https://www.microchip.com/en-us/tools-resources/archives>

1.3.4.1 Installationshinweise

Die Installation erfolgt auf die für das jeweilige Betriebssystem übliche Weise.

1.3.4.1.1 Installationshinweis Windows

Die Entwicklungsumgebung kann auf verschiedenen Betriebssystemen genutzt werden. Auf einigen davon unterliegen Pfad- und Dateinamen stärkeren Beschränkungen als bei Windows.

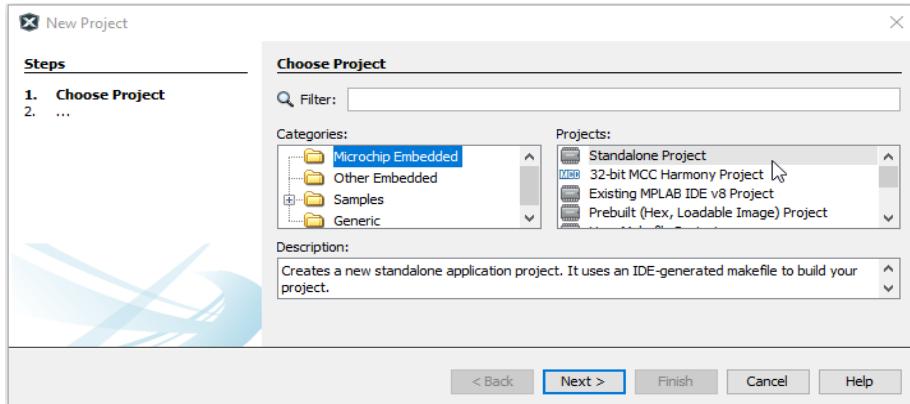
Dies kann in Einzelfällen unter Windows zu **Problemen** führen, wenn diese Pfade/Namen **Leerzeichen, Sonderzeichen, Umlaute usw.** enthalten. (`../Programme (x86)`).

2 Grundkonfiguration neuer Projekte

2.1 Neues Projekt anlegen

MPLAB X IDE über die übliche Vorgehensweise des entsprechenden Betriebssystems starten.

Nach Aufrufen des Menüs **File->New Project...** (oder über entsprechenden Button der Menüleiste) wird man automatisch schrittweise durch den Prozess der Projekterstellung geführt.



Zuerst wird die Art des Projektes abgefragt. Hier kann man komplett neue Projekte anlegen, Beispielprojekte öffnen oder auch Projekte aus der „alten“ IDE (*MPLAB ohne X*) importieren.

Das wichtigste ist natürlich zu lernen, wie ein neues Projekt angelegt wird. Hat man das einigermaßen verstanden, dann ist es auch absolut kein Problem ein vorhandenes Projekt der alten Form neu anzulegen ohne die Importfunktion zu benutzen.

Für das erste Beispielprojekt hier bitte „**Microchip Embedded**“ und „**Standalone Projekt**“ auswählen und mit „**Next >**“ bestätigen.

2.1.1 Select Device / Tool

Als nächstes wird der zu verwendenden Prozessortyp (**PIC18F24K22**) unter „**Device**“ ausgewählt. Vor der Auswahl kann diese optional unter „**Family**“ noch etwas eingeschränkt werden.

Im selben Dialog kann auch noch ein zur Verfügung stehendes Programmiergerät bzw. Debugger oder auch ein PC-seitiger Simulator ausgewählt werden, welcher im Projekt Verwendung finden soll. Im µC -Labor MCON steht das PICKit3 zur Verfügung, welches hier ausgewählt werden kann.

Die hier getroffenen Vor-Einstellungen sind nicht für alle Zeiten bindend können im später bei der Bearbeitung des Projektes verändert, oder ergänzt werden. Beispielsweise kann eine weitere Konfiguration angelegt werden, um mit dem integrierten Simulator bestimmte Details zu überprüfen.

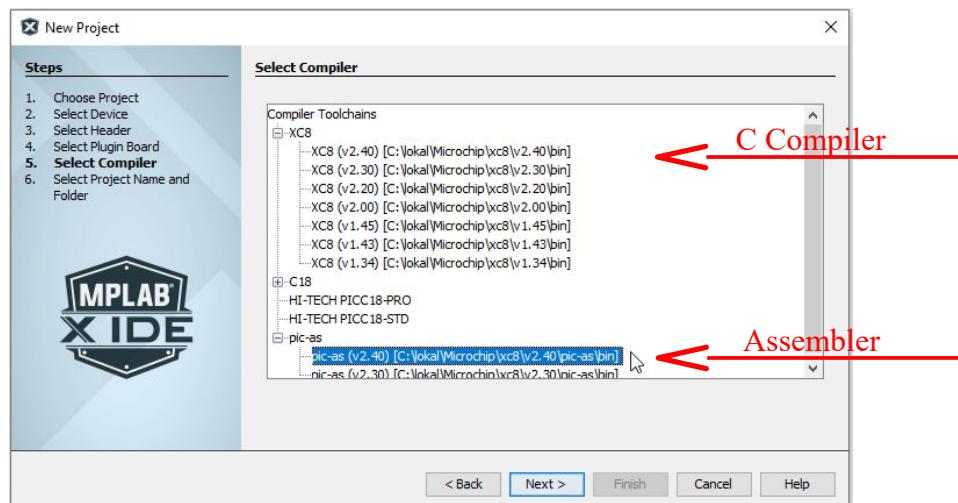
2.1.2 Select Header (nicht mehr automatisch angezeigt ab MPLAB X v3.x)

PICs mit niedriger Pin-Anzahl haben oft keine ICD (*In-Circuit-Debugger*) Module. Die zusätzliche Schnittstelle würde zu viele der verfügbaren Pins belegen, die dann nicht für das eigentliche Programm zur Verfügung stehen. Zum Debuggen dieser PICs benötigt man zusätzliche Hardware.

Verfügt der ausgewählte PIC über ein integriertes ICD-Modul, entfällt dieser Schritt ...

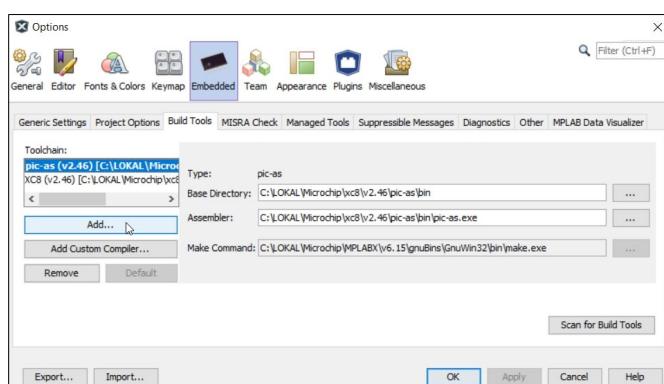
2.1.3 Select Compiler

Im nächsten Dialog erfolgt die Abfrage des zu verwendenden Compilers, bzw. Assemblers.



Alle Beispiele in dieser Anleitung beziehen sich auf den C-Compiler XC8 und den Assembler pic-as der im XC8 enthalten ist. Sind auf dem verwendeten Computer wie in der obigen Abbildung mehrere Versionen davon installiert, verwendet man am besten immer die aktuellste.

MPLAB-X „findet“ normalerweise alle verfügbaren Compiler und Assembler selbstständig. Wird beispielsweise der pic-as nicht angezeigt, oder möchte man manuell weitere hinzufügen oder entfernen, dann kann man das mit Hilfe des Menüs Tools->Options->Build Tools tun.



Falls nur der C-Kompiler XC8 angezeigt wird, dann kann man sich dessen Base Directory (z.B. .../Microchip/xc8/v2.46/bin) merken und mit dem Add... den Assembler hinzufügen.

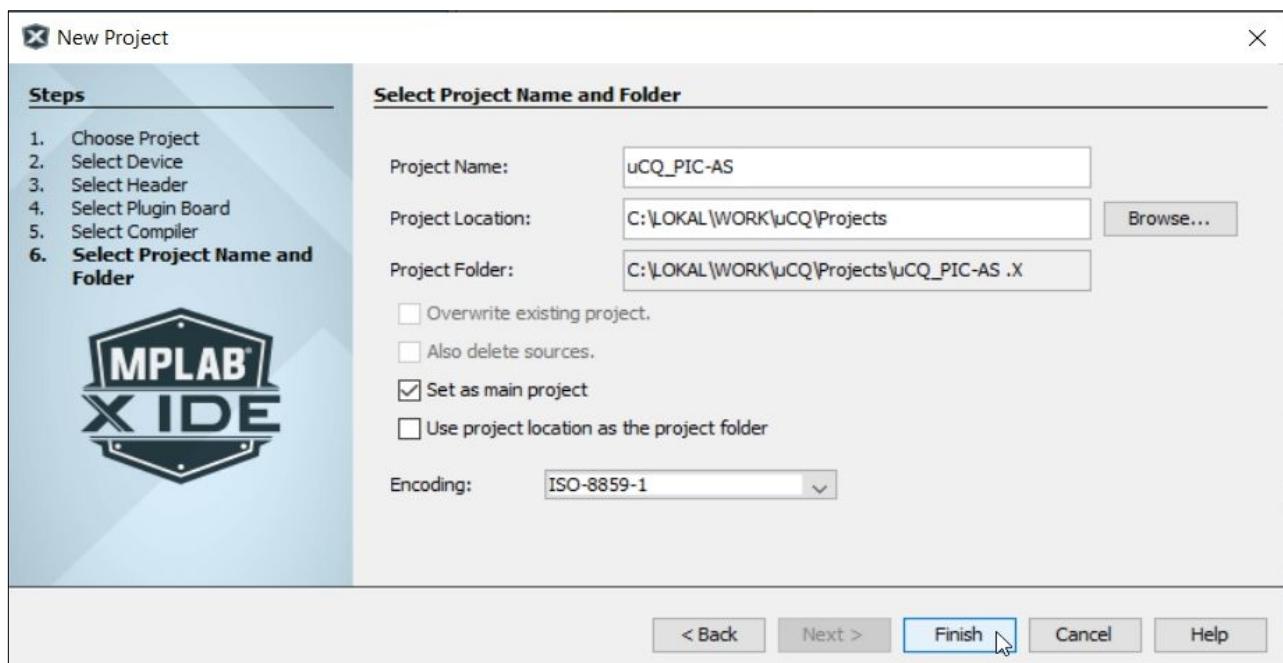
*Dessen Base Directory befindet sich fast am gleichen Ort und unterscheidet sich nur durch ein eingeschobenes /pic-as/.
(.../Microchip/xc8/v2.46/pic-as/bin).*

Für die ersten Projekte schadet es auf keinen Fall, wenn diese auch in der Sprache Assembler durchgeführt werden. Assembler ermöglicht einen wesentlich engeren Bezug zur Hardware des Controllers als die Sprache „C“. Assembler-Anweisungen entsprechen direkt der Maschinensprache des Controllers. Somit hat der Programmierer die volle Kontrolle (*und Verantwortung ;)* für jeden einzelnen Schritt, den das Programm durchläuft und auch die volle Kontrolle über alle integrierten Hardwaremodule.

Nachdem die grundsätzliche Arbeitsweise des Controllers verstanden wurde, kann man dann dazu übergehen auch in einer abstrakteren Sprache wie „C“ zu programmieren. Optimal ist es wenn man dann noch versteht, dass eine einfache Anweisung in „C“ sehr viel Maschinencode erzeugen kann.

2.1.4 Projektnamen und Speicherort vergeben

Der letzte Schritt der Projekterstellung besteht in der Vergabe eines geeigneten Namens und Speicherortes für das neue Projekt.



Leider kann es unter Windows Betriebssystemen bei vielen Programmen zu Problemen kommen, wenn auf Dateien/Projekte über einen Netzwerkpfad (`\hs-ulm.de\fs\...`) zugegriffen wird.

Auch auf den „Desktop“ auf den Pool-Rechnern der Hochschule wird auf diese Weise zugegriffen !!!

Das Projekt sollte deshalb zunächst möglichst auf einem lokalen Laufwerk (auch USB...) oder zumindest einem „gemapten“ Laufwerk gespeichert werden. (erkennbar am Laufwerksbuchstaben)

Wenn das Projekt problemlos plattformübergreifend verwendbar sein soll, ist es auch ratsam, für Pfade und Dateinamen nur ASCII Zeichen zu verwenden. (keine Umlaute, Klammern, Leerzeichen, ...)

Da das neu erstelle Projekt höchst wahrscheinlich auch gleich im Anschluss bearbeitet wird, ist es empfehlenswert die Checkbox „Set as main project“ zu aktivieren !

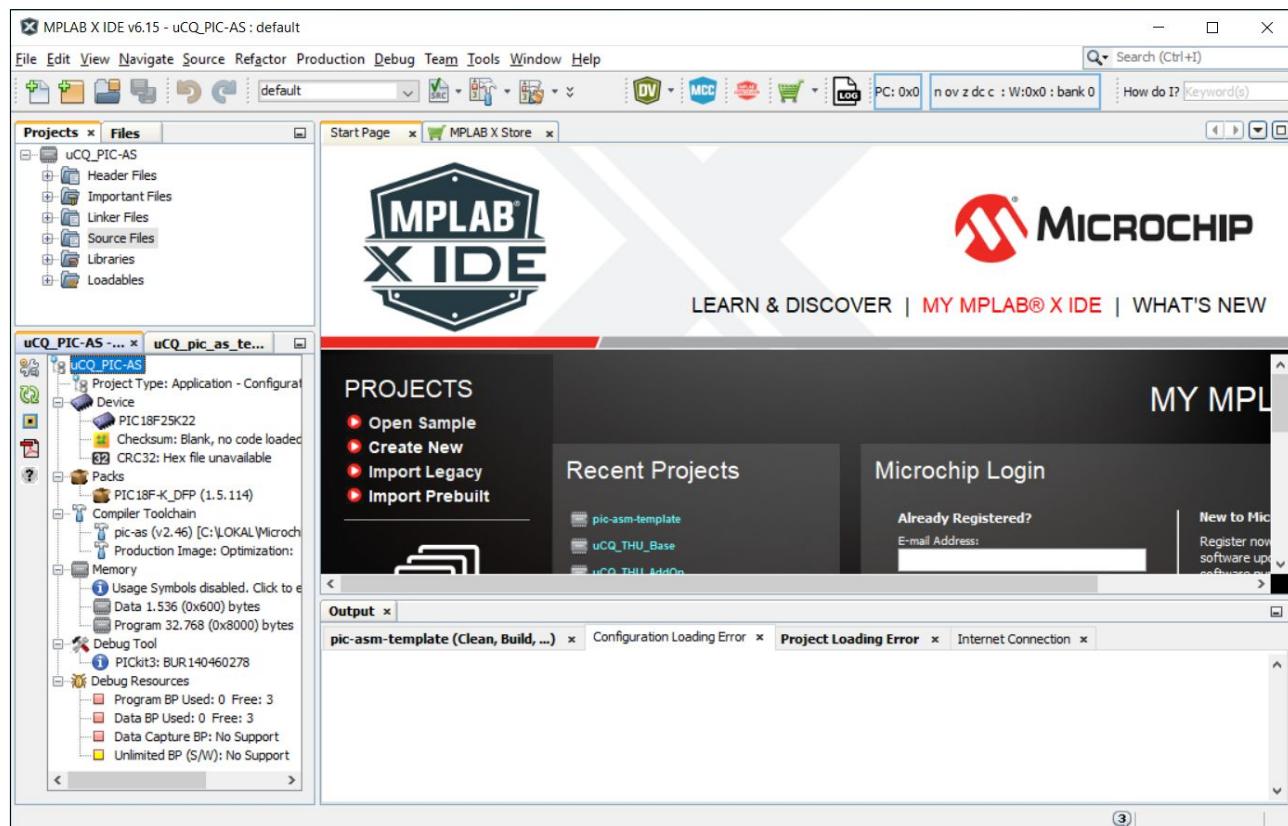
2.2 Erster Überblick über IDE

Die verschiedenen Fenster der IDE können prinzipiell nach persönlichem Geschmack angeordnet werden. Die einzelnen Abschnitte können durch Ziehen mit der Maus in ihrer Größe verändert und verschoben werden.

Über die Icons am oberen rechten Rand der können die Fenster minimiert, geschlossen oder gegebenenfalls wieder eingebettet werden. Standardmäßig startet die Oberfläche ähnlich wie unten abgebildet.

Links oben an der Seite befindet sich das Fenster „**Projects**“, welches eine übersichtliche Navigation zwischen verschiedenen Projekten und deren Quelltextdateien ermöglicht. Die beiden Fenster „**Files**“ und „**Classes**“ im Hintergrund sind für den Anfänger eher verwirrend und können zunächst ignoriert werden.

Links unten ist in der Abbildung unten das Fenster „**Dashboard**“ sichtbar, welches einige Informationen zum Projekt zusammengefasst darstellt.



Der größte Bereich oben wird später vom Editor-Fenster eingenommen welches vorerst noch keine Quelltextdatei anzeigt, da das neue Projekt noch keine enthält.

Unter dem Editorbereich sind die „**Output**“ Fenster angeordnet in denen die Meldungen der verschiedenen verwendeten Tools (*Compiler, Programmiergerät, Debugger* ...) angezeigt werden. Auch die Resultate von Suchanfragen erscheinen hier.

Wie die Anordnung der Fenster ist auch die Tool-bar konfigurierbar und kann den persönlichen Vorlieben angepasst werden. Ein Klick mit der rechten Maustaste in einem freien Bereich der Tool-bar öffnet einen Dialog über den man der Tool-bar neue Element hinzufügen, oder auch entnehmen kann.

Die Anordnung und Ansicht der Fenster kann gegebenenfalls im Menü *Window | Reset Windows* wieder auf die Standardeinstellungen zurückgestellt werden.

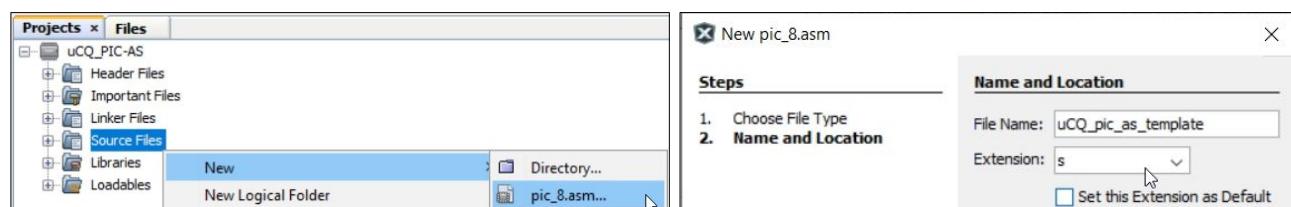
2.3 Grundgerüst eines µC Programms in Assembler

Im folgenden (*bis Kapitel 2.6 Configuration Bits*) wird der Aufbau einer solchen eigenen Basis für einen PIC18FxxK22 Controller beschrieben. Das Resultat ist auch im Anhang **TODO** zu finden.

2.3.1 Assembler File von der IDE erstellen lassen

Als Basis für eine neue Assembler Datei kann man von der IDE eine Datei erstellen lassen, die schon wichtige Elemente einer Assemblerdatei beinhaltet. Bis zur Version 6.20 der IDE und v2.46 des XC8 Compilers ist dies im Vergleich zu früheren Assemblerversionen allerdings sehr rudimentär umgesetzt.

Für 8-Bit PICs gibt es z.B. die Möglichkeit im Popupmenü des Projektfensters unter **New->pic8.asm** dem Projekt eine neue Datei hinzuzufügen, welche ein für den Anfänger eher unverständliches Elemente schon beinhaltet und auch eine kurze Erklärung dazu bietet.



Leider handelt es sich bei diesem Assembler-Template auch eher um eine Datei, welche einem C-Projekt hinzugefügt werden kann.

Die Codezeile mit „**psect**“ ist allerdings unbedingt erforderlich und der Rest wird in den folgenden Kapiteln dann angepasst.

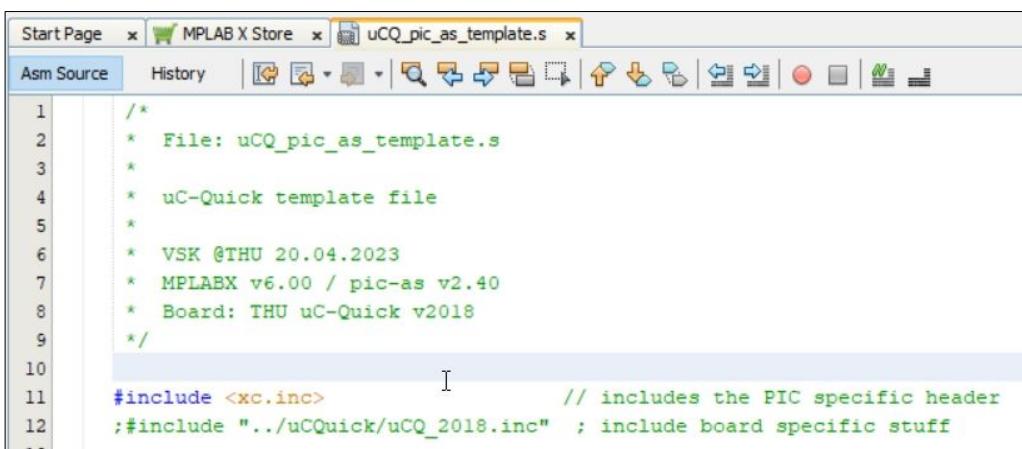
Als Erstes muss man die entsprechende „**psect**“ Zeile für die richtige PIC-Familie aktivieren, indem man die Kommentarzeichen entsprechend setzt.

2.3.2 Kommentare (; , //, /*.....*/)

Ein wichtiger, aber oft sträflich vernachlässigter Teil eines Programms sind die Kommentare. Im von der IDE erstellten Assembler File kann man relativ leicht erraten, dass es sich bei allem hinter einem Semikolon, um einen Kommentar handelt.

In Code unten sind noch weitere Möglichkeiten zu sehen. „//“ und „;“ gelten bis zum Zeilenende. Mit „/*“ wird ein Kommentar begonnen und mit „*/“ beendet. (auch über mehrere Zeilen)

Der Kopf jeder Datei sollte auf jeden Fall einen Kommentar enthalten um was es sich eigentlich handelt, wer die Datei erstellt hat und wann die Datei erstellt wurde. Auch innerhalb der anderen Abschnitten sollte nicht aus Faulheit mit Kommentaren gespart werden.



```
Start Page x MPLAB X Store x uCQ_pic_as_template.s x
Asm Source History | [File] [Edit] [Search] [Project] [Build] [Run] [Help]
1 /*
2 *   File: uCQ_pic_as_template.s
3 *
4 *   uC-Quick template file
5 *
6 *   VSK @THU 20.04.2023
7 *   MPLABX v6.00 / pic-as v2.40
8 *   Board: THU uC-Quick v2018
9 */
10
11 #include <xc.inc>           I           // includes the PIC specific header
12 ;#include "../uCQuick/uCQ_2018.inc" ; include board specific stuff
```

Kommentare helfen dem Programmierer schon beim Schreiben des Programms sich genau darüber klar zu werden und zu formulieren, was da eigentlich getan werden soll.

Spätestens beim Überarbeiten eines Programms sind sie von sehr großer Hilfe, da meistens nicht mal der ursprüngliche Entwickler auf den ersten Blick in ein komplexeres Programm erkennen kann was da getan wird und warum genau die jeweiligen Befehle verwendet wurden.

2.3.3 Assembler Direktiven

2.3.3.1 #include Direktive

#include <xc.inc> und #include "../uCQuick/uCQ_2018.inc"

Spezifiziert eine Datei deren Inhalt an dieser Stelle eingefügt werden soll. Handelt es sich bei der Include-Datei um eine vom System bereitgestellte Datei, dann kennzeichnet man das dadurch, dass der Dateiname in spitze Klammern <System_eigen.inc> eingebunden wird.

Benutzerdefinierte Dateien werden in Anführungszeichen gesetzt “**Benutzer.inc**“. Es können auch **Pfade** zu der einzufügenden Datei, ausgehend vom Projektverzeichnis, mit angegeben werden.

Müssen vom aktuellen Verzeichnis Ebenen im Verzeichnisbaum nach oben angegeben werden dann geschieht das durch Verwendung des bekannten “..“. Dies kann durch Aneinanderreihungen auch über mehrere Ebenen realisiert werden (“../../meineLib/xyzLib.inc”)

2.3.3.1.1 Die Funktion von Include-Dateien

In nicht total veralteten Systemen, welche über einen sogenannten Linker verfügen, enthalten Include Dateien **keinen Programmcode** sondern hauptsächlich Definitionen, Deklarationen und Makros, die dem Programmierer die Arbeit erleichtern.

Die Datei **xc.inc**, welche vom pic-as Assembler bereit gestellt wird, bindet ihrerseits lediglich weitere Dateien ein wie **pic18.inc**, die unter anderem wiederum **pic18_chip_select.inc** und diese letztendlich das Include-File des im Projekt spezifizierten Prozessors hinzufügt.

Ein Blick in die Datei .../xc8/v2.46/pic/include/proc/pic18f25k22.inc zeigt viele Zeilen wie:

```
4066 // Register: LATB
4067 #define LATB LATB
4068 LATB                                equ 0F8Ah
4069 // bitfield definitions
4070 LATB_LATB0_POSN                      equ 0000h
```

Dem Bezeichner LATB wird hier der Wert 0x0F8A zugewiesen, was der Adresse von LATB im Speicher entspricht.

Include-Dateien haben unter anderem eine Art Übersetzer Funktion. Der Assembler kann mit Registernamen wie LATB nichts anfangen. Er braucht die Adressen der Register, welche bei unterschiedlichen Controllern auch unterschiedlich sind. Ohne die prozessor-spezifischen Include-Dateien müsste der Programmierer immer die Adressen verwenden. (*auch EQU ist eine Direktive*). Nur die Include Datei ermöglicht die Benutzung von Registernamen wie z.B. LATB.

Projektspezifische Include-Dateien wie z.B. .../uCQuick/uCQ_2018.inc enthalten weitere Definitionen um den Code noch eine Stufe weiter zu abstrahieren.

```
41 ; -----
42 #define nLED_1      LATB,LATB2,ACCESS
43 #define LED_1_TRI   TRISB,TRISB2,ACCESS
44 #define nLED_2      LATB,LATB3,ACCESS
```

#define nLED_1 LATB, 2, ACCESS (*#define ist eine Preprocessor Direktive* ...)
beispielsweise ermöglicht das Einschalten von LED_1 über ein relativ leicht verständliches bcf nLED_1. Ohne die beiden Include-Dateien müsste der Code bcf H'0F8A', 2, 0 lauten

Für die in den Include-Dateien über die Direktiven EQU oder #define eingeführten programmierer-freundlichen **BEZEICHNER** werden bewusst GROSSBUCHSTABEN verwendet, um zu signalisierten (*einem anderen Programmierer*) dass es sich hierbei nicht um Variablen oder Unterfunktionen handelt.

2.3.3.1.2 Include Guards

Sogenannte "Include Guards" verhindern das mehrfache Einfügen des Textes einer Includedatei in eine Sourcedatei. Dies würde durch z.B doppelte oder mehrfache Definitionen zu Fehlern bei der Kompilierung führen. Die Anweisungen "#ifndef" (*if not defined*) in der ersten und "#endif" in der letzten Zeile gehören zusammen. Alles was dazwischen liegt wird nur dann eingefügt, wenn die Definition noch nicht vorhanden ist.

```
#ifndef _DATEINAME_INC
#define _DATEINAME_INC
....
#endif
```

Beim ersten Einfügen wird durch die Präprozessor Direktive "#define _DATEINAME_INC" in der zweiten Zeile ein Makro erstellt, das ein weiteres Einfügen verhindert, da das Makro bei weiteren Include-Versuchen schon (*nicht mehr nicht*) definiert ist. Die Konvention für den Namen des Makros ist hier der Dateiname mit einem Unterstrich beginnend und in Großbuchstaben.

2.3.3.2 Psect Direktive

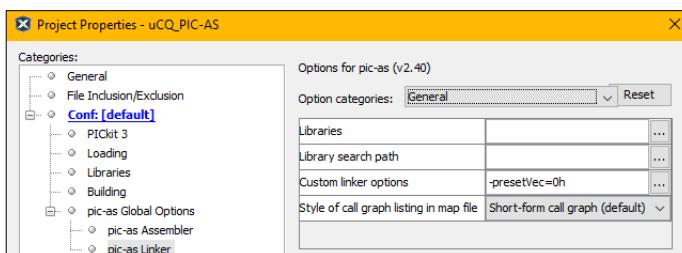
Wer klassische Assembler kennt, bei denen Programm und Datenadressen im Code deklariert wurden, vergisst dieses Vorgehen besser so schnell als möglich komplett. Moderne Assembler verfügen, wie auch C-Compiler über einen Linker, der verschiedene Objekt-Dateien zusammenfügen kann. Fest vorgegebene Adressen würden dies erheblich erschweren, sind meistens völlig unnötig und die Dateien mit festen Adressen sind fast unmöglich wieder zu verwenden.

2.3.3.2.1 Benutzer definierte Sections

Die Psect Direktive deklariert eine Program- oder Data-Section, der nur in Ausnahmefällen eine feste Adresse zugeordnet wird. Eine dieser wenigen feststehende Adressen ist z.B. die Adresse, auf die der Programmzeiger zu Beginn, oder nach einem Reset zeigt. Für das Main-Template eines Assemblerprogramms ändern wir deshalb den Abschnitt ab zu:

```
11 #include <xc.inc>           // includes the PIC specific header
12 ;#include "../uCQuick/uCQ_2018.inc" ; include board specific stuff
13
14 PSECT resetVec,class=CODE,reloc=2    ; define "-presetVec=0h" in linker options
15 resetVector:
16     goto Init
17
18 PSECT code
19 Init:
20 ;   initialisation code (runs only one time)
21 MainLoop:
22 ;   code for main programm (do this forever)
23     goto MainLoop
24 END      resetVector
```

In Zeile 14 wird eine Section mit dem Namen **resetVec** definiert. Dieser wird unter **Custom linker options** der Projekteigenschaften mit **-presetVec=0h** (**-p<section>**) die Adresse 0h zugewiesen.



Die einfachste Möglichkeit den Dialog mit den Project Properties zu öffnen bietet das **Drop-Down-Menü** in der Symbolleiste der IDE. **Customize...** öffnet den Dialog.

2.3.3.2.2 Psect Flags

Die Definition einer Psect umfasst mindestens noch **Flags**, welche die Linker-class bestimmen und alle Flags, die vom Defaultwert abweichen. Für PIC18 muss deshalb das **Reloc Flag** auf **2** gesetzt werden. ([siehe 2.3.1Assembler File von der IDE erstellen lassen bzw. Benutzerhandbuch](#))

2.3.3.2.3 Vordefinierte Sections

Der gesamte Programmcode und alle Datenobjekte die im Controller untergebracht werden müssen, bedingen der Zuordnung zu einer Psect. Diese kann wie *resetVec* oben selbst definiert werden, oder man greift für den Assembler vordefinierte Sections zurück.

Dazu muss <xc.inc> in der Sourcedatei eingebunden sein!

Das Benutzerhandbuch des Assemblers enthält die komplette Liste vordefinierter Sections, wie **code**, **edata**, **data**, **udata** und **udata_acs** (PIC18). Die vordefinierten Sections beinhalten auch schon die Linker-class und ggf. erforderliche Flags wie das Reloc Flag.

Im Programm-Speicherbereich nach dem **Reset-Vektor** an der Adresse **0h**, folgen beim PIC18 noch die beiden **Interrupt-Vektoren** an den Adressen **8h** und **18h**. Somit kann das eigentliche Programm im Allgemeinen nicht in der Section **resetVec** bzw. direkt im Anschluss daran liegen, weil der Platz für die Interruptvektoren freigehalten werden muss.

Wo sich das Haupt-**Programm** im Speicher befindet ist eigentlich egal. Deshalb legt man dieses mitsamt der Initialisierung des Systems in eine neue **Psect** namens **code**, welche keine fest vorgegebene Speicheradresse hat und vom Linker beliebig platziert werden kann.

An der Adresse des Reset-Vectors steht gewöhnlich nur ein Sprungbefehl (goto Init) zu der Adresse, an welcher der Linker das eigentliche Programm platziert hat.

2.3.3.3 End Direktive

Um die Warnung [:0:: warning: \(528\) no start record; entry point defaults to zero](#) zu vermeiden, sollte am Ende des Programms in der Hauptdatei des Projektes (*und nur in dieser*) die Direktive END mitsamt der Angabe des Einstiegspunktes (*hier resetVector*) des Programms stehen. Unterhalb dieser Zeile sollte nichts weiteres in der Main-Datei stehen.

2.3.3.4 Variablen

udata / udata_acs (udata_ovr, udata_shr, idata)

GPR_VAR UDATA / ACS_VAR UDATA_ACST sind Anweisungen welche dem Assembler / Linker vorgeben in welchem Bereich des Datenspeichers nachfolgende Variablen angelegt werden sollen und ob diese bei Programmstart initialisiert werden müssen (*udata -> uninitialized, idata ...)*

Die verschiedenen Datenspeicher-Bereiche ergeben sich bei 8bit PICs aus der Aufteilung des RAM in Banks, welche über ein zusätzliches **BANK-Selection-Register** adressiert werden. Dieses „**BSR**“ muss bei jedem Zugriff auf eine RAM Adresse neu eingestellt werden, wenn die Adresse in einer anderen BANK liegt als die Adresse auf welche beim vorhergehende Befehl zugegriffen wurde.

Der Zugriff auf einen bestimmter Bereich, welcher **Access RAM** genannt wird kann ohne Umwege über das BSR erfolgen. Fast alle „**Special-Function-Register (SFR)**“ der PIC18 liegen deshalb in diesem Bereich um einen schnellen Zugriff zu ermöglichen.

Den „**General-Purpose-Register (GPR)**“ Teil des Access RAM kann man für eigenen Daten nutzen, auf die im Programm häufig zugegriffen wird.

GPR_VAR und ACS_VAR sind für den Assembler bedeutungslose, willkürlich vergebene Label.

2.3.4 Assembler Direktiven

Neben dem eigentlichen Programmcode braucht der Assembler noch weitere Anweisungen von denen einige in den Templates auftauchen. Eine komplette Liste kann man sich im Handbuch für den Assembler/Linker anschauen. Im folgenden werden die im Template verwendeten Direktiven in der Reihenfolge ihres Auftauchens **kurz erklärt**. Für genauere Erklärungen und weitere Optionen ist der **User-Guide** des Assemblers gedacht.

2.3.4.1 list p=

(list – LISTING OPTIONS)

list p=18f25k22

Es gibt mehrere List Direktiven. **list p=** setzt den Prozessortyp. Dieser wird allerdings auch von der IDE gesetzt und dem Assembler mitgeteilt. Stimmen die beide nicht überein kommt eine Warnung.

2.3.5 Aufsplittung des Assembler Templates in mehrere Dateien

Um die Übersichtlichkeit und Modularität zu erhöhen, kann man das mitgelieferte Template in ein „**Config**“-Template, ein „**Main**“-Template, ein „**Interrupt**“-Template und ein „**eeprom**“-Template aufsplitten. Die Interrupt und eeprom Templates müssen nur bei Bedarf in das Projekt eingefügt werden und werden auch erst später in den entsprechenden Kapiteln behandelt.

Im Anhang 13.2 Assembler Templates sind entsprechende Dateien zu finden. Diese können als neues Gerüst für eigene Projekte verwendet werden.

2.3.5.1 Configuration Template

Die Datei **uCQ_config.asm** enthält nur noch den Abschnitt mit den „**CONFIG**“ Anweisungen. Diese werden zusätzlich mit Kommentaren und der Angabe der jeweiligen Optionen ergänzt.

2.3.5.2 Main Template

Der Code aus dem Template, der für jedes Programm unbedingt erforderlich ist, wird in eine neue Datei namens **uCQ_main.asm** geschrieben. Hier handelt es sich vor allen um den Reset-Vektor mit dem Sprungbefehl (*goto*) zum Startlabel *Init* (*Start*) des Programms. Dazu kommen die Beispiele für Variablendefinitionen und die Befehle für die Frequenzeinstellung des (*internen*) Oszillators.

2.3.6 Aufteilung der Hauptschleife in einmalige Initialisierung und Main-Loop

Der direkt auf das Startlabel folgende Code muss in der Regel nur ein einziges Mal beim Start des Programms durchlaufen werden. Er enthält die **Initialisierung** der zu benutzenden Pins und Hardware-Module. Danach folgt eine Endlosschleife in der das **Hauptprogramm** läuft.

Bei **Assembler Projekten** kann dies durch ein zusätzliches Label „**Main**“, nach der Initialisierungsphase, realisiert werden. Der Sprung am Ende wird von „**goto Start**“ auf „**goto Main**“ abgeändert.

```
list      p=PIC18F25K22      ; list directive to define processor
#include <p18f25k22.inc>          ; processor specific definitions (LATB EQU H'0F8A')
#include "../uCQuick/uCQ_2013.inc" ; project specific (#define nLED_1 LATB,LATB2,ACCESS)

GPR_VAR    udata      ;-----
;MYVAR_1    res       1      ; User variable linker places

ACS_VAR    udata_acs ;-----
counter_1  res       1      ; variable in ACCESS RAM

RES_VECT   code      0x0000 ; processor reset vector-----
      goto  Init      ; go to beginning of program (initialization)

MAIN_PROG  code      ; let linker place main program-----
Init
      movlw  0x00
      movwf  OSCCON
      bcf    LED_1_TRI
Main
      incfsz counter_1
      goto  Main
      btg    nLED_1
      goto  Main
      END
```

Das Template enthält ein kleines Testprogramm, welches eine LED blinken lässt. So kann man sehr leicht erkennen, ob das Programm und der PIC auch richtig funktionieren.

2.3.7 CONFIG Direktive

Mit der Config Direktive werden die Einstellungen der **Configuration Bits** für den PIC vorgenommen. Diese Konfiguration wird schon zum Zeitpunkt des Aufspielen des Programms auf den Controller und nicht erst bei der Ausführung des Programms eingestellt.

Ein einleuchtendes Beispiel dafür sollte die Konfiguration des Ausführungstaktes sein. z.B.:
CONFIG FOSC = INTIO7

Wenn kein externer Takt vorhanden ist, dann muss schon vor Ausführung des Programms festgelegt worden sein, dass der interne Takt benutzt werden soll, da sonst keine einzige Befehlszeile ausgeführt werden könnte. (*Also auch keine welche den internen Taktgeber aktivieren könnte*)

2.3.7.1.1 code

RES_VECT CODE 0x0000 ist eine Anweisung an den Linker zur Platzierung von Code im Programmspeicher. Es gibt verschiedene Fälle in denen Code genau an einer bestimmten Adresse stehen muss.

Zunächst ist das der Programm-Code der ausgeführt werden soll, wenn der PIC eingeschaltet, oder ein **Reset** durchgeführt wird. In beiden Fällen wird der Zeiger auf die Adresse des auszuführenden Befehls im Programmspeicher auf Null gestellt. Durch „**CODE 0x0000**“ erreicht man das der nachfolgende Code genau an dieser Stelle beginnt.

Der nächste Fall tritt in dem Moment auf, in welchem die Ausführung des Programms durch einen **Interrupt** unterbrochen wird. Der Programmzeiger wird hier wieder auf genau definierte Adressen gestellt, an denen der Interrupt Code stehen muss.

Ein PIC18 verfügt über zwei **Interrupt-Vektoren** (*Adressen*) die auf **0x0008** und **0x0018** liegen. Weil der Bereich zwischen den Vektoren jeweils nur 8 Programmzeilen entspricht, können dort meist nur Sprungbefehle stehen.

2.3.7.1.2 res

Die Anweisung „**MYVAR_1 res 1**“ dient dazu Speicher im RAM in der Größe von einem Byte (**1**) für Daten zu reservieren. Als Name für den Zugriff auf die Daten wurde **MYVAR_1** vergeben.

An welcher genauen Adressen (*die BANK und der genaue Ort in der BANK*) wurde dabei nicht spezifiziert. Wo genau innerhalb einer Bank ist völlig egal und wird komplett dem Assembler / Linker überlassen. Ob AccessBank oder nicht, ergibt sich aus der `udata` / `udata_acs` Direktive.



2.4 Grundgerüst eines C Programms

C Projekte sind normalerweise von Haus aus „*relocatable*“ solange der Programmierer nicht gezielt eingreift. Vieles, was man bei der Assemblerprogrammierung selber im Programmcode machen muss, ist bei Verwendung eines C-Compilers schon vorgegeben, da die Verwendung des Compilers gewisse zusätzlich notwendige Rahmenbedingungen mit sich bringt.

Verschiedene Ressourcen im PIC sind für den Compiler reserviert und dürfen nicht im eigenen Programmcode benutzt werden. Welche das sind und für was der Compiler die benötigt, kann man im User-Guide des Compilers detailliert nachlesen.

Im Gegenzug erledigt der C Compiler einiges im Hintergrund, worum man sich bei der Assembler-Programmierung selber kümmern muss. Dazu gehört z.B. die Behandlung der Reset- und Interrupt-Vektoren.

Hier wird wieder nur eine **sehr kurze Einführung** gegeben. Fertige Templates für PIC18 XC8 Projekte befinden sich im Anhang [13.3 C Templates](#). Für eine Zusammenfassung der wichtigsten C-Sprachelemente siehe [13.5 ANSI C mini-Guide](#). Eine sehr gute Einführung bietet auch der Eintrag [Fundamentals of the C Programming Language \(www\)](#) auf der Microchip Wiki Seite.

2.4.1 Runtime Startup Code

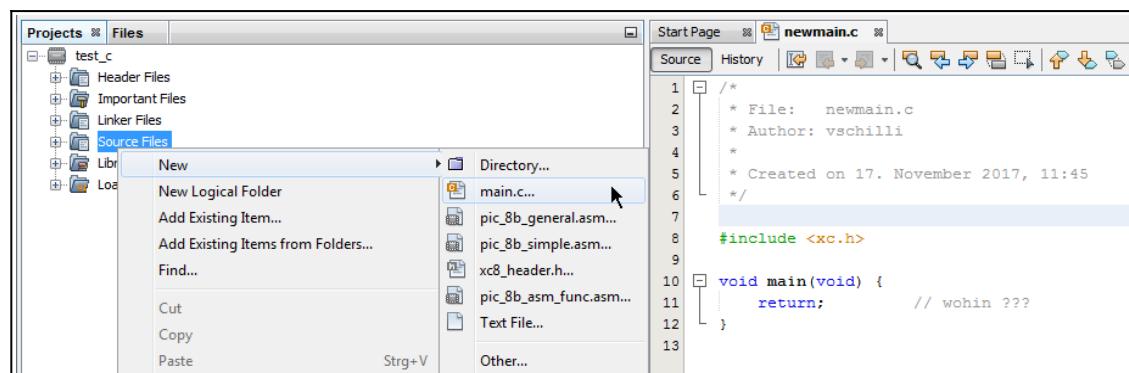
Wie oben schon angesprochen, erfordert es ein C Programm, dass einige Objekte vorinitialisiert werden müssen, bevor der eigene, im Projekt geschriebene, Code mit der Funktion *main()* zur Ausführung kommt. Das wird vom sogenannten „*Runtime Startup Code*“ erledigt. Dieser wird vom XC8 Compiler für jedes Projekt erzeugt und erledigt unter anderem die folgenden Dinge:

- Code für Reset- und Interrupt-Vektoren
- Initialisierung globaler Variablen mit gegebenenfalls vorgegebenen Werten
- Globale Variablen ohne vorgegebenen Wert werden Null gesetzt.
- Genereller Setup verschiedener, ausschließlich vom Compiler benutzter Register.

Im User Guide des Compilers finden sich bei Bedarf detaillierte Informationen zum Runtime Startup Code.

2.4.2 Erste Sourcecode Datei mit der Funktion main()

Wie im Assembler Projekt, kann man auch in einem C Projekt eine Main-Datei erstellen lassen. Wählt man im Menü *New* den Eintrag **main.c**, dann bekommt man eine Datei, die alles enthält, was für ein C Programm unbedingt erforderlich ist. Das ist erst mal nur die Funktion *main()*.



Selbstverständlich sollte man auch diese Datei wie weiter oben beim Assembler Projekt umbenennen. (z.B. *uCQ_main.c*)

2.4.2.1 void main(void) { return; } ???

Für Funktionen müssen in C Übergabe- und Rückgabe-Parameter angegeben werden. Der Rückgabe-Parameter steht dabei vor dem Funktionsnamen, die Übergabe-Parameter in der Klammer hinter dem Funktionsnamen. Wird nichts übergeben, dann wird das durch „**void**“ (leer) gekennzeichnet.

Im Gegensatz zu einer PC-Umgebung mit einem Betriebssystem wie z.B. Windows, existiert in einer einfachen Embedded Umgebung eigentlich nichts außer dem Runtime Startup Code von dem irgendwelche Parameter an *main()* übergeben werden könnten. Die Funktion *main()* sollte eigentlich auch niemals enden, weil es nichts gibt, was danach noch kommt und das einen Rückgabewert entgegen nehmen könnte. Deshalb darf das „**return;**“ in der erzeugten Funktion niemals ausgeführt werden!

Das embedded C Programm bleibt normalerweise genau wie das Assembler Programm, so lange in einer noch zu programmierenden Endlosschleife, bis die Energieversorgung abgestellt wird.

2.4.3 Kommentare in C „//“ und /*...*/

Die Sprache C bietet zwei Möglichkeiten für die Kommentierung des Quellcodes.

Der doppelte Schrägstrich „//“ in der obigen Abbildung (*// wohin ???*) entspricht dem Semikolon der Assembler Programmierung. Das Semikolon selbst hat in C die Bedeutung der Terminierung eines Statements. (*return;*)

Die zweite Möglichkeit bietet die Kombination /* ... */, wobei der eingeschlossene Text, egal über wie viele Zeilen er sich erstreckt, als Kommentar behandelt wird. (*im obigen Code für den Kopf der Datei benutzt*)

2.4.4 Die benötigten Include-Dateien

Damit die Register des PICs dem Compiler bekannt sind, muss eine prozessorspezifische Header-Datei mit dem Namen **p18f25k22.h** eingebunden werden. Dies wird beim XC8 Compiler über **#include <xc.h>** gemacht. Der Header **xc.h** bindet dann unter anderem (*und mit Hilfe einiger weiterer Includes*) die passende Datei für den im Projekt eingestellten PIC ein.

Die Einbindung über diesen Umweg hat den Vorteil, dass bei Auswahl eines anderen PICs (*z.B. PIC18F24K22 anstelle von PIC18F25K22*) nicht in allen Dateien des Projektes Änderungen vorgenommen werden müssen. Das geht dann automatisch.

Verwirrend? → xc.h öffnen und versuchen die Verknüpfungen bis zum PICheader zu finden ;-)

Neben der Header-Datei für den Prozessor hat ein Projekt gewöhnlich noch einen Projekt-Header, welcher projektspezifische Definitionen enthält. (*LED_1 ist am Ausgang B_2 angeschlossen ...*) Wenn dieser Projekt-Header auch das Einbinden des Systemheaders xc.h übernimmt, muss in den Source-Dateien (*.c) des Projektes nur der Projekt-Header eingebunden werden.

```
// uCQ_2013.h

#include <xc.h>                                // XC8 compiler header

#define IRCF_31KHZ      0b000                  // value for oscillator frequency setting
                                                // see data sheet
#define LED_1           LATBbits.LATB2        // LED_1 is connected at port B_2
#define LED_1_TRI       TRISBbits.TRISB2
#define mTOG_LED_1()    LED_1 ^= LED_1        //

#define OUTPUT_PIN      0                      // 0 is for output / 1 for input
```

ACHTUNG: Gemäß dem C-Standard muss jede nicht leere Zeile mit einem „newline“ abgeschlossen sein! (→ jede Datei muss mit einer leeren Zeile enden)

2.4.4.1 C-Namen von Registern und Bits des Mikrocontrollers

Die Namen der Special-Funktion-Register des Controllers entsprechen auch in C meist den Namen die auch im Datasheet verwendet werden. Diese „Variablen“ sind im Header für den jeweiligen Controller deklariert. Zugriff auf einzelne Bits erhält man über zusätzlich definierte Bit-Felder, die wie folgt aufgebaut sind: **REGISTERbits.BIT** (z.B. **LATBbits.LATB2**).

2.4.5 Funktionen (am Beispiel `__init()`)

Der Programmteil zur Initialisierung des Mikrocontrollers, welcher nur einmal bei Programmstart bzw. Reset ausgeführt werden soll und deshalb vor der Hauptschleife steht, kann recht umfangreich werden. Die Anzahl an Programmzeilen für die Initialisierung ist oft sogar umfangreicher als die des eigentlichen Programms in der Hauptschleife.

Es wird in einem späteren Kapitel sogar ein Programm folgen, in dem die Endlosschleife absolut gar nichts enthält und der uC trotzdem Aktionen ausführt.

Um die Funktion `main()` übersichtlich zu halten kann deshalb der Code zur Initialisierung in eine eigene Funktion gepackt werden, die vor der Endlosschleife in `main()` aufgerufen wird. Diese Funktion wurde hier `__init()` genannt. („Arduino“ Jünger würden sie `Setup()` nennen)

Wie beim Assembler Template kann bei der Initialisierung die Einstellung der Frequenz für den internen Oszillator und die Einstellung eines PINs für eine LED als Ausgang eingefügt werden.

```
// uCQ_main.c

#include "uCQ_2013.h"

void __init(void)
{
    OSCCONbits.IRCF = IRCF_31KHZ;           // usually defined in <project_name>.h
    OSCTUNEbits.PLLEN = 0;

    LED_1_TRI = OUTPUT_PIN;                 // ""
}

void main(void)
{
    __init();
...
}
```

Der C-Compiler muss jede Funktion kennen, bevor sie aufgerufen wird. Deshalb steht die Definition dieser Funktion in diesem ersten Beispiel bevor der Aufruf der Funktion kommt.

2.4.6 Ein einfaches C Programm

Die Main Funktion enthält letztendlich den Code, der nach der Initialisierung bis zum Abschalten der Stromversorgung oder einem Reset ausgeführt wird.

Die Endlos-Schleife wird über die Bedingung „**while(1){...}**“ gebildet.

```
void main(void)
{
    unsigned char counter_1 = 0;           // one byte variable, values 0..255

    __init();                           // initialization / setup

    while(1) {                         // endless loop
        if(++counter_1 == 0) {
            mTOG_LED_1();
        }
    }
}
```

Nicht vergessen: `__init()` (oder ein Prototyp davon) muss vor dem Aufruf in `main()` definiert sein!

2.4.7 Variablen in C

Lokale Variablen wie „**counter_1**“, sollten in C am Anfang eines Blocks in dem sie gültig sind deklariert werden. (nach einer öffnenden, geschweiften Klammer)

(Aktuelle Compiler wie XC8 erlauben die Deklaration auch an anderen Stellen im Quellcode, solange diese vor der Verwendung erfolgt. Übersichtlicher wird das dadurch aber nicht...)

In C haben **Variablen** immer einen bestimmten Typ, der festlegt, wie bestimmte Operationen damit ausgeführt werden müssen. Mit „**unsigned char**“ ist hier eine 8-Bit Variable ohne Vorzeichen definiert. d.h. **counter_1** kann Werte von 0..255 annehmen.

Der C Compiler überprüft die korrekte Benutzung der Variablen anhand ihres Typs.

Das Demoprogramm des Templates hat wieder die gleiche Funktion wie im Assembler Template. Es soll eine LED blinken lassen.

Hier wird das Umschalten für das Blinken über eine Bedingung „**if(...)**“ gesteuert.

Ist der Ausdruck in der Klammer wahr, dann wird der Block in den geschweiften Klammern (*die Anweisung zum Toggeln, also Umschalten, der LED*) ausgeführt, sonst nicht.

Das „**++**“ vor „**counter_1**“ entspricht dem „**inc**“ (increment) des Assmbler.

„**== 0**“ ist der Vergleich der Variablen mit der Konstanten NULL.

Dadurch das „**++**“ vor der Variablen steht, wird festgelegt, dass zuerst inkrementiert (pre-increment) und dann verglichen wird. Stünde „**++**“ hinter der Variablen, dann würde zuerst verglichen und dann inkrementiert. (post-increment).

Die Abfrage des Zählerwertes auf Null macht Sinn, weil der counter_1 maximal den Wert 255 (0b11111111) annehmen kann. Bei der nächsten Inkrementierung (255 +1) findet ein Überlauf statt und der Wert wird wieder Null (0b00000000).

2.4.8 Prototypen von Funktionen

Ein C Compiler übersetzt den Quellcode in Maschinensprache, in dem er die Datei zeilenweise vom Anfang bis zum Ende durcharbeitet. Jede Funktion muss vor ihrem Aufruf bekannt sein, damit überprüft werden kann, ob der Aufruf auch die richtigen Parameter benutzt.

Dies ist im ersten Beispiel mit lediglich zwei Funktionen leicht zu realisieren. Man schreibt einfach die aufgerufene Funktion vor die Aufrufende. Dann kennt der Compiler die Funktion vor dem Call.

In einem größeren Projekt mit möglicherweise hunderten Funktionen in vielen verschiedenen Dateien ist diese Vorgehensweise nicht praktikabel. Die Lösung des Problems sind sogenannte Prototypen, welche die Funktionen vorab beschreiben (deklarieren).

Ein solcher Prototyp besteht lediglich aus dem Kopf der Funktion mit dem Namen und den Parametern, der mit einem Semikolon abgeschlossen wird.

```
void __init(void);      // function prototype
void main(void)
{
    __init();          // function call
}
void __init(void)      // function definition
{
...
}
```

2.4.9 Dateiabschluss mit „newline“

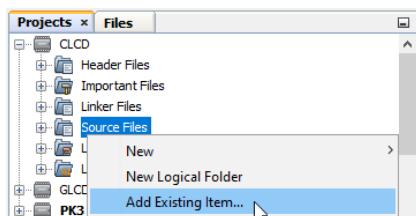
Gemäß dem C-Standard muss jede nicht leere Zeile mit einem „newline“ abgeschlossen sein. Daraus folgt, dass jede *.c oder *.h Datei mit einer leeren Zeile enden muss.

2.5 Vorhandene Dateien / Templates hinzufügen

in solches Template, wie es im nächsten Kapitel schrittweise erstellt werden soll, findet sich in der finalen Version auch im Anhang **TODO**

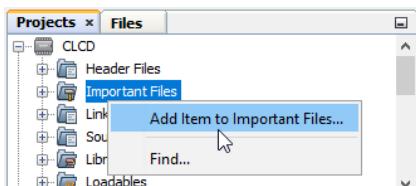
Vor dem Schreiben der ersten Quellcode Zeile bitte unbedingt den ersten Teil des Kommentarfeldes am Anfang bearbeiten und mit Namen, Datum usw. ergänzen.

Dem Projekt können jetzt noch weitere (*schon vorhandene*) Dateien hinzugefügt werden. Dabei kann es zum Beispiel sich um die eigens erstellten Templates handeln, welche ein in vielen Projekten verwendbares Grundgerüst für das zu erstellende Programm enthalten.



2.5.1 Dokumentation hinzufügen

Im Projekt können auch Links zu anderen Dokumenten gespeichert werden, die für die Programmierung oft (immer) benötigt werden. Ein Datenblatt des im Projekt verwendeten PICs, wäre hier eine gute Wahl. Gegebenenfalls ist es auch sehr vorteilhaft die Dokumentation (Schaltplan) einer verwendeten Plattform (uC_Quick Platine) schnell aufrufen zu können.
Unter **Important Files** mit **Add Existing Items** als Link einfügen, dann ist es immer griffbereit!



2.6 Configuration Bits (Assembler und C)

Die „Configuration Bits“ werden bei der Programmierung des Controllers gesetzt und normalerweise nicht während der Laufzeit des Programms verändert.

Bei den PIC18 Controllern befindet sich die Beschreibung der Konfigurations-Einstellungen im Data Sheet unter „*Special Features of the CPU – Configuration Bits*“ (**anschauen !!!**)

Damit das Programm auf der uCQ Hardware lauffähig ist, müssen unbedingt eingestellt werden:

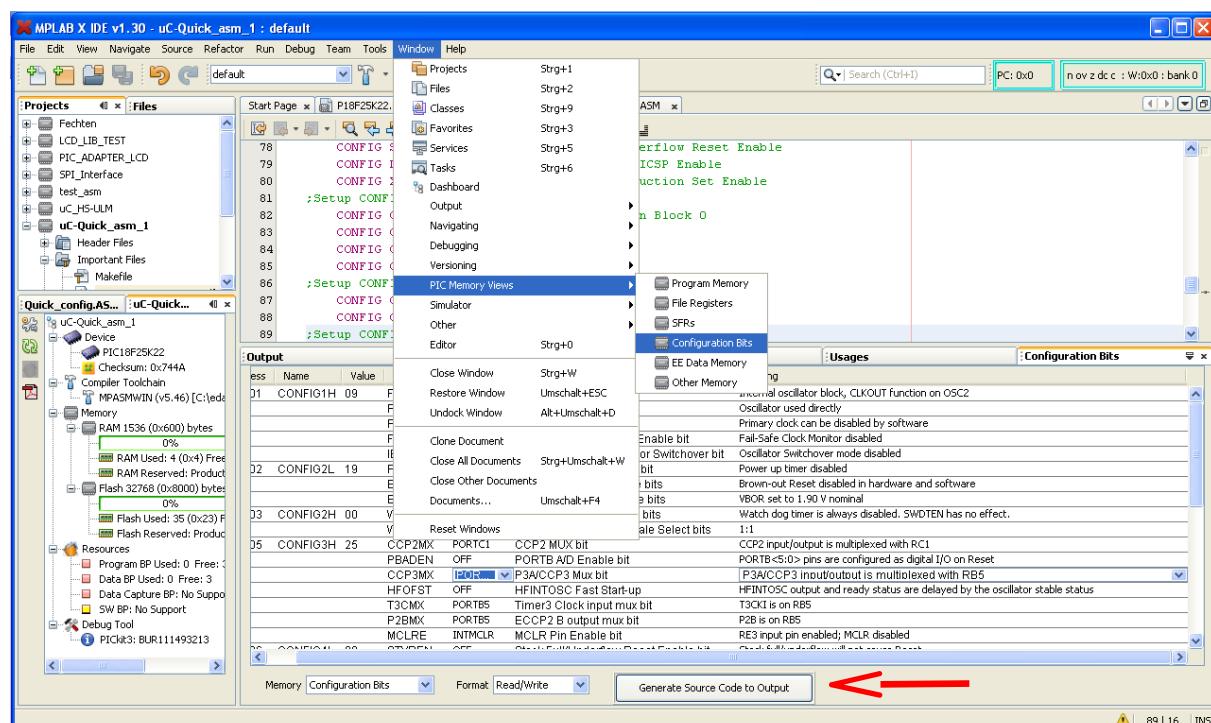
Oszillator = intern

Watchdog = AUS

LVP = AN

Am besten gewöhnt man sich gleich an „**ALLE**“ Config Bits zu setzen, dann erlebt man auch keine bösen Überraschungen wegen vermeintlicher „defaults“.

Mit MPLAB X wurde die Codeerzeugung für die Configuration Bits stark vereinfacht. Über das Menü **Window->PIC Memory View->Configuration Bits** kann ein Fenster geöffnet werden, in dem alle Bits nach Wunsch eingestellt werden. Für jede Einstellung erscheint auch eine Beschreibung.



Mit dem Button „**Generate Source Code to Output**“ kann dann der Quelltext für den im Projekt eingestellten Assembler/Compiler ausgegeben werden. Die folgende Abbildung zeigt den erzeugten Code eines XC8 Projektes.

Diesen kopiert man anschließend in eine Quelldatei. Die Datei kann man wieder über das Popup Menü im Projekt Fenster anlegen und z.B. „*uCQ_config.c*“ bzw. „*uCQ_config.s*“ nennen.

```
Output - Config Bits Source  Configuration Bits

// PIC18F25K22 Configuration Bit Settings

// 'C' source line config statements

// CONFIG1H
#pragma config FOSC = INTIO67    // Oscillator Selection bits (Internal oscillator block)
#pragma config PLLCFG = OFF      // 4X PLL Enable (Oscillator used directly)
#pragma config PRCLKEN = ON       // Primary clock enable bit (Primary clock enabled)
#pragma config FCMEN = OFF        // Fail-Safe Clock Monitor Enable bit (Fail-Safe Clock Monitor disabled)
#pragma config IESO = OFF         // Internal/External Oscillator Switchover bit (Oscillator Switchover mode disabled)
```

ACHTUNG: Gemäß dem C-Standard muss jede nicht leere Zeile mit einem „newline“ abgeschlossen sein! (→ jede Datei muss mit einer leeren Zeile enden)

2.7 Oszillator Konfiguration (Assembler und C)

Eine der wichtigsten Einstellungen ist die Konfiguration des Oszillators, welcher den Arbeitstakt des Controllers bereitstellt. **Ohne Arbeitstakt keine Ausführung des Programms !!!**

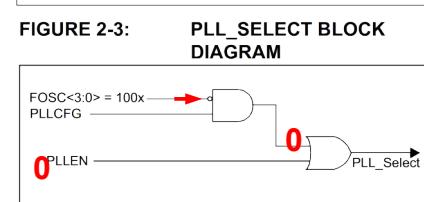
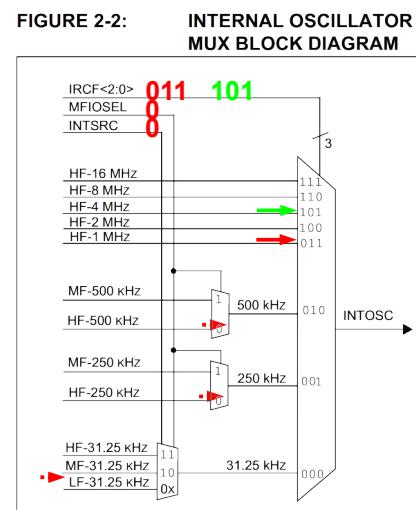
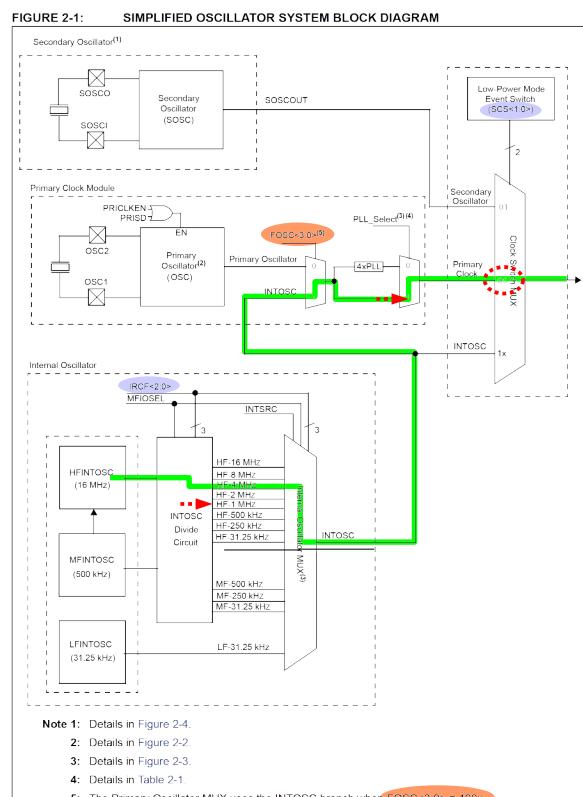
Das Datenblatt des PIC18F24K22 zeigt sechs verschiedene Takt Quellen und jede Menge Einstellmöglichkeiten

<u>Quellen:</u>	<u>Register bits:</u>
<ol style="list-style-type: none"> 1. RC External Resistor/Capacitor 2. LP Low-Power Crystal 3. XT Crystal/Resonator 4. INTOSC Internal Oscillator 5. HS High-Speed Crystal/Resonator 6. EC External Clock 	<ul style="list-style-type: none"> - FOSC<3:0> (CONFIG1H<6:4>) - PRISD (OSCCON2<2>) - PLLCFG (CONFIG1H<4>) - PLLEN (OSCTUNE<6>) - HFOFST (CONFIG3H<3>) - IRCF<2:0> (OSCCON<6:4>) - MFIOSEL (OSCCON2<4>) - INTSRC (OSCTUNE<7>) - PRICLKEN (CONFIG1H<5>)

Für jede dieser Auswahlmöglichkeiten bietet das Datenblatt eine Beschreibung, dazu meistens auch eine Abbildung. Die oben rot markierte Auswahl des internen Oszillators erfordert keinerlei externe Beschaltung und wird deshalb hier näher ausgeführt.

Zur Erleichterung verschiedener Berechnungen innerhalb der Beispiele soll ein Takt von **4MHz** eingestellt werden. Bedingt durch die Art der Befehlsverarbeitung bei der PIC18 Familie, die 4 Oszillator Takte für einen Befehlstakt benötigt, bedeutet dies, dass für die Verarbeitung eines einfachen Befehls genau **1μs** benötigt wird.

Für die erste Übersicht verwendet man am besten die Abbildungen aus dem Datenblatt.



Der Pfad des Taktes führt durch verschiedene Blöcke, welche über verschiedenen Steuerbits kontrolliert werden. Die Bits **FOSC<3:0>** gehören zu den Konfigurations-Bits, die schon bei der Programmierung des Controllers gesetzt werden und sind deshalb von größerer Bedeutung als die Bits der **OSCCON** Register, welche erst zur Laufzeit des Programms verändert werden können.

2.7.1 Oszillator Kontroll-Register – OSCCON / OSCCON2 / OSCTUNE

REGISTER 2-1: OSCCON: OSCILLATOR CONTROL REGISTER							
R/W-0	R/W-0	R/W-1	R/W-1	R-q	R-0	R/W-0	R/W-0
IDLEN		IRCF<2:0>		OSTS ⁽¹⁾	HFOIFS	SCS<1:0>	
bit 7							bit 0
Legend:							
R = Readable bit	W = Writable bit	U = Unimplemented bit, read as '0'	q = depends on condition				
-n = Value at POR	'1' = Bit is set	'0' = Bit is cleared	x = Bit is unknown				
bit 7	IDLEN: Idle Enable bit						
	1 = Device enters Idle mode on SLEEP instruction						
	0 = Device enters Sleep mode on SLEEP instruction						
bit 6-4	IRCF<2:0>: Internal RC Oscillator Frequency Select bits ⁽²⁾						
	111 = HFINTOSC – (16 MHz)						
	110 = HFINTOSC/2 – (8 MHz)						
	101 = HFINTOSC/4 – (4 MHz)						
	100 = HFINTOSC/8 – (2 MHz)						
	011 = HFINTOSC/16 – (1 MHz) ⁽³⁾						
	If INTSRC = 0 and MFIOSEL = 0:						
	010 = HFINTOSC/32 – (500 kHz)						
	001 = HFINTOSC/64 – (250 kHz)						
	000 = LFINTOSC – (31.25 kHz)						
	If INTSRC = 1 and MFIOSEL = 0:						
	010 = MFINTOSC – (500 kHz)						
	001 = MFINTOSC/2 – (250 kHz)						
	000 = MFINTOSC/16 – (31.25 kHz)						
bit 3	OSTS: Oscillator Start-up Time-out Status bit						
	1 = Device is running from the clock defined by FOSC<3:0> of the CONFIG1H register						
	0 = Device is running from the internal oscillator (HFINTOSC, MFINTOSC or LFINTOSC)						
bit 2	HFOIFS: HFINTOSC Frequency Stable bit						
	1 = HFINTOSC frequency is stable						
	0 = HFINTOSC frequency is not stable						
bit 1-0	SCS<1:0>: System Clock Select bit						
	1x = Internal oscillator block						
	01 = Secondary (SOSC) oscillator						
	00 = Primary clock (determined by FOSC<3:0> in CONFIG1H).						
Note 1: Reset state depends on state of the IESO Configuration bit.							
2: INTOSC source may be determined by the INTSRC bit in OSCTUNE and the MFIOSEL bit in OSCCON2.							
3: Default output frequency of HFINTOSC on Reset.							

REGISTER 2-2: OSCCON2: OSCILLATOR CONTROL REGISTER 2							
R/W-0	R-0/q	U-0	R/W-0/u	R/W-0/u	R/W-1/1	R-x/u	R-0/u
PLRDY	SOSCRUN	—	MFIOSEL	SOSCGQ ⁽¹⁾	PRISD	MFIOFS	LFIOS
bit 7							bit 0
Legend:							
R = Readable bit	W = Writable bit	U = Unimplemented bit, read as '0'	q = depends on condition				
'1' = Bit is set	'0' = Bit is cleared	x = Bit is unknown					
-n/n = Value at POR/BOR/Value at all other Resets							
bit 7	PLRDY: PLL Run Status bit						
	1 = System clock comes from 4xPLL						
	0 = System clock comes from an oscillator, other than 4xPLL						
bit 6	SOSCRUN: Secondary Oscillator Start bit						
	1 = System clock comes from secondary SOSC						
	0 = System clock comes from an oscillator, other than SOSC						
bit 5	Unimplemented: Read as '0'						
bit 4	MFIOSEL: MFINTOSC Select bit						
	1 = MFINTOSC is used in place of HFINTOSC frequencies of 500 kHz, 250 kHz and 31.25 kHz						
	0 = MFINTOSC is not used						
bit 3	SOSCGQ⁽¹⁾: Secondary Oscillator Start Control bit						
	1 = Secondary oscillator is enabled						
	0 = Secondary oscillator is shut off if no other sources are requesting it.						
bit 2	PRISD: Primary Oscillator Drive Circuit Shutdown bit						
	1 = Oscillator drive circuit is on						
	0 = Oscillator drive circuit off (zero power)						
bit 1	MFIOFS: MFINTOSC Frequency Stable bit						
	1 = MFINTOSC is stable						
	0 = MFINTOSC is not stable						
bit 0	LFIOS: LFINTOSC Frequency Stable bit						
	1 = LFINTOSC is stable						
	0 = LFINTOSC is not stable						
Note 1: The SOSCGQ bit is only reset on a POR Reset.							

2.7 Register Definitions: Oscillator Tuning							
REGISTER 2-3: OSCTUNE: OSCILLATOR TUNING REGISTER							
R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
INTSRC	PLLEN ⁽¹⁾	—	TUN<5:0>				
bit 7							bit 0
Legend:							
R = Readable bit	W = Writable bit	U = Unimplemented bit, read as '0'					
-n = Value at POR	'1' = Bit is set	'0' = Bit is cleared	x = Bit is unknown				
bit 7	INTSRC: Internal Oscillator Low-Frequency Source Select bit						
	1 = 31.25 kHz device clock derived from the MFINTOSC or HFINTOSC source						
	0 = 31.25 kHz device clock derived directly from LFINTOSC internal oscillator						
bit 6	PLLEN: Frequency Multiplier 4xPLL for HFINTOSC Enable bit ⁽¹⁾						
	1 = PLL enabled						
	0 = PLL disabled						
bit 5-0	TUN<5:0>: Frequency Tuning bits – use to adjust MFINTOSC and HFINTOSC frequencies						
	011111 = Maximum frequency						
	011110 =						
	000001 =						
	000000 = Oscillator module (HFINTOSC and MFINTOSC) are running at the factory calibrated frequency.						
	111111 =						
	111110 = Minimum frequency						
Note 1: The PLLEN bit is active for all the primary clock sources (internal or external) and is designed to operate with clock frequencies between 4 MHz and 16 MHz.							

In der Überblickdarstellung der Registers sind die Default- und Reset-Einstellungen aller Bits aufgelistet. Die „**Internal Oscillator Frequency Select bits**“ haben die Voreinstellung **0-1-1**, was einer Frequenz von 1 MHz entspricht. Für eine angestrebte Takt Frequenz von 4 MHz müssen die Bits **IRCF<2:0>** auf **1-0-1** geändert werden.

Die „**System Clock Select bits**“ haben die Voreinstellung **0-0**, was „Primary oscillator“ bedeutet. Diese Bits **müssen nicht** auf 1-x für „**Internal oscillator block**“ umgestellt werden.

Das Verändern des OSCCON Registers muss zur Laufzeit des Programms erfolgen. Das bedeutet natürlich, dass die Konfiguration zu diesem Zeitpunkt schon lauffähig sein muss !!!

Da die Default-Konfiguration „Primary oscillator“ lautet, muss dieser über die Konfiguration auf „Internal oscillator“ eingestellt sein !!!

Die anderen Bits des Registers sind "Read-Only" bzw. müssen vorerst nicht beachtet werden. Der so ermittelte Wert für das Register ist:

x-1-0-1-x-x-0-0 (oder x-1-0-1-x-x-1-x)

Ersetzt man die "dont care" (x) Bits mit "0", so führt das zum binären Wert:

MPLAB X Assembler: **B'01010000'** (hexadezimal H'50' oder 0x50)

MPLAB XC8: **0b01010000** (oder hexadezimal 0x50)

Das Register OSCTUNE ist für den angestrebte Frequenz nicht relevant, im OSCTUNE muss die die Einstellung für die 4xPLL (PhasedLockedLoop / Frequenzvervielfacher) überprüft werden.

2.7.2 Configuration Bits FOSC<3:0>

REGISTER 24-1: CONFIG1H: CONFIGURATION REGISTER 1 HIGH							
R/P-0	R/P-0	R/P-1	R/P-0	R/P-0	R/P-1	R/P-0	R/P-1
IESO	FCMEN	PRICKEN	PLLCFG			FOSC<3:0>	
bit 7							
Legend: R = Readable bit P = Programmable bit U = Unimplemented bit, read as '0' -n = Value when device is unprogrammed x = Bit is unknown							
bit 7	IESO ⁽¹⁾ : Internal/External Oscillator Switchover bit 1 = Oscillator Switchover mode enabled 0 = Oscillator Switchover mode disabled						
bit 6	FCMEN ⁽¹⁾ : Fail-Safe Clock Monitor Enable bit 1 = Fail-Safe Clock Monitor enabled 0 = Fail-Safe Clock Monitor disabled						
bit 5	PRICKEN: Primary Clock Enable bit 1 = Primary Clock is always enabled 0 = Primary Clock can be disabled by software						
bit 4	PLLCFG: 4 x PLL Enable bit 1 = 4 x PLL always enabled, Oscillator multiplied by 4 0 = 4 x PLL is under software control, PLLLEN (OSCTUNE<6>)						
bit 3-0	FOSC<3:0>: Oscillator Selection bits 1111 = External RC oscillator, CLKOUT function on RA6 1110 = External RC oscillator, CLKOUT function on RA6 1101 = EC oscillator (low power, <500 kHz) 1100 = EC oscillator, CLKOUT function on OSC2 (low power, <500 kHz) 1011 = EC oscillator (medium power, 500 kHz-16 MHz) 1010 = EC oscillator, CLKOUT function on OSC2 (medium power, 500 kHz-16 MHz) 1001 = Internal oscillator block, CLKOUT function on OSC2 1000 = Internal oscillator block 0111 = External RC oscillator 0110 = External RC oscillator, CLKOUT function on OSC2 0101 = EC oscillator (high power, >16 MHz) 0100 = EC oscillator, CLKOUT function on OSC2 (high power, >16 MHz) 0011 = HS oscillator (medium power, 4 MHz-16 MHz) 0010 = HS oscillator (high power, >16 MHz) 0001 = XT oscillator 0000 = LP oscillator						
Note 1: When FOSC<3:0> is configured for HS, XT, or LP oscillator and FCMEN bit is set, then the IESO bit should also be set to prevent a false failed clock indication and to enable automatic clock switch over from the internal oscillator block to the external oscillator when the OST times out.							

Wie in der Beschreibung für das OSCCON Register schon erläutert, müssen die "Oscillator Selection bits" auf "**"Internal oscillator block ..."**" gesetzt werden.

Die geschieht nicht zur Laufzeit des Programms, sondern zum Zeitpunkt des Programmierens durch das Programmiergerät.

Leider kann die erforderliche Anweisung dafür nicht im Datenblatt gefunden werden, sondern muss der Hilfe der IDE oder der Include-Datei entnommen werden. Hat man schon ein vernünftiges Template inklusive der "**"Config Settings"** erstellt, dann kann auch dieses weiterhelfen.

```
CONFIG FOSC = INTIO7      ; Oscillator (LP, XT, HSHP, HSMP, RC, RCIO6, ECHP, ECHPIO6,
;                                INTIO67, INTIO7, ECMPIO6, ECLP, ECLPIO6)
```

Die Definitionen INTIO67 und INTIO7 entsprechen offensichtlich den Zeilen:

1001 = Internal oscillator block, CLKO function on OSC2/RA6, port function on RA7

1000 = Internal oscillator block, port function on RA6 and RA7

2.7.3 Oszillator Einstellungen im Quelltext eintragen

Das Setzen der Konfiguration sollte bei den vorhandenen Templates eigentlich keiner weiteren Erläuterungen bedürfen. Man muss lediglich die erste Zeile des Konfigurationsblocks entsprechend anpassen.

Das Schreiben des OSCCON Registers sollte möglichst früh im Programmablauf erfolgen. Am besten vor allem Anderen. Der erforderliche Programmcode wird in die Initialisierungsroutinen der Templates eingefügt.

ASSEMBLER

```
Init
    movlw H'50'        ; Arbeitsregister mit Konstante laden
    movwf OSCCON,A    ; Arbeitsregister nach OSCCON schreiben
...
Main
```

C

```
Init()
{
    OSCCON = 0x50;
...
}
```

Der aus den verschiedenen Versionen generierte Maschinencode muss natürlich exakt der selbe sein. Das heißt, dass der C-Compiler genau wie der Assembler zwei Instruktionen erzeugt !
([3.1.7 Der erzeugte Maschinencode](#) kann später in der IDE angezeigt werden)

2.8 Basisinitialisierung aller Pins, Module und Variablen

Um eine möglichst große Anwendungsvielfalt zu gewährleisten, sind in den meisten Controllern viele Peripheriemodule integriert und beinahe jeder Anschlusspin ist mit mehreren Funktionen belegbar. Leider erfordert dies eine umfangreiche Konfiguration die noch vor der eigentlichen Programmierung des Hauptprogramms (*main*) in der Initialisierungsphase (*__init*) erfolgen muss.

2.8.1 Konfiguration für alle zu verwendenden Pins ermitteln

Für jeden (verwendeten) Pin muss die benötigte Einstellung im Data Sheet ermittelt werden. Dies betrifft die Signalrichtung (Ein- oder Ausgang), sowie die Konfiguration als Analogeingang, Digital-IO, oder einem bestimmten Hardwaremodul zugehörigen Pin.

Zudem empfiehlt es sich von Anfang an [2.9 Text Substitution Labels](#) einzuführen, welche die Pin-Namen im Programmcode durch geeignete Signal-Namen ersetzen. Dadurch wird die Lesbarkeit enorm erhöht und auch eine sehr einfache Änderung der Pinbelegung ermöglicht.

2.8.2 Peripheriemodule initialisieren

Werden Peripheriemodule wie z.B. Timer benutzt, so müssen diese natürlich aktiviert und eingestellt werden, damit sie mit der gewünschten Geschwindigkeit arbeiten.

Bei der Programmierung in **Assembler**-Projekten können dafür alle nötigen Informationen dem Datenblatt entnommen werden.

Einige C-Compiler stellen für die meisten Module Funktionen bereit, welche vor allem die Übersichtlichkeit erhöhen und den Code meist sehr viel verständlicher machen.

Nachteile ergeben sich aber in der Geschwindigkeit, da bei jedem Aufruf einer Funktion der momentane Kontext (Programmzeiger, Inhalt mehrerer Register) gesichert und danach wieder geladen werden muss. Der erforderliche zeitliche Aufwand wird von Anwendern meist total übersehen oder völlig falsch eingeschätzt. (→ [12.7.2 Context saving](#))

Auch bei Verwendung von Bibliotheks routinen ist der Blick ins Datenblatt unumgänglich !!!

2.9 Text Substitution Labels

Diese werden unter anderem dazu verwendet die Verständlichkeit des Programms zu erhöhen und Änderungen zu erleichtern. In der Datei "*uCQ_2013.inc*" bzw „*uCQ_2013.h*“ finden sich einige Beispiele. Zur Veranschaulichung hier ein Beispiel aus einem weiter unten folgenden C Projekt.

```
#define BUTTON      PORTBbits.RB2    // Taster Eingang an Port B2
#define LED          LATBbits.LATB5   // LED Ausgang an Port B5
#define LED_ON        0              // LED an bei 0V am Ausgang
#define LED_OFF       1              // 0V am Eingang wenn Taster gedrückt
```

Die Definition macht zusätzliche Kommentare im Quellcode praktisch unnötig.

```
if (BUTTON == BTN_ACTIVATED) {
    LED = LED_ON;
}
else{
    LED = LED_OFF;
}
```

Müssten für Taster oder LED später andere Pins verwendet werden oder würden sich die aktiven Pegel ändern, so wäre die Anpassung auch bei häufigem Vorkommen im Quellcode immer nur an einer einzigen, leicht zu findenden Stelle erforderlich.

Zusätzlich erzeugt eine solche Vorgehensweise eine gute Übersicht darüber, welche "Signale" an welchen Pins anliegen.

3 Digitale Ein- und Ausgänge (Hello World)

3.1 „Hello World“ für µC

Aufbauend auf den Basis-Templates (*im Anhang*) kann das erste kleine Projekt gestartet werden. Das "Hello World" Programm für Mikrocontroller kann aus der Abfrage einer Taste und der Ansteuerung einer LED bestehen. Bei einer gedrückten Taste soll eine LED leuchten.

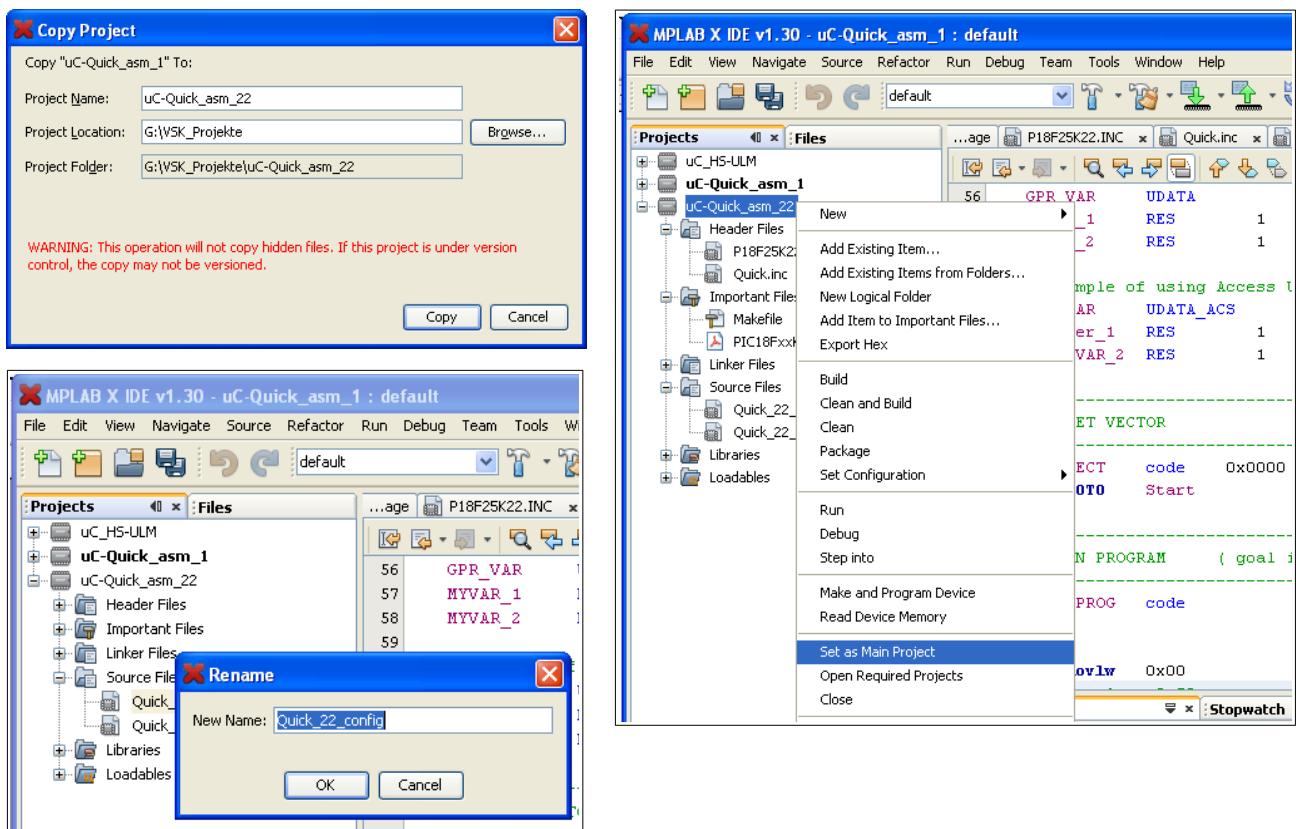
Die mühsam erstellten Templates sollen als Basis für alle weiteren Projekte dienen und deshalb nicht verändert werden. MPLAB X bietet eine sehr komfortable Art eine Kopie des Projektes in einem neuen Verzeichnis anzulegen. Einfach das vorhandene Projekt mit der rechten Maustaste anklicken und im Pop-Up Menü den Befehl „copy“ auswählen.

Im erscheinenden Dialog kann dann ein neuer Name für das Projekt eingegeben werden, der auch gleich als neuer Ordnername vorgeschlagen wird.

(endet der Ordner nicht mit „.X“ kann man das noch anfügen)

Dann sollten (*müssen*) nur noch die Source-Files des neuen Projektes angepasst werden. (z.B. **hello_main.asm** ...). Auch das geht am einfachsten über das Pop-Up Menü im Projektfenster und sollte inzwischen mit etwas Geduld auch ohne Anleitung durchführbar sein.

Zum guten Schluss wird das Projekt noch über das Pop-Up Menü zu „Main Project“ gemacht.

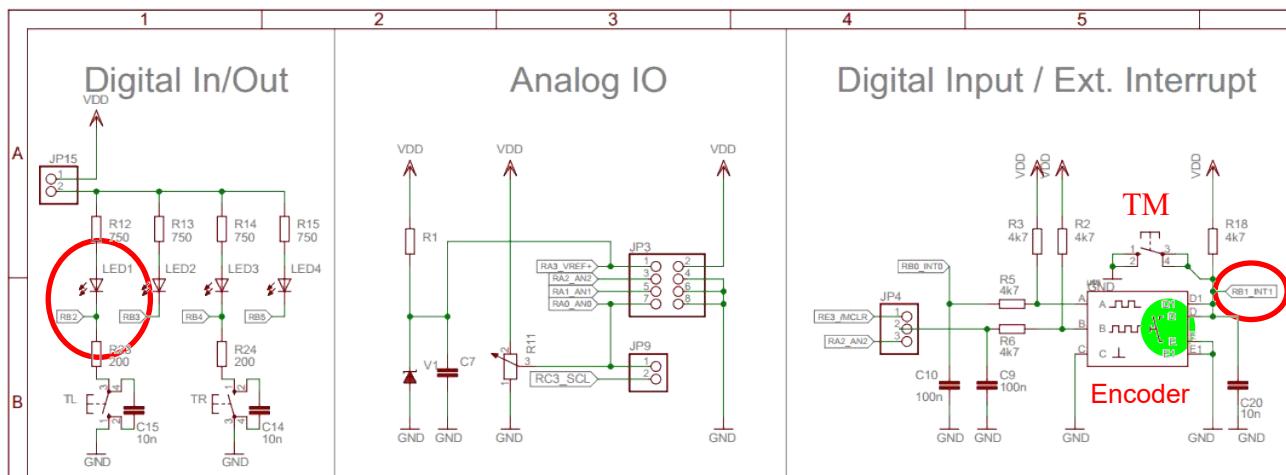


3.1.1 Neues C Projekt

Parallel zum Assembler Projekt kann ein entsprechendes „Hello World“ C Projekt, mit den Templates gestartet werden. Die Dateien könnten dann beispielsweise in „**hello_config.c**“, „**hello_main.c**“ und **hw_profile.h** umbenannt werden.

3.1.2 Pins für Taster und LED bestimmen

Soll das Projekt auf der uC_Quick Plattform realisiert werden, muss man anhand des Schaltplanes die Pins für Taster und LED festlegen.



Auf der Platine können bis zu vier LEDs und drei Taster bestückt werden. Die Funktion des Tasters TM kann dabei über der Tastfunktion des Encoders realisiert sein. Gewählt werden sollen TM oder der Encoder Taster und die LED1 (*an RB2*).

Es gibt allerdings noch weitere Details, die man dem Schaltplan entnehmen kann.

1. Die LEDs leuchten bei Low-Level an den Kathoden. (*zugehörigen Ausgängen des PICs*)
2. Wenn ein Taster betätigt wird, erscheint Low-Level ² an zugehörigen Eingängen (*Bei TL/TR leuchtet die zugehörige LED wenn der IC-Pin als Eingang konfiguriert ist*)
3. Bei nicht gedrückten Taster TM werden die Eingänge über R16 bzw. die LED am gleichen Pin und deren Vorwiderstand auf High-Level ³ gehalten.
4. Die Kondensatoren C14/C15 dienen der „Entprellung“ der Taster (*Google ... ;-*) Die Serienwiderstände R23/R24 verhindern eine Überschreitung des max. Ausgangstromes von 25mA (*Data Sheet ;-*), falls der Pin des Controllers als Ausgang konfiguriert ist, und der Taster trotzdem betätigt wird. ($5V / 25mA = 200 \Omega$)

² Low-Level wäre idealerweise 0V.

TL und TR stellen die Verbindung in Richtung GND (*Low-Level*) her. Am Eingang des PICs erscheint eine Spannung die sich aus der Reihenschaltung von Vorwiderstand, LED und Serienwiderstand des Tasters ergibt. (*Der Serienwiderstand von 200 Ohm verhindert, dass der zulässige Strom überschritten wird, falls der Pin ein Ausgang ist.*)

Bei Vernachlässigung des Eingangsstromes des Pins, einer Versorgungsspannung von 5V und einer Diodenspannung von ~2V kann man den Wert der Spannung am Eingang berechnen.

$$U_{in_low} = ((5V - 2V) / (750 + 200)) * 200 = \underline{0,63 \text{ V}}$$

³ High-Level entspricht idealerweise der Versorgungsspannung des PICs (z.B. 5V)

Tatsächlich verursacht der (sehr kleine) Eingangsstrom des Controller-Pins an der LED einen Spannungsabfall. (*die LED leuchtet nicht oder nicht richtig !.*)

$$U_{in_high \text{ gemessen}} \sim \underline{3,5 \text{ V}}$$

3.1.3 Config Settings überprüfen (einstellen)

Nochmal zur Erinnerung und zur Übung:

Immer ALLE Konfigurationseinstellungen überprüfen ob diese wirklich für das Projekt geeignet sind.

3.1.4 Pins initialisieren

Im Kapitel **I/O Ports** des Datenblattes findet man zunächst eine kurze Zusammenfassung der Funktion mit einem zugehörigen Diagramm. Leider ist beides nicht ganz vollständig, da nur die digitalen Eigenschaften beschrieben und gezeichnet sind.

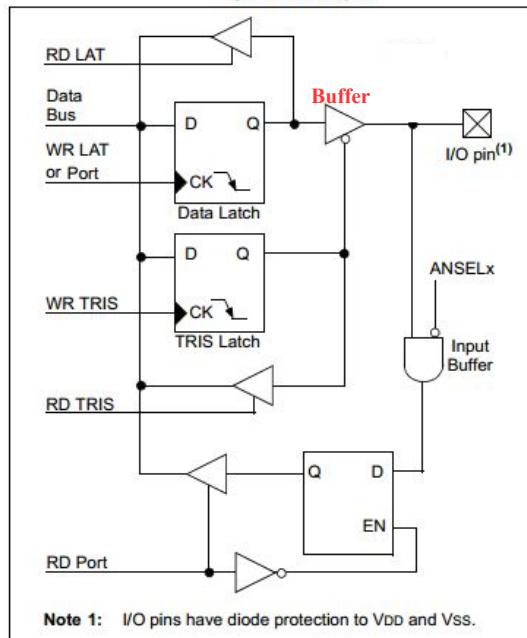
Da die meisten Pins sowohl als Ein- und als Ausgang genutzt werden können, muss man irgendwo die gewünschte Richtung festlegen. Dies geschieht im dem **I/O Pin** zugehörigen **TRIS** Register. Der Name **TRIS** für dieses „Richtungsregister“ kommt von „Tri-State“, was einen weiteren Zustand eines Ausgangs zusätzlich zu „High“ und „Low“ beschreibt. Dieser „dritte“ Zustand entspricht einer unterbrochenen Leitung.

Der Speicherort für den Wert des Ausgangs ist das „**Data Latch**“. Ob dieser Wert zum Pin durchkommt entscheidet aber der nachgeschaltete Tristate-Buffer, welcher vom im „**TRIS Latch**“ gespeicherten Wert gesteuert wird.

An dem kleinen Kreis am Steuereingang des Buffers kann man erkennen, das dieser mit einem Low-Signal (0) aktiviert wird.

Wenn man also den Pin als Ausgang verwenden möchte, muss man eine „0“ ins TRIS Register schreiben. Will man ein Eingangssignal erfassen, so muss der Ausgang mit einer „1“ im Tristate Register abgetrennt werden, damit das Eingangssignal nicht vom „Data Latch“ gestört wird.

FIGURE 10-1: GENERIC I/O PORT OPERATION



Note: On a Power-on Reset, RB<5:0> are configured as analog inputs by default and read as '0'; RB<7:6> are configured as digital inputs.

When the PBADEN Configuration bit is set to '1', RB<5:0> will alternatively be configured as digital inputs on POR.

EXAMPLE 10-2: INITIALIZING PORTB

```

MOVLB 0xF ; Set BSR for banked SFRs
CLRF PORTB ; Initialize PORTB by
; clearing output
; data latches
CLRF LATB ; Alternate method
; to clear output
; data latches
MOVLW 0F0h ; Value for init
MOVWF ANSELB ; Enable RB<3:0> for
; digital input pins
; (not required if config bit
; PBADEN is clear)
MOVLW 0CFh ; Value used to
; initialize data
; direction
MOVWF TRISB ; Set RB<3:0> as inputs
; RB<5:4> as outputs
; RB<7:6> as inputs

```

Im Data Sheet gibt es noch weiterführende Beschreibungen zu IO-Ports generell und nochmals für jeden Port speziell, da diese in den Details oft stark unterschiedlich ausgeführt sind.

Sehr wichtig sind immer die „Notes“ !!!

Hier erfährt man beispielsweise, dass der PORT standardmäßig zumindest teilweise als Analogport konfiguriert ist (*RB5:RB0*) und wie man dies ändern kann.

Meistens ist auch noch ein Beispiel für die Initialisierung des Ports vorhanden.

Basierend auf den neu gewonnenen Informationen sollten also nochmals die Configuration-Bits überprüft und gegebenenfalls **PBADEN** auf **OFF** eingestellt werden ;-)

3.1.4.1 Anlegen von Text Substitution Labels für LED und Taster

Die aus dem Schaltplan gewonnenen Informationen werden jetzt am besten sofort in das Projekt integriert. In die „globale“ Include-Datei „uCQ_2013.inc“ des ASM Projektes kann man dazu folgende Definitionen einfügen (siehe auch [3.1.4.2](#)). Ein kleines führendes „n“ im Signalnamen signalisiert hierbei den „low-active“ Charakter des Signals.

```
#define nBTN_ENC           PORTB,RB1,ACCESS      ; Input Pin for Button (low-active)
#define BTN_ENC_TRI        TRISB,TRISB1,ACCESS

#define nLED_1              LATB,LATB2,ACCESS     ; Output Pin for LED (low active)
#define LED_1_TRI           TRISB,TRISB2,ACCESS
```

3.1.4.1.1 Text Substitution Labels für C-Projekt

Für C-Projekte sind zusätzliche Definitionen in der Header-Datei nützlich:

```
#define BTN_ENC           PORTBbits.RB1          //button input
#define BTN_ENC_TRI        TRISBbits.TRISB1

#define LED_1               LATBbits.LATB2         //LED output
#define LED_1_TRI           TRISBbits.TRISB2

#define LED_ON              0                      // (low-active LED output)
#define LED_OFF             1
#define BTN_ACTIVATED       0                      // (low-active button input)

#define INPUT_PIN           1          // definitions for TRIS registers
#define OUTPUT_PIN          0
```

3.1.4.2 Initialisierungscode einfügen

Um Pin 2 von PORTB als Ausgang zu definieren muss nur Bit Nummer 2 im Register TRISB auf „0“ gesetzt werden. Die anderen sollten vorläufig auf Ihren Voreinstellungen („1“) belassen werden.

(*Es liegt ja immerhin im Bereich der Möglichkeiten, dass sich die Chipentwickler etwas dabei gedacht haben, als die Voreinstellungen (→ REGISTER FILE SUMMARY) festgelegt wurden*)

Für die Modifikation eines einzelnen Bits ist natürlich ein Bit-orientierter Befehl am besten geeignet. Da es davon nicht viele gibt, hier nochmals die Übersicht:

BIT-ORIENTED OPERATIONS									
BCF	f, b, a	Bit Clear f	1	1001	bbba	ffff	ffff	None	1, 2
BSF	f, b, a	Bit Set f	1	1000	bbba	ffff	ffff	None	1, 2
BTFS	f, b, a	Bit Test f, Skip if Clear	1 (2 or 3)	1011	bbba	ffff	ffff	None	3, 4
BTFS	f, b, a	Bit Test f, Skip if Set	1 (2 or 3)	1010	bbba	ffff	ffff	None	3, 4
BG	f, d, a	Bit Toggle f	1	0111	bbba	ffff	ffff	None	1, 2

„Bit **Clear f**“ hört sich schon so an, als könnte damit etwas zu „0“ gemacht werden. Die detaillierte Beschreibung gibt dann genauere Auskunft über die Parameter f, b und a:

BCF Bit Clear f									
Syntax:	BCF f, b, {a}								
Operands:	0 ≤ f ≤ 255 0 ≤ b ≤ 7 a ∈ {0,1}								
Operation:	0 → f(b)								
Status Affected:	None								
Encoding:	1001 bbba ffff ffff								
Description:	Bit 'b' in register 'f' is cleared. If 'a' is '0', the Access Bank is selected. If 'a' is '1', the BSR is used to select the GPR bank (default).								
Words:	1								
Cycles:	1								
Q Cycle Activity:	<table border="1"> <tr> <td>Q1</td><td>Q2</td><td>Q3</td><td>Q4</td> </tr> <tr> <td>Decode</td><td>Read register 'f'</td><td>Process Data</td><td>Write register 'f'</td> </tr> </table>	Q1	Q2	Q3	Q4	Decode	Read register 'f'	Process Data	Write register 'f'
Q1	Q2	Q3	Q4						
Decode	Read register 'f'	Process Data	Write register 'f'						
Example:	BCF FLAG_REG, 7, 0								
Before Instruction	FLAG_REG = C7h								
After Instruction	FLAG_REG = 47h								

Dank der schon vorgenommenen Definition der Text Substitution Labels, sehen die Initialisierungen dann folgendermaßen aus:

```
Init: ; initialization code
    movlw 0x50          ; set internal oscillator
    movwf OSCCON         ; frequency 4MHz
    bcf LED_1_TRI        ; output for LED
    bsf BTN_ENC_TRI      ; input for button

void __init() {
    OSCCON = 0x50;      // set internal oscillator 4MHz
    BTN_ENC_TRI = INPUT_PIN;
    LED_1_TRI = OUTPUT_PIN;
}
```

3.1.5 Taster abfragen und LED einschalten

Das Ein- und Ausschalten der LED kann natürlich analog zu der Pin-Konfiguration vorgenommen werden.

Auch für die Abfrage des Tasters ist in der Assemblersprache einer der Bit-orientierten Befehle angesagt. Die detaillierte Beschreibung des Befehls **BTFSC** beispielsweise lautet:

If bit 'b' in register 'f' is '0', then the next instruction is skipped.

Die Programmierung gestaltet sich aber schon etwas kniffliger, da das Überspringen (skip) eher einer negativen Logik entspricht, welche zu Beginn etwas verwirrend sein kann. Zudem gibt es meist auch verschiedene Lösungswege. Eine Möglichkeit würde in Umgangssprache etwa lauten:

1. Wenn der Taster nicht gedrückt ist, dann überspringe das Einschalten der LED
2. Wenn der Taster gedrückt ist, dann überspringe das Ausschalten LED

Die negative Logik des Tasters (gedrückt == „0“ oder „cleared“) erschwert möglicherweise das ganze noch etwas.

Eine Assembler Version des Programms mit der obigen Lösungsvariante könnte so aussehen:

```
RES_VECT    CODE      0x0000    ; processor reset vector
        goto      Init           ; go to beginning of program

Init          ; initialization code
        movlw    0x50            ; set internal oscillator
        movwf    OSCCON          ;   frequency 4MHz
        bcf     LED_1_TRI        ; output for LED
        bsf     BTN_ENC_TRI      ; input for button

Main          ; main code
        btfss   nBTN_ENC        ; button not activated ?
        bcf    nLED_1             ;   ? skip switch on
        btfsc   nBTN_ENC        ; button activated ?
        bsf    nLED_1             ;   ? skip switch off
        bra     Main
```

In „C“ kommt das Ganze auch ohne Kommentare schon etwas verständlicher rüber:

```
void main()
{
    __init();

    while(1) {
        if (BTN_ENC == BTN_ACTIVATED) {
            LED_1 = LED_ON;
        }
        else {
            LED_1 = LED_OFF;
        }
    }
}
```

Das Endergebnis sollte in beiden Fällen das gleiche sein und der vom C-Compiler erzeugte Code dem selbst geschriebenen Assembler-Code ähneln.

3.1.6 Assemblieren / Compilieren des Programms

Die Erzeugung des Maschinencodes, der später in den Controller geladen werden kann, wird über die Menüs **Production | Build Main Project** oder **Production | Clean and Build Main Project** gestartet.

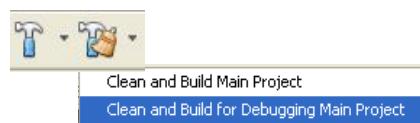
Der Verlauf des Prozesses wird im **Output** Fenster ([2.2 Erster Überblick über IDE am unteren Bildrand](#)?) dokumentiert.

Steht am Ende „**BUILD SUCCESSFUL**“, ist alles in Ordnung.

3.1.6.1 Build Configuration / Release oder Debug ?

Wenn das Programm auf der Zielplattform getestet bzw. Fehler gesucht werden sollen, kann dafür das im Mikrocontroller integrierte Debugmodul genutzt werden.

Damit die Entwicklungsumgebung mit dem InCircuitDebug-Modul kommunizieren kann, müssen zum Anwenderprogramm noch zusätzliches ICD Routinen hinzugefügt werden.



Wird der Build-Prozess über die Icons in der Menüleiste gestartet, dann kann über die „Drop-Down“ Liste ausgewählt werden, ob der Build mit oder ohne Debug-Option ausgeführt wird.

3.1.6.2 Fehlermeldungen

Falls bei diesem Prozess Fehler auftreten, dann werden diese im **Output** Fenster aufgelistet. Diese Meldungen müssen dann natürlich gelesen und zumindest der Versuch gestartet werden, die Meldungen auch zu deuten.

Farblich (blau) hervorgehobene Meldungen zeigen meist die Möglichkeit an, durch einen Doppelklick auf die Meldung an die zugehörige Problemstelle im Quelltext zu springen.

Einige häufig auftretende Fehler, die dem Anfänger oft Schwierigkeiten bereiten, sind im Anhang [13.1 Fehlermeldungen MPLAB / Linker / C18 / PICKit ...](#) aufgelistet.

Fehlermeldungen die im Anhang **noch nicht** enthalten sind und die trotzdem größere Probleme bereiten, bitte kopieren und inklusive des gezippten Projekts an Volker.Schilling-Kaestle@thu.de schicken.

3.1.7 Der erzeugte Maschinencode (XC8 Compiler)

Nachdem für ein Projekt ein **erfolgreicher Build** vorliegt, kann man sich den daraus resultierenden Maschinencode anschauen. Insbesondere wenn das Programm in der Sprache „C“ geschrieben wurde, kann es sehr aufschlussreich sein, zu sehen welche zusätzlichen Module vom Linker hinzugefügt wurden oder wie viele Maschinenbefehle eine Zeile des Sourcecodes zur Folge hat.

Ein einfaches Ansichtsfenster, das später auch zum Debuggen genutzt werden kann, erhält man über das Menü **Window | PIC Memory Views | Program Memory**. Es wird normalerweise im Output-Bereich der IDE angezeigt, lässt sich aber zum einfacheren Vergleich auch beliebig anders anordnen.

Address	Opcode	Label	DisAssy	Line
0000	EFF0		GOTO 0x7FE0	startup 1
0002	F03F		NOP	2
7FDE	FFFF		NOP	16.368
7FE0	0100		MOVLB 0x0	16.369
7FE2	EFF8		GOTO 0x7FF0	startup 16.370
7FE4	F03F		NOP	16.371
7FE6	0E50		MOVWL 0x50	init() 16.372
7FE8	6ED3		MOVWF OSCCON, ACCESS	16.373
7FEA	8293		BSF TRISB, 1, ACCESS	16.374
7FEC	9493		BCF TRISB, 2, ACCESS	16.375
7FEE	0012		RETURN 0	16.376
7FF0	ECF3		CALL 0x7FE6, 0 main()	main() 16.377
7FF2	F03F		NOP	16.378
7FF4	B281		BTFSC PORTB, 1, ACCESS	16.379
7FF6	D002		BRA 0x7FFC	16.380
7FF8	948A		BCF LATB, 2, ACCESS	16.381
7FFA	D7FC		BRA 0x7FF4	16.382
7FFC	848A		BSF LATB, 2, ACCESS	16.383
7FFE	D7FA		BRA 0x7FF4	16.384

Der XC8 Compiler füllt den Programmspeicher nicht immer vom Anfang an. Deshalb ist an der Adresse 0x0000 manchmal nur ein **GOTO** Befehl zum Beginn des Programms irgendwo im Speicher. Dazwischen befinden sich NOP Einträge (*No Operation, Opcode 0xFFFF*), die einem unbeschriebenen bzw. gelöschten Speicher entsprechen.

Beim Löschen werden alle Bits zu Einsen. Beim Programmierungsvorgang können nur Einsen zu Nullen umgewandelt werden, aber nicht umgekehrt. Deshalb erfordert jeder Programmierungsvorgang einen vorausgehenden Löschvorgang.

3.1.7.1 Disassembly Listing File

Für fortgeschrittene Programmierer enthält das Listing-File, welches über den Tab „**Files**“ des Projektfensters erreichbar ist (`<project_name>.production.lst`) jede Menge an Informationen.

```

126      112 ;incstack = 0
127      113 007FF0 ECF3 F03F call    _init ;wreg free
128      114 007FF4
129      115 :uCQ_main.c: 37: if (PORTBbits.RB1 == 0) {
130      116 007FF4 B281 btfsc  3969,1,c ;volatile
131      117 007FF6 D002 goto   117
132
133      118 :uCQ_main.c: 38: LATBbits.LATB2 = 0;
134      119 007FF8 948A bcf    3978,2,c ;volatile
135
136      121 :uCQ_main.c: 39: } else {
137      122 007FFA D7FC goto   116
138      123 007FFC
139

```

Hier kann man unter anderem für jede Zeile im C-Programm die dafür erzeugten Maschinenbefehle sehen. Die Text-Substitution-Labels wurden dabei schon mit ihrem realen Inhalt ersetzt.

3.1.8 Erstes Testen / Debuggen des Programms

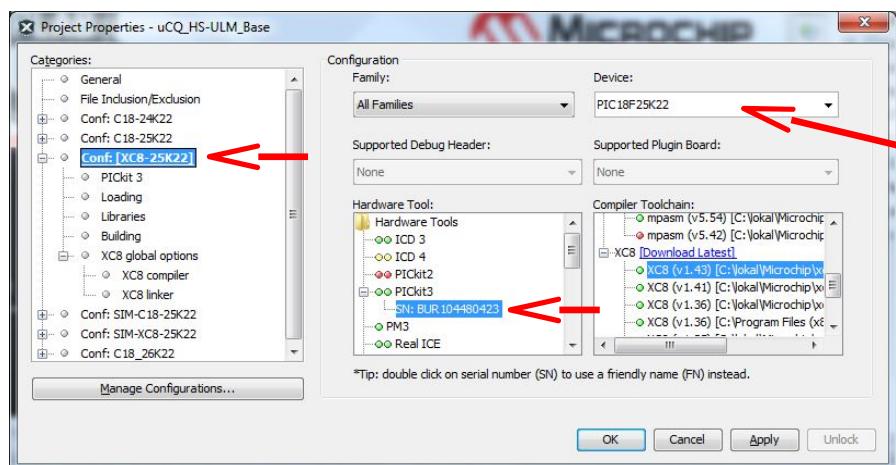
Bei der Entwicklung eines Mikrocontroller Programms muss gewöhnlich mehr Zeit für Tests und Fehlersuche aufgewendet werden, als für das reine „Schreiben“ des Programms. Nachdem der „Build“ Prozess zum ersten Mal erfolgreich abgeschlossen wurde, kann damit begonnen werden. Steht eine Hardware wie die uC-Quick Platine inklusive Debugger PICkit3 zur Verfügung, kann das Programm sofort in der Realität getestet werden. Alternativ kann man auch den Simulator [MPSIM](#) benutzen. Hierzu siehe die Beschreibung in [12.3 Simulator MPSIM](#)

3.1.8.1 ICD System

Alle InCircuitDebugger funktionieren nach dem gleichen Prinzip. Der eigentliche Debugger ist ein Hardwaremodul im PIC selbst. Für die Kommunikation ist ein zusätzliches Softwaremodul erforderlich, welches dem Projekt beim Debug-Build automatisch hinzugefügt wird. Tools wie z.B. PICkit3 sind eigentlich nur das Interface zwischen ICD im Controller und der IDE (*MPLAB-X*).

3.1.8.2 Einrichten des Debuggers (Power)

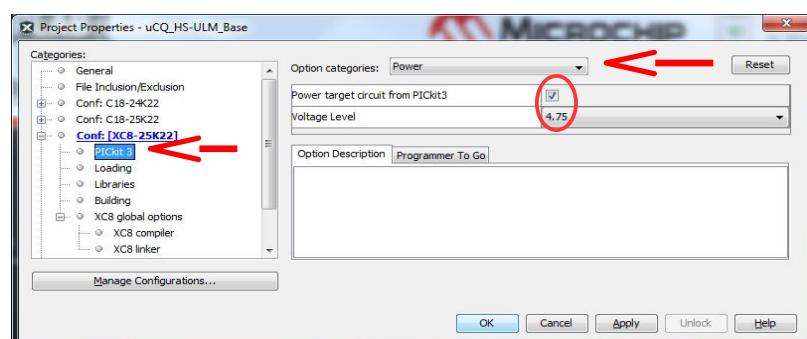
Wurde das zu verwendende Tool nicht schon bei der Projekterstellung konfiguriert, dann kann man dies jetzt im Projekt Properties Dialog nachholen. Der Dialog kann über das Menü **Production | Set Project Configuration | Customize ...**, die Drop-Down Box in der Tool-Leiste oder das Pop-Up Menü im Projekt-Fenster aufgerufen werden.



Ist hier wirklich genau
der Typ ausgewählt,
der sich auf der
angeschlossenen
Hardware befindet?

Bei der Auswahl ist es wichtig, dass nicht nur die Art des Tools (*PICkit3*), sondern auch das angeschlossene Tool selbst (*über Seriennummer oder Namen*) ausgewählt wird.

Soll das ICD-Tool (*PICkit*) auch die **Energieversorgung** für die angeschlossene Hardware zur Verfügung stellen, so muss dies im Untermenü für das Tool eingestellt werden. Meist kann der vorgegebene Wert für die Spannung beibehalten werden, da bei diesem Vorschlag der im Projekt eingestellte PIC schon berücksichtigt wurde. Ist die Hardware für andere Versorgungsspannungen konzipiert, dann muss diese eingehalten werden.

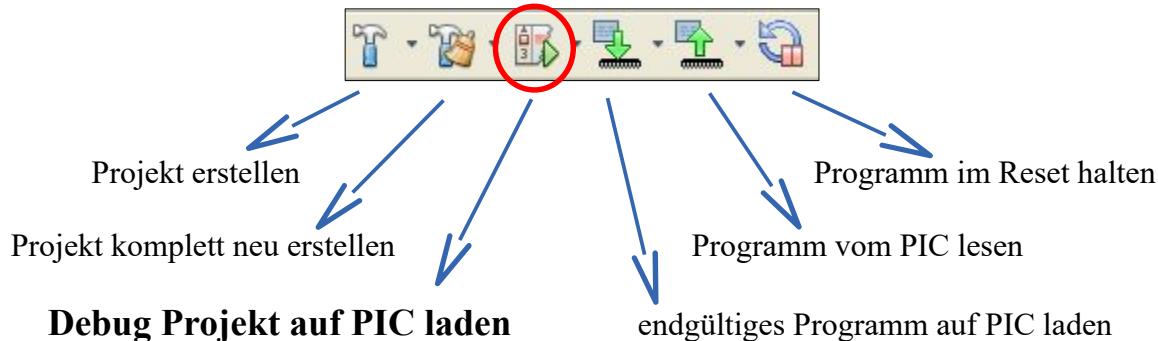


uC-Quick-2013 braucht 3,3V – 5V. An vielen USB Anschlüsse funktioniert es leider nur bis 4,75V
uC-Quick-2018 ist für 3,3V ausgelegt

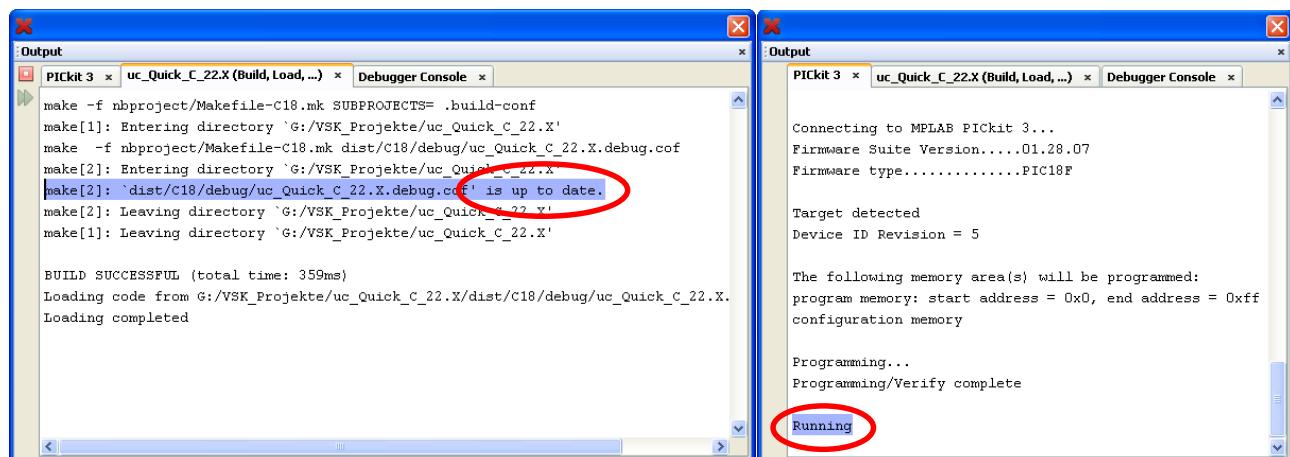
3.1.8.3 Programmieren der Hardware mit dem Debug-Code

Nach dem Übersetzen des Programms in Maschinensprache muss dieses nun als erstes über den Programmer/Debugger (*PICkit3*) in die Zielplattform geschrieben werden.

Dies geschieht normalerweise über das Menü **Debug | Debug Main Project** oder das entsprechende Symbol in der Toolbar.



Debug Main Project überprüft zuerst ob das Projekt aktuell ist und führt gegebenenfalls eine Neuerstellung durch. Danach wird die Kommunikation zum Debug-Tool und dem angeschlossenen PIC auf der Entwicklungsplattform aufgebaut. Wird ein passender PIC detektiert, dann wird das Programm auf diesen geladen und gleich gestartet.



Damit der beschriebene Vorgang auch bis zum Ende durchläuft, müssen verschiedene Bedingungen erfüllt sein. Häufige Fehlermeldungen sind z.B. :

Target device was not found. ...You must connect to a target device to use PICkit 3.

Eine der Grundvoraussetzungen ist eine korrekte Stromversorgung des Controllers auf der Zielhardware. Ist diese nicht vorhanden, dann wird er nicht erkannt.

Target Device ID (0x?????) does not match expected Device ID (0x5540).

Falsche Controller-Familie ausgewählt. Keine Kommunikation möglich.

Tool falsch oder gar nicht mit Zielplattform verbunden. (*Liest nur „Nullen“ bzw „Einsen“*)
Ist die „Target Device ID“ nicht 0x0000 oder 0x3FE0, 3FFF, 0xFFE0, 0xFFFF0000 (Masken), dann sind eingestellter und angeschl. PIC nicht identisch.

**Programming/Verify complete / The target device is not ready for debugging.
Please check your configuration bit settings and program the device before proceeding.**

Für das Debuggen muss der PIC mit der IDE kommunizieren. D.h. das Programm auf der Zielplattform muss laufen ! **Der Oszillator muss korrekt eingestellt sein und funktionieren !!!**

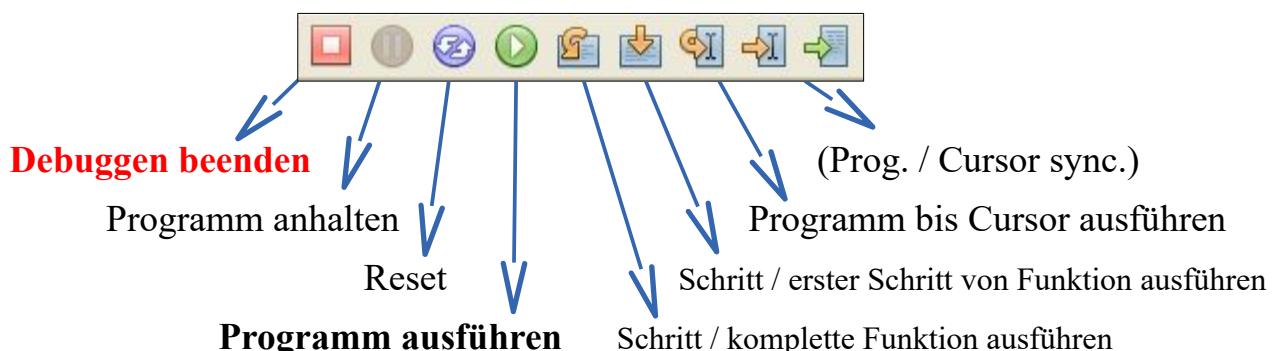
PICkit 3 is trying to supply 5,000000 volts from the USB port, but the target VDD is ...

Die Zielplattform kann vom PICkit nicht ausreichen versorgt werden. **Kleinere Spannung einst.**

3.1.8.4 Ausführen, Anhalten und Reset des Programms

Nach dem Programmieren der Hardware wird das Programm normalerweise gleich gestartet.
(kann man im Menü Tools | Options | Embedded | Generic Settings | Debug startup konfigurieren)

Die Folgende Abbildung zeigt die Menüleiste nachdem die „Pause ||“ Taste gedrückt wurde.



- **Continue:** Die LED_1 (an RB2) geht an, wenn die Taste TM (an RB1) gedrückt wird. Keine Bewegungen im Editor und im Disassembly-Listing.
- **Pause:** Programm stoppt. LED_1 reagiert nicht mehr auf Taster. Grüne Zeiger und grün hinterlegte Zeilen verweisen auf den Befehl der als nächstes ausgeführt werden würde.
- **Step ...:** LED reagiert verzögert. (dann wenn der grüne Zeiger auf dem Befehl stand)
Bei den Step-Funktionen bietet es sich manchmal an die zugehörigen Funktionstasten zu benutzen und diese gedrückt zu halten. (Programm läuft dann sehr langsam durch)
- **Reset:** Assembler Programm wird auf die Startadresse 0x0000 zurückgestellt. Bei C-Programmen auf die erste Anweisung der Funktion „main()“. *Soll Das Programm nicht automatisch bis zu „main()“ vorlaufen, kann dies im Menü Tools | Options | Embedded | Generic Settings | Reset @ geändert werden.*

Alle Funktionen des Debugger Menüs sollten jetzt ausgiebig getestet werden.

Bei C Programmen ist oft die zusätzliche Anzeige des Maschinen-Codes informativ und hilfreich.
 Menü -> **Window | PIC Memory Views | Program Memory**

Das Bild zeigt das Entwicklungsumgebungsfenster mit zwei Hauptbereichen:

- Quellcode (Source View):** Zeigt den C-Code der Funktion main():


```
32 void main(void)
33 {
34     __init();
35
36     while (1) {
37         if (BTN_ENC == BTN_ACTIVATED) {
38             LED_1 = LED_ON;
39         } else {
40             LED_1 = LED_OFF;
41         }
42     }
43 }
```
- Programm-Memory View (Program Memory View):** Zeigt die maschinellen Befehle (Opcode, Adresse, Label, Disassembly) für die Funktion main(). Die Zeile mit der Änderung des LED-Werts (Zeile 40) ist hellgrün hervorgehoben.

Line	Address	Opcode	Label	Disassembly
16.372	7FE6	0E50	_init	MOVlw 0x50
16.373	7FE8	6ED3		MOVwf OSCCON, ACCESS
16.374	7FEA	8293		BSF TRISB, 1, ACCESS
16.375	7FEC	9493		BCF TRISB, 2, ACCESS
16.376	7FEE	0012		RETURN 0
16.377	7FF0	ECD3	main	CALL 0x7FE6, 0
16.378	7FF2	F03F		NOP
16.379	7FF4	B281		BTFS PORTB, 1, ACCESS
16.380	7FF6	D002		BRA 0x7FFC
16.381	7FF8	948A		BCF LATB, 2, ACCESS
16.382	7FFA	D7FC		BRA 0x7FF4
16.383	7FFC	848A		BSF LATB, 2, ACCESS
16.384	7FFE	D7FA		BRA 0x7FF4

Falls die LED_1 immer leuchtet und nicht auf die Taste reagiert, bzw. das Programm nie in die Zeile zum Ausschalten springt, dann könnte das daran liegen, dass der zum Taster zugehörige Eingang nicht als digitaler Eingang konfiguriert wurde.

→ Zurück zum Kapitel Pins konfigurieren!

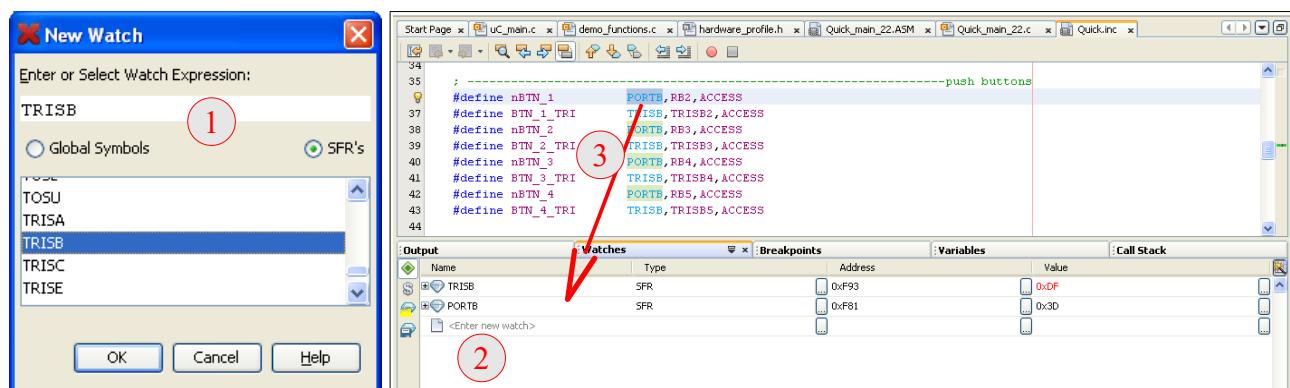
3.1.8.5 Anzeigen von Registern

Reagiert ein Programm nicht wie erwünscht, ist es natürlich notwendig nach den Ursachen zu forschen. Bei dem Programm „HelloPIC“ wäre z.B. die Anzeige der verwendeten Ein- und Ausgänge hilfreich. Im sogenannten „**Watches**“ Fenster kann man die gewünschten Register anzeigen lassen und Änderungen beobachten. Ähnlich zu „**Watches**“ ist das „**Variables**“ Fenster.

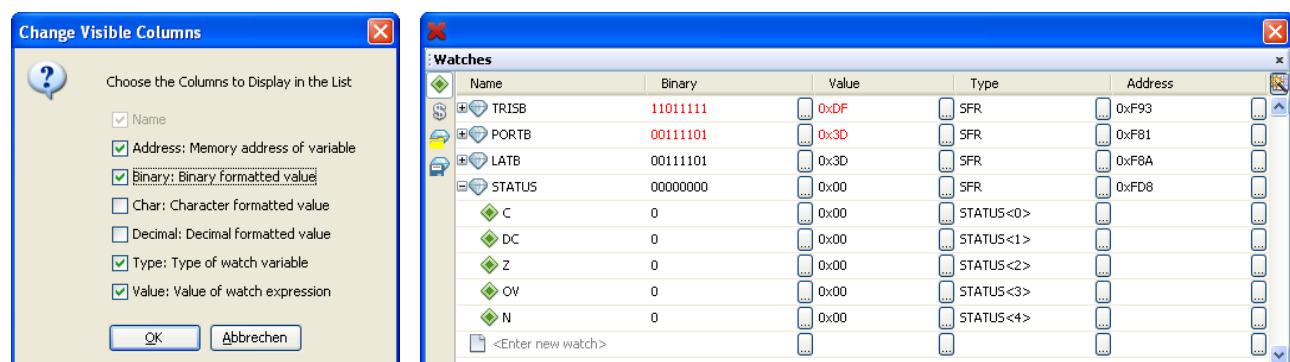
Ist im Output-Bereich kein Fenster mit dem Namen „**Watches**“ sichtbar, so kann man dies über das Menü **Window | Debugging | Watches** öffnen. Leider können die Port-Pins hier nicht mehr mit den vorher vergebenen (`#define`) Namen zur Anzeige ausgewählt werden, sondern nur über den „wirklichen“ Namen des beinhaltenden Registers. Die entsprechenden Register können der Projekt-Header-Datei oder dem Schaltplan entnommen werden.

Da einzelne Bits nicht separat ausgewählt werden können, sollten also die Register **PORTB**, **LATB** und auch das Richtungsregister **TRISB** ausgewählt werden. Dies kann auf mehrere Arten erfolgen.

1. über das Pop-Up Menü (klick rechte Maustaste) im Editor- oder Watch-Window (**New Watch...**)
2. direkte Eingabe unter **<Enter new watch>** im Watch Window (Groß- Klein-schreibung beachten)
3. durch Ziehen eines im Editor markierten Registers oder einer Variablen in das „**Watch**“ Fenster.



Die Anordnung einzelner Spalten kann durch Ziehen mit der Maus in der Titelleiste verändert werden. Ein Klick mit der rechten Maustaste in der Titelleiste öffnet den Dialog **Change Visible Columns**, in dem weitere Anzeigeformate ausgewählt werden können. Klick mit der linken Maustaste ändert bei bestimmten Spalten die Sortier-Reihenfolge der Watches (**Zeilen**)



Durch Anklicken des „+“ Symbols vor den Registrernamen können auch die einzelnen Bits separat angezeigt werden, was besonders bei Registern wie **STATUS** sehr nützlich sein kann.

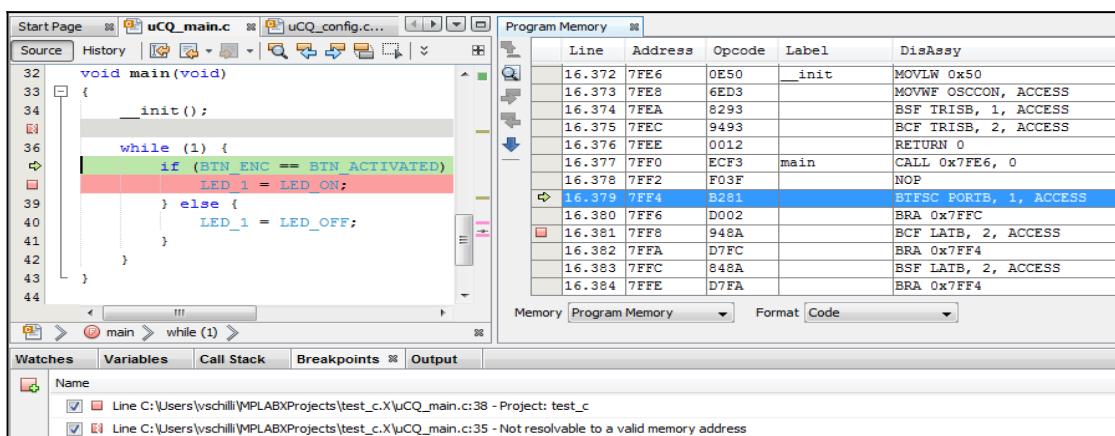
Die Werte der angezeigten Register können natürlich **nur bei einem Halt (Einzel schritt)** aktualisiert werden. Rot werden die Werte angezeigt, welche sich seit dem letzten Halt geändert haben.

Für das vorliegende einfache Programm kann nun überprüft werden, ob die Richtungsbits richtig gesetzt wurden, ob der Eingang PORTB_1 auf Tastendruck reagiert und ob das Bit für die LED im Register LATB bei Tastendruck geändert wird. (*Continue → Pause → Prüfen → Continue ...*)

3.1.8.6 Program Breakpoint

Bei größeren Programmen kann die schrittweise Ausführung schnell etwas mühsam werden und oft will man das Programm auch nur an einer bestimmten Stelle überprüfen. In diesem Fall kann man das Programm bis zu dieser Stelle in Echtzeit ausführen und dann mit Einzelschritten arbeiten.

Einen Haltepunkt (*Breakpoint*) an einer bestimmten Stelle im Programmcode kann man setzen, mit einem Klick im grauen Randbereich an der linken Seite des Editorfensters, oder über das Pop-up Menü des Editorfensters. Es erscheint dann ein **rotes Quadrat** an der jeweiligen Zeile.



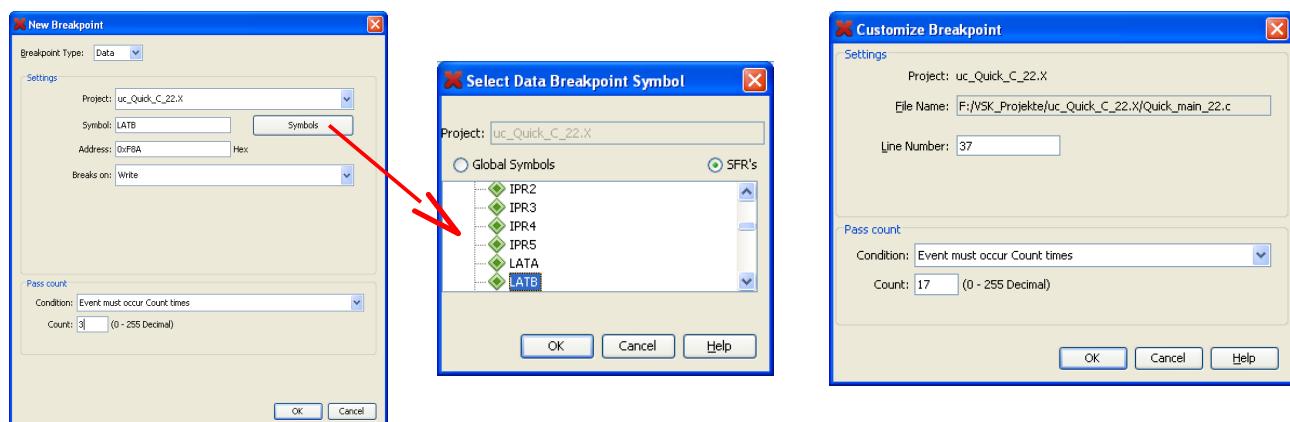
Wird ein **zerbrochenes Quadrat** angezeigt, dann ist das Setzen eines Haltepunktes an dieser Stelle nicht möglich. (*der Grund dafür ist leider nicht immer offensichtlich*)

Die Steuerung des Programms erfolgt genau wie oben schon beschrieben über die **Continue**, **Pause** und **Step** Befehle. Durchläuft das Programm einen Breakpoint, dann wird es automatisch gestoppt.

3.1.8.7 Data Breakpoint

Manchmal kann es vorkommen, dass ein Register unerwartet veränderte Werte aufweist und es ist nicht sofort ersichtlich welcher Teil des Programms diese Änderung hervorgerufen hat.

In diesem Fall kann man einen Daten-Haltepunkt einfügen um die entsprechende Stelle zu finden.



Datenhaltepunkte können nur über das Menü **New (Data) Breakpoint** eingegeben werden, welches über das Debugger Menü oder die Pop-Up Menüs von Editor und Breakpoint Window aktiviert wird. Bei diesen Haltepunkten kann man zusätzlich auswählen, ob das Programm bei **Schreib-** oder **Lesezugriff** gestoppt werden soll, oder nur wenn das Register einen **bestimmten Wert** erhält.

3.1.8.8 Breakpoint Pass-Count

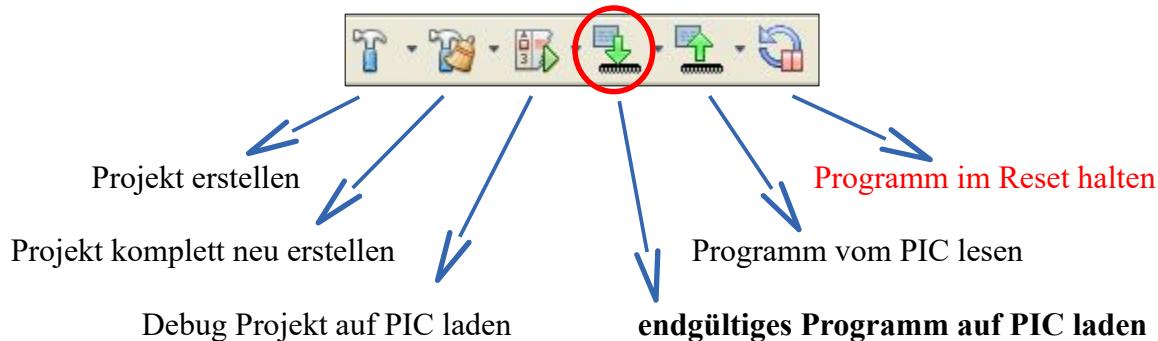
Sowohl bei Programm- als auch Daten- Haltepunkten kann man, über **Customize** im Breakpoint Fenster (Pop-Up Menü), zusätzlich noch einstellen, beim wievielen Eintreten der Bedingung angehalten werden soll. Hierfür muss man die **Condition** ändern (*Always Break → Event must occur Count times*) und **Count** entsprechend einstellen.

3.1.9 Programmieren der endgültigen Software

Ist die Programmentwicklung abgeschlossen, dann kann der Controller mit dem endgültigen Programm beschrieben werden, welches dann auch ohne ICD-System und ohne Verbindung zum Entwicklungsbrett lauffähig ist.

Nach dem Übersetzen des Programms in Maschinensprache muss dieses nun als erstes über den Programmer/Debugger (*PICKIT3*) in die Zielplattform geschrieben werden.

Dies geschieht normalerweise über das entsprechende Symbol in der Toolbar.



Läuft das Programm nach der Programmierung nicht wie erwartet, so kann dies verschiedene Ursachen haben:

1. Die Schaltfläche „Hold in Reset“ ist gedrückt (*anstelle des roten Pause Zeichens ist ein grünes Play Dreieck und es erscheint der Hinweis „Release from Reset“*)
2. Bleibt das Programm nach Entfernen des Programmiergerätes stehen, oder reagiert seltsam, dann ist vermutlich der /MCLR Pin aktiv geschaltet, hängt jedoch „in der Luft“ (*ist nirgends angeschlossen*)
3. Wurde die Zielplattform über das PICkit mit Strom versorgt, dann fehlt diese Stromversorgung nach abziehen vom PICkit ...
4. ...

3.2 Hello World II / LEDs umschalten

Das erste „hello world“ Beispiel für Mikrocontroller ist wie die meisten solcher Beispiele in der Realität wenig anwendbar. Normalerweise will man einen Schalter nicht gedrückt halten, damit das Licht an bleibt. Die Aktion, also das Ein- bzw. Ausschalten, soll beim Drücken ausgeführt werden und dann bestehen bleiben, bis man die Taste wieder drückt.

Dieses gewünschte Verhalten zu programmieren, führt bei Anfängern gewöhnlich zu einem kleinen „Aha Effekt“ in Bezug auf die Ausführungsgeschwindigkeit eines Programms. Die Ergebnisse der ersten Versuche verhalten sich oft wie eine Art Zufallsgenerator.

Das folgende Programm könnte ein solcher erster Ansatz sein. Um LEDs um zu schalten.

- Beim Start sollen LED_1 und LED_2 leuchten und LED_3 und LED_4 aus sein
- Bei jedem Tastendruck am Encoder dann soll umgeschaltet werden

Mit den vordefinierten Makros aus dem Anhang könnte das zunächst wie folgt aussehen:

```
#include "uCQ_2013.h"

void __init(void)
{
    mSET_LED_1_ON(); mSET_LED_2_ON();      // LED 1 & 2 ON
    mSET_LED_3_OFF(); mSET_LED_4_OFF();    // LED 3 & 4 OFF
    mALL_LED_OUTPUT();                    // set LED pins as output
    ENC_BTN_TRI = INPUT_PIN;             // set encoder button pin as input
    ENC_BTN_ANS = DIGITAL_PIN;          // set pin as digital input
}
void main(void)
{
    __init();                            // initialization

    while(1) {
        if(mGET_ENC_BTN()) {            // if encoder button is pressed
            mTOG_LED_1();              //   toggle the LEDs
            mTOG_LED_2();
            mTOG_LED_3();
            mTOG_LED_4();
        }
    }
}
```

Testet man dieses Programm, dann sieht man unter Umständen ein unerwartetes Verhalten:

- Beim Start leuchten die beiden linken LEDs wie gefordert
- Drückt man die Taste, leuchten alle (4) LEDs (*jede etwas schwächer als wenn nur zwei...*)
- Beim Loslassen der Taste leuchten entweder die linken, oder die rechten LEDs
- welche LEDs nach dem Loslassen leuchten ist nicht vorhersagbar (*Zufallsgenerator*)

Das zu beobachtende Verhalten kommt daher, dass bei der Abfrage des Tasters nicht berücksichtigt wurde, ob dieser vorher nicht gedrückt war. Somit ist die Bedingung, ob der Taster gedrückt ist, immer wahr, solange er gedrückt bleibt. Die LEDs werden also immer umgeschaltet (*schwächer*).

Weil der komplette Durchlauf der Schleife, inklusive der Abfrage des Tasters und dem Umschalten der LEDs, bei der Standardeinstellung des Oszillators nur 32µs dauert, geschieht dies jedoch alles 31250 mal pro Sekunde. Welche LEDs dann beim Loslassen gerade eingeschaltet sind ist zufällig.

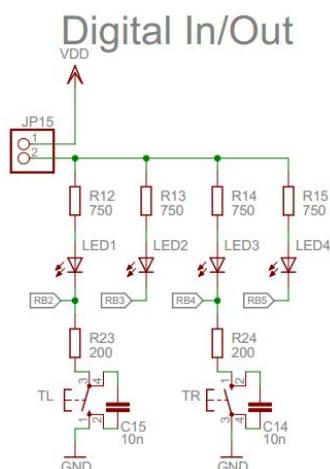
Als Problemlösung könnte man bei diesem sehr einfachen Programm auf das Loslassen warten:

```
mTOG_LED_4();
while (mGET_ENC_BTN()){}; // do nothing while waiting for button release
}
```

3.3 Pins „gleichzeitig“ als Ein- und Ausgang benutzen

Es ist möglich, in einem Programm die gleichen Pins für Tasten und LEDs zu benutzen. Natürlich geht das nicht wirklich zur gleichen Zeit, sondern es muss nacheinander geschehen.

Durch die hohe Ausführungsgeschwindigkeit merkt der Bediener von dieser eigentlich sequenziellen Benutzung nichts. Für seine Wahrnehmung passiert noch immer alles gleichzeitig.



Die Taster L/R sind z.B. zusammen mit den LEDs 1 und 3 an den gleichen Pins angeschlossen. Mit Taster TL sollen die LEDs 1 + 2, mit TR die LEDs 3 + 4 gesteuert (*umgeschaltet*) werden.

Beim Umschalten ist noch ein zusätzlicher Effekt zu berücksichtigen. Die Kondensatoren, die an den Schaltern zur Entprellung dienen, brauchen etwas Zeit, um bei geöffnetem Schalter aufgeladen zu werden, falls vorher der zugehörige Ausgang vorher Low-Pegel geführt hat. Im ersten Moment wirken sie dann wie ein Kurzschluss, also wie ein geschlossener Schalter.

Die erforderliche Zeit zum Laden kann man leicht über die Ladezeitkonstante $\tau = R_{23} \cdot C_{15}$ berechnen. Nach 5τ , also $\sim 10\mu\text{s}$ ist der Kondensator annähernd vollständig geladen.

Zur Überbrückung der erforderlichen Wartezeit kann man das XC8 Compiler Makro `delay_us(x)` nutzen.

```
#include "uCQ_2013.h"

#define _XTAL_FREQ 1000000 // 1MHz? -> PIC data sheet...

void __init(void)
{
//    OSCCONbits.IRCF = IRCF_1MHZ; OSCTUNEbits.PLLEN = 0; // (default setting)
    mSET_LED_1_ON(); mSET_LED_2_ON(); // LED 1 & 2 ON
    mSET_LED_3_OFF(); mSET_LED_4_OFF(); // LED 3 & 4 OFF
    mALL_LED_OUTPUT(); // set LED pins as output
}

void main(void)
{
    __init(); // initialization

    while(1){
        BTN_L_TRI = INPUT_PIN;
        __delay_us(10); // -> XC8 user guide...
        if(mGET_BTN_L()){
            mTOG_LED_1();
            mTOG_LED_2();
            while (mGET_BTN_L()) {}
        }
        LED_1_TRI = OUTPUT_PIN;

        // add code for button_R and LED_3+4
    }
}
```

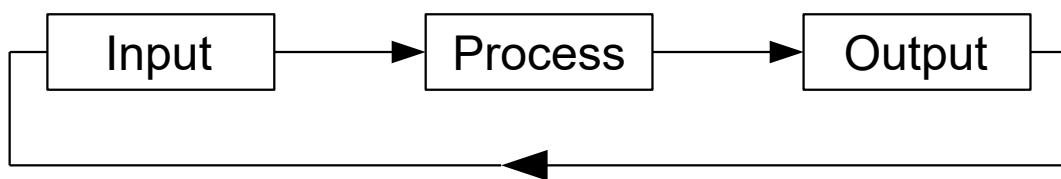
Das `delay_us(x)` Makro des Compilers braucht die Information, wie schnell der Prozessor getaktet wird um eine Zählschleife zu generieren. Das Delay entsteht durch die Abarbeitung dieser Zählschleife. Der Default Wert des internen Oszillators ist 1MHz. Zum Abarbeiten der einfachsten Befehle werden 4 Takte des Oszillators benötigt. Dies entspricht dann 4us.

Aus diesen Rahmenbedingung kann man schließen, dass bei relativ langsamer Taktfrequenz und gleichzeitig kleinen Delays, diese nicht unbedingt sehr genau werden. (hier auch nicht erforderlich)

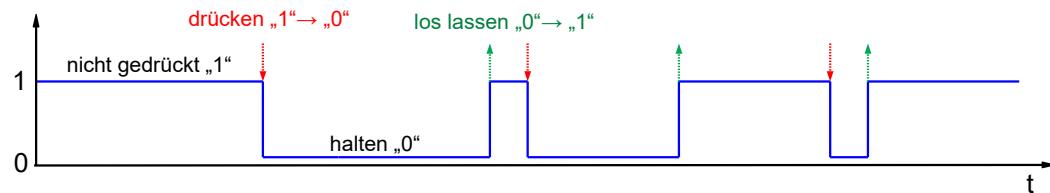
3.4 Hello World III / IPO Pattern (input-process-output)

Die Lösung aus den vorhergehenden Kapiteln, das Drücken eines Tasters durch das nachfolgende Warten auf das Loslassen zu erkennen, führt sehr schnell zu einem neuen Problem. Was passiert, wenn zwei Taster gleichzeitig gedrückt werden?

Man kann nicht auf das Loslassen eines Tasters warten, weil in der Zwischenzeit der andere beliebig oft betätigt werden könnte und man dies dann nicht mitbekommt. Ein einfaches und bewährtes Muster zur Lösung dieses Problems führt über das IPO Model. Wie der Name schon sagt, werden bei Anwendung dieses Modells zunächst sämtliche Eingangssignale erfasst. Danach werden die Eingangssignale verarbeitet und am Ende findet eine Ausgabe statt.



Das IPO Modell alleine löst allerdings nicht das Problem der Unterscheidung, ob ein Taster gerade gedrückt wurde, oder ob er schon länger gedrückt ist. Im zeitlichen Verlauf stellt sich das so dar:



Die Aktion soll beim Drücken des Tasters, also dem Wechsel des Eingangssignals von logisch „1“ nach logisch „0“ ausgelöst werden. Um dies erkennen zu können, braucht man nicht nur den momentanen Zustand des Eingangssignals vom Taster, sondern auch den vorhergehenden. Man muss den Zustand also speichern. Da auch der Folgezustand der LEDs während der Verarbeitung geändert wird, und die Pins erst bei der Ausgabe aktualisiert werden, muss auch dafür eine Variable angelegt werden. Im folgenden Beispiel werden „globale Variablen“ verwendet, die sowohl in der Funktion `main()` als auch in der Funktion `__init()` verändert werden können.

```
#include "uCQ_2013.h"      // (uCQ_2018.h)

#define _XTAL_FREQ 1000000          // 1MHz? -> PIC data sheet...

unsigned char buttons_ago, buttons_now, leds;    // use global variables

void __init(void)
{
    mSET_LED_1_ON(); mSET_LED_2_ON(); mSET_LED_3_OFF(); mSET_LED_4_OFF();
    mALL_LED_OUTPUT();
    BTN_L_ANS = BTN_R_ANS = DIGITAL_IN;

    buttons_ago = 0b00010100;        // initialize as not pressed
    leds = LATB & 0b00111100;       // initialize leds variable
}
```

Globale Variablen werden außerhalb der Funktionen definiert. Ihr Gültigkeitsbereich beschränkt sich zunächst auf die Datei in der sie definiert wurden.

Variablen muss vor dem ersten Lesezugriff ein Wert zugewiesen werden. Dies Initialisierung betrifft den vorhergehenden Zustand der Taster und auch den Zustand der LEDs, weil diese ja umgeschaltet werden sollen und dafür der Einschalt-Zustand bekannt sein muss.

Für die Variablen werden 8-Bit Typen (*unsigned char*) verwendet, in denen die kompletten Zustände des PORTS bzw. des LATCHS, an dem Taster und LEDs hängen, Platz finden.

Die nicht relevanten Bits werden dann jeweils über **UND-Verknüpfungen maskiert**. In den Masken steht zu diesem Zweck nur an den relevanten Stellen eine „1“.

Man kann solche Masken für verschiedene Zwecke anlegen und mit verschiedenen logischen Operatoren verwenden um Bits zu löschen, zu setzen, oder auch umzuschalten (*toggeln*).

- Löschen Operator & alle Bits mit „0“ in der Maske werden gelöscht ($\rightarrow 0$)
- Setzen Operator | alle Bits mit „1“ in der Maske werden gesetzt ($\rightarrow 1$)
- Toggeln Operator ^ alle Bits mit „1“ in der Maske werden getoggled ($1 \rightarrow 0, 0 \rightarrow 1$)

```
#define MASK_BTNS          ???
#define MASK_LEFT_BTN        ???
#define MASK_TGL_LEFT_LEDS  ???

void main(void)
{
    __init();

    while(1 {

// input -----
        BTN_L_TRI = BTN_R_TRI = INPUT_PIN;
        __delay_us(10);
        buttons_now = PORTB & MASK_BTNS;           // read all buttons
        LED_1_TRI = LED_3_TRI = OUTPUT_PIN;

// process -----
        if(buttons_now != buttons_ago){           // something happened
            if( (buttons_ago & MASK_LEFT_BTN) > (buttons_now & MASK_LEFT_BTN) ){
                leds = leds ^ MASK_TGL_LEFT_LEDS;
            }

            // add code for the right part of the board

            buttons_ago = buttons_now;           // prepare for next run
        }

// output -----
        LATB = leds;
    }
}
```

Etwas unschön ist am obigen Programm noch, dass die LEDs, welche sich den Pin mit einem Taster teilen, beim Drücken leuchten. Das kommt daher, dass solange der Pin zur Abfrage des Tasters als Eingang fungiert, nur der Taster und nicht der Controller bestimmt, ob die LED leuchtet oder nicht.

Da das Programm im Moment noch ohne bestimmte Zeitscheibe abläuft, dauert der Teil in dem die Pins als Input geschaltet sind, fast so lange wie der Rest. Bei gedrücktem Taster leuchtet die LED also fast die Hälfte der Zeit. Abhilfe kann man **vorläufig** schaffen, indem man die Schleife durch ein Delay abbremst. Dafür kann man die Funktion `__delay_ms()` verwenden, über die man sich im UserGuide des Compilers informieren kann. Damit der Compiler die benötigte Anzahl von Maschinenzyklen für die Zeit berechnen kann, muss das Makro `_XTAL_FREQ` definiert werden!

```
// output
    LATB = leds;

// slow down
    __delay_ms(x);    // insert x for running the loop ~100 times/s
}
```

3.5 Eigenschaften von I/O Pins

3.5.1 Maximale Ausgangsströme der I/O Pins (25mA?)

Auf der Titelseite vom PIC18(L)F2X/4XK22 Data Sheet werden **High-Current Sink/Source** Ströme von **25 mA/25 mA** angepriesen. Schaut man nicht noch etwas genauer hin, kann man sich eventuell wundern, dass der Ausgang doch nicht ganz das liefert, was man erwartet hat.

Die versprochenen 25mA gelten leider nur bei einer Versorgungsspannung von 5V und über die Spannung welche dabei aufrecht erhalten werden kann sollte man sich auch noch klar werden.

Im Kapitel **28.0 DC AND AC CHARACTERISTICS GRAPHS AND CHARTS** kann man sich die etwas detaillierteren Zusammenhänge in zwei Diagrammen anschauen

FIGURE 28-85: PIC18(L)F2X/4XK22 OUTPUT LOW VOLTAGE

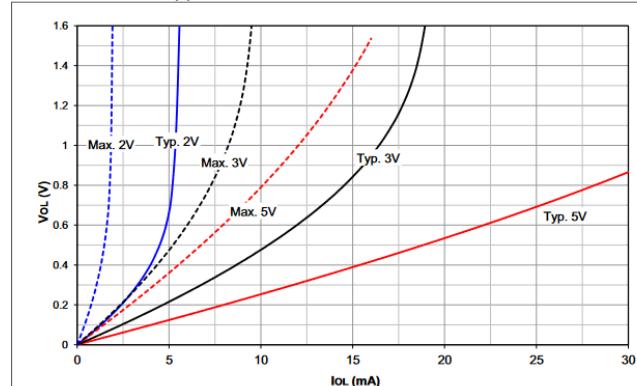
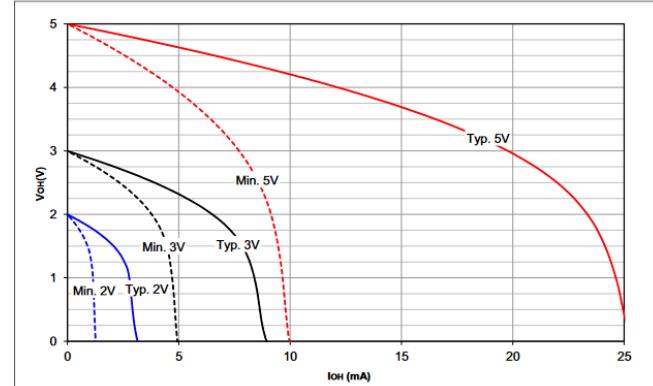


FIGURE 28-86: PIC18(L)F2X/4XK22 OUTPUT HIGH VOLTAGE



Will man direkt an einem Ausgang eines mit 5V versorgten PICs eine blaue LED mit einer *Forward Voltage* von ~3,5V mit einem Strom von 20mA betreiben, so ist nur eine „**Active-Low**“ Ansteuerung möglich. D.h die LED muss so angeschlossen werden, dass sie bei Low-Pegel am Ausgang des PICs leuchtet, also aktiv wird und bei High-Pegel nicht leuchtet.

Der Grund dafür ist, dass bei einem geforderten Strom von 20mA am Logic-High Ausgang die Spannung typischer Weise auf 3V absinkt. D.h. Spannungsverlust im PIC selbst beträgt hier 2V. (*Der max. Strom bei 3,5V liegt typischerweise nur bei 15mA.*)

Bei der Active-Low Ansteuerung hingegen, steigt der Low-Pegel bei 20mA deutlich weniger an, als der High-Pegel abfällt. Hier entsteht bei einer Versorgung mit 5V nur ein Spannungsabfall von ~0,5V und die blaue LED könnte problemlos betrieben werden. Der erforderliche Vorwiderstand wäre $R_V = V_{CC} - V_{OL} - V_{LED} / I_{LED} = (5V - 0,5V - 3,5V) / 20mA = 1V / 20mA = 50\Omega$.

Ist die Versorgungsspannung kleiner als 5V sinkt die Belastungsfähigkeit der Ausgänge recht stark, wovon die High-Pegel weiterhin viel stärker betroffen sind, als die Low-Pegel.

Fazit: Die Leistungsfähigkeit der PIC18FxxK22 Ausgänge ist also sehr stark abhängig von der Versorgungsspannung und bei Active-Low Ansteuerungen weit besser als bei Active-High.

3.5.2 Logic Level von TTL und Schmitt-Trigger Eingängen

PIC Eingänge von können verschiedene Charakteristiken aufweisen. Diese können sogar am selben Pin je nach Konfiguration unterschiedlich sein.

Im Datenblatt finden sich die Bezeichnungen *Analog*, *TTL*, *CMOS* und *ST*. Für digitale Eingänge werden TTL und Schmitt-Trigger Charakteristiken verwendet, die sich folgendermaßen unterscheiden:

Bei **TTL** und gibt es einen festen Spannungspegel, bei dem das Signal von logisch 0 zu 1 wechselt und umgekehrt auch von 1 auf 0 beim selben Wert der Eingangsspannung.

ST (Schmitt-Trigger) Eingänge verhalten sich etwas anders. Zum Wechsel von einer logischen 0 zur 1 ist eine höhere Spannung erforderlich als beim Wechsel von 1 nach 0.

Daraus folgt, dass wenn ein Signal gerade von 0 nach 1 gewechselt hat, ist ein größerer Abfall erforderlich um wieder zur logischen 0 zu gelangen und umgekehrt nach einem Wechsel von 1 nach 0 ein größerer Anstieg um wieder eine logische 1 zu erhalten.

Die damit erzielte Charakteristik ist vorteilhaft bei sich relativ langsam ändernden Eingangssignalen, die möglicherweise zusätzlich mit kleinen Störsignalen verunreinigt sind. Ohne Schmitt-Trigger Charakteristik könnte so eine einzige langsame Änderung viele Änderungen der internen logischen Signale zur Folge haben.

ST Eingänge werden meist dann verwendet, wenn Änderungen der Eingänge direkte Auswirkungen haben. Interrupt-, Timer-, CCP-Module ...

Auf der uC-Quick Platine wird das Signal des Einstellers R11 auf den Pin RA0_AN0 gegeben. Ist dieser Pin nicht als Analog-Input konfiguriert so weist er TTL Charakteristik auf.

TODO Bilder ...

Zusätzlich kann man das Signal mit dem Jumper JP9 auf den PIN RC3_SCL leiten, welcher eine Schmitt-Trigger Eingangsschaltung besitzt.

Mit einem kleinen Programm aus dem uCQ_HS-ULM_Base Projekt kann man jetzt die Unterschiede sichtbar machen.

Die im Programm mit den jeweiligen Eingängen verknüpften LEDs verhalten sich unterschiedlich. Hat man ein Multimeter zur Hand, kann man auch die entsprechenden Spannungspegel ermitteln.

```
#include "uCQuick/uCQ_201x.h"

void TTL_ST_toLED(void)
{
//----- __init__()
    TTL_IN_TRI = INPUT_PIN;
    ST_IN_TRI = INPUT_PIN;
    LED_1_TRI = OUTPUT_PIN;
    LED_2_TRI = OUTPUT_PIN;
    mSET_LED_3_OFF();
    LED_3_TRI = OUTPUT_PIN;
    mSET_LED_4_OFF();
    LED_4_TRI = OUTPUT_PIN;
//----- main()
    while(1) {
        // VDD = 5V
        LED_1 = !TTL_IN; // ON/OFF ~1.3V
        LED_2 = !ST_IN; // ON ~ >2.9V OFF ~ <1.4V
    }
}
```

3.5.3 Interne WEAK PULL-UPS

Pull-Up (*oder* Pull-Down) Widerstände benötigt man beispielsweise bei Eingängen die den Zustand eines Schalters bzw. Taster abfragen sollen.

Wenn der Schalter geschlossen ist, dann wird der Eingang fest mit einem bestimmten Spannungspegel verbunden und dieser kann eindeutig einer logischen 0 oder 1 zugeordnet werden.

Bei geöffnetem Schalter ohne zusätzliche Beschaltung „hängt der Eingang in der Luft“.

Das Fehlen einer äußeren Spannung bedeutet hier aber nicht, dass der Eingang auf 0V liegt.

Vielmehr ist es so, dass durch den offenen Eingang kein Bezug zum 0V Potential hergestellt werden kann und der ermittelte Logikpegel zufällig entsteht, welcher in der Regel sogar High entspricht.

Den fehlenden Bezug bei geöffneten Schaltern stellen die zusätzlichen Pull-Up oder Pull-Down Widerstände her.

Beim PIC18FxxK22 können alle Pins von PORTB individuell und zusätzlich PORTE 3 mit einem internen Pull-Up Widerstand konfiguriert werden. Die Pull-Ups sind nur aktiv, wenn der jeweilige Pin auch als Eingang konfiguriert ist. Nach einem Power-On-Reset sind die Pull-Ups deaktiviert.

Um den ungefähren Wert des Widerstandes zu bestimmen kann man aus der Tabelle **DC Characteristics: Input/Output Characteristics** den **Weak Pull-up Current** und die zugehörigen Versorgungsspannung entnehmen und dann mit $R = U/I$ ausrechnen.

$$R = 5V / 130\mu A = 38,5 \text{ kOhm.}$$

TODO Beispiel ...

3.6 I/O Übung 1: Taster lassen LED-Anzeige rotieren

Als abschließendes Übung zum Kapitel digitale Ein- und Ausgänge kann man ein Programm entwerfen, welches eine aktive LED auf der Übungsplatine mit der linken Taste nach links und mit der rechten Taste nach rechts verschiebt. Wandert die leuchtende LED auf einer Seite raus, dann soll die erste LED der anderen Seite leuchten, sodass immer eine LED leuchtet.

Für ein Assembler Programm suche man dafür im Data Sheet nach „rotate“.

Bei einem C Programm wäre das Stichwort „shift“

4 Zeitgesteuerte Abläufe

Sollen Vorgänge zeitgesteuert ablaufen, kann man dies dadurch erreichen, dass man den Controller zählen lässt. Die Zeit die für einen Zählschritt vergeht, kann relativ einfach berechnet werden und man muss dann nur noch festlegen, wie weit gezählt werden muss, um eine bestimmte Zeitspanne vergehen zu lassen. Dies ist jedoch nur für sehr kurze Zeitspannen (*wenige Taktzyklen*) wirklich sinnvoll, da der Controller während des Zählens blockiert ist. Auch die Funktion `_delay_ms()` am Ende der [IPO Pattern Variante](#) wird so realisiert und deshalb nur in seltenen Fällen verwendet.

Da die Problemstellung weit verbreitet ist, verfügen praktisch alle Mikrocontroller über integrierte "Timer" Module, die diese Aufgabe parallel zum normalen Programmablauf übernehmen können und die „signalisieren“, wenn eine voreingestellte Zeitspanne abgelaufen ist.

Die Signalisierung kann aktiv oder passiv erfolgen. Beim passiven Modus wird lediglich ein "Flag" gesetzt, welches im Hauptprogramm zyklisch abgefragt werden muss. Die dann auszuführenden Aktionen werden somit auch zeitverzögert abgearbeitet. Eben dann, wenn das Hauptprogramm wieder an der Abfrage „vorbeikommt“.

Dieses "Flag" Prinzip gleicht Briefkästen welche mit kleinen roten Fähnchen (*Flags*) versehen sind. Wirft jemand etwas in den Briefkasten, dann klappt er das Fähnchen hoch.

Der Besitzer des Briefkasten sieht irgendwann das hochgeklappte Fähnchen. Er leert den Briefkasten und klappt das Fähnchen wieder nach unten.

Beim aktiven Modus wird dagegen das Hauptprogramm unterbrochen und die erforderlichen Anweisungen sofort ausgeführt. Das Hauptprogramm muss den Timer so nicht ständig beobachten oder abfragen, sondern bekommt durch den "Interrupt" signalisiert, dass die voreingestellte Zeit abgelaufen ist. ()

Dieses Prinzip gleicht dem bei der Verwendung eines Weckers. Es wäre unsinnig die ganze Nacht über immer wieder zu überprüfen zu müssen, ob es schon Zeit zum Aufstehen ist. Geradezu absurd wäre es natürlich bis zum Aufstehen zu zählen.

Welches Prinzip im Einzelfall verwendet wird hängt immer von der Anwendung ab.

Es ist meist völlig unwichtig und wäre auch extrem lästig "sofort" auf den Einwurf eines Briefes oder gar einer Zeitung reagieren zu müssen. (*Zeitungen kommen oft bevor der Wecker klingelt*)

4.1 Blink Variante 1 – Zählen (`delay`)

Zunächst muss natürlich wieder ein neues Projekt angelegt werden.

Für ein neues „C“ Projekt kann man folgendermaßen vorgehen.

- Neues Projekt erzeugen (*copy... im Projektfenster ???*)
- Die Source Dateien nach dem Kopieren umbenennen (*ZaehlBlinker_main.c ...*)
- Die Köpfe der Quell-Dateien mit den Beschreibungen sinnvoll editieren.

Angenommen die **LED_3** soll im Sekundenrhythmus blinken. Dann ergibt sich jeweils eine Zeit von einer halben Sekunde für die Zustände „Aus“ und „Ein“. Jetzt muss überlegt werden, wie weit bei einer gegebenen Geschwindigkeit gezählt werden muss, um diese Zeit ungefähr zu erreichen.

Die Geschwindigkeit mit der gezählt wird ist von der [Oszillatorkonfiguration](#) abhängig. Würde der in den Templates voreingestellte Befehlstakt von 1MHz beibehalten, müsste man 500.000 Befehle abarbeiten um $\frac{1}{2}$ Sekunde verstreichen zu lassen. Wie weiter unten noch ersichtlich wird ist die Verarbeitung von großen Zahlen für einen 8-Bit Controller etwas mühsam, deswegen sollte der Oszillatorkonfiguration auf eine niedrigere Taktrate eingestellt werden, damit die Zählwerte nicht so groß werden. (*auch mit Blick auf den Energieverbrauch wäre es sinnvoll die niedrigst mögliche Taktrate einzustellen. Leider wird dadurch aber auch die Debug-Kommunikation sehr langsam*)

Bei der C-Programmierung von Mikrocontrollern sollte man generell darauf achten, dass für Variablen immer der kleinstmögliche Daten-Typ verwendet wird. Unnötig „große“ Datentypen verringern die Verarbeitungsgeschwindigkeit enorm und verschwenden Speicherplatz. Die zur Verfügung stehenden Datentypen erfährt man im **Userguide** des Compilers, der wiederum im „**docs**“ Ordner des Installationsverzeichnisses zu finden ist.

Die Suche nach „**Data Types**“ führt unter anderem zum Eintrag „**Integer Data Types**“ mit zwei zugehörigen Tabellen:

5.4.2 Integer Data Types

The MPLAB XC8 compiler supports integer data types with 1, 2, 3 and 4 byte sizes as well as a single bit type. **Table 5-3** shows the data types and their corresponding size and arithmetic type. The default type for each type is underlined.

TABLE 5-3: INTEGER DATA TYPES

Type	Size (bits)	Arithmetic Type
bit	1	Unsigned integer
signed char	8	Signed integer
unsigned char	8	Unsigned integer
signed short	16	Signed integer
unsigned short	16	Unsigned integer
signed int	16	Signed integer
unsigned int	16	Unsigned integer
signed short long	24	Signed integer
unsigned short long	24	Unsigned integer
signed long	32	Signed integer
unsigned long	32	Unsigned integer
signed long long	32	Signed integer
unsigned long long	32	Unsigned integer

TABLE 5-4: RANGES OF INTEGER TYPE VALUES

Symbol	Meaning	Value
CHAR_BIT	bits per char	8
CHAR_MAX	max. value of a char	127
CHAR_MIN	min. value of a char	-128
SCHAR_MAX	max. value of a signed char	127
SCHAR_MIN	min. value of a signed char	-128
UCHAR_MAX	max. value of an unsigned char	255
SHRT_MAX	max. value of a short	32767
SHRT_MIN	min. value of a short	-32768
USHRT_MAX	max. value of an unsigned short	65535
INT_MAX	max. value of an int	32767
INT_MIN	min. value of a int	-32768
UINT_MAX	max. value of an unsigned int	65535
SHRTLONG_MAX	max. value of a short long	8388607
SHRTLONG_MIN	min. value of a short long	-8388608
USHRTLONG_MAX	max. value of an unsigned short long	16777215
LONG_MAX	max. value of a long	2147483647
LONG_MIN	min. value of a long	-2147483648

Der kleinste Datentyp, den der XC8 Compiler unterstützt, wäre (1) **bit**. Da dieser Typ allerdings nur die Werte **0** oder **1** annehmen kann, kommt er für eine Zählschleife wohl eher nicht in Frage.

Auch die nächst größeren **char** Varianten (8 Bits) haben noch einen sehr kleinen Wertebereich von max. **0..255**. Das ist von den oben berechneten 500.000 Befehlszyklen pro $\frac{1}{2}$ s noch sehr weit weg.

Mit einer 16 Bit Variable vom Typ **unsigned short** kann man immerhin schon bis **65.535** zählen.

Wenn man vorerst annimmt, das Zählen könnte in einem Befehl erledigt werden, muss man eine dafür passende Frequenz bestimmen. Maximal 65.535 Befehle pro $\frac{1}{2}$ s entsprechen 131070 Befehlen/s.

Aus dem Datenblatt des PICs erfährt man die am besten passende (*nächst kleinere*) Einstellung für das **OSCCON** Register. ($\rightarrow 500\text{kHz}; \sim 125.000 \text{ Befehle/s} \rightarrow 62.500 / \frac{1}{2}\text{s}$)

Die anschließend durchzuführende Berechnung ist, bedingt durch die 8-Bit Architektur der PIC18, besonders bei der Programmierung in „C“ nicht so einfach.

Beginnen kann man empirisch mit der Annahme, dass ein Zählschritt einem Befehlsschritt entspräche. Dann schaut man, was dabei heraus kommt.

Mit der Anweisung `for(time = 62500; time > 0; time--);` kann man versuchen die entsprechende Zeit ablaufen zu lassen. Das gesamte Programm, mit Initialisierung, und einer zusätzlichen Option eine LED über eine Taste einzuschalten, könnte wie folgt aussehen:

```
#include "uCQ_2013.h"

unsigned short time;

void __init(void)
{
    OSCCONbits.IRCF = IRCF_500KHZ; OSCTUNEbits.PLLN = 0; // 500kHz clock
    ENC_BTN_TRI = INPUT_PIN; ENC_BTN_ANS = DIGITAL_PIN;
    LED_2_TRI = OUTPUT_PIN; // LED button
    LED_3_TRI = OUTPUT_PIN; // LED blink
    mSET_LED_3_ON();
}
...
```

```

void main(void)
{
    __init();

    while(1{
        for (time = 0; time < 62500; time++) {} // wait 1/2 sec. ???
        mTOG_LED_3(); // then toggle LED

        if(mGET_ENC_BTN()) {mSET_LED_2_ON();} // easy test if program is running
        else {mSET_LED_2_OFF();}
    }
}

```

Wenn das Ganze kompiliert, in den PIC geladen und gestartet wird, stellt man leider fest, dass die LED um den Faktor 10 langsamer blinkt als beabsichtigt :-(

Ein Blick in das [Disassembly Listing File](#) gibt einen ersten Hinweis darauf, warum dem so ist:

	;uCQ_main.c: 29: for (time = 0; time < 62500; time++) {}		
007FE2 0E00	movlw	0	// WREG mit 0 laden (1)
007FE4 6E02	movwf	_time+1,c	// und in time_HB (high byte) speichern (1)
007FE6 0E00	movlw	0	// WREG mit 0 laden (1)
007FE8 6E01	movwf	_time,c	// und in time_LB (low byte) speichern (1)
007FEA 1665:			
->007FEA 0E24	movlw	36	// Am Beginn der Schleife 36 (62500_LB) (1)
# 007FEC 5C01	subwf	_time,w,c	// ins WREG laden und von time_LB abziehen (1)
# 007FEE 0EF4	movlw	244	// 244 (62500_HB) ins WREG laden (1)
# 007FF0 5802	subwfb	_time+1,w,c	// und inkl. Übertrag von time_HB abziehen (1)
# 007FF2 B0D8	btfsc	status,0,c	// Ist Ergebnis = 0 ? (1/2)
# 007FF4 D003	bra	u30	// Ja -> Schleife verlassen (2/0)
# 007FF6 4A01	infsnz	_time,f,c	// Nein -> time_LB hochzählen und wenn 0 (1/2)
# 007FF8 2A02	incf	_time+1,f,c	// time_HB auch hochzählen (1/0)
<-007FFA D7F7	bra	1665	// Zur Abfrage am Anfang der Schleife (2)
007FFC u30:			
	;uCQ_main.c: 30: LATBbits.LATB4 ^= 1; // mTOG_LED_3()		
007FFC 788A	btg	3978,4,c	// 3978 ist Adresse von LATB (1)
007FFE D7EC	bra	118	// while(1){ (2)
008000			__end_of_main:

Der vom Compiler erzeugte Maschinen-Code ist in hohem Maße vom verwendeten Compiler und auch von dessen Optimierungseinstellungen abhängig. Der abgebildete Code entspricht schon sehr gut der best möglichen Umsetzung, was bei den ersten freien Versionen des XC8 Compilers noch nicht der Fall war. Deshalb bitte immer eine aktuelle Version des Compilers verwenden!

(alleine der Code zum toggeln der LED_3 umfasste 13 Befehle)

Die Schleife enthält 9 Befehle von denen *bra am Ende* immer 2 Cycles und die Kombination aus *infsnz und incf* auch immer 2 Cycles benötigt. Der Sprung von *btfsc über bra* wird außer im letzten Durchlauf immer durchgeführt, d.h. nochmals 2 Cycles. Insgesamt 10 Cycles für einen Durchlauf der Zählschleife. Bei 62.500 Durchläufen a 10 Cycles = 625.000 kann man die zusätzlichen 4 für die Initialisierung der Zählvariable *time*, 1 zum toggeln, weitere 2 für die *while(1)* Schleife und 6 für die Tasterabfrage vernachlässigen. ($13/625.000 = 0,00208\%$)

Das Ganze könnte man jetzt korrigieren, indem man den Endwert der Schleife anpasst (6250). Leider hat das Programm aber noch einen weiteren Nachteil, der es in einer realen Umgebung unbrauchbar macht. Beobachtet man LED_2 und LED_3 beim Betätigen des Tasters, dann stellt man fest, dass LED_2 erst dann reagiert, wenn LED_3 umschaltet.

Die Taste wird während der Warteschleife nicht abgefragt !!!

Nochmal zur Erinnerung: Auch die *__delay_xs()* Funktionen arbeiten nach dem Zählprinzip!

4.2 Blink 1 in Assembler Sprache

Neues Projekt anlegen mit den **main- und config-Templates** oder über „**Copy ...**“ im Projektfenster (Pop-Up Menü)

Bei der Assembler Programmierung wird das Problem auf eine etwas andere Weise angegangen. Von Anfang an ist klar, dass der Controller eigentlich nur 8-Bit Werte verarbeiten kann.

Für eine „minimal“ Schleife (*mit bis zu 256 Durchläufen*) braucht man mindestens 2 Befehle, die insgesamt meistens 3 Cycles an Rechenzeit verbrauchen:

```
loop    decfsz variable ; 1 (2) Cycles
        goto    loop    ; 2 Cycles
```

Für die während der C-Programmierung berechneten 62.500 Cycles muss die Schleife also 62.500 Cycles / (256*3 Cycles) = ~81,4 mal durchlaufen werden. Das Programm kann wie folgt aussehen:

```
UDATA_ACS
time_LB RES 1
time_HB RES 1

RES_VECT code 0x0000      ; processor reset vector
          goto _Init           ; go to beginning of program (initialization)

MAIN_PROG code
_Init
    movlw 0x20
    movwf OSCCON
    bcf LED_2_TRI
    bcf LED_3_TRI
    bsf ENC_BTN_TRI

_Main
    btfss nENC_BTN
    bcf nLED_2
    btfsc nENC_BTN
    bsf nLED_2
    clrf time_LB
    movlw .81
    movwf time_HB

Loop
    decfsz time_LB      ; Innere Schleife
    goto Loop
    decfsz time_HB      ; Äussere Schleife
    goto Loop

    btg nLED_3           ; LED umschalten (blink)
    goto Main

END
```

Blinkt auf Anhieb mit der richtigen Frequenz !!!

(die Taste wird aber auch hier nicht abgefragt während der PIC mit Zählen beschäftigt ist)

4.3 Vertiefung: Zufallsgenerator / Reaktionstest (Prof. Groß ...)

Wer sich etwas tiefer in Assembler einarbeiten will, kann das etwas eintönige Blinken aus der vorherigen Aufgabe in ein kleines Spiel zum Testen der Reaktionsschnelligkeit erweitern, bzw. abändern.

Für den Reaktionstest kann ein Pseudo-Zufallsgenerator verwendet werden, der (*fast*) zufällige Bit-Kombinationen erzeugt welche mit den LEDs visualisiert werden können.

Der Spieler muss dann immer bei einer bestimmten LED-Anzeige eine Taste drücken. Schafft er das nicht, dann ist das Spiel beendet (*verloren*).

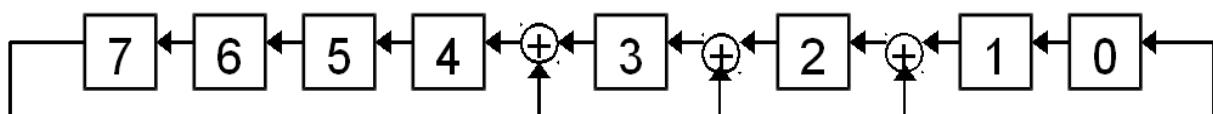
4.3.1 LED-Anzeige und Taste

Für den Taster soll der Encoder Button (ENC_BTN) und für die Anzeige sollen LEDs 1..4 verwendet werden. Die Initialisierung ändert sich also zu:

```
_Init          ; initialization code
    movlw 0x00      ; set internal oscillator
    movwf OSCCON    ;   frequency 31250 Hz
    bcf LED_1_TRI  ; output
    bcf LED_2_TRI  ; output
    bcf LED_3_TRI  ; output
    bcf LED_4_TRI  ; output
    bsf ENC_BTN_TRI; input
_Main         ; main code
    btfss nENC_BTN ; Taster nicht betätigt -> ueberspringe
```

Die Tastenabfrage während des im Programmablaufs sollte vorerst gelöscht und später an einer anderen Stelle wieder eingefügt werden ! (*die Zeitschleifen bleiben*)

4.3.2 Der „Pseudo“-Zufallsgenerator



Für den Zufallsgenerator kann man ein „*Linear rückgekoppeltes Schieberegister*“ nach Fibonacci oder Galois verwenden. (*Abbildung Galois*). Dieses besteht aus einem 8-Bit Muster welches geschoben oder besser „rotiert“ wird.

Zusätzlich zur Rotation ist natürlich noch eine Modifikation des Musters beim Rotieren erforderlich. Die Modifikation wird durch die XOR-Verknüpfung einzelner Bits erzeugt.

Die XOR Operation bewirkt im dargestellten Beispiel eine Invertierung der Bits beim Rotieren von 1→2, 2→3 und 3→4 in Abhängigkeit des Zustandes von Bit 7. Ist Bit 7 eine „1“, wird invertiert, sonst nicht.

Im Mikrocontroller Programm kann natürlich nicht gleichzeitig rotiert und die XOR Operation ausgeführt werden. Diese Operationen müssen sequentiell durchgeführt werden.

Invertiert man vor dem Rotieren, dann sind davon die Bits 1,2 und 3 in Abhängigkeit von Bit 7 betroffen. Nach dem Rotieren wären es die Bits 2, 3 und 4 in Abhängigkeit von Bit 0.

4.3.2.1 Rotieren

Der zum Rotieren eines Registers benötigte Befehl findet sich im Datenblatt im Kapitel Instruction Set. Bei Nutzung der Suchfunktion eines PDF-Viewers reicht womöglich auch die Eingabe von „rotate“ um auf die richtige Spur zu gelangen ;-)

In der obigen Abbildung kann man dann noch erkennen, dass ein Befehl zum rotieren nach links benötigt wird. Bei zusätzlichem Lesen der Beschreibungen und Betrachten der kleinen Abbildungen sollte zur Erkenntnis führen, dass **RLNCF** der zum vorliegenden Problem am besten passende Befehl ist.

Für das zu rotierende Register kann direkt das Register **LATB** gewählt werden, da die LEDs an diesem angeschlossen sind und in diesem Projekt keine weiteren Ausgangssignale am selben Port verwendet werden. Anstelle der Blinkfunktion wird der Befehl **RLNCF** auf das Register LATB angewendet.

Damit die Rotation sichtbar wird, und später der Zufallsgenerator auch funktionieren kann muss LATB noch mit einem Wert $\neq 0$ vorgeladen werden.

Die Änderungen können / sollten sofort getestet werden !

```

    movlw  B'01000100'      ; Startwert != 0
    movwf  LATB,A
_Main
...
    btg    nLED_3          ; LED umschalten (blink)
    rlncf  LATB,F,A       ; Rotation
    goto   _Main

```

RLNCF Rotate Left f (No Carry)									
Syntax:	RLNCF f {,d {,a}}								
Operands:	0 ≤ f ≤ 255 d ∈ [0,1] a ∈ [0,1]								
Operation:	(f<n>) → dest<n + 1> (f<0>) → dest<0>								
Status Affected:	N, Z								
Encoding:	0100 01da ffff ffff								
Description:	The contents of register 'f' are rotated one bit to the left. If 'd' is '0', the result is placed in W. If 'd' is '1', the result is stored back in register 'f' (default). If 'a' is '0', the Access Bank is selected. If 'a' is '1', the BSR is used to select the GPR bank. If 'a' is '0' and the extended instruction set is enabled, this instruction operates in Indexed Literal Offset Addressing mode whenever f ≤ 95 (5Fh). See Section 25.2.3 "Byte-Oriented and Bit-Oriented Instructions in Indexed Literal Offset Mode" for details.								
Words:	1								
Cycles:	1								
Q Cycle Activity:	<table border="1"> <tr> <th>Q1</th><th>Q2</th><th>Q3</th><th>Q4</th></tr> <tr> <td>Decode</td><td>Read register 'f'</td><td>Process Data</td><td>Write to destination</td></tr> </table>	Q1	Q2	Q3	Q4	Decode	Read register 'f'	Process Data	Write to destination
Q1	Q2	Q3	Q4						
Decode	Read register 'f'	Process Data	Write to destination						
Example:	XORWF REG, 1, 0 Before Instruction REG = 1010 1011 After Instruction REG = 0101 0111								

4.3.2.2 XOR Verarbeitung

Im Befehlssatz der PIC18 Controller sind zwei Befehle enthalten die zur XOR Verarbeitung geeignet sind. Bei Verwendung von **XORWF** kann man sich einige (*MOV.*) Befehle sparen.

„W“ muss somit mit einer Maske für die XOR-Verknüpfung geladen werden, welche dann direkt auf LATB angewendet werden kann. Die erforderliche Bitkombination der Maske ergibt sich aus der Position der zu invertierenden Bits.

Für die Abfrage ob invertiert werden muss, kann man den bekannten Befehl **BTFS** verwenden.

(Überspringe das Invertieren wenn entsprechendes Bit = 0 ...)

Beachten muss man natürlich noch, dass das Ergebnis der Verknüpfung wieder im Register LATB (File) stehen soll, welches im Access Speicherbereich liegt.

Das kann dann wie folgt aussehen:

```

    rlncf  LATB,F,A      ; Rotation
    movlw  B'00011100'    ; XOR Maske laden
    btfsc  LATB,0,A       ; Abfrage Bit 0
    xorwf  LATB,F,A       ; Invertierung
    goto   _Main

```

XORWF Exclusive OR W with f									
Syntax:	XORWF f {,d {,a}}								
Operands:	0 ≤ f ≤ 255 d ∈ [0,1] a ∈ [0,1]								
Operation:	(W) .XOR. (f) → dest								
Status Affected:	N, Z								
Encoding:	0001 10da ffff ffff								
Description:	Exclusive OR the contents of W with register 'f'. If 'd' is '0', the result is stored in W. If 'd' is '1', the result is stored back in the register 'f' (default). If 'a' is '0', the Access Bank is selected. If 'a' is '1', the BSR is used to select the GPR bank. If 'a' is '0' and the extended instruction set is enabled, this instruction operates in Indexed Literal Offset Addressing mode whenever f ≤ 95 (5Fh). See Section 25.2.3 "Byte-Oriented and Bit-Oriented Instructions in Indexed Literal Offset Mode" for details.								
Words:	1								
Cycles:	1								
Q Cycle Activity:	<table border="1"> <tr> <th>Q1</th><th>Q2</th><th>Q3</th><th>Q4</th></tr> <tr> <td>Decode</td><td>Read register 'f'</td><td>Process Data</td><td>Write to destination</td></tr> </table>	Q1	Q2	Q3	Q4	Decode	Read register 'f'	Process Data	Write to destination
Q1	Q2	Q3	Q4						
Decode	Read register 'f'	Process Data	Write to destination						
Example:	XORWF REG, 1, 0 Before Instruction REG = AFh W = B5h After Instruction REG = 1Ah W = B5h								

4.3.3 Zeitsteuerung und Tastenabfrage

Die Tastenabfrage muss in die Zeitschleifen integriert werden, damit der Tastendruck auch während der Wartezeit erkannt werden kann. Zusätzlich muss das Programm abspeichern, dass eine Taste gedrückt wurde, da die Kontrolle der Richtigkeit erst nach Ablauf der Wartezeit erfolgt. Vor Beginn einer neuen Warteschleife (*Anzeige einer neuen Kombination*) muss dann ein möglicher Tastendruck aber auch wieder gelöscht werden !

Die Speicherung des Tastendrucks kann in einem Bit einer neu angelegten Variable erfolgen.

Der Name der Variablen könnte „flags“ lauten und für die Taste könnte das Bit 0 verwendet werden.

```
Wait    decfsz time_LB      ; Innere Schleife
        goto   Wait
        btfss  nENC_BTN   ; Taster nicht betätigt -> speichern ueberspringen
        bsf    flags,0,A   ; speichern
        decfsz time_HB      ; Äussere Schleife
        goto   Wait
```

(Nützlich wäre hier ein **#define BUTTON flags,0,A** ;-)

4.3.4 Kontrolle der Reaktion

Zur Auswertung der Reaktion sollte man sich zunächst klar werden welche Fälle auftreten können.

- Der Spieler kann die Taste gedrückt haben oder nicht.
- Unabhängig davon kann eine gültige LED Anzeige vorliegen oder nicht.

Daraus ergeben sich vier Möglichkeiten:

- Taste gedrückt, Muster korrekt → Spiel geht weiter
- Taste gedrückt, Muster falsch → Spielende
- Taste nicht gedrückt, Muster korrekt → Spielende
- Taste nicht gedrückt, Muster falsch → Spiel geht weiter

4.3.4.1 Maskierung

Bei den erforderlichen Kontrollen besteht das sehr häufig auftretende Problem, dass nur bestimmte Bits (*die LEDs*) überprüft werden dürfen. Dies kann man wieder durch eine Maskierung lösen. Für diese Maskierung gibt es zwei Möglichkeiten:

- nicht relevante Bits werden zu Einsen gemacht → OR Befehl mit „1“
- nicht relevante Bits werden zu Nullen gemacht → AND Befehl mit „0“

```
LATB(xxxxxxxx) & Maske(00111100) => 00xxxx00
```

```
LATB(xxxxxxxx) | Maske(11000011) => 11xxxx11
```

Somit ist der Zustand der nicht relevanten Bits definiert und kann gezielt in die Mustererkennung einbezogen werden.

Wenn die Rotation direkt an LATB erfolgt, dann kann die Maskierung nicht direkt an LATB durchgeführt werden, das sonst der Zufallsgenerator beeinflusst würde. Man muss gegebenenfalls einen Zwischenspeicher anlegen

4.3.4.2 Mustererkennung

Zur vorhandenen Tastenerkennung kann man mit Hilfe der Maskierung jetzt die Mustererkennung programmieren. Die Mustererkennung kann gleich nach der Rotation/Invertierung mittels unterschiedlicher Methoden erfolgen. Man könnte die LEDs beispielsweise einzeln mit den Bit-Test Befehlen abfragen. Bei nur 4 Bits wäre das gerade noch effektiv aber unflexibel in Bezug auf eine Erweiterung mit verschiedenen Kombinationen. Zudem sollen hier natürlich neue Befehle eingeführt werden ;-)

Die erste Wahl würde vermutlich auf einen Vergleichsbefehl (*Compare f with WREG ...*) fallen.

TABLE 25-2: PIC18(L)F2X/4XK22 INSTRUCTION SET

Mnemonic, Operands	Description	Cycles	16-Bit Instruction Word		Status Affected	Notes
			MSb	LSb		
BYTE-ORIENTED OPERATIONS						
ADDWF	f, d, a	Add WREG and f	1	0010 01da ffff ffff	C, DC, Z, OV, N	1, 2
ADDWFC	f, d, a	Add WREG and CARRY bit to f	1	0010 00da ffff ffff	C, DC, Z, OV, N	1, 2
ANDWF	f, d, a	AND WREG with f	1	0001 01da ffff ffff	Z, N	1, 2
CLRF	f, a	Clear f	1	0110 101a ffff ffff	Z	2
COMF	f, d, a	Complement f	1	0001 11da ffff ffff	Z, N	1, 2
CPFSEQ	f, a	Compare f with WREG, skip =	1 (2 or 3)	0110 001a ffff ffff	None	4
CPFSGT	f, a	Compare f with WREG, skip >	1 (2 or 3)	0110 010a ffff ffff	None	4
CPFSLT	f, a	Compare f with WREG, skip <	1 (2 or 3)	0110 000a ffff ffff	None	1, 2
DECf	f, d, a	Decrement f	1	0000 01da ffff ffff	C, DC, Z, OV, N	1, 2, 3, 4

Weitere Möglichkeiten sind die Addition oder Subtraktion eines Wertes zum maskierten Wert und die Überprüfung des Ergebnisses auf Null.

Die Subtraktion ist insofern einfacher, da genau die Kontrollkombination subtrahiert werden muss. Bei der Addition müsste das 2er-Komplement addiert werden.

Ist die gültige Kombination nicht veränderbar bieten Subtraktions- und Additions-Verfahren den weiteren Vorteil, dass Operationen mit Konstanten zur Verfügung stehen.

TABLE 25-2: PIC18(L)F2X/4XK22 INSTRUCTION SET (CONTINUED)

Mnemonic, Operands	Description	Cycles	16-Bit Instruction Word		Status Affected	Notes
			MSb	LSb		
LITERAL OPERATIONS						
ADDLW	k	Add literal and WREG	1	0000 1111 kkkk kkkk	C, DC, Z, OV, N	
ANDLW	k	AND literal with WREG	1	0000 1011 kkkk kkkk	Z, N	
IORLW	k	Inclusive OR literal with WREG	1	0000 1001 kkkk kkkk	Z, N	
LSFR	f, k	Move literal (12-bit) 2nd word to FSR(f) 1st word	2	1110 1110 00ff kkkk	None	
MOVLB	k	Move literal to BSR<3:0>	1	0000 0001 0000 kkkk	None	
MOVLP	k	Move literal to WREG	1	0000 1110 kkkk kkkk	None	
MULLW	k	Multiply literal with WREG	1	0000 1101 kkkk kkkk	None	
RETLW	k	Return with literal in WREG	2	0000 1100 kkkk kkkk	None	
SUBLW	k	Subtract WREG from literal	1	0000 1000 kkkk kkkk	C, DC, Z, OV, N	
XORLW	k	Exclusive OR literal with WREG	1	0000 1010 kkkk kkkk	Z, N	

Eine Mustererkennung über Subtraktion könnte wie folgt aussehen. (#define REQUEST flags,1,A)

```

...
xorwf LATB,F,A      ; Invertierung
movf LATB,W
andlw B'00111100'    ; Maskierung LEDs
sublw B'00101000'    ; gueltige Bitkombination (AUS-AN-AUS-AN)
bnz   _Main
bsf   REQUEST        ; gueltige Kombination liegt an
bra   _Main

```

4.3.4.3 Kontrolle

Als letztes muss jetzt noch die Kontrolle der vier möglichen Fälle implementiert werden. Das kann durch Anwendung von Bit-Test Befehlen auf die vorbereiteten „Flags“ erfolgen.

Die Anforderung „Spielende“ kann man durch eine Endlosschleife erzeugen. Im folgenden Beispiel ist diese Endlosschleife durch den Befehl „bra \$“ gegeben. Das „\$“ Zeichen steht hier für die aktuelle Adresse. Der Befehl ist also äquivalent zu „Spielende bra Spielende“

```

btfs  BUTTON      ; Taster ?
bra  no_btn
btfs  REQUEST     ; Taster & gueltige Kombi ?
bra  $
bra  new_LED      ; Taster falsch gedrueckt
no_btn btfsc REQUEST ; kein Taster & keine gueltige Kombi ?
bra  $
new_LED clrflags ; alten Tastendruck und Request loeschen
rlncf LATB,F,A   ; Rotation

```

Die angegebene Kontrollabfrage muss zwischen der Zeitschleife und der Rotation stehen ...

4.4 Blink Variante 2 – Timer Flags

Das Zählen im Hauptprogramm der Variante 1 hat den Nachteil, dass während dessen andere Vorgänge blockiert sind. Dies kann man vermeiden, indem man einfach eines der Peripheriemodule zählt. Peripheriemodule arbeiten parallel und unabhängig vom Hauptprogramm und müssen nur von diesem eingestellt bzw. kontrolliert werden.

Die Controller der PIC18 Familie haben immer mehrere Timer Module. Ein Blick in das Datenblatt des PIC18F24K22 zeigt sieben Timer mit den Nummern 0-6. Die Timer haben verschiedene Eigenschaften, welche für Anfänger natürlich nicht sofort zu durchschauen sind. Die Timer 1,3,5 bzw. 2,4,6, können zudem im Verbund mit anderen Peripheriemodulen genutzt werden um komplexere Funktionen zu realisieren. Diese Funktionen sollen vorerst nicht blockiert werden. Timer 0 hat dagegen den größten Zählbereich. → **Verwendung von Timer 0 in diesem Abschnitt.**

11.0 TIMER0 MODULE

The Timer0 module incorporates the following features:

- Software selectable operation as a timer or counter in both 8-bit or 16-bit modes
- Readable and writable registers
- Dedicated 8-bit, software programmable prescaler
- Selectable clock source (internal or external)
- Edge select for external clock
- Interrupt-on-overflow

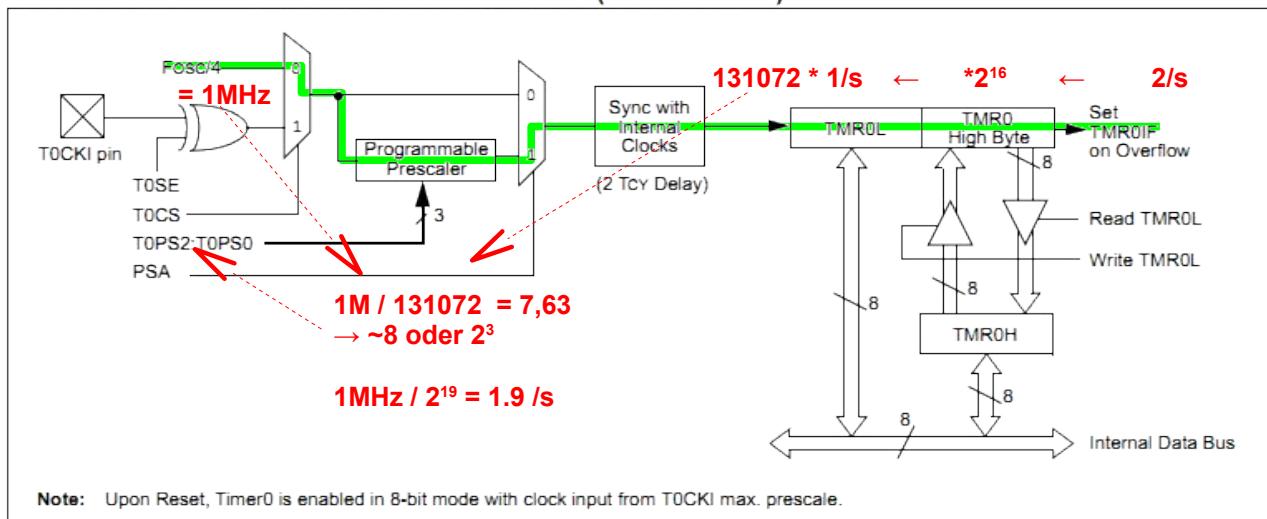
Die Unterscheidung zwischen Counter- und Timer- Modus besteht eigentlich nur darin, dass ein Timer mit einer festen Frequenz (*intern oder extern*) zählt. Der Counter zählt „Ereignisse“ an Pins, die in beliebigen Zeitabständen auftreten können.

Der Umfang des Zählbereichs ergibt sich aus dem Registerumfang (8-bit / 16-bit) und dem verfügbaren Prescaler ($2^{0..8}$).

Maximal sind also $2^{16} * 2^8 = 2^{24} = 16.777.216$, Minimal $2^8 * 2^0 = 256$ Takte nötig, um einen Überlauf zu erzeugen.

Im vorliegenden Beispiel soll die Frequenz für $F_{osc} = 4MHz$ und die Blinkfrequenz wieder 1Hz betragen. Daraus kann der Mindestwert für den Prescaler berechnet bzw. ermittelt werden, ob die Blinkfrequenz mit nur einem Timerdurchlauf realisierbar ist. Das folgende Diagramm aus dem Datenblatt wurde hierfür mit den Berechnungsschritten für die gegebene Aufgabe ergänzt:

FIGURE 11-2: TIMER0 BLOCK DIAGRAM (16-BIT MODE)



Den Prescaler des Timers kann man mit der Zählfrequenz (linken Seite), geteilt durch die max. möglichen Zählschritte /s (rechte Seite, Zählbereich*Umschaltfrequenz) berechnen → 7,63.

Die nächstliegende Einstellmöglichkeit ist 8. Der Fehler zur gewünschten Blinkfrequenz beträgt damit $0.1 / 2 = 5\%$, was möglicherweise eine zu tolerierende Abweichung wäre. Will man den Wert genauer einhalten, kann man dies durch Ändern des Timerwertes während des Durchlaufes erzielen.

4.4.1 Die „ungenaue“ Methode

Wenn die 5% Abweichung kein Problem darstellt ist die Programmierung der Blinkfunktion sehr einfach. Anhand des Datenblattes wird der Wert bestimmt, welcher ins Timer0 Kontrollregister geschrieben werden muss. Im Hauptprogramm muss man dann nur noch das Timer0 Interruptflag beobachten das bei einem Überlauf des Timers gesetzt wird. Dann toggelt man die LED und löscht auch das Flag wieder.

REGISTER 11-1: T0CON: TIMER0 CONTROL REGISTER							
R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1
TMR0ON	T08BIT	T0CS	T0SE	PSA	T0PS2	T0PS1	T0PS0
bit 7							bit 0
bit 7	TMR0ON: Timer0 On/Off Control bit 1 = Enables Timer0 0 = Stops Timer0						
bit 6	T08BIT: Timer0 8-bit/16-bit Control bit 1 = Timer0 is configured as an 8-bit timer/counter 0 = Timer0 is configured as a 16-bit timer/counter						
bit 5	T0CS: Timer0 Clock Source Select bit 1 = Transition on T0CKI pin 0 = Internal instruction cycle clock (CLKO)						
bit 4	T0SE: Timer0 Source Edge Select bit 1 = Increment on high-to-low transition on T0CKI pin 0 = Increment on low-to-high transition on T0CKI pin						
bit 3	PSA: Timer0 Prescaler Assignment bit 1 = Timer0 prescaler is NOT assigned. Timer0 clock input bypasses prescaler. 0 = Timer0 prescaler is assigned. Timer0 clock input comes from prescaler output.						
bit 2-0	T0PS2:T0PS0: Timer0 Prescaler Select bits 111 = 1:256 prescale value 110 = 1:128 prescale value 101 = 1:64 prescale value 100 = 1:32 prescale value 011 = 1:16 prescale value 010 = 1:8 prescale value 001 = 1:4 prescale value 000 = 1:2 prescale value						
Legend: R = Readable bit W = Writable bit U = Unimplemented bit, read as '0' -n = Value at POR '1' = Bit is set '0' = Bit is cleared x = Bit is unknown							

T0CON = 0b-1-0-0-X-0-010;

(bit 4 hat bei internem Clock keine Bedeutung und kann sowohl 0 als auch 1 gesetzt werden)

Wieder neues Projekt anlegen, Dateien umbenennen und editieren ...

```

RES_VECT    code      0x0000 ; processor reset vector
goto        _Init       ; go to beginning of program (initialization)

MAIN_PROG   code
_Init
    movlw 0x50          ; let linker place main program
    movwf OSCCON         ; initialization code
    ; set internal oscillator
    bcf   LED_2_TRI     ; frequency 4 MHz
    bcf   LED_3_TRI     ; output for button LED
    bsf   BTN_ENC_TRI   ; output for blink LED
    ; input for button

    movlw 0x82          ; TMR0ON T08BIT T0CS T0SE PSA T0PS2 T0PS1 T0PS0
    movwf T0CON          ; Timer0 adjust and activate

_Main
    btfss nENC_BTN     ; main code
    bcf   nLED_2         ; button not pressed -> skip
    btfsc nENC_BTN     ; switch on
    bsf   nLED_2         ; button pressed -> skip
    btfsc nENC_BTN     ; switch off

    btfss INTCON,TMR0IF ; Timer0 overflow?
    bra  _Main           ; - no -> main loop
    btg   nLED_3         ; - yes -> toggle LED
    bcf   INTCON,TMR0IF ; and clear flag
    bra  _Main           ; Main
END

```

4.4.1.1 Übung: Blink Variante 2 in C

Die drei neuen Zeilen, die für das „C“ Projekt benötigt werden, sollten keine allzu große Herausforderung sein. Viel Spaß beim Ausprobieren ...

4.4.2 Die exakte Methode (*nur für Freaks*)

Will man mit dem Timer 0 eine Blinkfrequenz von exakt 1Hz einstellen, dann ist das nicht mehr ganz so einfach.

- Die erste Voraussetzung ist, dass der Timer länger braucht als gewünscht.
(*Den Timer-Durchlauf zu verlangsamen ist noch komplizierter als ihn zu beschleunigen*)
- Aus der voreingestellten Geschwindigkeit kann man errechnen, um wie viel kleiner der eigentliche Endwert des Timers sein sollte. ($1.000.000 / 2 / 8 = 62.500$)
- Die Differenz zum maximalen Wert ist die Zeit, um die der Timer dann nach jedem Überlauf vorgestellt werden muss. ($2^{16} - 62.500 = 65.536 - 62.500 = 3036$)
- Das Vorstellen erreicht man durch Addition einer noch zu bestimmenden Konstanten zum Timerwert. Bei der Berechnung der Konstanten müssen die Rechenschritte berücksichtigt werden, die für die Addition gebraucht werden. ($+ 3036 - ??? = + 0x0BDC - ???$)
- Im Timer0 Block Diagramm weiter oben ist zu sehen, dass die beiden 8-Bit Register des Timers in einer bestimmten Reihenfolge beschrieben werden müssen. Zuerst wird ein Buffer für TMR0H geladen welcher erst beim Schreiben von TMR0L in das eigentliche Timerregister übernommen wird. (*Würden die Register im direkten Modus geschrieben, könnte dabei ein Überlauf vom LOW in das HIGH Register verloren gehen*)
- Auch bei der Addition im LOW Register könnte es zu einem Überlauf kommen, der bei der Addition des HIGH Registers nicht mehr berücksichtigt werden kann, da diese ja vorher erfolgen muss. ($TMR0L + 0xDC - ???$)
- Bei der Programmierung in „C“ wird alles noch viel schlimmer ;-)

Fazit:

Es ist sehr schwierig auf diese Weise ein wirklich exaktes Timing zu realisieren.
(*In vielen Übungsbeispielen wird hierfür trotzdem einige Energie aufgewandt*)

Überlassen wir das lieber den Profis ;-) und benutzen statt dessen in einem späteren Kapitel ein erweitertes Modul, welches diese häufig auftretende Problemstellung automatisch löst.

Für alle die es dennoch versuchen wollen:

Das erzielte Genauigkeit kann mit Hilfe des Simulators [MPSIM](#) kontrolliert werden. Siehe hierzu „Zeitmessungen“ im Abschnitt [12.3 Simulator MPSIM](#)

4.5 Übung: Hello World IV, IPO mit Time-Slot

Zur Vertiefung der neuen Kenntnisse kann man die `_delay_ms()` Funktion aus dem [IPO Pattern](#) durch den Einsatz eines Timers und der Abfrage des Überlauf-Flags ersetzen.

Dies hat zusätzlich den Vorteil, dass die Zeit für einen Hauptschleifendurchlauf auch dann konstant bleibt, wenn zusätzlicher Code hinzukommt. Das funktioniert natürlich nur solange die Abarbeitung des Codes innerhalb der eingestellten möglich ist.

Die Hauptschleife soll jetzt allerdings ~8 mal pro Sekunde, oder einmal alle 131,072ms durchlaufen werden. Die Durchläufe kann man dann durch Toggeln einer LED sichtbar machen und erhält eine Blinkfrequenz von ~4Hz.

Man kann die konstante Zeitscheibe der Hauptschleife leicht überprüfen, indem man zusätzlich in der Hauptschleife `_delay_ms(???)` aufruft um die Verarbeitung von weiterem Programmcode zu simulieren. Die Blinkfrequenz darf sich erst ändern, wenn das eingegebene Delay den Schleifendurchlauf auf mehr als 131,072ms verlängert!

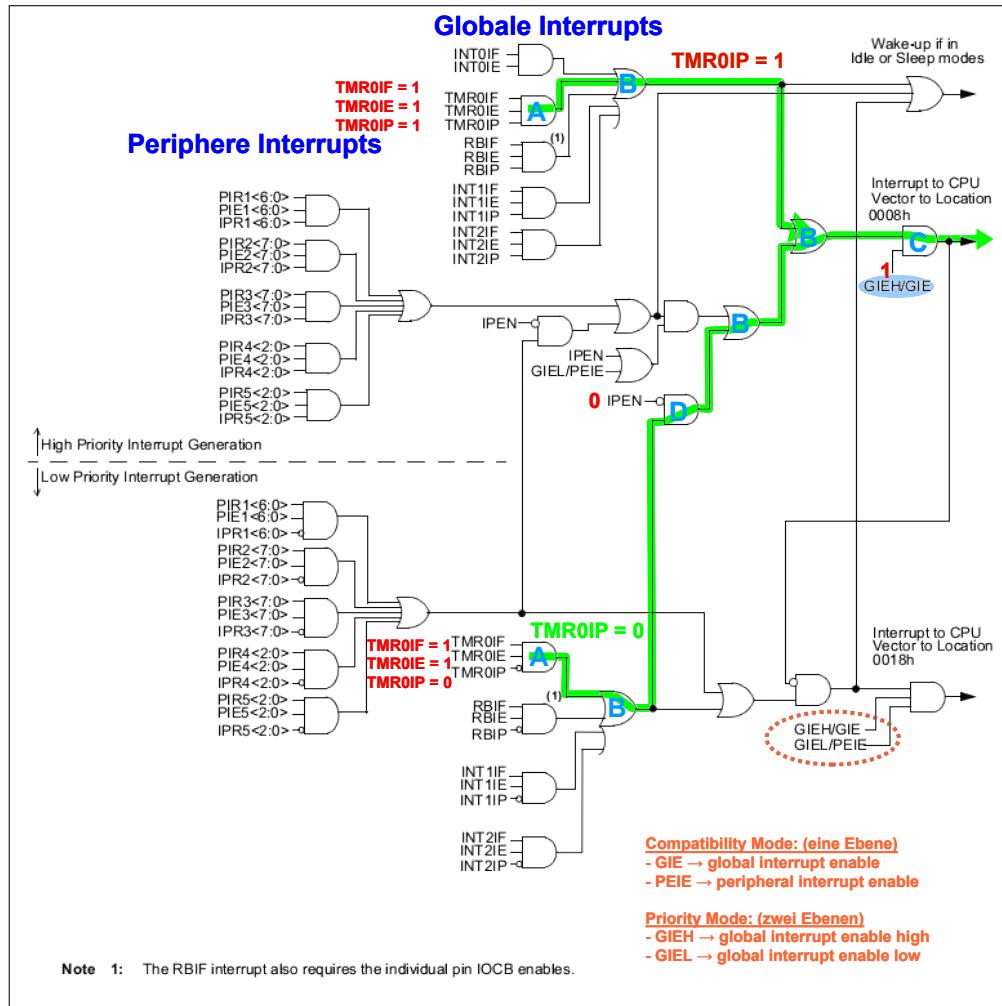
5 Interrupts (siehe auch [1.2.4 Interrupt-System](#))

5.1 Timer_0 Interrupt (Blink Variante 3)

Die Abfrage (Polling) des Interrupt-Flags von Timer0 aus Blink Variante 2 ersetzt man sinnvoller durch die Implementierung eines richtigen Interrupts. Der Interrupt unterbricht die Ausführung des Hauptprogramms sofort und führt möglichst kurze Befehlsfolgen aus. Danach wird wieder an die Stelle des Hauptprogramms zurück gesprungen, an der es unterbrochen wurde.

Damit ein Interrupt ausgelöst werden kann, müssen verschiedene Bedingungen erfüllt werden, die man am besten im Diagramm für die Interrupt-Logik erkennen kann.

FIGURE 9-1: PIC18 INTERRUPT LOGIC



Das Logik-Diagramm enthält Pfade des Interrupts durch verschiedene Logiksymbole und beginnt immer an einer logischen **UND**-Verknüpfung (**A**) von drei Bits. Bei der **UND**-Verknüpfung müssen alle Eingänge logisch „1“ sein, damit am Ausgang keine „0“ ist.

Wenn man **TMR0IF** (*Timer 0 Interrupt Flag*) als das „Signal“ betrachtet, dann sind die beiden anderen **TMR0IE** (*Timer 0 Interrupt Enable*) und **TMR0IP** (*Timer 0 Interrupt Priority*) die Schalter, die bestimmen, ob das Signal passieren kann und müssen dafür in logisch „1“ sein.

Die **UND**-Verknüpfung (**A**) im unteren Teil des Bildes hat an einem Eingang einen kleinen Kreis, welcher das Symbol für eine Invertierung darstellt. Damit wird das Bit **TMR0IP** invertiert und das **UND**-Gatter wird mit **TMR0IP = 0** durchlässig.

Der Pfad beginnt also abhängig von **TMR0IP** in der oberen oder unteren Bildhälfte.

Als nächstes muss das Interrupt-Signal einen (B) Block passieren. Alle diese (B) Symbole stellen **ODER**-Verknüpfungen mit unterschiedlicher Anzahl von Eingängen dar. Jede logische „1“ an einem beliebigen Eingang hat hier eine logische „1“ am Ausgang zur Folge und das Signal kann quasi ungehindert passieren.

Das Signal aus der oberen Bildhälfte mit **TMR0IP (Priority)** „1“ steht jetzt schon vor der letzten Schranke (C), dem **Global-Interrupt-Enable (GIE)** Bit. Ist dieses logisch „1“, wird ein Interrupt ausgelöst und der Program-Counter springt auf die Adresse **0x0008**, den **High-Priority-Interrupt Vector**. Dabei werden die momentane Programmadresse auf dem **Hardware-Stack** und die Register **WREG**, **STATUS** und **BSR** auf einem speziellen Fast-Register-Stack gesichert.

Das Signal aus der unteren Bildhälfte muss noch eine weitere **UND**-Verknüpfung (D) passieren. Da das Steuersignal **IPEN (Interrupt-Priority-Enable)** hier wieder invertiert wird (*Kreis*), kann das Signal im **Compatibility Mode** ($IPEN = 0$; *Priority-Mode nicht aktiviert*) auch von unten kommen. Im Umkehrschluss bedeutet dies, dass es im Compatibility-Mode keine Rolle spielt, ob die Priorität (**TMR0IP**) auf „High“ oder „Low“ eingestellt ist.

Besteht keine unbedingte Notwendigkeit den **Priority Mode** zu nutzen, dann sollte man diesen auch nicht aktivieren. Das System bleibt so schneller und einfacher zu handeln.

5.1.1 Timer_0 Interrupt Initialisierung

Alle erforderlichen Einstellungen müssen nun nur noch in die Initialisierung eingefügt werden. Dabei sollten die Steuerbits die weiter rechts im Diagramm auftauchen (*GIE*) erst als letzte Befehle der Initialisierung gesetzt werden um von vornherein jegliche „Unterbrechung“ während der Initialisierung zu unterbinden.

```
void __init()
{
    OSCCONbits.IRCF = IRCF_4MHZ; OSCTUNEbits.PLLN = 0;
    ENC_BTN_TRI = INPUT_PIN; ENC_BTN_ANS = DIGITAL_PIN;
    LED_2_TRI = OUTPUT_PIN; // LED button
    LED_3_TRI = OUTPUT_PIN; // LED blink
    mSET_LED_3_ON(); // TIMER_0 setup
    TOCON = 0x82; // TMROON T08BIT TOCS TOSE PSA TOPS2 TOPS1 TOPSO

    RCONbits.IPEN = 0; // disable IR priority mode (do not enable)
    INTCONbits.TMR0IE = 1; // enable timer_0 IR
    INTCONbits.GIE = 1; // global IR enable
}
```

5.1.2 Interrupt Programmcode

Für die Implementierung des Interrupt-Codes kann dem Projekt eine Kopie vom Template für die entsprechende Sprache hinzugefügt werden. In diesen Templates sind alle elementaren Strukturen vorhanden, die für die Behandlung von Interrupts erforderlich sind. Im folgenden werden die wichtigsten Punkte angesprochen. (*Ausführliche Informationen findet man im Datenblatt ;-)*

5.1.2.1 Interrupt Vektor Adressen

Wird ein Interrupt ausgelöst, dann wird der normale Programmablauf unterbrochen. Der Programmzeiger springt sofort auf eine fest vorgegebene Adresse, den Interrupt Vektor. Genau an dieser Adresse muss der auszuführende Programmcode stehen.

Die (*gegebenenfalls aus dem Datenblatt ermittelten*) Adressen der zwei möglichen Vektoren müssen dem Linker des Assemblers explizit mitgeteilt werden, damit dieser den Code auch genau da platziert.

Für den **Assembler Code** besteht die Anweisung aus „CODE“ + Adresse des Vektors:

```
HI_INT_VECTOR      CODE    0x0008
bra    HighInt ; go to high priority interrupt routine
LOW_INT_VECTOR     R      0x0018
bra    LowInt ; go to low priority interrupt routine
```

(Die Labels „HI_INT_VECTOR“ und „LOW_INT_VECTOR“ sind nicht unbedingt erforderlich)

Als eigentlicher Programmcode steht an der Vektor Adresse meist nur ein Sprungbefehl. Das ist vor allem darauf zurückzuführen, dass bei Verwendung beider Interruptvektoren (*High* und *Low*) für den Programmcode des ersten Vektors sonst nur der Speicherbereich bis zum zweiten Vektor zu Verfügung stände ($0x0008$ bis $0x0018 = 16\text{ Bytes} = \text{max. 8 Instruktionen}$).

Der Sprungbefehl löst dieses Problem für den Vektor an der niedrigeren Adresse ($0x0008$).

Wird der zweite Vektor nicht benutzt, besteht auch das Problem des eingeschränkten Programm-Speicherplatzes nicht und der Code kann auch direkt am Vektor platziert werden (*ohne Sprung*). Der Programmcode für den zweiten Vektor könnte eigentlich immer an der Vektoradresse stehen.

Beim **XC8-Compiler** ist die Angabe der Vektoradressen nicht nötig. Die Adressen sind in einer Datenbank des Compilers abgespeichert. Der bei einem Interrupt auszuführende Code muss beim C-Compiler in eine Funktion eingepackt werden, die allerdings unter keinen Umständen manuell aufgerufen werden sollte! Zur Kennzeichnung dieser Interrupt-Funktion sind auch hier spezielle Argumente nötig.

```
void __interrupt(high_priority) high_isr(void)
{
...
}
void __interrupt(low_priority) low_isr(void)
{
...
}
```

Mit dem Specifier „**__interrupt()**“ wird dem Compiler Signaliert, dass es sich um eine Interrupt Service Funktion handelt. Die Zuordnung zu entsprechenden Vektor erfolgt durch „**low_priority**“ und „**high_priority**“.

5.1.2.2 Context Saving / Restore

Wenn ein Programm durch einen Interrupt unterbrochen wird, dann muss die Adresse an der das Programm unterbrochen wurde und mehrere Datenregister zwischengespeichert werden, damit das Programm nach Abarbeitung des Interrupts ohne Datenverlust weitergeführt werden kann.

Die Programm-Adresse wird auf dem Hardwarestack des Programmspeichers abgelegt.

Die zu sichernden Datenregister sind das Arbeitsregister „**WREG**“, das Statusregister „**STATUS**“ und das Register in dem der aktuell ausgewählte Datenspeicherbereich „**BSR**“ (*Bank-Select-Reg.*) eingetragen ist. Die Controller der PIC18 Familie verfügen dafür über einen „Fast-Register-Stack“, der allerdings nur einen Satz der drei Register automatisch aufnehmen kann.

Die Wiederherstellung des Kontextes aus dem Fast-Register-Stack wird durch den Parameter „**FAST**“ initiiert, welcher dem Befehl zum Rücksprung (*retfie, siehe Datasheet*) beigefügt wird.

Der „Fast-Register-Stack“ enthält immer den Kontext, der bei Auftreten des letzten Interrupts automatisch gesichert wurde. Wird ein Interrupt niedriger Priorität von einem höher priorisierten unterbrochen, dann wird der vorher gesicherte Kontext (*der des Hauptprogramms*) überschrieben.

Bei **Assembler** Programmen mit Verwendung von Interrupt-Prioritäten muss deshalb nur der Kontext im niedriger priorisierten Interrupt manuell gesichert und wieder hergestellt werden. Der Rücksprung erfolgt dann ohne den Parameter **FAST**.

```

HighInt:
; *** high priority interrupt code goes here ***
    retfie FAST
LowInt:
    movff STATUS,STATUS_TEMP ;save STATUS register
    movff WREG,WREG_TEMP ;save working register
    movff BSR,BSR_TEMP ;save BSR register
; *** low priority interrupt code goes here ***
    movff BSR_TEMP,BSR ;restore BSR register
    movff WREG_TEMP,WREG ;restore working register
    movff STATUS_TEMP,STATUS ;restore STATUS register
    retfie

```

Bei Verwendung des **C Compilers** wird das Sichern und Wiederherstellen des Kontextes, dem Programmcode durch den Compiler selbst hinzugefügt (*User Guide* ...).

Zusätzlich zu den Registern die beim Assemblerprojekt gesichert werden, kommen beim C-Compiler noch einige (*bis zu 18?*) weitere Register hinzu, die er für verschiedene interne Zwecke benutzt. In der „free“ Version des XC8 Compilers wird hierbei evtl. nicht optimiert, d.h. überprüft, ob das Sichern erforderlich ist. Das Sichern und Wiederherstellen von bis zu 18? Registern benötigt natürlich auch einiges an Zeit, die vergeht, bis der eigentliche Interrupt Code ausgeführt wird.

Genauere Informationen, welche Register gesichert werden, findet man wie immer im User-Guide des C-Compilers, oder auch im Listing des erzeugten Maschinencodes.

5.1.2.3 Bestimmung der IR-Quelle und Löschen des IR-Flags

Werden in einem Programm mehrere Interrupts benutzt, für die der selbe Vektor gilt, dann muss zunächst ermittelt werden, welche Quelle den Interrupt ausgelöst hat.

Dazu werden nacheinander die Interrupt Flags „**IF**“ aller in Frage kommenden Quellen überprüft.

Besteht in einem Programm die Möglichkeit, dass das Flag zwar gesetzt ist, aber der Interrupt gar nicht aktiviert ist, dann muss auch dies anhand des Interrupt-Enable Bits „**IE**“ überprüft werden !

Die meisten IR-Flags müssen explizit gelöscht werden (*IF = 0*) wenn der IR behandelt wurde. Ansonsten wird der Interrupt sofort wieder ausgelöst.

```

#include "uCQ_2018.h"

void __interrupt(high_priority) high_isr(void)
{
    if(INTCONbits.TMR0IE && INTCONbits.TMR0IF){      // if timer 0 overflow
        mTOG_LED_3(); // then toggle LED
        INTCONbits.TMR0IF = 0; // and clear flag
        return;
    }
//    if (interrupt_2_enabled && interrupt_2_flagged){ ...
//        return;
    }
    while(1){} // Endloschleife -> finde unvorhergesehene interrupts
}

```

Vor allem während der Entwicklungsphase eines Programms, ist es nützlich Fallen aufzustellen, in denen das Programm hängen bleibt wenn unberücksichtigte Ereignisse eintreten. Der Aufbau des Templates enthält eine solche Falle. Hier werden nacheinander die in Frage kommenden Quellen abgefragt. Ist eine gültige Quelle identifiziert, wird der zugehörige Programmcode ausgeführt und danach wird sofort die Interruptroutine beendet ohne weitere Quellen zu überprüfen.

Wird keine der vermeintlich erwarteten Quellen als gültig erkannt, verfängt sich das Programm in der Endlosschleife am Ende. Da das Programm dann nicht mehr reagiert, wird man gewöhnlich beim Debuggen auch ohne das explizite Setzen eines Haltepunktes sehr schnell darauf aufmerksam. Beim simplen Anhalten des Programms sieht man sofort an welchem Punkt es steht und kann den Fehler suchen, der dafür verantwortlich ist, dass ein unerwarteter Interrupt ausgelöst wurde.

5.1.3 Das Hauptprogramm *main()* der Blink Variante 3

Da das Umschalten der LED_3 für das Blinken jetzt von der Interrupt Routine übernommen wird, fällt der Code dafür aus dem Hauptprogramm *main()* heraus und es bleibt nur noch die Abfrage des Tasters. Im abschließenden Überblick sieht die main-Datei dann ungefähr folgendermaßen aus:

```
#include "uCQ_2018.h"

void __init(void);

void main(void)
{
    __init();

    while(1) {
        if(mGET_ENC_BTN())
            mSET_LED_2_ON();
        else
            mSET_LED_2_OFF();
    }
}

//-----
void __init(void)
{
    OSCCONbits.IRCF = IRCF_4MHZ; OSCTUNEbits.PLLEN = 0;

    ENC_BTN_TRI = INPUT_PIN; ENC_BTN_ANS = DIGITAL_PIN;

    LED_2_TRI = OUTPUT_PIN;      // LED button
    LED_3_TRI = OUTPUT_PIN;      // LED blink
    mSET_LED_3_ON();           // TIMER_0 setup
    T0CON = 0b10000010;         // TMR0ON T08BIT TOCS T0SE PSA T0PS2 T0PS1 T0PS0

    RCONbits.IPEN = 0;          // Compatibility mode - one IR level/vector
    INTCONbits.TMR0IE = 1;       // enable timer_0 IR
    INTCONbits.GIE = 1;          // global IR enable
}
```

5.2 Interrupt mit Special-Event-Trigger (CCP + Timer)

Genau genommen ist die Tastenabfrage der Blinkvarianten 2 und 3 nicht wirklich „realtime“. Lediglich das Verhältnis der (*im Beispiel sehr kurzen*) Laufzeit der Hauptprogrammschleife im Verhältnis zur Geschwindigkeit mit der ein Mensch eine Taste drücken kann bzw. registriert, dass diese das Aufleuchten einer LED auslöst, lässt diesen Eindruck entstehen.

In der neuen Version soll ein zusätzlicher Timer benutzt werden, um in einem Interrupt einen Taster regelmäßig abzufragen. Wird der Taster gedrückt, dann soll eine LED umgeschaltet werden.

Zur Fähigkeit des Hauptprogramms unterbrechen zu können soll noch die präzise Einstellung der Zeiten für die Tastenabfrage hinzukommen. Capture-Compare Module im Verbund mit einem Timer (1, 3 oder 5) eignen sich dafür wesentlich besser als Timer0 mit oder ohne korrigiertem Zählerwert.

5.2.1 Capture Compare Modul mit Special-Event-Trigger

Das Capture-Compare Modul muss für die geplante Anwendung im **Compare Modus** mit zusätzlichem **Special-Event-Trigger** betrieben werden. Dabei wird der Wert eines Timers mit einem im Compare Register abgelegten Wert verglichen. Erreicht der Timer den Vergleichswert, wird ein Interrupt Flag gesetzt und der Timer wieder auf Null zurückgestellt. Der Timer läuft also nicht bis zum Ende, sondern nur bis zum Vergleichswert.

Der Vergleichswert entspricht dabei genau einem berechneten Wunschendwert. Durch die Nutzung des Compare Modus kann so die ganze Prozedur der Differenzberechnung und Berücksichtigung der notwendigen Korrekturbefehle aus der „genauen Methode“ für Timer_0 entfallen.

Die benötigten Moduseinstellung für das CCP-Modul erfolgten über das zugehörige CCP-Control Register **CCPxCON**. Die Verknüpfung mit den Timern erfolgt im **CCPTMRSx** Register.

REGISTER 14-1: CCPxCON: STANDARD CCPx CONTROL REGISTER							
U-0	U-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
—	—	DCxB<1:0>	—	CCPxM<3:0>	—	—	bit 0
Legend: R = Readable bit W = Writeable bit u = Bit is unchanged x = Bit is unknown '1' = Bit is set '0' = Bit is cleared							
bit 7-6 Unused bit 5-4 DCxB<1:0> : PWM Duty Cycle Least Significant bits Capture mode: Unused Compare mode: Unused PWM mode: These bits are the two LSbs of the PWM duty cycle. The eight MSbs are found in CCPRxL. CCPxM<3:0> : ECCPx Mode Select bits 0000 = Capture/Compare/PWM off (resets the module) 0001 = Reserved 0010 = Compare mode: toggle output on match 0011 = Reserved 0100 = Capture mode: every falling edge 0101 = Capture mode: every rising edge 0110 = Capture mode: every 4th rising edge 0111 = Capture mode: every 16th rising edge 1000 = Compare mode: set output on compare match (CCPx pin is set, CCPxIF is set) 1001 = Compare mode: clear output on compare match (CCPx pin is cleared, CCPxIF is set) 1010 = Compare mode: generate software interrupt on compare match (CCPx pin is unaffected, CCPxIF is set) 1011 = Compare mode: Special Event Trigger (CCPx pin is unaffected, CCPxIF is set) TimerX selected by CXTSEL bits is reset ADON is set, starting A/D conversion if A/D module is enabled ⁽¹⁾							
Note 1: This feature is available on CCP5 only.							

Das Block Diagramm des Compare-Mode gilt für die Timer1, Timer3 und Timer5 und die Compare-Module CCP1 bis CCP5.

REGISTER 14-3: CCPTMRS0: PWM TIMER SELECTION CONTROL REGISTER 0							
R/W-0	R/W-0	U-0	R/W-0	R/W-0	U-0	R/W-0	R/W-0
C3TSEL<1:0>	—	—	C2TSEL<1:0>	—	—	C1TSEL<1:0>	bit 0
bit 7 C1TSEL<1:0>: CCP1 Timer Selection bits 00 = CCP1 – Capture/Compare modes use Timer1, PWM modes use Timer2 01 = CCP1 – Capture/Compare modes use Timer3, PWM modes use Timer4 10 = CCP1 – Capture/Compare modes use Timer5, PWM modes use Timer6 11 = Reserved							

Beispiel für Programmierung in „C“

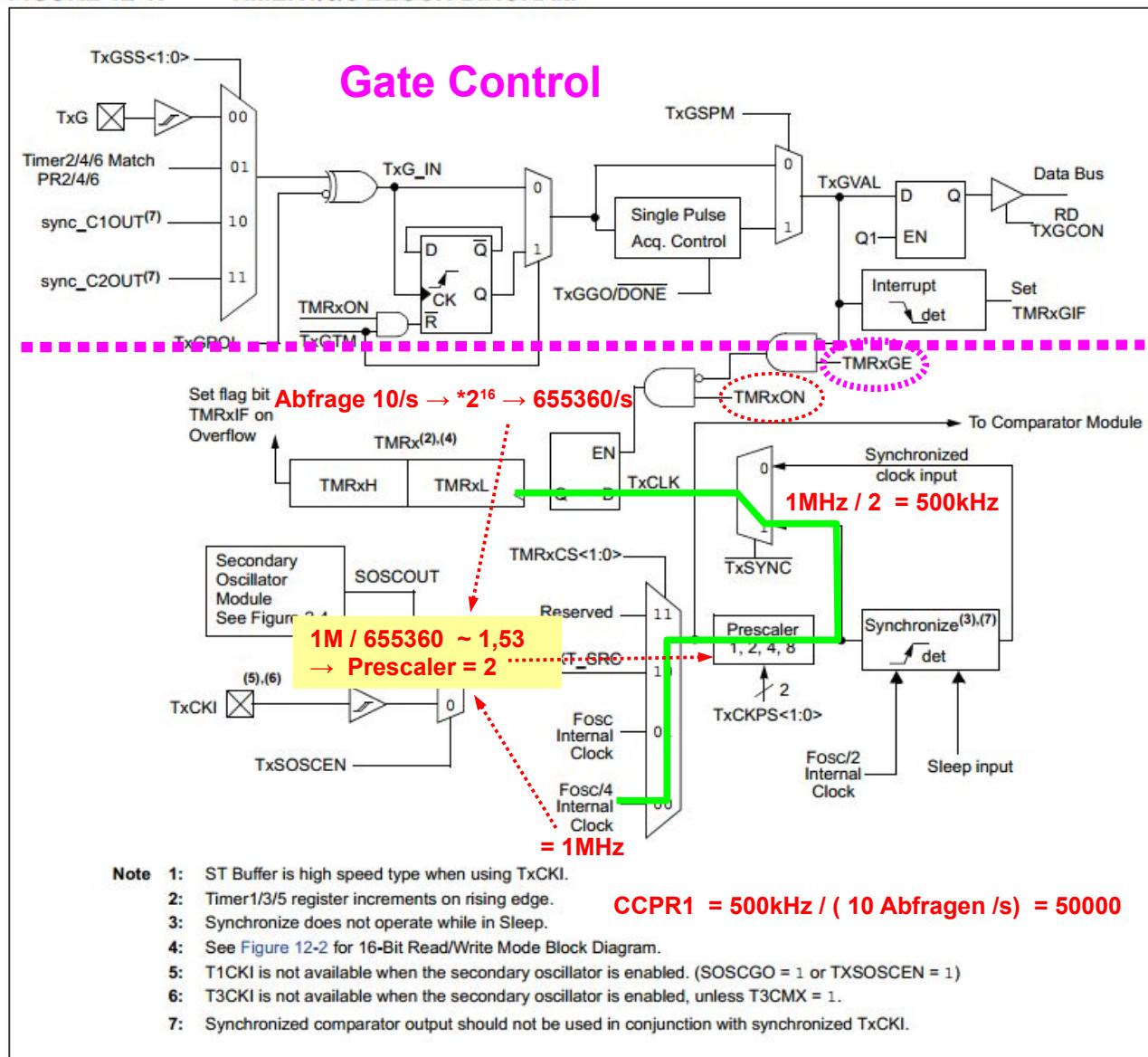
```
// Compare Mode with Special Event Trig.
CCP1CONbits.CCP1M = 0b1011;
// timer <-> ccp module (CCP1 / TMR1)
CCPTMRS0bits.C1TSEL = 0;
```

5.2.2 Timer 1

Angenommen der kürzeste (*schnellste*) Tastendruck, der noch registriert werden soll, sei eine Zehntel Sekunde. Dann müssen Timer1 und CCPR1 Register so konfiguriert werden, dass der CCP1 Interrupt alle 100ms ausgelöst wird.

Analog zu den Berechnungen für Timer0 in den Vorübungen kann man anhand des wesentlich komplexeren **Timer 1 Block Diagramms** folgende Überlegungen anstellen:

FIGURE 12-1: TIMER1/3/5 BLOCK DIAGRAM



Die komplette **Gate-Control** Funktionalität im oberen Teil des Diagramms, mit der das Zählen des Timers über den Enable-Eingang (EN) des Flip-Flops vor dem Timer Register nochmals zusätzlich gesteuert werden kann, wird nicht benötigt und kann abgeschaltet werden. (*bzw. muss nicht aktiviert werden*)

Dazu dient das **TMR1GE** Bit (*Timer 1 Gate Enable*) im Register **T1GCON** (*Timer 1 Gate Control*). Setzt man dieses auf logisch „0“ dann werden alle Signale aus der Gate-Control Logik durch die **UND**-Verknüpfung blockiert. (*Das Bit TMR1ON würde bei „0“ auch blockieren, muss aber natürlich „1“ sein, damit der Timer läuft*)

Aus der gewünschten Abtastfrequenz von 10/s und der Anzahl max. Schritte (2^{16}) bis zu einem Timer-Überlauf erhält man die maximal erlaubte Zähl-Geschwindigkeit (*Frequenz*) von 655360 /s.

Das Verhältnis aus der Eingangsfrequenz des Zählers (*z.B 1MHz bei 4MHz Fosc*) und der berechneten max. geeigneten Frequenz ist wieder die Vorgabe für die Einstellung des Prescalers. Das rechnerische Ergebnis von ~1,53 wird durch den nächst höheren einstellbaren Wert (2) ersetzt. Somit kann die tatsächliche Zälfrequenz **TxCLK** mit $1\text{MHz}/2 = 500\text{kHz}$ berechnet werden.

Letztendlich kann man aus der Zählgeschwindigkeit und der Anzahl der Abfragen des Tasters pro Sekunde den Wert für das Compare Register bestimmen.

$$\text{CCPR1} = 500\text{kHz} / 10/\text{s} = 500\text{kHz} / 10\text{Hz} = 50\text{k} = 50000$$

Bei Timer_0 bestand nur die Möglichkeit die interne Oszillatorfrequenz geteilt durch vier (*Fosc /4*) als Taktfrequenz für den Timer aus zu wählen. Timer_1 bietet weitere Möglichkeiten, die über zwei Bits **TMR1CS<1:0>** im Register **T1CON** eingestellt werden müssen. Die für die Berechnungen angenommene Einstellung entspricht hier dem Wert **0b00** (*T1CONbits.TMR1CS = 0b00;*)

5.2.3 C-Code Compare mode `_init()` und `main()`

Werden die ermittelten Einstellungen in die Initialisierung eingefügt, dann kann das bei einem C-Projekt folgendermaßen aussehen:

```
void __init()
{
    // 4MHz internal clock -> 1MHz instruction/(timer) clock
    OSCCONbits.IRCF = IRCF_4MHZ; OSCTUNEbits.PLLEN = 0;

    ...initialize pins
    ...Timer0 setup

    // use CCP1 for button
    CCP1CONbits.CCP1M = 0b1011; // Compare Mode with Special Event Trigger
    CCP1CONbits.C1TSEL = 0b00; // timer <-> ccp module (CCP1 / TMR1)
    CCP1CONbits.CCPR1L = 50000; // Fosc/4 / prescaler / F_blink = 1MHz /2 /10Hz = 50000

    // use TMR1 for button
    T1GCONbits.TMR1GE = 0; // Timer1/ counts regardless of gate function
    T1CONbits.TMR1CS = 0b00; // Timer1 clock source is instruction clock (FOSC/4)
    T1CONbits.T1CKPS = 1; // 2^1 -> 2 Prescale value
    T1CONbits.TMR1ON = 1; // switch on Timer1

    ...Initialize interrupt system
}
```

Da die Tastenabfrage in den Interrupt verlegt wird, muss das Hauptprogramm in diesem Beispiel nichts tun oder könnte sogar „**schlafen**“.

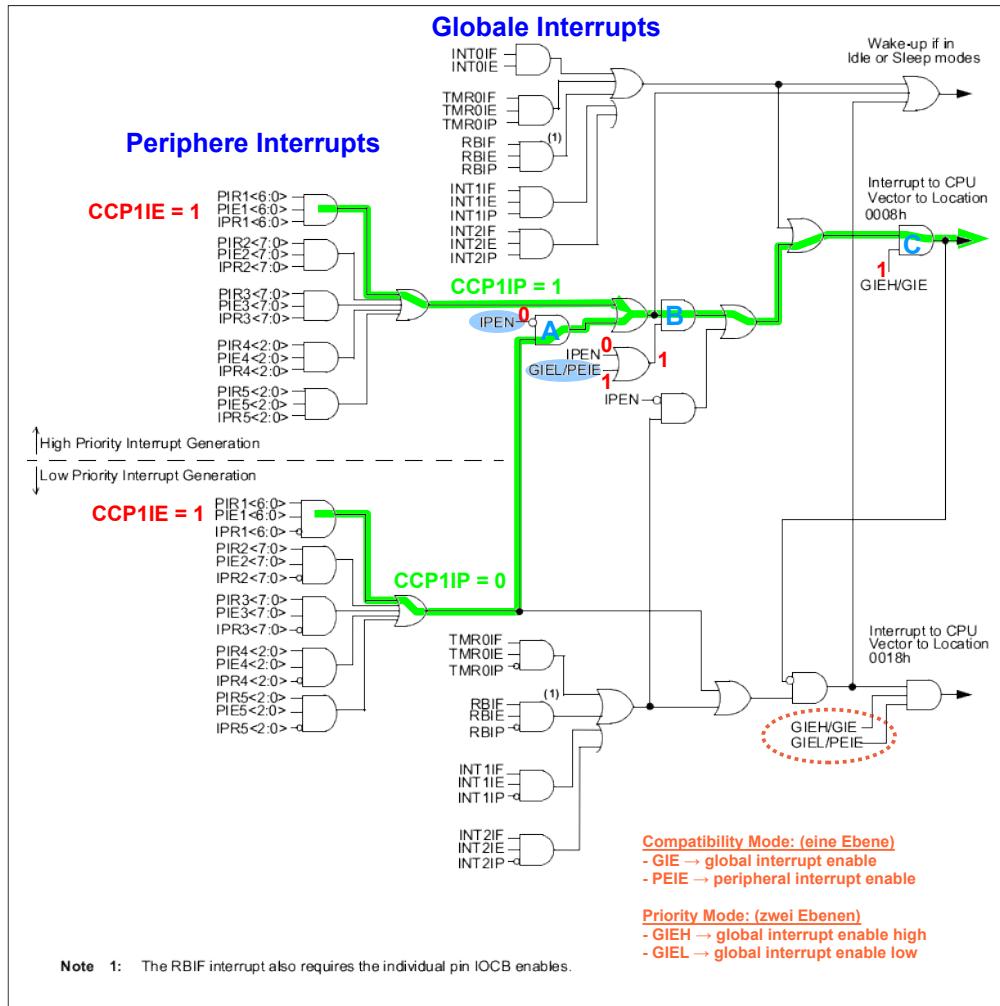
```
void main()
{
    __init(); // C18 calls this function in startup code

    // OSCCONbits.IDLEN = 1; // ??? want to know about? -> see DATASHEET ;-
    while(1){
        Nop(); // not really a function - try [ctrl]+[click] ???
        // Sleep(); // alternative - no function neither ;-
        Nop();
    }
}
```

5.2.4 CCP Interrupt Initialisierung

Damit das CCP Modul auch einen Interrupt auslösen kann, muss natürlich noch das Interrupt System eingestellt werden. Dazu nimmt man am besten wieder das entsprechende Diagramm aus dem Datenblatt zur Hand. In der folgenden Abbildung ist der Weg des CCP1 Interrupts dargestellt.

FIGURE 9-1: PIC18 INTERRUPT LOGIC



Da alle „Globalen“ Interrupts aufgezeichnet sind und der CCP1 IR nicht dabei ist, muss er zu den „**Peripheren Interrupts**“ gehören. Weil es recht viele davon gibt, sind nur die Register aufgeführt, welche die Steuerbits beinhalten.

Für das Auslösen eines IR sind von den **UND** bzw. **ODER** Verknüpfungen im obigen Diagramm vor allem die UND Verknüpfungen (**A B C**) maßgeblich. Sie funktionieren als Torschaltungen die den IR passieren lassen oder eben nicht. Für die Passage ist eine „1“ am jeweils anderen Eingang nötig. Ein kleiner Kreis vor einem der Eingänge invertiert das Signal. Vor einem Kreis braucht man also eine Null um das Tor freizugeben.

- **A:** Das „Priority“ System soll nicht verwendet werden. Deshalb muss das Bit IPEN (**Interrupt-Priority-ENable**) „0“ sein. Durch die Invertierung wird das Tor geöffnet !
- **B:** Zur Freischaltung muss zumindest eines der Bits IPEN oder PEIE „1“ sein. Da IPEN schon auf „0“ festgelegt ist, muss das Peripheral-interrupt-ENable Bit „1“ gesetzt werden.
- **C:** Die Letzte Schranke ist das **Global-Interrupt-Enable** Bit.

Die sich ergebenden zwei Wege zeigen, dass es bei nicht aktivierte IR-Prioritäten letztendlich keine Rolle spielt ob das IP-Bit gesetzt ist oder nicht. Bei gesetztem IP-Bit führt der obere Weg zum Ziel, bei gelösctem IP-Bit der Weg von unten.

Alle erforderlichen Einstellungen müssen in die Initialisierung eingefügt werden. Dabei sollten die Steuerbits die weiter rechts im Diagramm auftauchen (**PEIE, GIE**) erst als letzte Befehle der Initialisierung gesetzt werden um von vornherein jegliche „Unterbrechung“ während der Initialisierung zu unterbinden.

```

void __init()
{
...Oszillator, Pins, CCP and Timer init

    RCONbits.IPEN = 0;           // Compatibility mode - one IR level/vector
    INTCONbits.TMR0IE = 1;       // enable timer_0 IR
    PIE1bits.CCP1IE = 1;         // interrupt enable - CCP1
    INTCONbits.PEIE = 1;         //   - CCP1 IR is a "peripheral IR"
    INTCONbits.GIE = 1;          //   - global IR enable bit
}

```

5.2.4.1 IR-Quelle bestimmen, IR-Flag löschen und Code einarbeiten

Zur Erinnerung:

Werden in einem Programm mehrere Interrupts benutzt, für die der selbe Vektor gilt, dann muss zunächst ermittelt werden, welche Quelle den Interrupt ausgelöst hat.

Dazu werden nacheinander die Interrupt Flags „**IF**“ aller in Frage kommenden Quellen überprüft.

Besteht in einem Programm die Möglichkeit, dass das Flag zwar gesetzt ist aber der Interrupt gar nicht aktiviert ist, dann muss auch dies anhand des Interrupt-Enable Bits „**IE**“ überprüft werden !

Die Tastenabfrage erfolgt 10 mal pro Sekunde. Bei einem „langsamen“ Drücken der Taste also vermutlich auch mehrmals. Damit der Taster die LED bei jeder Betätigung nur ein mal toggelt, muss man sich irgendwie merken, ob die Taste auch bei der vorherigen Abfrage schon gedrückt war!

Wenn man in einer Variable mitzählt, wie oft die Taste schon gedrückt war, dann kann man sehr einfach ein Verhalten erzielen, wie man es z.B. von Tastaturen kennt. Bleibt man etwas länger auf einer Taste dann wird die Aktion (*bei einer Tastatur das Zeichen*) wiederholt.

```

#include "uCQ_2018.h"

#define autorepeate 10;           // <<-- ???
unsigned char btn_checker;

void __interrupt(high_priority) high_isr(void)
{
    if(INTCONbits.TMR0IE && INTCONbits.TMR0IF){      // if timer 0 overflow (blink IR)
        mTOG_LED_3();                                // then toggle LED
        INTCONbits.TMR0IF = 0;                         // and clear flag
        return;
    }
    if (PIE2bits.CCP2IE && PIR2bits.CCP2IF){        // check button IR
        PIR2bits.CCP2IF = 0;                           // clear flag

        if(mGET_BTN_ENC()){
            if (btn_checker++ == 0) mTOG_LED_3();
            if (btn_checker >= autorepeate) btn1_checker = 0;
        }
        else{
            btn_checker = 0;
        }
        return;
    }
    while(1){}           // endless loop → detect unexpected interrupts
}

```

5.2.5 Übung: Blink Variante 4 mit CCP_2 & Timer_3

Als kleine Vertiefung kann man jetzt durch Verwendung einer weiteren CCP-Timer Kombination, anstelle von Timer_0, auch noch das sekündliche Blinken genau einstellen. Dazu kann irgend eines der verbliebenen CCP Module und mit Timer_3 oder Timer_5 verwendet werden.

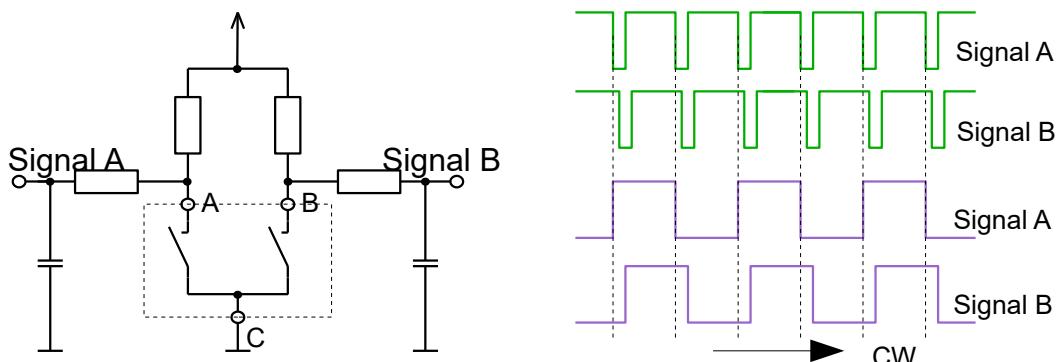
5.3 Pin Interrupts INTx (Drehgeber rotiert LEDs)

Signaländerungen an Eingangspins muss man nicht unbedingt mit Hilfe eines Timers kontrollieren. Diese können auch direkt zum Auslösen eines Interrupt genutzt werden. Das ist besonders dann wichtig, wenn auf bestimmte Events sofort reagiert werden sollte. Abfragen von Bedienelementen wie Eingabetasten oder ähnliches gehören hier eigentlich nicht dazu, weil der Controller immer so schnell sein wird, dass der Benutzer keine Verzögerungen bemerkt. Da die Vorgehensweise aber identisch ist, bzw. auf einfachen Demoboards meist nur Taster oder ähnliches zur Verfügung stehen, kann man die Funktionalität der Pin Interrupts auch mit diesen einfachen Signalen testen.

5.3.1 Drehgeber (Inkrementalgeber)

Auf den uCQ-2013 Boards sind Drehgeber (*Rotary Encoder*) vorhanden, welche beim Drehen Impulse erzeugen, die über einen Interrupt-Pin ausgewertet werden können. Drehgeber haben eine bestimmte Anzahl von Positionen pro Umdrehung. Die Positionen sind bei den als Bedienelement ausgeführten Gebern beim Drehen meist gut spürbar.

Die Erzeugung der Impulse erfolgt nach einem einfachen Schalterprinzip und kann so realisiert sein, dass der Schalter bei jedem Positionswechsel seine Stellung ändert (*Signalverlauf in der Abbildung violett dargestellt*), oder nur während des Wechsels geschlossen wird und im folgenden Ruhezustand wieder geöffnet ist (*grün dargestellt*). Für die Erkennung der Drehrichtung werden zwei Signale (*A und B*) verwendet, welche zeitverzögert (*winkelverzögert*) schalten. Aus dem Zustand des einen (*z.B. Signal B*) bei der Änderung des Anderen (*Flanke Signal A*) kann man die Drehrichtung ableiten. **Gestrichelte Linien im Diagramm liegen auf Flanken, nicht Rasten!**



Achtung: Für die Auswertung der Encoder über Interrupt müssen die Schalter unbedingt mit Hilfe der Widerstände und Kondensatoren in der obigen Abbildung entprellt sein!

Der zeitliche Verlauf der Signale A und B wird im Datenblatt zum Drehgeber meist im Uhrzeigersinn dargestellt (*CW, Clock Wise*).

Bei der Realisierung als Tastschalter (*grün*) bedeutet ein High-Pegel an Signal B bei fallender Flanke des Signals A, dass der Geber im Uhrzeigersinn gedreht wurde.

Bei einer Drehung gegen den Uhrzeigersinn (*Signalverlauf von rechts nach links*), hat Signal B bei der fallenden Flanke von A Low-Pegel.

Ist der Drehgeber als Wechselschalter ausgeführt (*violett*), dann müssen beide Flanken eines Signals ausgewertet werden. Bei einer steigenden Flanke an A und Low-Pegel an B ($\uparrow L$) oder fallender Flanke an A und High-Pegel an B ($\downarrow H$) gilt Drehung im Uhrzeigersinn.

Bei $\downarrow L$ oder $\uparrow H$ ergibt sich dann eine Drehung gegen den Uhrzeigersinn.

Auf den fertig bestückten uC-Quick Boards, welche in der Bibliothek am Standort OE ausgeliehen werden können, sind Drehgeber vom grünen Typ verbaut. Die Bausätze enthalten Drehgeber vom violetten Typ

5.3.2 INTx Interrupt

Für einfache Nutzbarkeit im zu schreibenden Quellcode kann man zuerst wieder Text-Substitution-Label für die verwendeten Pins definierten. Auf den uCQ-2013 Platinen sind die Pins B_0 und A_2 mit den Encoder-Signalen A und B verbunden. Pin B_0 ist auch als Interrupt-Eingang konfigurierbar und erhält deshalb den Namen „**ENC_INT**“. Das zweite Signal ist dann für die Richtungsauswertung zuständig und erhält deshalb den Namen „**ENC_DIR**“

```
// in ucQ_2018.h

#define ENC_INT           PORTBbits.INT0          // A -> interrupt pin int_0/b_0
#define ENC_INT_TRI       TRISBbits.TRISB0
#define ENC_INT_ANS       ANSELBbits.ANSB0
#define ENC_DIR           PORTAbits.RA2          // B -> direction pin a_2
#define ENC_DIR_TRI       TRISAbits.TRISA2
#define ENC_DIR_ANS       ANSELAbits.ANSA2
```

Die Einstellungen des INT0 Interrupts befinden sich in den Registern INTCON und INTCON2.

Für die Freigabe und Erkennung des Interrupts werden die Bits INT0IE (*enable*) und INT0IF (*flag*) im INTCON benötigt.

Zur Einstellung der aktiven Flanke das Bit INTEDG0 (*edge*) im Register INTCON2.

Für alle diese Bits, bzw. Kombinationen kann man wieder Text-Substitution-Label definieren.

9.9 INTn Pin Interrupts

External interrupts on the RB0/INT0, RB1/INT1 and RB2/INT2 pins are edge-triggered. If the corresponding INTEDGx bit in the INTCON2 register is set (= 1), the interrupt is triggered by a rising edge; if the bit is clear, the trigger is on the falling edge. When a valid edge appears on the RBx/INTx pin, the corresponding flag bit, INTxF, is set. This interrupt can be disabled by clearing the corresponding enable bit, INTxE. Flag bit, INTxF, must be cleared by software in the Interrupt Service Routine before re-enabling the interrupt.

```
// in ucQ_2018.h

#define ENC_IR INTCONbits.INT0IE && INTCONbits.INT0IF
#define mENC_IR_DIS() INTCONbits.INT0IE = 0
//          // _|__|__|__|__|__|__ (toggles)
#endifif ENCODER_TOGGLE
#define ENC_DIR_UP !INTCON2bits.INTEDG0
#define ENC_DIR_DOWN INTCON2bits.INTEDG0
#define mENC_IR_EN() INTCON2bits.INTEDG0 = !ENC_INT; INTCONbits.INT0IE = 1
#define mENC_IR_RST() INTCON2bits.INTEDG0 ^= 1; INTCONbits.INT0IF = 0
#else
#define ENC_DIR_UP 1 // _|__|__|__|__|__|__ (pulses)
#define ENC_DIR_DOWN 0
#define mENC_IR_EN() INTCON2bits.INTEDG0 = 0; INTCONbits.INT0IE = 1
#define mENC_IR_RST() INTCONbits.INT0IF = 0
#endifendif
```

Ein „**ENC_IR**“ (*Encoder-Interrupt*) wird dann angefordert, wenn der Interrupt geflagt und aktiviert (*enabled*) ist.

Bei der Aktivierung des Interrupts mit **mENC_IR_EN()** wird zusätzlich die aktive Flanke für den Interrupt eingestellt. Diese ist bei den pulsenden Encodern immer die gleiche (*hier die fallende*). Bei den umschaltenden Encodern ist die nächste Flanke abhängig vom momentanen Zustand. Bei Zustand High, ist die nächste Flanke fallend, bei Zustand Low ist die Flanke steigend.

Die Drehrichtung **ENC_DIR_UP** / **ENC_DIR_DOWN** ist bei toggelnden Encodern auch von der aktiven Flanke abhangig. Die aktive Flanke muss bei diesen auch in jedem Interrupt umgestellt werden. Dieses Toggeln wurde hier mit in das Makro **mENC_IR_RST()** (*reset*) aufgenommen.

5.3.3 Flags - Kommunikation zwischen Interrupt und main()

Das folgende Demo-Beispiel für die Auswertung der Drehgeber Pulse ist wie alle IR-Beispiele in eine Hauptprogramm-Funktion und in die Interrupt-Routine aufgesplittet. Im Hauptprogramm finden die Initialisierung der Pins und später die Auswertung von in der IR-Routine gesetzten „Flags“ (*Merkern*) statt.

Die Flags sind Informationen die nur ein Bit (*Ja/Nein; I/O*) benötigen. Einzelne Bits sind in C nicht vorgesehen. Um nicht für eine ein-Bit Information ein Byte an Speicherplatz zu verbrauchen, kann man das Konstrukt „**bit field**“ verwenden.

Dieses „**bit field**“ wird genau wie bei den SFR (*z.B. PORTBbits*) in den Prozessor-Headern als **Struct** definiert. Bei einem Bitfeld enthält die Struktur nur Integer Elemente. Die Größe der jeweiligen Elemente in Bits, wird hinter dem Doppelpunkt angegeben.

Damit man auch auf alle Bits auf einmal zugreifen kann, bildet man eine **Union** des Bitfeldes mit einem zusätzlichen **unsigned char** Typ. Diese Union kann man mit **typedef** als neuen Typ definieren, den man dann Variablen auch zuweisen kann.

```
// new project_header.h

#include uCQ_2018.h

typedef union {                                // type declaration
    unsigned char all;
    struct {
        unsigned encUp      : 1;      // 1 bit - flags encoder movement clockwise
        unsigned encDown   : 1;      // "" counter clockwise
        unsigned bit2_7    : 6;      // 6 bits not yet used
    };
}demoFlags_t;                                    // name of new type

extern demoFlags_t flags;                    // declaration variable of new type as extern
```

5.3.3.1 Variablen (Flags) in mehreren Source Dateien verwenden

Wenn die Variable in mehreren Dateien (*z.B. interrupt.c und main.c*) verwendet wird, dann müssen die obigen Deklarationen in einem Header stehen, den beide *.c Dateien einbinden. Es ist nicht sinnvoll, dafür *uCQ_2013.h* zu verwenden, da in diesem nur hardwarespezifische Dinge stehen sollten. Hier sollte man unbedingt eine **neue Header-Datei** mit einem zum aktuellen Projekt passenden Namen anlegen und diesen dann in die *.c Dateien des Projektes einbinden. Dieser neue Project-Header kann dann auch die Einbindung von *uCQ-2013.h* übernehmen.

Eine Variable, darf nur in einem einzigen *.c File **definiert** sein. Damit von anderen *.c Dateien darauf zugegriffen werden kann, wird die Variable im neuen Project-Header als **extern deklariert**.

```
#include "project_header.h"

demoFlags_t flags;                            // defining variable of new type
unsigned char dummy = 0b00000100; // a global auxiliary variable (LEDs RB2..RB5)

void __init(void)
{
    mALL_LED_OUTPUT();
    ENC_INT_TRI = INPUT_PIN; ENC_INT_ANS = DIGITAL_PIN;
    ENC_DIR_TRI = INPUT_PIN; ENC_DIR_ANS = DIGITAL_PIN;

    mENC_IR_CLR(); mENC_IR_EN();

    flags.all = 0; LATB = ~dummy; // initializing new variable & LEDs
    INTCONbits.GIE = 1;
}
```

Die Bitfeld Variable *flags* wird dann noch in der Funktion *__init()* initialisiert.

5.3.4 Setzen der Flags im Interrupt

In der Interrupt Service Routine wird zunächst mit Hilfe der vorher erstellten Makros überprüft, ob ein Interrupt durch den Encoder angefordert wurde. Ist das der Fall, dann wird die Dreh-Richtung ausgewertet und in den Flags gespeichert. Am Ende muss das Interrupt-Flag wieder gelöscht und gegebenenfalls die aktive Flanke für den nächsten Interrupt umgeschaltet werden.

```
#include "project_header.h"

void __interrupt(high_priority) high_isr(void)
{
    if(ENC_IR){
        if(ENC_DIR == ENC_DIR_UP) flags.encUp = 1;
        else flags.encDown = 1;

        mENC_IR_RST();           // clear flag and toggle edge if necessary
        return;
    }
    while(1);                  // (detect unexpected IR sources)
}
```

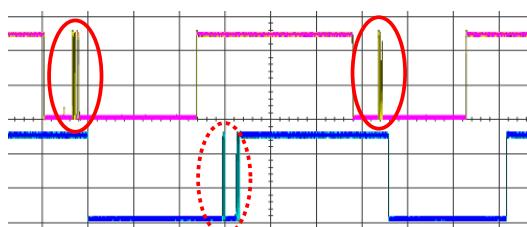
5.3.5 Abfrage der Flags in main()

In der `main()` Funktion werden die Flags nun über eine `if()` Bedingung abgefragt, und das Rotieren der LEDs mittels Shift-Left (`<<`) bzw. Shift-Right (`>>`) Operatoren durchgeführt.

Für diesen Zweck und zur besseren Verständlichkeit, wird zunächst die Hilfsvariable **dummy** angelegt. *Als kleine Übung kann man versuchen, alle Aktionen direkt mit LATB auszuführen.*

```
void main(void)
{
    while(1){
        if(flags.encUp){
            dummy = dummy << 1;                                // shift left (LED2->3->4->5)
            if(dummy > 0b00100000) dummy = 0b00000100;      // LEDs are low-active
            LATB = ~dummy;
            flags.encUp = 0;
        }
        if(flags.encDown){
            dummy = dummy >> 1;                                // shift right (LED5->4->3->2)
            if(dummy < 0b00000100) dummy = 0b00100000;      // LEDs are low-active
            LATB = ~dummy;
            flags.encDown = 0;
        }
    }
}
```

5.3.6 Mögliche Probleme bei der Encoder Auswertung über Pin Interrupts



Prellen der Kontakte bei Encodern

Bei der einfachen Auswertung über Interrupts, dürfen die Encodersignale auf keinen Fall beim Drehen prellen (wackeln). Ohne Entprellung werden viele ungewollte Interrupts ausgelöst und die Auswertung wird durch diese „falschen“ Interrupts völlig unbrauchbar!

Siehe auch:

[11.4 State Machine zur Auswertung von Drehgebern](#)

6 Bibliotheken, Funktionen, Plib/MCC

6.1 Überblick

„Libraries“ (*Funktionssammlungen*) kann man in verschiedene Bereiche unterteilen, die im folgenden kurz beschrieben werden.

6.1.1 System-Bibliotheken

Die System-Bibliotheken werden dem Programm im Hintergrund über Anweisungen an den Linker automatisch hinzugefügt. Siehe dazu auch Kapitel [1.3.3 Linker](#) ff.

6.1.1.1 Start-Up Code

Bei einem C Programm müssen vor Ausführung der main() Funktion einige Objekte initialisiert werden. Dies wird vom sogenannten „Startup-Code“ übernommen.

Beim XC8 wird der Statup-Code immer für das jeweilige Projekt speziell erzeugt. Nähers bitte den User Guides der Compiler entnehmen. (oder Kapitel [1.3.3 Linker](#) und [2.5.1 Der Startup Code](#))

6.1.1.2 Prozessor unabhängige Bibliotheken

Die C Standard Bibliotheken für mathematische Funktionen, Zeichenkettenverarbeitung und ähnliches sind beim XC8 Compiler in den Bibliotheken im Installationsordner .../XC8/v1.4x/lib enthalten. Auf den Funktionsumfang der allgemeinen C Bibliothek wird weiter unten in [6.2 General Software Library](#) noch detaillierter eingegangen.

6.1.1.3 Prozessor-spezifische Bibliotheken

Die „Plib“ des alten C18 Compilers, welche bis zur Version v1.34 auch beim XC8 Compiler standardmäßig in der Installation vorhanden war, ist für die Verwendung der prozessorspezifischen Hardwaremodule (*Timer* ...) zuständig. Einen Einblick in Funktionsumfang und Verwendung gibt das Kapitel [6.3 Peripheral Library](#)

6.1.1.4 MCC - MPLAB Code Configurator

Der (*relativ neue*) „MPLAB Code Configurator“ ersetzt die bisherige PLIB. Der Zugriff auf die Hardwaremodule erfolgt dann nicht mehr über vor-kompilierten Code in einer Bibliothek, sondern man kann sich den Code mit Hilfe einer graphischen Oberfläche für die jeweiligen Module erstellen lassen. Eine Einführung findet man in der Microchip Developer Help
<https://microchipdeveloper.com/mcc:start> – Ein Beispiel in [6.10 Microchip Code Creator Beispiel](#)

6.1.2 Eigene (nicht zum System gehörende) Bibliotheken

Bibliotheken müssen nicht immer vor-kompiliert sein. Es kann sich auch um eine Sourcecode-Sammlung handeln, deren Dateien man im Bedarfsfall einfach in das Projekt einbindet.

Dieses Vorgehen bietet sich an, wenn bestimmte Funktionen immer wieder in verschiedenen Projekten benötigt werden. Das kann z.B die Ansteuerung eines LCD (*Liquid-Crystal-Display*) oder eines bestimmten Sensors sein. Alle dazu benötigten Funktionen packt man zusammen in eine oder mehrere Source- und deren zugehörige Header-Dateien.

Die Sourcecode-Bibliotheken kopiert man im Normalfall nicht in die Verzeichnisse der jeweiligen Projekte, sondern hält sie in einem separaten Bibliotheksverzeichnis, auf das dann verschiedenste Projekte zugreifen können. So vermeidet man unterschiedliche Versionen die durch Erweiterungen oder Fehlerkorrekturen in unterschiedlichen Projekten entstehen würden. Änderungen wirken sich auf alle Projekte aus und müssen demnach auch sehr sorgfältig durchgeführt werden.

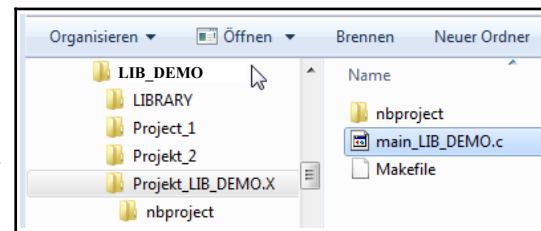
6.2 Eine eigene Sourcecode-Bibliothek erstellen / verwenden

Wer bisher noch kein Softwareprojekt realisiert hat, bei dem bestimmte, in mehreren Projekten wiederverwendbare, Teile in separate Dateien bzw. Verzeichnisse ausgelagert wurden, der sollte das folgende Übungsprojekt durcharbeiten.

Zunächst legt man am besten einen Übungsordner an, in dem die erforderlichen Strukturen später gut ersichtlich werden. Nennen wir ihn **LIB_DEMO**.

In diesem Ordner kann man dann ein neues Projekt mit dem Namen **Project_LIB_DEMO** und einen weiteren Ordner **LIBRARY** anlegen.

Im Projekt brauchen wir natürlich ein Sourcefile mit der *main()* Funktion und den wichtigsten Configuration Bits. Diese Datei wurde in der Abbildung rechts **main_LIB_DEMO.c** genannt.



Für die erste eigene Bibliothek soll ein möglichst einfaches Thema gewählt werden, welches keine bisher unbenutzten Komponenten verwendet und nicht durch unnötige Komplexität vom eigentlichen Aufbau der Bibliotheksstruktur ablenkt. Die Behandlung von digitalen Ausgängen an denen LEDs angeschlossen sind sollte inzwischen beherrscht werden und kann deshalb hier als Beispiel dienen. (*auch wenn es nicht wirklich sinnvoll wäre eine solche Bibliothek zu benutzen*)

Die LED-Bibliothek soll Funktionen bereit stellen um die vier LEDs auf der uC-Quick Plattform zu benutzten. Der Funktionsumfang soll die Konfiguration des Pins als Ausgang, sowie Ein- Aus- und Um-schalten der LED beinhalten. Im Testprojekt könnten die LED-Funktionen dann wie folgt eingesetzt werden.

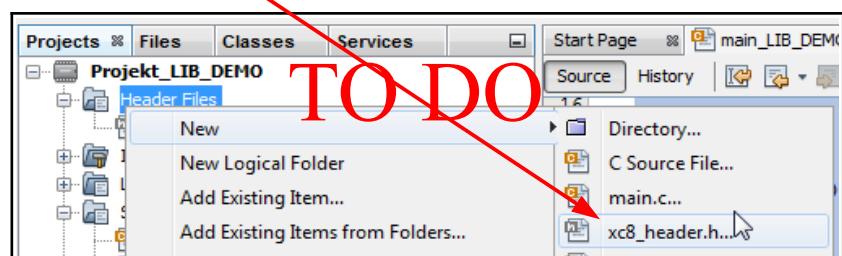
```
/*
 * File: main_LIB_DEMO.c
 * Author: vschilli
 * Created on 12. April 2018, 09:08
 */
#include <xc.h>
#include "../LIBRARY/LEDfunctions.h"

#pragma config FOSC = INTIO67
#pragma config WDTEN = OFF
#pragma config LVP = ON

#define _XTAL_FREQ 1000000 // default 1MHz (macro needed for __delay... functions)

void main(void)
{
    LED_setup(1);
    //...
    // LED_on(4);

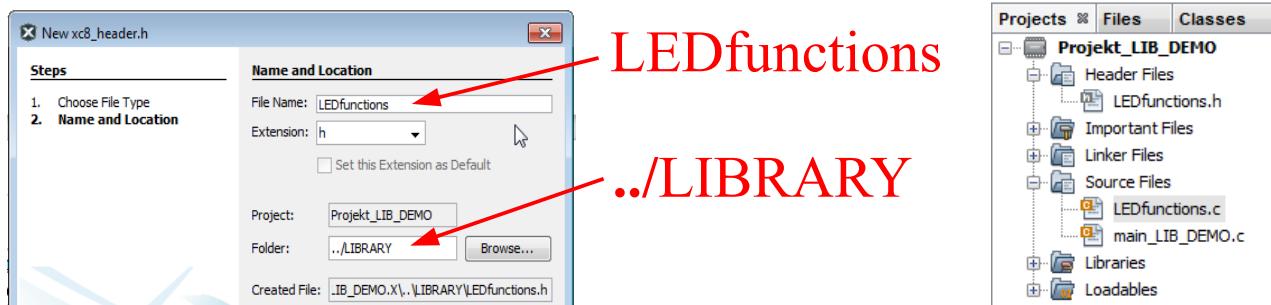
    while(1) {
        LED_on(1);
        LED_toggle(2);
        LED_on(3);
        __delay_ms(500); // (((C18 -> Delay1KTCYx(125);)))
        LED_off(1);
        __delay_ms(500); // (((C18 -> Delay1KTCYx(125);)))
        LED_toggle(2); LED_toggle(3); LED_toggle(4);
    }
}
```



Damit der C-Compiler die Datei **main_LIB_DEMO.c** in eine Objektdatei übersetzen kann, muss er die hier benutzten Funktionen kennen. Deshalb schreibt man die Prototypen der LED-Funktionen in den Header **LEDfunctions.h**, der im Bibliotheksordner angelegt werden sollte. Das Einbinden des Headers in die Projektdatei erfolgt mit dem kompletten Pfad. **#include "../LIBRARY/LEDfunctions.h"**

Den neuen Header kann man über das Pop-Up Menü im Projektfenster der IDE anlegen.

Im sich öffnenden Dialog kann/muss man dann den gewünschten Speicherort auch relativ zum Projektordner angeben, wenn dieser im Dateiverzeichnis oberhalb des Projektverzeichnisses liegt.



Für das vorbereitete Verzeichnis LIBRARY gibt dazu man unter Folder: „..../LIBRARY“ ein.
Auf die selbe Weise kann man auch noch das „C Source File“ **LEDfunctions.c** anlegen.

In die neu angelegte Source Datei kommen die Definitionen der Funktionen zur Ansteuerung der LEDs. Im zugehörigen Header werden wie schon erwähnt die Prototypen der Funktionen deklariert.

```
/* File: LEDfunctions.h
 * Author: VSK HS-Ulm
 * Comments:
 * Revision history:
 */

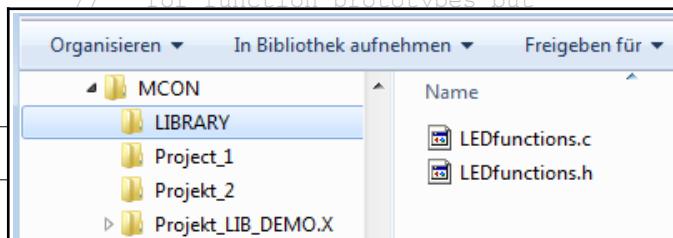
#ifndef LED_FUNCTIONS_H // This is a guard condition so that contents of this file
#define LED_FUNCTIONS_H // are not included more than once.

// LED library prototypes
extern void LED_setup(unsigned char ledNr); // Init LED off and TRIS
extern void LED_on(unsigned char ledNr); // the keyword extern is superfluous
extern void LED_off(unsigned char ledNr); // for function prototypes but
//...TODO

#endif /* LED_FUNCTIONS_H */

/* File: LEDfunctions.c
...
#include <xc.h>
//#include "LEDfunctions.h" /* sometimes useful, here not necessary */

void LED_setup(unsigned char ledNr)
{
    if(ledNr == 1){ TRISBbits.TRISB2 = 0; }
    else if(ledNr == 2){ TRISBbits.TRISB3 = 0; }
    else Nop(); // TODO
}
void LED_on(unsigned char ledNr)
{
    switch(ledNr){
        case 1: LATBbits.LATB2 = 0; break;
        case 2: Nop(); // TODO
        // ...
        default: break;
    }
}
void LED_off(unsigned char ledNr)
{
    LATBbits.LATB2 = 1;
    Nop(); // TODO
}
// TODO...
```



6.3 Sourcecode und Header Dateien

Das wichtigste Unterscheidungsmerkmal zwischen Sourcecode- und Header- Dateien ist:

Die Sourcecode-Datei enthält alles, was die Reservierung von Speicherplatz im Programm oder Datenspeicher zur Folge hat! (Definitionen von Funktionen und Variablen)

Die Header-Datei enthält nichts, was eine Reservierung von Speicherplatz im Programm oder Datenspeicher zur Folge hat! (nur Deklarationen von Variablen (als extern), Prototypen von Funktionen, Text Substitution Makros, ...)

Die Header Datei sollte nur die Prototypen der Funktionen enthalten, die für die Verwendung in anderen Source-Dateien vorgesehen sind und auch nur die Variablen sollten als extern deklariert werden, die für eine globale Verwendung vorgesehen sind!

Auch im Einführungsbeispiel des vorherigen Kapitels enthält die Source-Datei die Definitionen und der Header lediglich die Prototypen der **Funktionen**.

Der Header ist nur eine Art Inhaltsverzeichnis der Sourcecode-Datei, welches dem Compiler die Beschreibungen der Funktionen in Form von Prototypen bereit stellt, damit dieser die ordnungsgemäße Verwendung dieser innerhalb anderer Dateien überprüfen kann.

Diese Beschreibungen werden anderen Source-Dateien mittels der Direktive „#include ...“ hinzugefügt. Die Include-Direktive bewirkt, dass der Inhalt des entsprechenden Headers in die Datei eingefügt wird.

Werden in der Sourcecode-Datei auch **globale Variablen** verwendet, auf die von anderen Source-Files zugegriffen werden darf, dann verhält es sich genau wie bei den Funktionen.

Die Definition der Variablen muss einmalig in der Source-Datei der Library erfolgen.
Im Header steht lediglich eine Deklaration, also wieder nur eine Beschreibung der Variablen.

Diese Deklaration im Header wird mit dem Codewort „**extern**“ ergänzt, welches dem Compiler nach dem Einfügen des Textes der Library-Header-Datei in die fremde Source-Datei signalisiert, dass die **Variable** schon in einer anderen Source-Datei (der der Lib.) definiert wurde und nicht nochmals benötigt wird.

Enthielte eine Header Datei Definitionen von Funktionen oder Variablen, dann würden diese bei jedem Einfügen des Header in eine Sourcecode-Datei neu definiert, also im Speicher angelegt und wären somit mehrfach (*in mehreren vom Compiler aus den Source-Dateien erzeugten Object-Dateien*) vorhanden.

Auch wenn der Linker dies erkennt und möglicherweise im Sinn des Programmierers korrigiert, sollte man das unter allen Umständen vermeiden!

In bestimmten Fällen multipler Definition werden vom Linker auch folgende Fehler ausgelöst:

(482) symbol "" is defined more than once in "*"*

oder

error: redefinition of "" ... note: previous definition is here...*

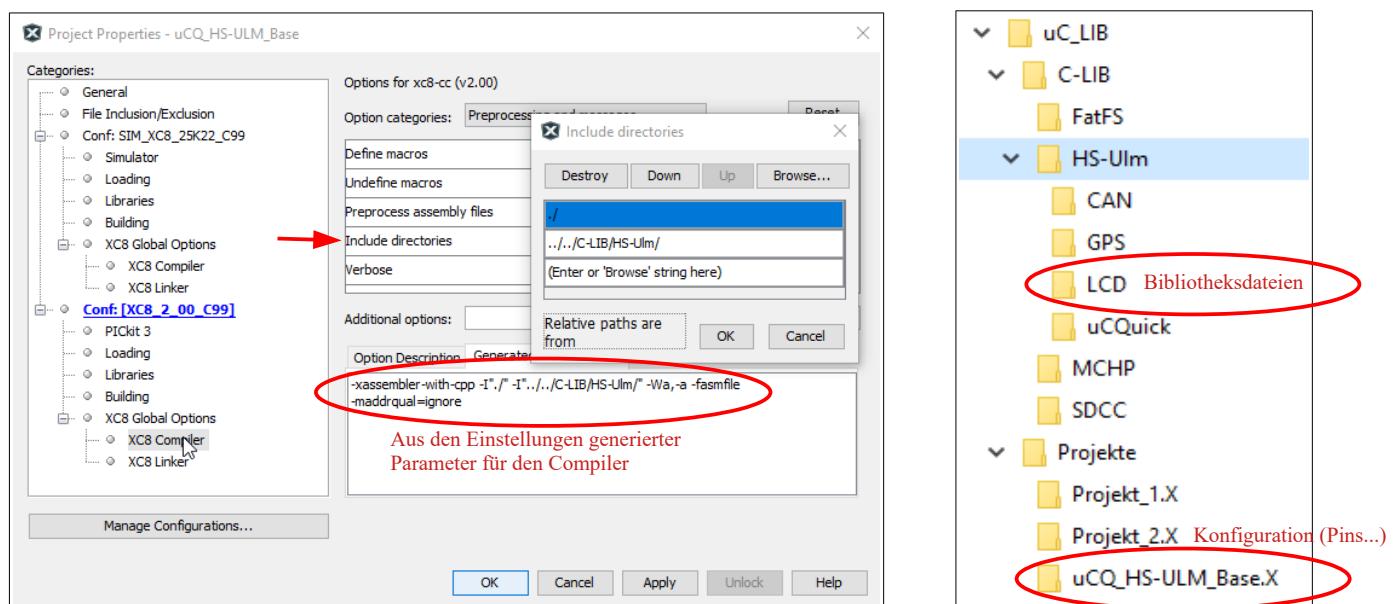
6.4 Compiler Include Directories der IDE

Spätestens wenn der Compiler Sourcen übersetzen soll, die Include-Files einbinden, welche in einem Verzeichnis liegen, das zum Zeitpunkt der Erstellung der Source-Datei nicht bekannt ist, dann stellt sich die Frage, wie man dem Compiler die Location übermitteln kann.

Dieser Fall tritt auch ein, wenn der Pfad zu einer Include-Datei für verschiedene Projekte variabel ist, wie im später folgenden Beispiel der LCD-library noch zu sehen sein wird.

In diesem Fall handelt es sich um eine Bibliothek, deren Source- und Header-File in einem Bibliotheksordner liegen. Zusätzlich wird noch ein weiterer Header benötigt, in dem die Signalbelegung für das Display projektabhängig festgelegt wird. Dieser Konfigurations-Header wird auch von der Bibliothek benötigt, befindet sich aber in verschiedenen Projektverzeichnissen und somit ist der Pfad dahin natürlich variabel, bzw. aus Sicht der Bibliotheksdateien unbekannt.

Für diese Fälle gibt es die Möglichkeit dem Compiler die Pfade als Parameter beim Aufruf zu übergeben. Die IDE bietet einen Dialog an, in dem man die Pfade eintragen kann und erzeugt daraus dann die erforderlichen Parameter, die an den Compiler übergeben werden.



Der Eintrag „„/“ in der obigen Abbildung steht hier für das Projektverzeichnis selber und ermöglicht es dem Compiler beispielsweise einen Header mit projektbezogenen Konfigurationseinstellungen zu finden welcher beim Übersetzen einer Bibliotheksdatei eingebunden werden muss.

Eigentlich ist diese Methode auch allgemein gegenüber längeren Pfaden bei den #include Direktiven zu bevorzugen. Beim zweiten Eintrag in der Abbildung handelt es sich beispielsweise um den Pfad zum Grundverzeichnis einiger Bibliotheken, relativ zu Projekt uC_Q_HS-ULM_Base.X

Im Projekt kann man dann die Include Direktiven wie folgt verwenden:

```
#include "uCQuick/uCQ_2013.h"
#include "LCD/LCD_lib_busy.h"
#include "GPS/gpsNMEA.h"
```

Siehe hierzu auch:

<https://microchipdeveloper.com/xc16:set-the-include-directory-path>

<https://microchipdeveloper.com/tls2101:include-directive>

6.5 LCD Bibliothek (Character Displays)

(Funktionen für Display-Anzeigen)

Die große Mehrheit an LCD Displays basiert auf dem quasi Standard Controllerchip HD44780 oder einem dazu kompatiblen. Es gibt eine Unmenge an Bibliotheken, Beispielen zur Ansteuerung, und guten Einführungen zu diesem Thema. z.B. <http://sprut.de/electronic/lcd/index.htm>

Solche „Standard“ LCDs haben ein paralleles Interface mit 8 Datenleitungen **D7..D0** und 3 Steuerleitungen **E (Enable)**, **RS (Register Select)** und **RW (Read/Write)**

Bei den 8 Datenleitungen besteht meistens auch die Möglichkeit nur 4 davon zu benutzen (*D7..D4*) und die eigentlich 8-Bit breiten Daten in zwei Zugriffen zu übertragen. Das spart Pins am Controller und natürlich auch Leitungen oder Leiterbahnen. Die Übertragung dauert dafür natürlich länger.

Wenn man auf das Display nur schreibend zugreift, dann kann man auch auf das Steuersignal **RW** verzichten indem man **RW** am Display auf GND legt.

Dabei handelt man sich allerdings den Nachteil ein, dass nach jedem Schreibzugriff eine bestimmte Zeit abgewartet werden muss, bevor der nächste Schreibzugriff erfolgen darf.

Kann man dagegen lesend auf das Display zugreifen, hat man die Möglichkeit ein „**Busy Flag**“ abzufragen, um zu sehen, ob das Display noch mit der Verarbeitung des letzten Befehls beschäftigt ist. Die Wartezeit durch das Pollen des Busy-Flags fällt meist deutlich kürzer aus.

6.5.1 Die XLCD Bibliothek der Microchip Compiler

Die PLIB der Compiler C18 und XC8 bietet Bibliotheksfunktionen zur Ansteuerung von HD44780 kompatiblen Displays an. Leider sind die Funktionen schon vorkompiliert in den Bibliotheken enthalten. Dadurch sind auch die Pins am jeweiligen Controller schon festgelegt, welche zur Ansteuerung des LCD verwendet werden müssen.

Will man andere Pins zur Ansteuerung des LCD verwenden, dann muss man unter anderem die Pinbelegung im Header „xlcd.h“ anpassen und die Library neu erstellen, da die Funktionen in der Lib. mit dem Pinning in der originalen Header Datei erzeugt wurden.

Alternativ dazu kann man auch die Sourcefiles der XLCD Bibliothek dem Projekt hinzufügen und sich das Kompilieren der Prozessor-Bibliothek sparen.

Selbst wenn die angesprochene Anpassung gelingen würde, könnten die XLCD Funktionen für die uC-Quick Plattform nicht verwendet werden, weil die Datensignale D7...D4 wie bei eigentlich allen LCD-Bibliotheken immer am selben Port und in festgelegter Reihenfolge entweder die unteren oder die oberen vier Bits belegen müssen. Ein Blick in den uC-Quick Schaltplan ...

6.5.2 LCD_lib_busy der Hochschule Ulm

LCD_lib_busy ist eigentlich gar keine Bibliothek, sondern nur Sourcecode der dem Projekt einfach hinzugefügt und mit diesem kompiliert wird.

Für jedes Signal kann ein beliebiger Pin am PIC verwendet werden, solange er die erforderlichen elektrischen Eigenschaften aufweist. Bestimmte Gruppierungen von Signalen an Ports oder Reihenfolgen sind nicht erforderlich.

Die „Bibliothek“ besteht aus der Quelldatei **LCD_lib_busy.c** und dem zugehörigen Header **LCD_lib_busy.h**, die vom Benutzer nie verändert werden.

Dazu kommt noch eine projektspezifische Konfigurationsdatei **lcd_config.h** zum Anpassen des Pinnings und Timings an das jeweilige Projekt.

Jedes Projekt hat seine eigene **lcd_config.h** Datei im Projektverzeichnis.

LCD_lib_busy.c und **LCD_lib_busy.h** befinden sich normalerweise nicht direkt im aktuellen Projektverzeichnis, da diese Dateien von mehreren Projekten gemeinsam genutzt werden.

6.5.2.1 lcd_config.h

Für diese projektspezifische Datei gibt es das Template **lcd_config_template.h** (*in .../C-LIB/HS-Ulm/LCD*), welches man sich in sein Projektverzeichnis kopieren kann. Dort muss dann natürlich der Name abgeändert („*_template*“ entfernen) und der Inhalt dem Projekt angepasst werden.

Für alle Boards die in den Laboren vorhanden sind und die ein passendes LCD haben, enthält das Template schon die passenden Pin-Definitionen. Diese können über ein #define aktiviert werden.

Zusätzlich zum Pinning muss unbedingt noch das Timing korrekt eingestellt werden. Besonders in der **Initialisierungsphase**, wenn noch kein gültiges **Busy-Flag** abgefragt werden kann, ist die Einhaltung einiger Delays extrem wichtig. So muss nach dem Anlegen der Spannungsversorgung meist mindestens 15ms gewartet werden, bis die erste Kommunikation stattfinden darf.

Danach folgen nochmals längere Wartezeiten von ~4ms. Um diese Zeiten einzuhalten wurde das Makro **LCD_DELAY_5MS()** definiert, welches dann für die 15ms einfach 3 mal aufgerufen wird und 1 mal für die 4ms. Eine längere Wartezeit ist kein Problem, es darf nur nicht kürzer sein.

Nach der Initialisierung muss nur noch das **Bus-Timing** beachtet werden. Hier müssen die Daten mindestens 1us lang anliegen, bevor mit dem Enable-Signal (*E*) die Übernahme veranlasst wird. Bei Oszillatorfrequenzen bis 4MHz ist die Mindestzeit schon automatisch durch die Befehlsverarbeitung gewährleistet. Taktet man den PIC höher, dann können über das Makro **LCD_DELAY_1US()** Nop() Befehle (*no operation*) eingefügt werden, die jeweils auch vier Oszillator-Takte verbrauchen.

Ein Blick in die Datei lcd_config_template.h:

```
#ifndef _LCD_CONFIG_H
#define _LCD_CONFIG_H

#include <xc.h>
#include "a_global_header.h"      // #define _XTAL_FREQ ...

#define LCD_TIMEOUT 100          // max nr. of busy checks ...

#define LCD_DELAY_5MS() __delay_ms(5)
#define LCD_DELAY_1US() __delay_us(1)

// uncomment the board in use or define a custom pinning ;-) -----
#define uC_QUICK
//    #define PICDEM2p_2002
//    ...

#if defined uC_QUICK
    #warning "LCD-pinning for uC-Quick board !!!"
    #define LCD_E      LATCbits.LATC1
    #define LCD_E_DIR  TRISCbits.TRISC1
    #define LCD_RW    LATCbits.LATC0
    #define LCD_RW_DIR TRISCbits.TRISCO
    #define LCD_RS    LATAbits.LATA5
    #define LCD_RS_DIR TRISAbits.TRISA5
    ...

#elif defined  PICDEM2p_2002
    ...

```

Wird, wie bei dieser Bibliothek, das Busy-Flag des Displays abgefragt, dann muss man darauf achten, dass dies nicht zu einer Endlosschleife führen kann (*z.B. bei Defekt*). Mit **LCD_TIMEOUT** kann man hier einstellen, wann z.B. ein **while(LCD_Busy()){};** abgebrochen wird.

Für die hier verwendeten Delay Makros **__delay_us(1)** und **__delay_ms(5)** muss auch das Makro **_XTAL_FREQ** irgendwo definiert sein. Da dieses meist auch noch in anderen Dateien des Projektes verwendet wird, macht man das am besten in einem zusätzlichen Header der dann überall eingebunden wird.

6.5.2.2 LCD_lib_busy (.h / .c)

Die Source Datei **LCD_lib_busy.c** enthält die Definition der LCD Funktionen und der intern genutzten Variablen.

Im Header **LCD_lib_busy.h** sind die Prototypen der Funktionen und einige Definitionen für LCD Befehle hinterlegt. Vieles was auf den ersten Blick wie eine Funktion aussieht ist, ist nur ein Makro welches auch in der Header Datei definiert ist.

Im Kopf der Header Datei befindet sich auch eine Auflistung der Funktionen mit einer kurzen Beschreibung.

```
//#####
// filename:          LCD_lib_busy.h
//##### header file LCD library for HD44780, HD47780, ST7036 ...
//#####
//
//      Author:          V.SchK
//      Company:         THU
//
//      Revision:        2.0
//      Date:            May 2019
//      Assembled using XC8 2.00+
//
//      todo      - add comments ;-
//      -
//#####
//      FUNCTIONS:
//
//-- void LCD_Init(void)
//      must be called before using other LCD functions !!!
//
//-- void LCD_Clear(void)
//
//-- void LCD_SetCursor(unsigned char row,unsigned char column)
//      set the write cursor to given row and column (both start with 0)
//
//-- void LCD_CharOut(unsigned char character)
//      writes character at the present cursor position
//
//-- void LCD_TextOut(unsigned char row, unsigned char col, unsigned char *text)
//      writes a RAM string
//      -> unsigned char string[] = {"Hello"};
//      LCD_TextOut(0,0,string);
//
//-- void LCD_ConstTextOut(unsigned char row, unsigned char col, const char *text)
//      writes a ROM string
//      -> LCD_ConstTextOut(0,0,"Hello");
//
//-- void LCD_ValueOut(unsigned char row, unsigned char col, short value)
//      writes a (signed) number at the give coordinates
//
//-- void LCD_ValueOut_00(unsigned char row, unsigned char col, short value,
//      unsigned char min_dig);
//      writes a number with leading zeros to get at least the specified digits
//      -> short number = 7;
//      LCD_ValueOut_00(0,0,number,4); // -> "0007" at the top left of the display
//
//#####
#ifndef _LCD_LIB_BUSY_H
#define _LCD_LIB_BUSY_H

#include "lcd_config.h" // pin and power configuration file
...
```

Auch das Einbinden der Konfigurationsdatei **lcd_config.h** geschieht im Header **LCD_lib_busy.h**

6.5.3 LCD_lib_busy Beispiel

Durch Modifikation des Beispiels aus Kapitel [5.3 Pin Interrupts INTx \(Drehgeber rotiert LEDs\)](#) kann man eine einfache LCD Ausgabe realisieren.

Über die im Interrupt gesetzten [Flags](#), wird eine Variable inkrementiert, bzw. dekrementiert und anschließend angezeigt.

```
void __init(void)
{
    OSCCONbits.IRCF = IRCF_4MHZ; OSCTUNEbits.PLLN = 0;// known CLK for LCD_init()

    LCD_Init(); // initialize LCD (absolute must!)
    LCD_ConstTextOut(0,0,"uC-Quick"); LCD_ConstTextOut(1,0," HS-Ulm ");

    ENC_INT_TRI = INPUT_PIN; ENC_DIR_TRI = INPUT_PIN; // encoder setup
    mENC_IR_CLR(); mENC_IR_EN();

    flags.all = 0;

    INTCONbits.GIE = 1;
}
void main(void)
{
    short row = -1;
    unsigned char poem[12][9] = {"Dunkel ", "wars ", "der Mond", "schien ",
                                "helle. ", "Als ein ", "Auto ", "blitze- ",
                                "schnelle", "langsam ", "um die ", "Ecke fhr"};

    __init();
    while(1){
        if(flags.encUp){
            if(row < 11){
                row++;
                LCD_ConstTextOut(0,0," "); // delete old value (complete row)
                LCD_ValueOut(0,1,row+1); // display row number
                LCD_TextOut(1,0,&(poem[row][0])); // display row content
            }
            flags.encUp = 0;
        }
        if(flags.encDown){
            if(row > 0){
                ...
            }
        }
    }
}
```

Damit die Bibliothek verwendet werden kann, müssen die Dateien **LCD_lib_busy.c/h** und die Konfigurationsdatei **lcd_config.h** dem Projekt hinzugefügt werden.

LCD_lib_busy.c/h sollten **schreibgeschützt und außerhalb** des Projektpfades in einem Bibliotheksverzeichnis stehen, damit nicht unsinnig und unübersichtlich viele Kopien dieser Dateien entstehen. Die Datei lcd_config.h ist projektspezifisch. Sie sollte deshalb auch in jedem Projektverzeichnis, speziell für dieses Projekt vorhanden sein.

Bevor das LCD benutzt werden kann, muss es initialisiert werden. Diese Initialisierung ist bei den Charakter Displays relativ kompliziert und erfordert die Einhaltung eines speziellen Timings. Genaue Informationen zum Timing muss man den Datenblättern der jeweiligen Displays, bzw. den darin verbauten Controllern entnehmen. Einstellungen für das Timing sind in [lcd_config.h](#) vorzunehmen. **Das Timing kann man nur einstellen, wenn man die Taktfrequenz kennt!**

Nach der Initialisierung können die Funktionen der Bibliothek wie im Beispiel verwendet werden.

Falls lcd_config.h beim Übersetzen der Bibliothek vom Compiler nicht gefunden werden kann, bitte zurück zu Kapitel 6.4 Compiler Include Directories der IDE

6.6 GLCD Bibliothek (Graphik Display)

FOTO FOTO

Einfache Graphik-Displays kann man sich wie eine primitive Matrix aus einzelnen Punkten vorstellen. Auch die Version 2018 der uC-Quick Platine enthält solch ein einfaches Display mit 84x48 Punkten, welches aus alten Nokia Mobiltelefonen vom Typ 5110 oder 3110 stammt.

Dieses Display hat ein SPI Interface und ist im Vergleich zu einem Zeichen-Display mit HD44780 kompatiblem Interface vergleichsweise einfach anzusteuern. Die Initialisierung ist Zeit-unkritisch und es gibt nur wenige Befehle. Dafür gibt es hier aber keine integrierte Tabelle für Schriftzeichen und man muss diese wie alles andere auch als Bitmusterfolgen übertragen. Die Muster müssen dafür im Programm des Mikrocontrollers angelegt sein.

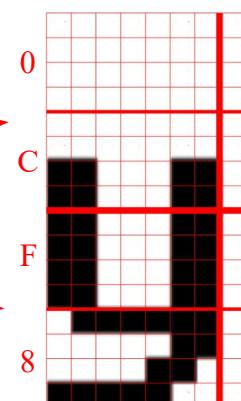
6.6.1 GLCD Zeichensätze

Die Datenübertragung zur Matrix erfolgt Byte weise und man stellt sich die 84x48 Punkte am besten als 84 Spalten und 6 Zeilen mit einer Höhe von 8 Punkten (ein Byte) vor. Damit Zeichen effektiv zum Display übertragen werden können, wurde für die Höhe der Zeichen das Äquivalent einer ganzzahligen Zeilenhöhe, also 8 Punkte (eine Zeile) und 16 Punkte (zwei Zeilen) gewählt und zwei Zeichensätze implementiert, die in den Dateien *charSet8x5.c* und *charSet16x7.c* zu finden sind.

```
const char charSet8x5[] = {
    0xFF, 0xFF, 0xFF, 0xFF, 0xFF, //0x00    black box
...
    0x3C, 0x40, 0x30, 0x40, 0x3C, //0x77    w
    0x44, 0x28, 0x10, 0x28, 0x44, //0x78    x
    0x0C, 0x50, 0x50, 0x50, 0x3C, //0x79    y
    0x44, 0x64, 0x54, 0x4C, 0x44, //0x7A    z
    0x00, 0x04, 0x36, 0x41, 0x00, //0x7B    {
    0x00, 0x00, 0x7F, 0x00, 0x00, //0x7C    |
    0x00, 0x41, 0x36, 0x04, 0x00, //0x7D    }
    0x08, 0x08, 0x2A, 0x1C, 0x08, //0x7E    ->
    0x08, 0x1C, 0x2A, 0x08, 0x08 //0x7F    ←
};

const char charSet16x7[] = {
    0x00, 0x3E, 0x51, 0x45, 0x43, 0x3E, 0x00, //0x00    black box
    0x00, 0x3E, 0x51, 0x45, 0x43, 0x3E, 0x00,
...
    0xC0, 0xC0, 0x00, 0x00, 0x00, 0xC0, 0xC0, //0x79    y
    0x8F, 0x9F, 0x90, 0x90, 0xD0, 0x7F, 0x3F,
    0xC0, 0xC0, 0x40, 0x40, 0xC0, 0xC0, 0x40, //0x7A    z
    0x18, 0x1C, 0x16, 0x13, 0x11, 0x18, 0x18,
    0x80, 0x80, 0xF0, 0x78, 0x08, 0x08, 0x00, //0x7B    {
    0x00, 0x00, 0x0F, 0x1F, 0x10, 0x10, 0x00,
    0x00, 0x00, 0x78, 0x78, 0x00, 0x00, 0x00, //0x7C    |
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x08, 0x08, 0x78, 0xF0, 0x80, 0x80, 0x00, //0x7D    }
    0x10, 0x10, 0x1F, 0x0F, 0x00, 0x00, 0x00,
    0x20, 0x30, 0x10, 0x30, 0x20, 0x30, 0x10, //0x7E    ~
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x01, 0x71, 0x09, 0x05, 0x03, 0x00, //0x7F    DEL
    0x00, 0x7F, 0x09, 0x09, 0x09, 0x01, 0x00
};
```

Die Zeichensätze enthalten nur die Zeichen selber. Die Zwischenräume (aus Leerspalten) werden in den Zeichenausgabefunktionen eingefügt.



6.6.2 GLCDnokia (.h / .c)

Die Bibliothek für die Ansteuerung des Displays befindet sich mehr oder weniger noch im Aufbau. Damit das Display der uCQ-Boards der neueren Version 2018 analog zum Zeichendisplay früherer Versionen angesteuert werden kann, wurde begonnen die meist benutzten Funktionen zu implementieren. Die Funktionsnamen entsprechen denen der Zeichen-Display Bibliothek, mit einem führenden „G“ (für Graphik). Zeichenausgabefunktionen für doppelte Zeichenhöhe enthalten zusätzlich die Ziffer „2“ im Funktionsnamen.

```
/*
 * File:    GLCDnokia.h
 * Author:  vschilli
 *
 * Created on 27. Juli 2017, 16:30
 */

#ifndef GLCDNOKIA_H
#define GLCDNOKIA_H

#include "glcdNokia_config.h"

extern const char charSet8x5[];
extern const char charSet16x7[];
...

extern void GLCD_Init(void);
extern void GLCD_ClearRow(unsigned char row);
extern void GLCD_Clear2Row(unsigned char row);
extern void GLCD_Clear(void);
extern void GLCD_Write(unsigned char data, unsigned char dc);
extern void GLCD_WriteChar(unsigned char c);

extern char GLCD_CharOut(unsigned char row, unsigned char column, unsigned char c);
extern void GLCD_TextOut(unsigned char row, unsigned char col, unsigned char *text);

extern char GLCD_Char2Out(unsigned char row, unsigned char column, unsigned char c);
extern void GLCD_Text2Out(unsigned char row, unsigned char col, unsigned char *text);
extern void GLCD_Value2Out_00(unsigned char r, ..., unsigned char min_dig);

#define     GLCD_Value2Out(r,c,v) GLCD_Value2Out_00(r,c,v,1)
#define     GLCD_SetCursor(row, column) GLCD_Write(DSPL_X | column, CMD); \
                                GLCD_Write(DSPL_Y | row, CMD);

#endif /* GLCDNOKIA_H */
```

6.6.3 glcdNokia_config.h

Wie die Bibliothek für Zeichen Displays, gibt es auch für die GLCD Bibliothek eine Konfigurationsdatei, in der das Pinning an das aktuelle Projekt angepasst werden kann.

```
#ifndef GLCDNOKIA_CNGF_H
#define GLCDNOKIA_CNGF_H
...
#define GLCD_DC      LATAbits.LATA5
#define GLCD_nRES   LATCbits.LATC0
#define GLCD_nCS    LATCbits.LATC1
#define GLCD_CLK    LATCbits.LATC3
#define GLCD_DATA   LATCbits.LATC5
#define GLCD_DC_TRI   TRISAbits.TRISA5
#define GLCD_nRES_TRI TRISCbits.TRISC0
#define GLCD_CS_TRI   TRISCbits.TRISC1
#define GLCD_CLK_TRI  TRISCbits.TRISC3
#define GLCD_DATA_TRI TRISCbits.TRISC5

#define GLCD_DLY()
//Nop();
#endif /* GLCDNOKIA_CNGF_H */
```

6.7 General Software Library

Die Beschreibung der verfügbaren Funktionen findet man im User-Guide des Compilers...

6.7.1 Delay Funktionen

6.7.2 Debug Funktionen

6.7.3 Reset Funktionen

6.7.4 Datentyp Umwandlungen

6.7.5 Speicher und Zeichenketten Verarbeitung

6.7.6 Ausgabe Funktionen für Zeichenketten

6.7.7 Funktionen zur Zeichen Klassifizierung

6.8 Peripheral Library (PLIB)

PLIB stammt vom älteren Michrochip C18 Compiler und enthält Funktionen zur Verwendung der im Controller integrierten Peripheriemodule. PLIB war beim XC8 Compiler nur bis zur Version v1.34 voll integriert. Bei späteren Versionen konnte die Bibliothek separat installiert werden. Der Download sollte unten auf der Seite des XC8 Compilers noch verfügbar sein.

Leider funktioniert die Bibliothek, bedingt durch Änderungen der Prozessor-Header-Files des XC8 Compilers nicht mehr und muss vor Verwendung neu erstellt werden!

Der Sourcecode vieler PLIB Funktionen für PIC18FxxK22 Controller, ist in den uCQ Demoprojekten auf der [Hochschulseite](#) enthalten und auf dieser Seite auch separat zum Download verfügbar. Die Dateien *plib18fxxk22.c* und *plib18fxxk22.h* können älteren Projekten zugefügt werden um eine bessere Kompatibilität mit den aktuellen Versionen des XC8 Compilers zu erhalten.

Die PLIB Dokumentation findet man als [PIC18F Peripheral Library Help Document.pdf](#).

6.9 Software Peripheral Library

Nicht vorhanden in neuen ...

Bla ...

6.10 Microchip Code Creator Beispiel

Bla ...

7 Puls-Weiten-Modulation (*fast analog*)

Digitale Ausgänge bieten zunächst nur die Möglichkeit etwas entweder „Aus“ oder eben „Ein“ zu schalten. Durch sehr schnelles Umschalten, kann man aber auch quasi analoge Effekte erzielen und den Eindruck viel feinerer Abstufungen als nur Ein/Aus erzeugen. Das ist immer dann der Fall, wenn Zeitkonstanten eines Systems wesentlich größer sind, als die Periode der Ein/Aus Schaltvorgänge. So kann man beispielsweise Drehzahlen von Motoren, den Helligkeitseindruck einer LED, oder Heizungen sehr einfach fein abgestuft einstellen und auch Töne erzeugen.

7.1 LED - Ein/Aus oder doch unterschiedlich hell ?

Ohne zusätzliche Hardware kann man Puls-Weiten-Modulation an LEDs testen. Auf der uCQ Platine sind 4 LEDs vorhanden die sich für unterschiedliche Implementierungen eignen. Im folgenden werden verschiedene Beispiele vorgestellt eine PWM zu realisieren. Die Variation der Helligkeit, wird in Beispielen über den Encoder wie in [5.3 Pin Interrupts INTx \(Drehgeber rotiert LEDs\)](#) gesteuert. Der Code für die Interrupt Service Routine kann von diesem Beispiel übernommen werden.

7.1.1 PWM durch manuelles Zählen und Vergleichen

Die primitivste Art ein PWM Signal zu erzeugen beruht darauf, bis zu einem bestimmten Wert zu zählen und dabei den momentanen Zählerstand mit einem vorgegebenen Wert zu vergleichen.

Man kann beispielsweise immer wieder bis 100 zählen und dabei am Anfang eine LED einschalten. Wenn man beim gewünschten Tastgrad (*engl. Duty Cycle*) angelangt ist, schaltet man die LED wieder aus. Auf diese Weise bekommt man den Tastgrad direkt in Prozent. Spielen Periodendauer und Schrittweite der Tastgrade nur eine untergeordnete Rolle, dann lässt man den Zähler einfach wie im folgenden Beispiel überlaufen.

Weil der Helligkeitseindruck nicht linear mit dem Tastgrad verläuft und 2^8 Helligkeitsstufen über den Drehgeber ohne zusätzliche Programmlogik etwas mühsam einzustellen wären, wird im folgenden Beispiel der Vergleichswert durch ein Schiebesystem realisiert.

```
void PWMcounter(void)
{
    unsigned char counter = 0, compare = 0b00011111; // (31)
//----- __init()
    mALL_LED_OUTPUT(); // LED pins as outputs

    ENC_INT_TRI = INPUT_PIN; ENC_INT_ANS = DIGITAL_IN; // encoder setup
    ENC_DIR_TRI = INPUT_PIN; ENC_DIR_ANS = DIGITAL_IN;
    mENC_IR_CLR(); mENC_IR_EN();

    flags.all = 0;

    INTCONbits.GIE = 1;
//----- main()
    while(1){
        if(flags.encUp){
            compare = (compare << 1)+1; // 0 → 1 → 3 → 7 → 15 → 31 → 63 → 127 → 255
            flags.encUp = 0;
        }
        if(flags.encDown){
            compare = compare >> 1; // 255 → 127 → 63 → 31 → 15 → 7 → 3 → 1 → 0
            flags.encDown = 0;
        }
        if(++counter == compare) mALL_LED_OFF();
        else if(counter == 0) mALL_LED_ON(); // led on at counter overflow (0)
    }
}
```

Dieses erste Beispiel mittels manuellem Zählen mag für sehr langsame Systeme wie z.B Heizungen durchaus funktionell sein. Sobald die Hauptschleife des Programms aber etwas umfangreicher wird, kann schon die Helligkeit einer LED nicht mehr vernünftig eingestellt werden.

Angenommen der Durchlauf würde eine tausendstel Sekunde dauern, dann wäre die Periodendauer der PWM $\sim \frac{1}{4}$ Sekunde ($255/1000$). Der Effekt wäre dann eine mit einer Frequenz von 4Hz blinkende LED mit veränderbarer Einschaltzeit. (*Test durch Einfügen von `_delay_ms(1);`*)

Man könnte jetzt das Zählen, Vergleichen und Ein/Aus-schalten in einen Timer-Interrupt verlegen, um eine vom restlichen Programmablauf unabhängiges Verhalten zu erzielen. Meistens gibt es aber elegantere Lösungen, die auch in den nächsten Kapiteln behandelt werden.

7.1.2 PWM mit CCP-Modul im PWM Modus (10 Bit)

Mikrocontroller haben natürlich spezielle Module eingebaut die solche häufig auftretenden Aufgabenstellungen wie eine PWM übernehmen. Einmal konfiguriert, erfolgt die Ausgabe des Signals autark vom Hauptprogramm, das gegebenenfalls nur noch die Parameter verändert.

Beim PIC18 nennt sich dieses spezielle Modul „Capture-Compare-PWM“ (CCP). Die Namensgebung beschreibt die drei Modi, in denen es betrieben werden kann. In allen drei Modi ist noch ein zusätzliches Timer-Modul erforderlich, mit dem es verknüpft werden muss. Wie immer schaut man sich am besten die Abbildung im Datasheet an um einen ersten Überblick des entsprechenden Modus zu bekommen.

Die PIC18FxxK22 verfügen über fünf etwas unterschiedlich ausgebauten PWM-Module. Es gibt Module mit einem Ausgang, mit zwei Ausgängen zum Ansteuern von Halbbrücken und mit vier Ausgängen zum Ansteuern von Vollbrücken. Die Versionen mit mehreren Ausgängen werden „Enhanced PWM“ Module genannt. Diese kann man aber auch wie die einfachen Module mit nur einem Ausgang konfigurieren.

FIGURE 14-4: SIMPLIFIED PWM BLOCK DIAGRAM

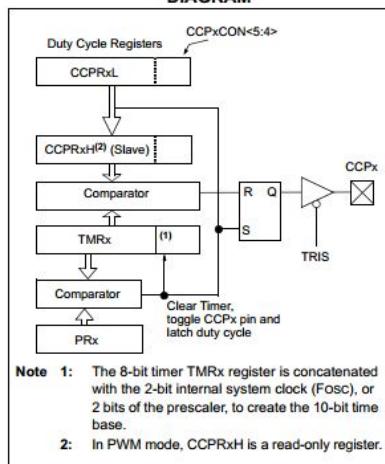
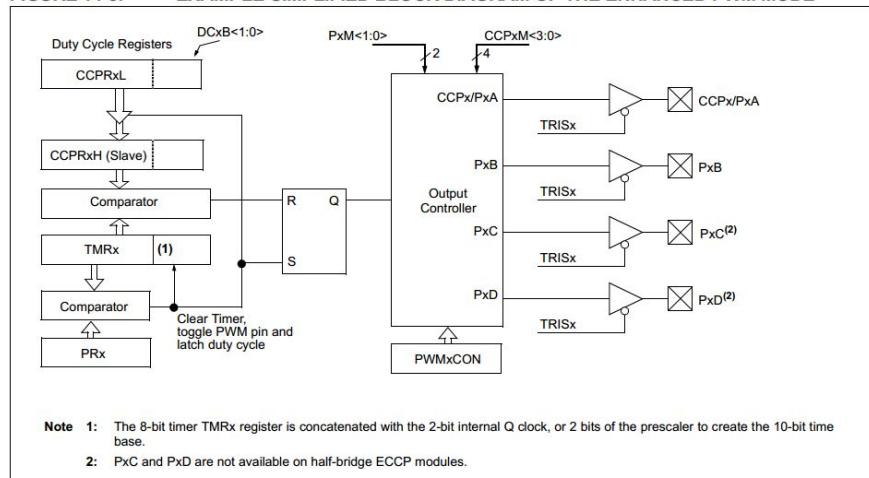


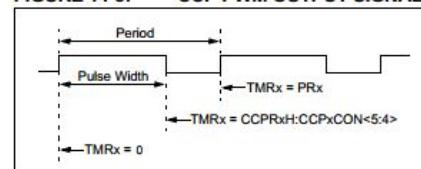
FIGURE 14-5: EXAMPLE SIMPLIFIED BLOCK DIAGRAM OF THE ENHANCED PWM MODE



Alle Zähler und Vergleichswerte welche für die PWM gebraucht werden sind in den Modulen hardwaremäßig integriert.

Der Zählerwert wird durch den Timer TMRx realisiert, welcher auch mit einer fest voreingestellten Frequenz „zählt“.

FIGURE 14-3: CCP PWM OUTPUT SIGNAL



Die Vergleichswerte für Pulsweite und Periode werden mit den beiden Komparatoren verglichen, wobei der Vergleich mit dem Register PRx die Periode des PWM-Signals ergibt und der Vergleich vom TMRx mit dem 8-Bit Registers CCPRxH und zwei zusätzlichen Bits im CCPxCON mit dem internen Systemtakt die Pulsweite. (*Figure 14-3 und Note 1: FIGURE 14.-4*)

Weil auf der uC-Quick Hardware am CCP2-Output eine LED angeschlossen ist, wird im Folgenden Beispiel das „Enhanced PWM Half-Bridge“ Modul ECCP2 mit Timer 2 verwendet.

Die generelle Vorgehensweise findet sich natürlich im Datasheet in den Kapiteln **SETUP FOR PWM OPERATION** bzw. **SETUP FOR ECCP PWM OPERATION**. Für die einfache Helligkeitssteuerung einer LED reicht es die folgenden Konfigurationen vorzunehmen.

- PWM-Mode einstellen (*CCPxM Bits im CcxCON Register*)
- Timer auswählen (*CCPTMRSx Register*)
- Timer-Geschwindigkeit konfigurieren (*TxCON*)
- Periode einstellen (*PRx*)
- Duty Cycle einstellen (*CCPRLx*)
- Timer starten (*TxCON*)
- PWM-Output konfigurieren (*TRIS Register, evtl. Configuration Bits*)

```
void PWMccpPWM(void)
{
//----- __init__()
    OSCCONbits.IRCF = IRCF_1MHZ;

    ENC_INT_TRI = INPUT_PIN;   ENC_INT_ANS = DIGITAL_IN; // encoder setup
    ENC_DIR_TRI = INPUT_PIN;   ENC_DIR_ANS = DIGITAL_IN;
    mENC_IR_CLR(); mENC_IR_EN();

//    LED_2_TRI = INPUT_PIN;      // LED2 is connected to CCP2 (RB3) → disable for setup
#warning "make sure #pragma config CCP2MX = PORTB3 !!!"

    CCP2CONbits.CCP2M = 0b1110;           // PWM mode PxA active low
    CCP2CONbits.P2M0 = CCP2CONbits.P2M1 = 0; // single output
    CCPTMRS0bits.C2TSEL = 0;             // CCP2-TMR2
    CCPR2L = 31;                        // duty cycle for pwm 31/256
    PR2 = 255;

    T2CONbits.T2CKPS = 0;                // no prescaler
//    T2CONbits.T2OUTPS = 0;              // no postscaler
    T2CONbits.TMR2ON = 1;                // -> ~1kHz (IRCF_1MHZ)

    LED_2_TRI = OUTPUT_PIN;             // LED2 is connected to CCP2 (RB3)

    flags.all = 0;

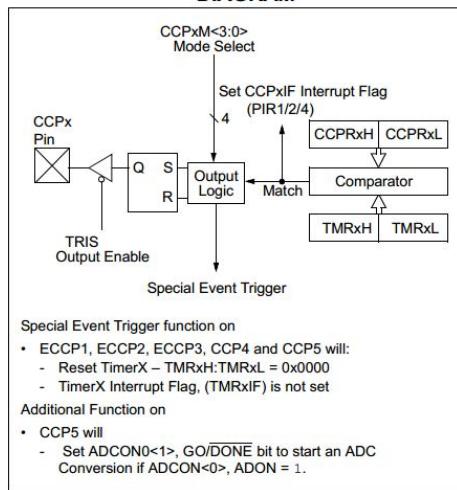
    INTCONbits.GIE = 1;

//----- main()
    while(1){
        if(flags.encUp){
            CCPR2L = (CCPR2L << 1)+1; // 0 → 1 → 3 → 7 → 15 → 31 → 63 → 127 → 255
            flags.encUp = 0;
        }
        if(flags.encDown){
            CCPR2L = CCPR2L >> 1;     // 255 → 127 → 63 → 31 → 15 → 7 → 3 → 1 → 0
            flags.encDown = 0;
        }
    }
}
```

7.1.3 PWM mit CCP-Modul im Compare Modus (16 Bit)

Im Compare Modus wird normalerweise nur eine einzige Aktion (*ein einmaliger Signalwechsel*) ausgeführt. Es ist aber auch möglich ein periodisches Signal zu erhalten, wenn man den Überlauf des auch hier beteiligten Timers nutzt, um den Prozess wieder neu zu starten. Doch zunächst wieder der Blick ins Datenblatt ...

FIGURE 14-2: COMPARE MODE OPERATION BLOCK DIAGRAM



Im Blockschaltbild für den Compare-Mode ist jetzt nur ein Komparator enthalten. Nur der Umschaltzeitpunkt ist über den Vergleich der TMRx mit den CCPRxH/L Registern einstellbar.

Die Periode des Signals ergibt sich aus einem vollen Durchlauf von TMRx. Die 16-Bit Breite der Register ergeben bei kleineren Taktraten unter Umständen sehr lange Perioden!

Ein weiteres Problem kann auftreten, wenn sehr kurze Pulse eingestellt werden sollen. Bis der Compare-Mode neu gestartet wird hat der Timer schon einige Zyklen gezählt und der Vergleichswert muss dementsprechend angepasst werden.

Im folgenden Beispiel werden Funktionen der [Plib](#) verwendet und wieder wegen der Einfachheit nur das Register CCPRxH für den Vergleichswert angepasst. Der Restart des Compare müsste auch in einem High-Prio-Interrupt ausgeführt werden.

```
void PWMccpCompare(void)
{
//----- init()
OSCCONbits.IRCF = IRCF_8MHZ;
OSCTUNEbits.PLLEN = 1; // ->32MHz

ENC_INT_TRI = INPUT_PIN; ENC_INT_ANS = DIGITAL_IN; // encoder setup
ENC_DIR_TRI = INPUT_PIN; ENC_DIR_ANS = DIGITAL_IN;
mENC_IR_CLR(); mENC_IR_EN();

OpenECompare2(COM_INT_OFF & ECOM_HI_MATCH & ECCP_2_SEL_TMR12, (31<<8)+10);

OpenTimer1(TIMER_INT_OFF & T5_16BIT_RW & T1_SOURCE_FOSC_4 & //~60Hz (32MHz)
           T1_PS_1_2 & T1_OSC1EN_OFF & T1_SYNC_EXT_OFF,
           TIMER_GATE_OFF);

LED_2_TRI = OUTPUT_PIN; // LED2 is connected to CCP2 (RB3)

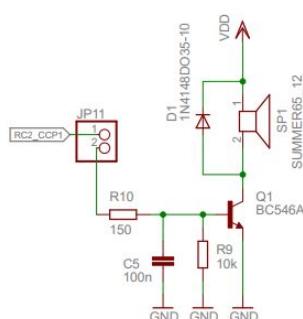
flags.all = 0;

INTCONbits.GIE = 1;

//----- main()
while(1){
    if(PIR1bits.TMR1IF){ // should be done in High-ISR of course ;-)
        if (flags.encUp) {
            CCPR2H = (CCPR2H << 1) + 1; // 0->1->3->7->15->31->63->127->255
            flags.encUp = 0;
        }
        if (flags.encDown) {
            CCPR2H = CCPR2H >> 1; // 255->127->63->31->15->7->3->1->0
            flags.encDown = 0;
        }
        CCP2CON = ECOM_HI_MATCH; // restart compare
        PIR1bits.TMR1IF = 0;
    }
}
}
```

7.2 Ton am Signalgeber generieren über PWM

Ein weitere Möglichkeit zur Nutzung einer PWM Signals ist die Erzeugung von Tönen über einen Signalgeber (Lautsprecher). Ein solcher ist am Ausgang des CCP1 Moduls angeschlossen.



Im Gegensatz zur Helligkeitsregelung, bei der die Pulsdauer verändert wurde, wird hier die Periode verändert, um unterschiedliche Töne zu erhalten.

Ansonsten ist das Beispiel sehr ähnlich. Es wurde lediglich noch um die Ausgabe der eingestellten Frequenz auf dem Display ergänzt. Diese ist natürlich *optional* und kann auch weg gelassen werden.

```
void Sound_PWM(void){    //PWM
    unsigned short frequency = 300;

//----- __init()
OSCCONbits.IRCF = IRCF_4MHZ;

LCD_Init();
LCD_ConstTextOut(0,0,"uC-Quick");
LCD_ConstTextOut(1,0," Sound ");

ENC_INT_TRI = INPUT_PIN;      // encoder setup
ENC_DIR_TRI = INPUT_PIN;
mENC_IR_CLR(); mENC_IR_EN();

CCP1CONbits.CCP1M = 0b1100;          // PWM mode
CCP1CONbits.P1M = 0;                // single
CCPTMRS0bits.C1TSEL = 0;           // CCP1-TMR2
OpenTimer2(TIMER_INT_OFF & T2_PS_1_16 & T2_POST_1_1); // 1MHz:16
PR2 = 62500 / frequency;           // duty cycle 50%
CCPR1L = PR2 >> 1;
SPEAKER_TRI = OUTPUT_PIN;

flags.all = 0;

INTCONbits.GIE = 1;
//----- main()
while(1){
    if(flags.encUp){
        if(frequency < 800) {frequency = frequency + 10;}
        else {frequency = frequency + 100;}
        if(frequency > 15600){frequency = 15600;}
        PR2 = (62500 / frequency) -1;           // 1MHz : 16
        CCPR1L = PR2 >> 1;
        LCD_ConstTextOut(1,0,"     Hz ");       // wipe out old value
        LCD_ValueOut(1,1,frequency);
        flags.encUp = 0;
    }
    if(flags.encDown){
        if(frequency < 800) {frequency = frequency - 10;}
        else {frequency = frequency - 100;}
        if(frequency < 250) {frequency = 250;}
        PR2 = (62500 / frequency)-1;           // 1MHz : 16
        CCPR1L = PR2 >> 1;
        LCD_ConstTextOut(1,0,"     Hz ");       // wipe out old value
        LCD_ValueOut(1,1,frequency);
        flags.encDown = 0;
    }
}
}
```

Verständnisfrage: Warum ändert sich der Ton bei hohen Frequenzen nicht bei jedem Schritt ?

7.3 Mehrere PWM Signale über einen Timer (special Event)

Benötigt man mehrere unabhängige PWM Signale, oder mehr Signale als PWM Module vorhanden sind, dann kann man die Signale unter Umständen auch beliebig viele, mit unterschiedlichen Perioden und Tastverhältnissen, mit nur einem Timer erzeugen.

Wie beim aller ersten Beispiel wird dabei das Zählprinzip angewendet. Das Zählen erfolgt allerdings in einem Timer-**Interrupt**, der beispielsweise alle Millisekunde ausgelöst wird. Für Jedes Signal kann man dann unabhängige Perioden- und Umschalt-Werte, sowie Zählvariablen verwenden und bei erfolgreichen Vergleichen beliebige Pins schalten.

```
if (MS_TMR_IR) {  
    if (++counter_1 == DUTYCYL_1) {  
        LED_1 = LED_OFF;  
    }  
    else if(counter_1 == PEROIDE_1) {  
        LED_1 = LED_ON;  
        counter_1 = 0;  
    }  
    ... // other PWMs  
    CLR_MS_TMR_IR();  
}
```

Nachteil:

Damit dies einigermaßen sinnvoll realisiert werden kann, sollte die Periodendauer der PWM Signale nicht übermäßig kurz sein.

Vorteil:

Die Wahl der Pins für die Ausgabe der Signale unterliegt nicht der Einschränkung durch die Belegung über die CCP Module.

7.4 PWM Vertiefungsübungen

Als Selbstkontrolle, ob die verschiedenen Möglichkeiten zur Erzeugung von PWM-Signalen auch verstanden wurden, kann man sich an den folgenden Aufgaben austoben.

7.4.1 PWM Übung 1: Helligkeit von LED_4 einstellen (Port B5)

Auch die LED_4 des uC-Quick Boards ist an einem Pin angeschlossen der als PWM-Ausgang konfiguriert werden kann. Als kleine Vertiefung kann man das Beispiel auf diese LED ändern.

Achtung; Auch hier muss man die Configuration-Bits beachten!

7.4.2 PWM Übung 2: Tonhöhe (Periode) und Lautstärke (Pulsdauer)

Abhängig davon, ob der Encoder beim Drehen gedrückt ist, Tonhöhe oder Lautstärke ändern.

7.4.3 PWM Übung 3: Automatischer Durchlauf

LED von dunkel nach hell, oder Ton von tief nach hoch, wiederholend Timer gesteuert ändern.

7.4.4 PWM Übung 4: Vier Signale an den vier LEDs (ein Timer)

4 LEDs mit unterschiedlichen Helligkeiten und/oder unterschiedlich schnell blinken lassen.
z.B: 10%, 20%, 40%, 80% Helligkeit, oder 10Hz, 15Hz, 20Hz, 25Hz Flimmern,
oder 10Hz mit 10% Helligkeit, 15Hz mit 20% Helligkeit, ...

8 Analoge Signale

8.1 Analoge Signale erfassen (ADC)

17.0 ANALOG-TO-DIGITAL CONVERTER (ADC) MODULE

The Analog-to-Digital Converter (ADC) allows conversion of an analog input signal to a 10-bit binary representation of that signal. This device uses analog inputs, which are multiplexed into a single sample and hold circuit. The output of the sample and hold is connected to the input of the converter. The converter generates a 10-bit binary result via successive approximation and stores the conversion result into the ADC result registers (ADRESL and ADRESH).

In der realen Welt gibt es viele analoge Signale, die erst in eine digitale Form umgewandelt werden müssen, um in einem digitalen System wie einem µC oder auch PC verarbeitet werden zu können. Zu diesem Zweck sind in den meisten µC Analog-Digital-Converter integriert.

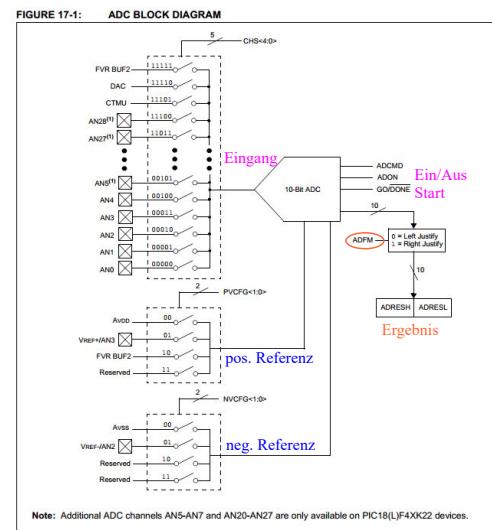
Es gibt verschiedene ADC Wandlertypen, die nach ganz unterschiedlichen Verfahren arbeiten. In den einfacheren Controllern sind meist Module enthalten, die nach dem Prinzip der sukzessiven Approximation arbeiten.

8.1.1 10-Bit ADC im PIC18FxxK22

Der im PIC18FxxK22 integrierte ADC ist ein 10-Bit Wandler. Der ADC kann mit verschiedenen Eingängen verbunden werden, es kann aber nur jeweils ein Signal zur gleichen Zeit gewandelt werden. Sollen mehrere Signale gewandelt werden, dann muss das nacheinander durch Umschalten des Eingangs erfolgen.

Für die Wandlung werden Referenzspannungen benötigt, die festlegen, für welche analogen Spannungen die min. und max. digitalen Werte gelten sollen. Wie bei den analogen Eingängen, gibt es auch hier mehrere Einstellmöglichkeiten.

Das digitale 10-Bit Resultat einer Wandlung passt nicht in ein 8-Bit Register. Hierfür werden zwei Register benötigt. Die Aufteilung der 10 Bit auf die beiden 8-Bit Register kann je nach Anwendung auf zwei verschiedene Arten erfolgen.



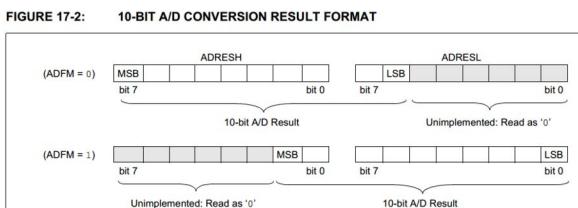
8.1.1.1 ADC Channel Auswahl – Start einer Messung

Die Auswahl des zu wandelnden Eingangssignals erfolgt im Register **ADCON0**, welches auch für den manuellen Start einer Messung und die Aktivierung des ADC Moduls zuständig ist.

8.1.1.2 ADC Referenz-Spannungen

Die Konfiguration der Referenzspannungen des ADC geschieht über das Register **ADCON1** zuständig. Über jeweils zwei Bits können hier interne, sowie externe Spannungs-Signale als Referenzen für die minimalen und maximalen digitalen Werte eingestellt werden.

8.1.1.3 ADC Result und Format

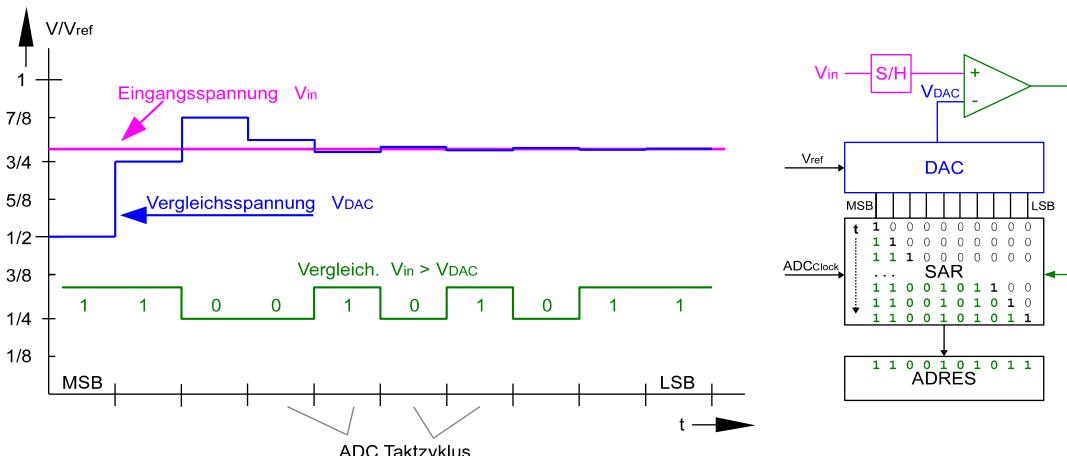


Das Ergebnis einer AD-Wandlung steht im **ADRES** Registerpaar ADRESH und ADRESL. Die Aufteilung der Bits erfolgt abhängig vom Wert des Format-Bits **ADCON2bits.ADFM**. Das am besten geeignete Format ist abhängig vom Verwendungszweck des Wandlungsresultats.

8.1.2 SAR Wandler Prinzip (Sukzessive Approximation Register)

Das sukzessive Approximationsverfahren ist ein Verfahren, in dem der Wert eines Eingangssignals durch schrittweise Annäherung eines Vergleichswertes bestimmt wird. Dieser Vergleichswert kann über einen Digital-Analog-Converter bereitgestellt werden. Ein SAR ADC besteht also aus einem DAC, einem Register zum Ansteuern des DAC und einem Komparator (*Vergleicher*).

Der Komparator kann bei einem Vergleich nur angeben, welches Signal das größere ist. Man erhält also bei einem Vergleich nur die Information von einem Bit. Durch eine schrittweise Verfeinerung des Vergleichswertes bekommt man aber eine Auflösung des Vergleichssignals und somit des Messwertes von $2^{\text{Anzahl der Vergleiche}}$. Bei den 10-bit Auflösung der im PIC18FxxK22 integrierten Module folgt daraus, dass 10 Vergleiche durchgeführt werden.



Der digitale Vergleichswert, zum Ansteuern des DAC, kommt aus dem SAR und hat beim Start der Wandlung lediglich eine „1“ im MSB (*most significant bit*), alle anderen Bits sind zunächst „0“.

Bei einem Wandler mit unendlicher Auflösung entspräche dies genau der Hälfte des Maximalwertes einer nicht vorzeichenbehafteten binären Zahl.

Die vom DAC erzeugte Spannung V_{DAC} wird so auf die Mitte des Bereiches zwischen den Referenzspannungen des DAC eingestellt ($\frac{1}{2}(V_{ref+} + V_{ref-})$).

Abhängig vom Ergebnis des Vergleichs der Eingangsspannung V_{in} mit V_{DAC} , wird das aktuell zu prüfende Bit im SAR entweder „0“ oder „1“ gesetzt. Im Anschluss wird das nächst niederwertigere Bit im SAR auf „1“ gesetzt und erneut verglichen. Das Ganze wird so lange weiter geführt, bis alle Bits bestimmt sind und das Ergebnis in das ADRES (*Analog Digital Result*) Register übernommen werden kann.

8.1.2.1 Analog Digital Conversion Clock

Die im Kapitel zum SAR Wandler Prinzip angesprochenen Vergleiche und die Steuerung des SAR benötigen eine bestimmte Mindestzeit. Wird diese unterschritten, dann ist die Richtigkeit des Ergebnisses der Wandlung nicht mehr gewährleistet.

Deshalb kann (*muss*) der ADC Takt des PIC18FxxK22, in Abhängigkeit vom Prozessortakt, auf diese Mindestzeit (*hier $\geq 1\mu s$*) eingestellt werden.

TABLE 17-1: ADC CLOCK PERIOD (TAD) Vs. DEVICE OPERATING FREQUENCIES

ADC Clock Period (TAD)		Device Frequency (Fosc)			
ADC Clock Source	ADCS<2:0>	64 MHz	16 MHz	4 MHz	1 MHz
Fosc/2	000	31.25 ns ⁽²⁾	125 ns ⁽²⁾	500 ns ⁽²⁾	2.0 μs
Fosc/4	100	62.5 ns ⁽²⁾	250 ns ⁽²⁾	1.0 μs	4.0 μs ⁽³⁾
Fosc/8	001	400 ns ⁽²⁾	500 ns ⁽²⁾	2.0 μs	8.0 μs ⁽³⁾
Fosc/16	101	250 ns ⁽²⁾	1.0 μs	4.0 μs ⁽³⁾	16.0 μs ⁽³⁾
Fosc/32	010	500 ns ⁽²⁾	2.0 μs	8.0 μs ⁽³⁾	32.0 μs ⁽³⁾
Fosc/64	110	1.0 μs	4.0 μs ⁽³⁾	16.0 μs ⁽³⁾	64.0 μs ⁽³⁾
FRC	x11	1-4 μs ^(1,4)			

Legend: Shaded cells are outside of recommended range.

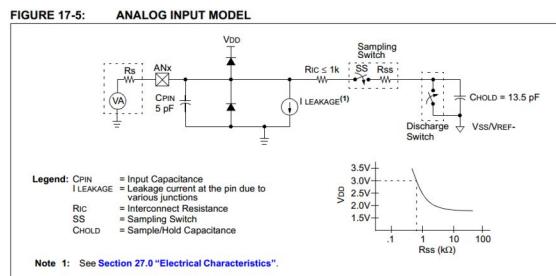
Die Einstellung der ADC Taktfrequenz erfolgt im Register ADCON2.

Sinnvolle Werte kann man der Tabelle im Data-Sheet entnehmen.

8.1.3 Acquisition Requirements / Automatic Acquisition Time

Im Blockschaltbild des weiter oben besprochenen SAR Wandlers, blieb bisher ein kleines Element unbeachtet, welches mit **S/H** beschriftet war. Das **S/H** steht hier für **Sample and Hold** und beschreibt die Art des ADC Eingangs.

Um Störungen durch Änderung der Spannung am Eingangspin zu vermeiden, wird direkt bei der Messung **nicht** das Signal am Eingangspin gewandelt. Das Eingangssignal wird vielmehr vor der Wandlung dazu verwendet, einen Kondensator aufzuladen (*sample*), und dann abgetrennt. Der Kondensator hält dann während der Wandlung den erworbenen Spannungslevel konstant (*hold*).



Im Diagramm des Analog Input Modells kann man relativ einfach die Funktionsweise des **Sample and Hold** Schaltkreises erkennen.

Vor dem Start der Wandlung ist der **Sampling-Switch** SS geschlossen und die Spannung am Kondensator folgt der Spannung am verbundenen Eingangspin. Beim Start der Wandlung wird SS dann geöffnet.

Der **Discharge-Switch** DS ist dabei die ganze Zeit geöffnet. Er wird erst nach erfolgter Wandlung geschlossen um C_{HOLD} zu entladen.

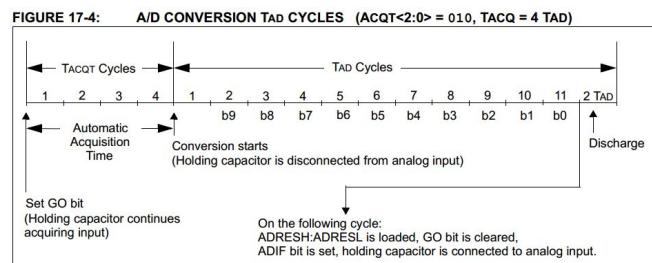
Damit C_{HOLD} vor einer Messung auf den Wert des Spannungssignals am Eingangspin aufgeladen werden kann, muss er lange genug mit diesem verbunden sein. Dies ist vor allem beim Umschalten des Eingangspins und sofortigem Start einer Messung oder dem sofortigen Start einer neuen Messung nach dem Entladen durch den Discharge Switch nicht gewährleistet.

Deshalb muss man bestimmte Wartezeiten berechnen die vor dem Start der Wandlung eingehalten werden müssen. Die Ladezeit von C_{HOLD} hängt direkt von den Werten der Bauteile ab, die im Analog Input Modell enthalten sind. Zu den Ladezeiten von C_{HOLD} kommen noch andere Effekte hinzu, die im **Data-Sheet** Kapitel **A/D Acquisition Requirements** beschrieben werden.

Im Berechnungsbeispiel zur Acquisition Time, das im Data-Sheet enthalten ist, sind schon sehr ungünstige Bedingungen berücksichtigt. Leider sind dabei einige Parameter auch nicht weiter im Data-Sheet aufgeführt (z.B. T_{AMP} (Amplifier Settling Time)) und erschienen zudem im Vergleich mit anderen PICs ähnlichen Alters recht hoch ($T_{AMP} = 5\mu s$). Das Ergebnis der (*worst case?*) Beispielrechnung im Data-Sheet ist $T_{ACQT} = 7,45\mu s$. Das heißt, mit dieser Wartezeit man kann sicher sein, dass die Wandlung korrekt ausgeführt wird.

Damit man keine Warteschleife in der Software programmieren muss, bietet das ADC Modul die Möglichkeit einer automatischen Verzögerung, die sich **Automatic Acquisition Time** nennt. Sie ist abhängig von der **AD Clock Periode** T_{AD} man kann sie wie diese im Register ADCON2 als Vielfache 2, 4, 6, 8, 12, 16 oder 20 von T_{AD} einstellen.

Nach dem Setzen des Startbits für den ADC wird dann automatisch die eingestellte Wartezeit eingefügt, bevor C_{HOLD} vom Eingang getrennt wird und die Wandlung startet.



Bei T_{AD} = 1μs und T_{ACQT} ~8μs ergibt sich so die Einstellung **8 T_{AD}** (ADCON2bits.ACQT = 0b100;) Eine komplette Wandlung inklusive Aquisition Time und Discharge dauert damit 21 T_{AD} (8+11+2)

8.1.4 ADC Initialisierung und Einfache Messung

Zunächst soll in einem einfachen Beispiel die Analogspannung an einem Poti gemessen und an den vier LEDs visualisiert werden.

Als erstes müssen dafür wieder anhand des Schaltplanes die entsprechenden Pins identifiziert werden, damit der AD-Wandler Eingang und LED Pins entsprechend konfiguriert werden können!

Da auf den uCQ Boards die externe Referenzspannungsquelle normalerweise nicht bestückt ist, wird die Versorgungsspannung als Referenz für den ADC eingestellt. Die Einstellung für T_{AD} und die **Acquisition Time** ergeben sich aus dem vorigen Kapitel.

Auf den vier zur Verfügung stehenden LEDs, können auch nur vier Bits visualisiert werden. Dabei ist es am sinnvollsten, die vier „most significant bits“ darzustellen. Damit diese nicht über zwei Register verteilt sind, sollte das Format „left justified“ gewählt werden.

```
#include <xc.h>
#include "uCQ_2013.h"

void __init(void)
{
    OSCCONbits.IRCF = IRCF_1MHZ; OSCTUNEbits.PLLEN = 0;      // -> 1MHz (default)

    mALL_LED_OUTPUT();
    POTI_TRI = INPUT_PIN; POTI_ANS = ANALOG_IN;

    ADCON0bits.CHS = ?;           // poti is connected to analog channel ?
    ADCON1bits.NVCFG = 0;        // ADref- connected to AVss (GND)
    ADCON1bits.PVCFG = 0;        // ADref+ connected to AVdd ()
    ADCON2bits.ADCS = 0;         // Tad = 1/(Fosc/2) = 2us
    ADCON2bits.ACQT = 0b010;     // 4*Tad = 8us
    ADCON2bits.ADFM = 0;         // left -> 8 most significant bits in ADRESH
    ADCON0bits.ADON = 1;         // switch on ADC module

//    OpenADC(ADC_FOSC_2 & ADC_LEFT_JUST & ADC_4_TAD,           // ADCON2
//            ADC_POTI & ADC_INT_OFF,                         // ADCON0
//            ADC_TRIG_CCP5 & ADC_REF_VDD_VDD & ADC_REF_VDD_VSS); // ADCON1
}
```

In der Schleife des einfachsten Hauptprogramms zur Demonstration der A/D Wandlung, kann man jetzt einfach eine Messung starten, auf das Ende der Messung warten und das Ergebnis anzeigen.

```
void main(void)
{
    __init();

    while(1) {
        ADCON0bits.GO = 1;           // start conversion
        while(ADCON0bits.NOT_DONE){} // wait for completion
        LATB = ~(ADRESH >> 2);    // display result
    }
}
```

Der Start der Wandlung wird das Setzen von ADCON0bits.GO (*bit0*) initiiert. Ob die Messung abgeschlossen ist, kann man durch Beobachten des selbigen erfahren. Es bleibt so lange „1“, bis das Resultat vorliegt. NOT_DONE ist nur ein anderer Name für das gleiche Bit (*union*).

An PORTB sind nur diejenigen Pins als Ausgang konfiguriert, an denen eine LED angeschlossen ist. Deshalb sind auch nur die zugehörigen Bits in LATB von Bedeutung. Die andern können einfach mit irgendwas beschrieben werden, ohne dass daraus negative Seiteneffekte entstehen. Das Positionieren der vier „*most significant bits*“ Resultats auf den LEDs kann man durch Schieben ($>>$) erreichen, die nötige Invertierung aufgrund des low-aktiven Charakters, durch Invertieren (\sim).

8.1.5 Timer -> ADC (Special-Event-Trigger 2)

In den „normalen“ Analog-Datenerfassungs-Anwendungen erfolgt das Einlesen der Analogwerte in genau definierten Zeitabständen (1/Sample-Rate). Dies ist sowohl für die Anzeige der Daten auf einer Zeitachse als auch für die Anwendung etwaiger digitaler Filterfunktionen elementar.

Aktuelle PIC18 Mikrocontroller bieten für diese Standardaufgabe eine recht komfortable Lösung. Beim PIC18FxxK22 kann ein Timer im Verbund mit dem CCP5-Modul über Special-Event-Trigger, genau mit der gewünschten Abtastrate, Analog-Digital-Wandlungen automatisch starten. Damit der ADC auch CCP als Trigger nimmt, muss **ADCON1bits.TRIGSEL** entsprechend gesetzt werden!

Eingestellt werden soll eine Abtastrate von 10/sec. Welcher Timer (*1, 3 oder 5*) im Verbund mit CCP5 arbeitet, wird im Register CCPTMRS1 eingestellt. Für die Einstellungen von Timer Prescaler, Compare Wert und F_{Osc} gibt es viele verschiedene Möglichkeiten. Eine davon wäre:

```
void __init(void)
{
    OSCCONbits.IRCF = IRCF_1MHZ; OSCTUNEbits.PLLEN = 0;      // -> 1MHz (default)
    POTI_TRI = INPUT_PIN; POTI_ANS = ANALOG_IN;

    OpenADC(ADC_FOSC_2 & ADC_LEFT_JUST & ADC_4_TAD,           // ADCON2
            ADC_POTI & ADC_INT_ON,                           // ADCON0
            ADC_TRIG_CCP5 & ADC_REF_VDD_VDD & ADC_REF_VDD_VSS); // ADCON1

    OpenTimer5(TIMER_INT_OFF & T5_16BIT_RW & T5_SOURCE_FOSC_4 &
               T5_PS_1_1 & T5_OSC1EN_OFF & T5_SYNC_EXT_OFF,
               TIMER_GATE_OFF);
    CCPTMRS1bits.C5TSEL = 2; // timer <-> ccp module (CCP5 / TMR5)
    CCPR5 = 25000;          // Fosc/4 / prescaler / Fadc = 1MHz/4 / 1 /10Hz
    CCP5CONbits.CCP5M = 0b1011; // Compare Mode with Special Event Trigger

    flags.all = 0;

    LCD_Init();
    LCD_ConstTextOut(0,0,"SEC      ");
    LCD_ConstTextOut(1,0,"ADC ????");

    INTCONbits.PEIE = 1;
    INTCONbits.GIE = 1;
}
```

Im vorliegenden Fall ist nur der Zeitpunkt interessant, an dem die Wandlung beendet ist und ein neuer Wert zur Verfügung steht. Deshalb wird auch nur der IR des ADC aktiviert!

Nach erfolgter Initialisierung muss das Programm nur noch auf die Interrupts reagieren, die ausgelöst werden, wenn ein neuer Analog-Wert fertig gewandelt wurde und zur Speicherung bzw. Weiterverarbeitung bereit steht. Die Kommunikation der IR-Routine mit dem Hauptprogramm findet wieder über Flags statt.

```
void main(void)
{
    unsigned short sec = 0;

    __init();
    while(1) {
        if(flags.newADC) {
            LCD_ConstTextOut(1,1,"      "); // delete old value
            LCD_ValueOut(1,1,ADRES);
            flags.newADC = 0;
        }
        if(flags.newSec) {
            LCD_ValueOut(0,4,++sec);
            flags.newSec = 0;
        }
    }
}
```

Zur einfachen Kontrolle der Abtastrate wurde im Interrupt auch noch das Flag „newSec“ ergänzt. Wenn die Anzeige auf dem LCD dann im Sekundenrhythmus hochzählt, kann man einigermaßen sicher sein, dass alles korrekt konfiguriert wurde.

```
unsigned short adc_value = 0;
unsigned char tsec = 0;

void high_isr(void)
{
    if (ADC_IR) {
        if (adc_value != ADRES) {
            adc_value = ADRES;
            flags.newADC = 1;
        }
        if (++tsec >= 10) {
            flags.newSec = 1;
            tsec = 0;
        }
        mADC_IR_CLR();
        return;
    }
}
```

8.1.6 ADC Vertiefungsübungen

8.1.6.1 ADC Übung 1: Poti → PWM Sound Frequenz

Poti über ADC einlesen und damit Frequenz von PWM Sound modifizieren

8.1.6.2 ADC Übung 2: Zweiter ADC Channel → Lautstärke

ADC Übung 1 mit einem zweiten ADC Kanal (AN1) ergänzen und sequenziell wandeln.
Mit dem Ergebnis das Tastverhältnis der PWM und damit die Lautstärke ändern.

8.2 Analoge Signale ausgeben (DAC)

22.0 DIGITAL-TO-ANALOG CONVERTER (DAC) MODULE

The Digital-to-Analog Converter supplies a variable voltage reference, ratiometric with the input source, with 32 selectable output levels.

The input of the DAC can be connected to:

- External VREF pins
- VDD supply voltage
- FVR (Fixed Voltage Reference)

The output of the DAC can be configured to supply a reference voltage to the following:

- Comparator positive input
- ADC input channel
- DACOUT pin

The Digital-to-Analog Converter (DAC) can be enabled by setting the DACEN bit of the VREFCON1 register.

VORSICHT: Der DAC Ausgang liegt auf PORTA_2, der auf den uCQ Boards normalerweise mit Signal_B (Richtung) des Drehgebers verbunden wird. Für die DAC Beispiele bitte deshalb JP4 entfernen!

8.2.1 DAC Initialisierung und einfache Ausgabe

...

```
void Analog_Out(void)
{
    unsigned char dac_value = 0;

//----- _init()
    BTN_L_TRI = INPUT_PIN;
    BTN_R_TRI = INPUT_PIN;
    ENC_BTN_TRI = INPUT_PIN;

    VREFCON0bits.FVRS = 3;          // 4.096 V
    VREFCON0bits.FVREN = 1;
    VREFCON1bits.DACPSS = 2;        // FVR
    VREFCON1bits.DACNSS = 0;        // VSS
    VREFCON1bits.DACOE = 1;
    VREFCON1bits.DACEN = 1;
    TRISAbits.TRISA2 = INPUT_PIN;

//----- main()
    while(1) {
        dac_value = 0;
        if(mGET_BTN_L()) dac_value += 4;
        if(mGET_BTN_R()) dac_value += 8;
        if(mGET_ENC_BTN()) dac_value += 16;

        VREFCON2 = dac_value;
    }
}
```

Aufgabe: Analogwert mit Encoder hoch runter

Aufgabe: Zusätzlich am LCD anzeigen

8.2.2 Ausgabe eines Sägezahnsignals

```
...
void Sawtooth_Out(void)
{
//----- __init()
    VREFCON0bits.FVRS = 3;      // 4.096 V
    VREFCON0bits.FVREN = 1;
    VREFCON1bits.DACPSS = 2;    // FVR
    VREFCON1bits.DACNSS = 0;    // VSS
    VREFCON1bits.DACOE = 1;
    VREFCON1bits.DACEN = 1;

//----- main()
    while(1){
        VREFCON2++;
    }
}
```

Aufgabe: Programm erweitern, dass Dreieck-Signal (stufenweise hoch und runter) entsteht.

8.2.3 Ausgabe einer beliebigen Signalform

```
...
void Signal_Out(void) {

    unsigned char signal_idx = 0;
    unsigned char signal_data[32] = {6, 7, 8, 7, 6, 6, 6, 4, 22, 31,
                                    0, 3, 6, 6, 8, 9, 11, 12, 11, 8,
                                    6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6};

//----- __init()
    VREFCON0bits.FVRS = 3;      // 4.096 V
    VREFCON0bits.FVREN = 1;
    VREFCON1bits.DACPSS = 2;    // FVR
    VREFCON1bits.DACNSS = 0;    // VSS
    VREFCON1bits.DACOE = 1;
    VREFCON1bits.DACEN = 1;

//----- main()
    while(1){
        VREFCON2 = signal_data[signal_idx];
        if(++signal_idx >= 32) signal_idx = 0;
    }
}
```

Aufgabe: Programm ändern, dass Sinus ausgegeben wird.

Aufgabe: Programm erweitern, dass die Frequenz eingestellt werden kann.

8.2.4 Erzeugung über PWM und RC-Filter

8.3 CTMU ???????

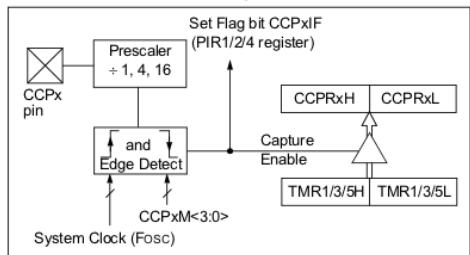
8.3.1 Kapazitiver Schalter

...

9 Zeiten und Frequenzen messen

Größen im Zeitbereich, wie Periodendauer, Pulsbreite und Tastverhältnis, von rechteckförmigen Signalen lassen sich sehr einfach mit Hilfe des CCP Moduls im **Capture Mode** messen.

FIGURE 14-1: CAPTURE MODE OPERATION BLOCK DIAGRAM



Dies wird dadurch ermöglicht, dass die Flanken eines zu untersuchenden Signals, die Speicherung des momentanen Wertes eines mitlaufenden Timers auslösen.

Der korrespondierende Timer ist die Zeitbasis für die Messung. Aus den Differenzen von nacheinander gespeicherten Werten kann man Periodendauer, Pulsbreite und Tastverhältnis eines Signals direkt ermitteln.

Aus der Periodendauer könnte man auch die **Frequenz** berechnen, indem man den Kehrwert bildet.

Je nachdem, welche Größe man messen will, muss man die sensitiven Flanken (*Edge Detect*) des zu untersuchenden Signals konfigurieren. Es gibt die Möglichkeit ein Capture bei jeder fallenden, **oder** jeder (ersten), jeder vierten, oder jeder sechzehnten steigenden Flanke auszulösen. Der Vorteiler ist dabei für die Messung periodischer Signale mit höheren Frequenzen sehr nützlich.

Zur Messung von Größen nicht rechteckförmiger Signale (z.B. EKG Signal) müssen diese vorher entsprechend aufbereitet werden, damit markante Teile (z.B. die R-Zacke von EKG) die Schaltschwellen der CCPx Pins durchlaufen. Im Capture Mode haben diese Pins Schmitt Trigger Charakteristik.

Frequenzen kann man mit zwei Timern bestimmen, von denen einer das Messintervall vorgibt. Der zweite wird als Counter konfiguriert wird und zählt die Perioden eines Signals innerhalb des Messintervalls. Bei einem Intervall von einer Sekunde erhält man direkt die Frequenz in Hz (1/s).

9.1 Periodendauer eines Signals / Zeit Puls zu Puls

Die Periodendauer eines Signals ist die einfachste Konstellation, da hier die Zeit immer zwischen gleichen Flanken (*zwei steigenden oder zwei fallenden*) gemessen werden muss.

Im folgenden Beispiel wird Timer1 so konfiguriert, dass er mit einer Frequenz von 1MHz zählt. Ein Schritt hat dann die Länge von einer Mikrosekunde.

Der komplette Durchlauf des Timers dauert **2¹⁶ µs**. Dies entspricht 65,536ms und es können somit nur Signale analysiert werden, deren Frequenz größer als **~15,3Hz** ist!

Das CCP1 wird auf Capture Mode gestellt, sensitiv auf jede positive Flanke und mit Timer1 verknüpft. Die Verarbeitung des Capture Events erfolgt per Interrupt. Der Eingang für CCP1 ist C2

```
#include "uCQ_2013.h"

#define CAPTR_IR      PIE1bits.CCP1IE && PIR1bits.CCP1IF
#define mCAPTR_IR_EN() PIE1bits.CCP1IE = 1
#define mCAPTR_IR_DIS() PIE1bits.CCP1IE = 0
#define mCAPTR_IR_CLR() PIR1bits.CCP1IF = 0

union DemoFlags flags;
unsigned short capture_value, old_capture;

void __init(void)
{
    OSCCONbits.IRCF = IRCF_4MHZ; OSCTUNEbits.PLLEN = 0;           // -> 4MHz
    TRISCBits.TRISC2 = INPUT_PIN; ANSELCbits.ANSC2 = DIGITAL_PIN;
```

```

OpenTimer1(TIMER_INT_OFF & T1_16BIT_RW & T1_SOURCE_FOSC_4 &
           T1_PS_1_1 & T1_OSC1EN_OFF & T1_SYNC_EXT_OFF,
           TIMER_GATE_OFF);
OpenECapture1(CAPTURE_INT_ON & ECAP_EVERY_RISE_EDGE & ECCP1_SEL_TMR12);

LCD_Init();
LCD_ConstTextOut(0,0,"period =");
LCD_ConstTextOut(1,0,"      us");

old_capture = 0;
flags.all = 0;

INTCONbits.PEIE = 1;
INTCONbits.GIE = 1;
}

void main(void)
{
    __init();

    while(1){
        if(flags.newCapture){
            LCD_ConstTextOut(1,0,"      "); // wipe old value
            LCD_ValueOut(1,0,capture_value);
            flags.newCapture = 0;
        }
    }
}

void __interrupt(high_priority) high_isr(void)
{
    if (CAPTR_IR){
        capture_value = (unsigned short)CCPR1 - old_capture;
        old_capture = (unsigned short)CCPR1;
        flags.newCapture = 1;
        mCAPTR_IR_CLR();
        return;
    }
}

```

Die Zeit für die Periode (*capture_value*) ergibt sich aus der Differenz vom Timer Wert und des vorherigen. Das Ergebnis ist aufgrund des Zweierkomplements auch bei einem Überlauf korrekt, solange der Timer innerhalb einer zu messenden Periode nicht zwei mal überläuft.

9.1.1 Übung Capture 1: Automatische Bereichsumschaltung

Als kleine Vertiefung kann man versuchen, den Timer automatisch auf eine längere Laufzeit um zu schalten, wenn eine Überschreitung der maximal möglichen Messzeit detektiert wird.

Sinnvoll wäre ein weiterer Messbereich, der Zeiten erfassen kann, welche um den Faktor 100 länger sind. Die Umschaltung sollte am besten in beide Richtungen funktionieren.

9.1.2 Übung Capture 2: Keine falschen Werte bei der ersten detektierten Flanke

Bei der ersten detektierten Flanke existiert noch kein vorheriger Wert, mit dessen Hilfe man eine gültige Differenz bilden könnte. Dieses Problem kann man versuchen mit Hilfe des Flags **firstLoop** zu lösen. (*Das Flag sollte nicht unbedingt in der IR ausgewertet werden*)

9.2 Pulsdauer-Messungen mit dem Capture Modul (Ultraschall Entfernungsmesser SRF04 / SRF05)

Bei beliebten Ultraschallsensoren zur Entfernungsmessung wie den SRF04/05 von **ROBOT ELECTRONICS** (<https://robot-electronics.co.uk>), wird die gemessene Entfernung mittels eines Pulses übertragen, dessen Länge direkt proportional zur Entfernung ist. Zur Messung der Pulslänge kann man das Capture Modul benutzen, wobei man allerdings die sensitive Flanke während der Messung umstellen muss.

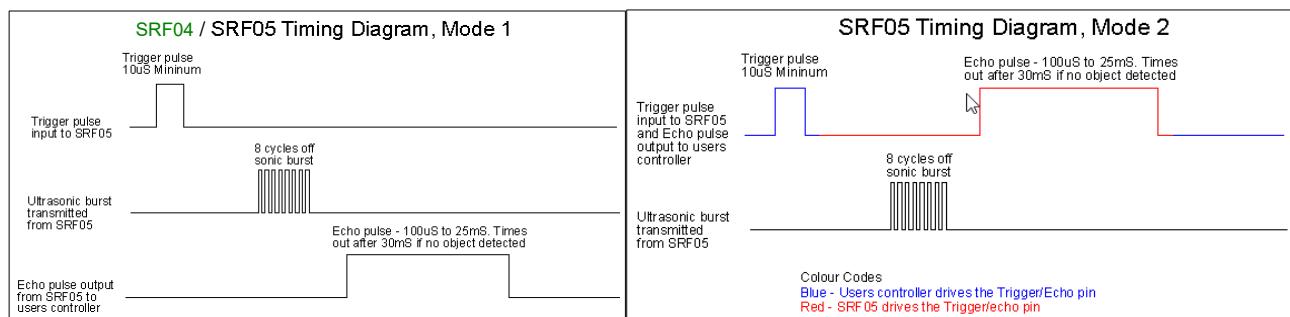
Beim SRF05 können Trigger und Messwert (*Echo*) über die selbe Signalleitung (*Mode 2*) übertragen werden. Kompatibel zum SRF04, kann man aber auch zwei Leitungen nutzen (*Mode 1*).

Fotos

Die Sensoren messen nur auf Anforderung. Um eine Messung zu triggern (*starten*) muss vom Mikrocontroller ein kurzer Impuls von mindestens 10µs Länge gesendet werden. Der Sensor startet daraufhin die Entfernungsmessung, indem er Ultraschallpulse (*sonic burst*) aussendet.

Sofort nach der Aussendung der Ultraschallpulse setzt der Sensor den Pulse Output auf High-Pegel und hält diesen solange, bis er ein Echo detektiert. Falls innerhalb von 30ms kein Echo detektiert wird, wird die Messung abgebrochen. Es gibt dann kein Objekt im Messbereich des Sensors!

Der Initiator der Messung (*unser Mikrocontroller*) muss die Länge des zurückgegebenen Pulses messen und daraus den Abstand eines detektierten Objektes berechnen, oder auch eventuelle Fehlerbedingungen auswerten. (*Abstandunterschreitung, keine Antwort, ...*)



Die Pulslänge pro cm Abstand kann man über die Schallgeschwindigkeit c_s bestimmen.

c_s beträgt **343m/s** oder **0,0343 cm/µs** und damit benötigt der Schall $1/c_s = 29,15\mu\text{s}/\text{cm}$.

Weil der Ultraschall den Weg zwischen reflektierendem Objekt und dem Sensor zwei mal zurück legen muss, ergibt sich eine Pulslänge von 58,3µs pro cm Entfernung zum Objekt. Die ermittelte Pulslänge in µs muss also durch 58,3 geteilt werden um die Entfernung zu berechnen.

Die Messung kann laut Datenblatt der Sensoren maximal alle 50ms oder 20 mal pro Sekunde wiederholt werden. Wenn man die maximale Messfrequenz nicht voll ausschöpft und nur alle 65,536ms oder ~15 mal pro Sekunde misst, dann kann man das Ganze mit einem einzigen 16Bit Timer und einem weiteren Capture Modul pro Sensor erledigen. Der Timer zählt im µs Takt und läuft bei 65.535µs über. Bei jedem Überlauf startet man einfach eine neue Messung.

Das folgende Codebeispiel stammt aus einem Projekt, das verschiedene Konfigurationen für beide Sensorarten (*SRF04 / SRF05*) bietet. Um dies möglichst einfach umzusetzen, wurden zunächst folgende Makros definiert:

```
#define _XTAL_FREQ 4000000

#define SRF_TMR_IR          PIE1bits.TMR1IE && PIR1bits.TMR1IF
#define mSRF_TMR_IR_EN()    PIE1bits.TMR1IE = 1
#define mSRF_TMR_IR_DIS()   PIE1bits.TMR1IE = 0
#define mSRF_TMR_IR_CLR()   PIR1bits.TMR1IF = 0
#define SRF_IR               PIE1bits.CCP1IE && PIR1bits.CCP1IF
#define mSRF_IR_EN()         PIE1bits.CCP1IE = 1
#define mSRF_IR_DIS()        PIE1bits.CCP1IE = 0
#define mSRF_IR_CLR()        PIR1bits.CCP1IF = 0

#define SRF05_SIG            PORTCbits.RC2
#define SRF05_ANS             ANSELDbits.ANSC2
#define SRF05_TRIG            LATCbits.LATC2
#define SRF05_TRI             TRISCbits.TRISC2
#define SRF04_SIG              PORTCbits.RC2
#define SRF04_SIG_TRI          TRISCbits.TRISC2
#define SRF04_TRIG             LATCbits.LATC5
#define SRF04_TRIG_TRI         TRISCbits.TRISC5
#define SRF04_SIG_ANS          ANSELDbits.ANSC2

// SRF05 trigger is generated by switching I/O
#if defined(ADDON_SRF05)
    #define mINIT_SRFPINS() SRF05_TRIG = 1; SRF05_TRI = INPUT_PIN; \
                           SRF05_ANS = DIGITAL_PIN
    #define mSRF_TRIGGER()  SRF05_TRI = 0; __delay_us(10); SRF05_TRI = 1
#elif defined(ADDON_SRF04)
    #define mINIT_SRFPINS() SRF04_TRIG_TRI = OUTPUT_PIN; \
                           SRF04_SIG_TRI = INPUT_PIN; \
                           SRF04_SIG_ANS = DIGITAL_PIN
    #define mSRF_TRIGGER()  SRF04_TRIG = 1; __delay_us(10); SRF04_TRIG = 0
#endif

#define CAPTURE_MODULE_OFF     0x00
#define CAPTURE_FALLING_EDGE   0x04
#define CAPTURE_RISING_EDGE    0x05
```

Beachten sollte man noch, dass das CCP1 Modul zu Anfang deaktiviert wird, damit es im Falle des für Trigger und Echo geteilten Pins beim SRF05 Mode2 nicht durch den Triggerpuls für den Sensor selbst ausgelöst wird. (*Später kann dies noch für eine kleine Sicherheitsabfrage verwendet werden*)

Die Initialisierung der benötigten Komponenten kann wie folgt aussehen.

```
void __init(void)
{
    OSCCONbits.IRCF = IRCF_4MHZ; OSCTUNEbits.PLLEN = 0;

    mINIT_SRFPINS();

    OpenTimer1(TIMER_INT_ON & T1_16BIT_RW & T1_SOURCE_FOSC_4
              & T1_PS_1_1 & T1_OSC1EN_OFF & T1_SYNC_EXT_OFF,
              TIMER_GATE_OFF);

    CCP1MRS0bits.C1TSEL = 0b00; // use TIMER_1 for CCP_1
    CCP1CONbits.CCP1M = CAPTURE_MODULE_OFF;
    mSRF_IR_CLR(); mSRF_IR_EN();

    LCD_Init();
    LCD_ConstTextOut(0,0," SRF-0X "); LCD_ConstTextOut(1,0,"      cm ");

    flags.all = 0;

    INTCONbits.PEIE = 1; // enable peripheral interrupts
    INTCONbits.GIE = 1; // enable global interrupts
}
```

Im Timer IR muss der Sensor getriggert und danach das Capture Modul mit steigender sensitiver Flanke aktiviert werden. Im Capture IR werden die Zeiten ermittelt und die sensitive Flanke von steigend auf fallend umgeschaltet, bzw. das Modul nach erfolgter Messung wieder deaktiviert.

```
void __interrupt(high_priority) high_isr(void)
{
    static uint16_t timeSRF_edge1;

    if (SRF_TMR_IR) //----- TIMER IR
    {
        mSRF_TMR_IR_CLR();
        mSRF_TRIGGER();
        CCP1CONbits.CCP1M = CAPTURE_RISING_EDGE; // wait for rising edge
        flags.all = 0;
        return;
    }
    if (SRF_IR) // ----- SRF_IR
    {
        mSRF_IR_CLR();
        if (CCP1CONbits.CCP1M == CAPTURE_RISING_EDGE) {
            CCP1CONbits.CCP1M = CAPTURE_FALLING_EDGE;
            timeSRF_edge1 = (uint16_t)CCPR1;
        }
        else {
            CCP1CONbits.CCP1M = CAPTURE_MODULE_OFF;
            timeSRF = (uint16_t)CCPR1 - timeSRF_edge1;
            flags.newSRF = 1;
        }
        return;
    }
    while(1){;} // (detect unexpected IR sources)
}
```

Das folgende Hauptprogramm zeigt lediglich die gemessenen Werte mit drei Stellen an.

(Der Wert in μs wird nicht durch 58,3 geteilt, sondern nur durch 58. Resultierender Fehler ~0,5%)

```
uint16_t timeSRF;
void main(void)
{
    __init();

    while(1) {
        if(flags.newSRF) {
            LCD_ValueOut_00(1, 1, timeSRF/58, 3);
            flags.newSRF = 0;
        }
    }
}
```

9.2.1 Übung Capture 3: Sensor Defekt Erkennung

Das Programm zur Abstandsmessung kann sehr einfach mit einer Erkennung erweitert werden, ob der Sensor überhaupt antwortet, oder ob beispielsweise die Verbindungsleitung defekt ist. Dies ist besonders wichtig, wenn es sich um die Messung eines Sicherheitsabstandes handelt.

Die Implementierung kann durch die Abfrage vom momentanen Mode des Capture Moduls **vor dem Start der Messung** im Timer IR erfolgen. Setzen Sie im Fehlerfall ein neu anzulegendes flag und zeigen Sie bei gesetztem Flag eine Meldung am Display an.

9.2.2 Übung Capture 4: Sicherheitsabstand

Ergänzen Sie eine Überprüfung auf einen Mindestabstand innerhalb der Interrupt-Routine. Verwenden Sie zum Vergleichen bitte eine Mindestzeit, die Sie aus dem Mindestabstand einmalig berechnen können. Zeigen Sie die Abstandsunterschreitung auf dem Display an. (*noch ein flag*)

10 Kommunikation

asdf

Einleitung bla, bla...

10.1 RS232 / COM-Port / UART (auch virtuell über USB oder BT)

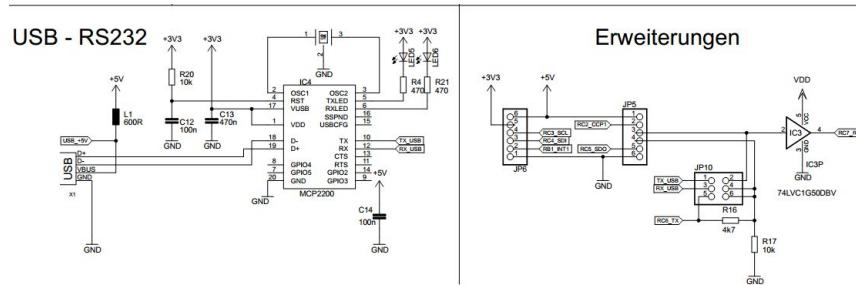
Verbindungen auf dem RS-232 Standard werden schon seit den frühen 1960er Jahren für die Datenübertragung zwischen Computersystemen genutzt. In vielen neueren Systemen ist diese auch als COM-Port bezeichnete frühere „Standartschnittstelle“ zwar oft nicht mehr vorhanden, doch gibt es viele USB und BlueTooth Adapter, welche diese bei Bedarf auf der PC Seite bereitstellen.

10.1.1 Die wichtigsten Grundlagen (tiefer Einblicke -> Wikipedia)

- Es wird stehn gewöhnlich zwei Datenleitungen zur Verfügung (*für jede Richtung eine*) Daten können damit gleichzeitig in beiden Richtungen übertragen werden (*Vollduplex*).
- Neben den Datenleitungen können auch noch einige zusätzliche sogenannte „Handshake“ Signale implementiert werden, die dem Kommunikationspartner verschiedene Zustände übermitteln. (*z.B. RTS → Request to Send, Device möchte Daten senden*)
- Die Datenübertragung erfolgt asynchronen, d.h. es gibt kein gemeinsames Taktsignal. Die Kommunikationspartner müssen (*unabhängig voneinander*) mit der gleichen Geschwindigkeit arbeiten.
- Hat der PC noch einen „echten“ COM-Port oder wird ein handelsüblicher USB-RS232 Konverter verwendet, dann ist eine Pegelanpassung (und Invertierung) notwendig. Dafür benötigt man ein zusätzliches Bauteil (*z.B. MAX232*)
Die Logikpegel „1“ und „0“ oder „High“ und „Low“ die am µC und innerhalb des PCs „1,8V..5V“ und „0V“ betragen, werden auf „-6V..-12V“ und „+6V..+12V“ umgesetzt.
Das Signal erscheint dadurch „invertiert“ (High → negative Spannung, Low → pos. Sp.)
Um die Übertragungssicherheit (Fehler) zu erhöhen, wird das Signal auf der Übertragungsstrecke verstärkt.
- Bei Verwendung eines USB-UART-Converter Bausteines, BlueTooth Chips oder ähnlichem, muss auf korrekte Logikpegel (*5V / 3,3V*) geachtet werden!
- Serielle Datenübertragung über RS232 ist „relativ“ langsam im Vergleich zur Arbeitsgeschwindigkeit eines Controllers. Während des Versendens eines Datenbytes kann ein Controller leicht hunderte von Arbeitsschritten ausführen.
- Im Mikrocontroller ist meist ein Modul enthalten, dass auch noch andere Standards implementieren kann. Ein **EUSART** Modul wie im PIC18xxK22 steht z.B. für einen **Enhanced-Universal-Synchronous-Asynchronous-Receiver-Transmitter** und muss zur Verwendung als UART erst entsprechend konfiguriert werden.

10.1.2 Hardwarevoraussetzungen

Im Schaltplan der uC-Quick-2013 Platine kann man sich anschauen, wie die Transmitter (*RC6_TX*) und Receiver-Signale (*RC7_RX*) des PIC über den Level-Konverter (*IC3*) und den Jumper *JP10* mit dem USB-UART-Converter IC4 verbunden werden können. (*1-2 und 3-4 verbinden*)



Der Logikpegel des USB Konverters beträgt 3,3V. Wird der PIC mit 5V versorgt, erkennt er das Sendesignal TX_USB des Konverters im High-Zustand nicht, weil der Eingang RC7_RX im Schmitt-Trigger Modus konfiguriert wird. ([Datenblatt...](#)) Die Anpassung des Pegels erfolgt hier automatisch dadurch, dass IC3 mit der selben Spannung versorgt wird wie der µC.

Die Sendeleitung RC6_TX des PICs hat bei einer Versorgung mit 5V einen zu hohen Pegel und kann durch den Spannungsteiler R16-R17 abgeschwächt werden. (*Bei 3,3V → JP10 5-6 verbinden*)

10.1.3 EUSART Modul Grundkonfiguration

Als erstes muss natürlich wie immer im Datenblatt des Controllers nachgeschaut werden wie und über welche Register das Modul konfiguriert wird. Für den PIC18F2xK22 wäre das im Kapitel:

16.0 ENHANCED UNIVERSAL SYNCHRONOUS ASYNCHRONOUS RECEIVER TRANSMITTER (EUSART)

The EUSART module includes the following capabilities:
<ul style="list-style-type: none"> • Full-duplex asynchronous transmit and receive • Two-character input buffer • One-character output buffer • Programmable 8-bit or 9-bit character length • Address detection in 9-bit mode • Input buffer overrun error detection • Received character framing error detection • Half-duplex synchronous master • Half-duplex synchronous slave • Programmable clock and data polarity

The operation of the EUSART module is controlled through three registers:

- Transmit Status and Control (TXSTAx)
- Receive Status and Control (RCSTAx)
- Baud Rate Control (BAUDCONx)

These registers are detailed in [Register 16-1](#), [Register 16-2](#) and [Register 16-3](#), respectively.

For all modes of EUSART operation, the TRIS control bits corresponding to the RXx/DTx and TXx/CKx pins should be set to '1'. The EUSART control will automatically reconfigure the pin from input to output, as needed.

When the receiver or transmitter section is not enabled then the corresponding RXx/DTx or TXx/CKx pin may be used for general purpose input and output.

In den folgenden Unterkapiteln werden die nötigen Schritte für die Konfiguration erläutert.

10.1.3.1 EUSART Register und Pins

Die erste Überraschung für Anfänger kann schon daraus bestehen, dass der Pin von dem Daten gesendet werden sollen (**TXx/CKx**) als Eingang konfiguriert werden muss. Gleich darunter wird ein Hinweis darauf gegeben, warum dem so ist. Weiterhin erfährt man, dass die Register **TXSTAx**, **RXSTAx** und **BAUDCONx** für die Kontrolle des Moduls verantwortlich sind.

Der PIC18F2xK22 verfügt über 2 USART Module. Das „x“ in den Registerbezeichnungen muss durch die Nummer des Moduls ersetzt werden.

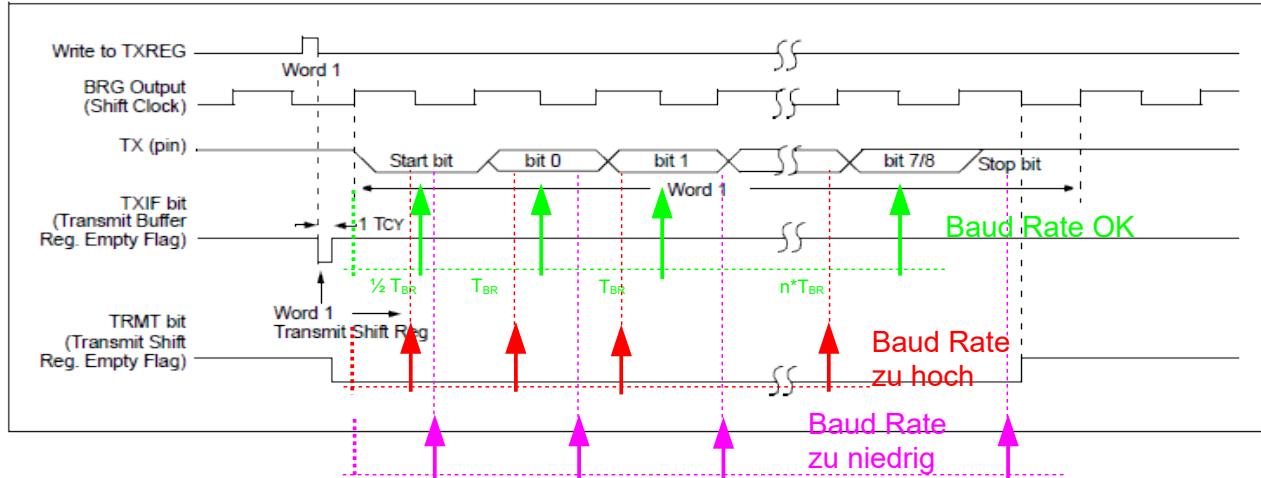
10.1.3.2 Die Baud Rate

Die „Baud Rate“ ist das Maß für die Geschwindigkeit (*hier Bits pro Sekunde*) mit der die Daten übermittelt werden. Da die Kommunikation über keine Synchronisationssignale (*Clocks*) verfügt, muss diese Zeit natürlich für beide Endstellen (*PC und PIC*) genau gleich eingestellt werden!

Die farbigen Pfeile in der folgenden Grafik sollen verdeutlichen, wie eine ungenau eingestellte Baud Rate sich auf die Datenübertragung auswirkt. Die Pfeile zeigen an, zu welchem Zeitpunkt der Empfänger versucht das übertragene Signal zu ermitteln.

Als Signal für den Übertragungsbeginn dient dem Empfänger das „Start-Bit“, welches immer ein Wechsel des Signals aus dem Ruhe-Zustand „1“ nach „0“ ist. **Danach versucht er im vorgegebenen Zeitabständen das übertragene Signal zu ermitteln.** Nach der Übertragung der Datenbits wechselt das Signal im sogenannten „Stop-Bit“ immer wieder in den „Idle“-Zustand „1“.

FIGURE 16-3: ASYNCHRONOUS TRANSMISSION



In der obigen Abbildung kann man erkennen wie sich Fehler in der Baudrate aufsummieren. Sieht die Abweichung beim Startbit noch relativ harmlos aus, so werden die Distanzen im weiteren Verlauf immer größer.

→ Die Baudrate sollte mindestens auf 3% genau eingestellt werden.
Dann beträgt die max. Abweichung beim 10. Bit (*Stop-Bit*) ungefähr 1/3 Bit (30%)

Die Erzeugung des Taktes, der für das Sampling der einzelnen Bits verantwortlich ist, erfolgt durch den Baudraten Generator. Prinzipiell funktioniert dieser genauso wie ein Timer in Verbindung mit einem Capture Compare Modul. Die Periode ist über das Registerpaar SPBRGH:SPBRG einstellbar. Die Berechnung des erforderlichen Wertes für das Registerpaar erfolgt nach einer Formel die im Datenblatt zu ermitteln ist. Stehen wie im folgenden Beispiel mehrere Formeln für verschiedene Konfigurationen zur Verfügung, muss man diejenige herausfinden, welche für den vorliegenden Fall am besten geeignet ist.

TABLE 16-3: BAUD RATE FORMULAS

Configuration Bits			BRG/EUSART Mode	Baud Rate Formula
SYNC	BRG16	BRGH		
0	0	0	8-bit/Asynchronous	$F_{osc}/[64 (n+1)]$
0	0	1	8-bit/Asynchronous	$F_{osc}/[16 (n+1)]$
0	1	0	16-bit/Asynchronous	
0	1	1	16-bit/Asynchronous	$F_{osc}/[4 (n+1)]$
1	0	x	8-bit/Synchronous	
1	1	x	16-bit/Synchronous	

Legend: x = Don't care, n = value of SPBRGHx, SPBRGx register pair.

Den Wert für des SPBRG Register erhält man durch Umstellen der Formeln nach „n“.

$$n = (F_{osc} / (X * \text{Baud Rate})) - 1 \quad / \quad X = 64, 16 \text{ oder } 4$$

Angenommen die Taktfrequenz des Controllers sei 4MHz und die Daten sollen mit 56.7k BAUD (57600 Bits/Sekunde) übertragen werden.

Je nach Formel erhält man für „n“ die Werte 0.085, 3.34, und 16.36.

Der Fehler entsteht dadurch, dass nur Integerwerte für das SPBRG Register in Frage kommen. Diese ergeben sich durch Auf- oder Abrunden der berechneten Werte.

Anhand des Betrages der gerundet werden muss, kann man schon ungefähr abschätzen wie hoch der Fehler sein wird. Die genaue Formel zur Fehlerberechnung lautet:

$$\text{Fehler} = (1/(SPBRG+1) - 1/(n+1)) * (n+1)$$

Formel	$n = (4M / (64 * 57k6)) - 1$	$n = (4M / (16 * 57k6)) - 1$	$n = (4M / (4 * 57k6)) - 1$
n	0,085	3,34	16,36
SPBRG	0	3	16
Fehlerrechnung	$(1 - 1 / 0,085) * 1,085$	$(1/4 - 1 / 4,34) * 4,34$	$(1/17 - 1 / 17,36) * 17,36$
Fehler	8,5%	8,5%	2,12%

Wenn der Fehler $\leq 3\%$ sein soll, dann kommt nur die Konfiguration BRG16 = 1 und BRGH = 1 in Frage. (*SYNC muss natürlich „0“ sein, da der Übertragungsmodus asynchron ist*)

BRGH ist Bit-2 im TXSTA Register und BRG16 ist Bit-3 im BAUDCON.

10.1.3.3 USART initialisieren

Mit den bisher gewonnenen Informationen und den Beschreibungen im Datenblatt können jetzt die Register vorkonfiguriert werden. Bits für bislang unbekannte Funktionen sollte man dabei am besten im „Default“ Zustand belassen.

Die Aktivierung des USART Moduls erfolgt über das Setzen den SPEN Bits im Register RCSTAx ! Inklusive der Richtungsfestlegung für die Pins ergibt sich damit für die Initialisierung:

```

TRISCBits.TRISC6 = 1;           // disable LAT output for TX pin
TRISCBits.TRISC7 = 1;           // disable LAT output for RX pin

TXSTA1bits.SYNC = 0;            // asynchronous

BAUDCON1bits.BRG16 = 1;         // 16-bit Baud Rate Generator
TXSTA1bits.BRGH = 1;            // high speed

SPBRG1 = 16; SPBRGH1 = 0;       // (4M / (4 * 57k6)) - 1 ~ 16

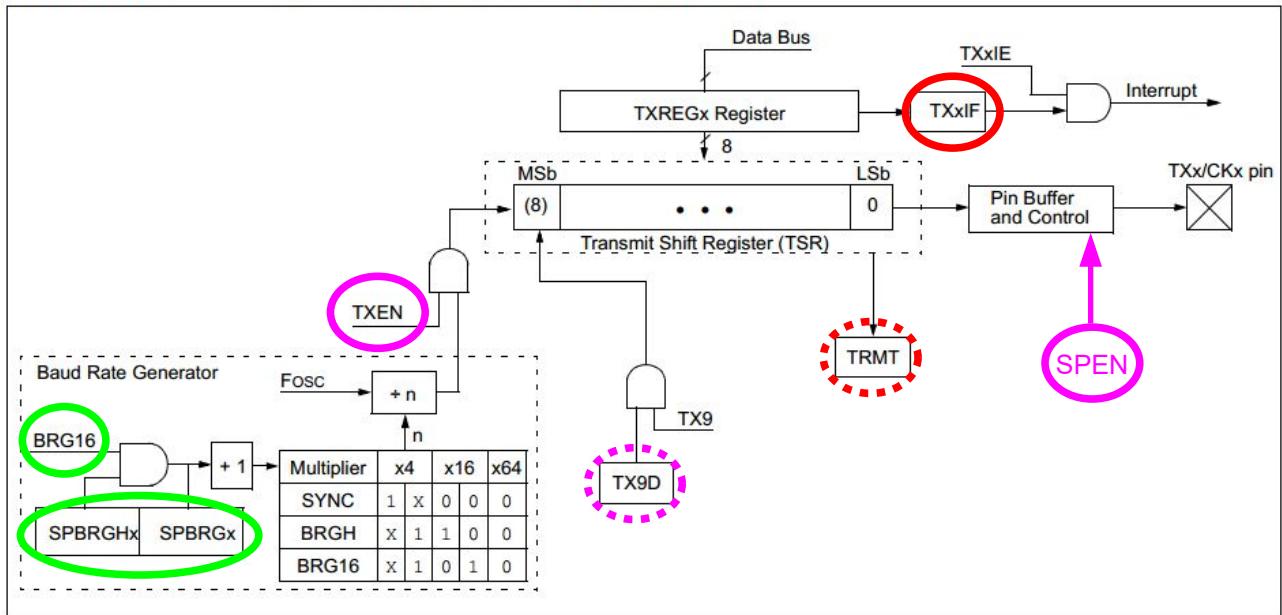
```

```
RCSTA1bits.SPEN = 1; // Serial Port Enable
```

10.1.4 Senden von Daten

Anhand der Beschreibung und des Blockdiagramms für den Sender werden jetzt die noch fehlenden Voraussetzungen geschaffen um Daten vom PIC an einen angeschlossenen PC senden zu können.

FIGURE 16-1: EUSART TRANSMIT BLOCK DIAGRAM



Zunächst muss noch das **TXEN** Bit (EN = „enable“) gesetzt werden. Das Bit **TX9D** zur Auswahl von acht oder neun Datenbits sollte auf „0“ bleiben, da der PC im Standard-Modus auch auf eine 8-Bit Übertragung eingestellt ist.

Der „Initialisierung“ wird demnach noch hinzugefügt:

```
TXSTAbits.TX9 = 0;
TXSTAbits.TXEN = 1;
```

Sind alle Vorbereitungen abgeschlossen wird das Senden einfach durch Beschreiben des Registers TXREG initiiert. Dieses „Pufferregister“ übergibt die Daten dann an das Transmitter-Shift-Register“ welches die einzelnen Bits dann nacheinander an den TX Pin schiebt.

Die Datenübergabe von TXREG an TSR ist nur möglich, wenn TSR „leer“ ist, bzw. eine vorangegangene Transmission abgeschlossen ist. Dieser Vorgang geschieht automatisch und muss nicht durch den Programmcode überwacht werden. Sofort nach der Übergabe ist TXREG wieder frei obwohl das Senden der Daten noch einige Zeit in Anspruch nimmt.

Die Prüfung ob das TXREG zur Aufnahme von Daten bereit ist muss dagegen die Software durchführen. Vor jedem Beschreiben erfolgt deshalb die Abfrage des Transmitter-Interrupt-Flags TXIF. Ist dies „1“ so ist das TXREG frei. Ist TXIF = 0 dann muss gewartet werden, bis es frei wird.

ACHTUNG: Nach dem Beschreiben von TXREG dauert es ... bis das TXIF reagiert. Die Abfrage darf deshalb nicht unmittelbar danach erfolgen !!!

Das eigentliche Senden eines Datenbytes erfordert mit den Start- und Stop-Bits bei der eingestellten Baudrate 10/57600 s. Die Ausführung eines Befehls (*z.B. Füllen von TXREG*) bei der Taktfrequenz von 4MHz 1/1000000 s. Muss das Programm einen kompletten Zyklus „warten“, dann könnten in dieser Zeit 173 Befehle ausgeführt werden!

Würde der PIC bei der eingestellten Rate ohne Pause senden, dann kämen beim PC ~5760 Bytes/s an. Die übermittelten Daten sind eigentlich nur Bit-Kombinationen aus Einsen und Nullen und können nur unter Verwendung des ASCII (*American Standard Code Information Interchange*) Codes vernünftig dargestellt werden.

Werden diese Daten auf dem PC als Zeichen interpretiert, entspräche das ungefähr 5760 Zeichen und somit einer voll beschriebenen Textseite pro Sekunde. Die „online“ Darstellung wäre etwas schwierig und die Verfolgung durch einen menschlichen Betrachter unmöglich.

Für das erste Testprogramm soll deshalb nur dann ein Datenbyte gesendet werden, wenn eine Taste auf der PIC-Platine betätigt wird.

Durch Inkrementieren des zu sendenden Zeichens kann man auf einfache Weise verschiedene Zeichen senden. Weil aber nicht alle Bit-Kombinationen als Zeichen darstellbar sind, sollte man durch eine zusätzliche Abfrage sicherstellen, dass nur darstellbare Zeichen gesendet werden.

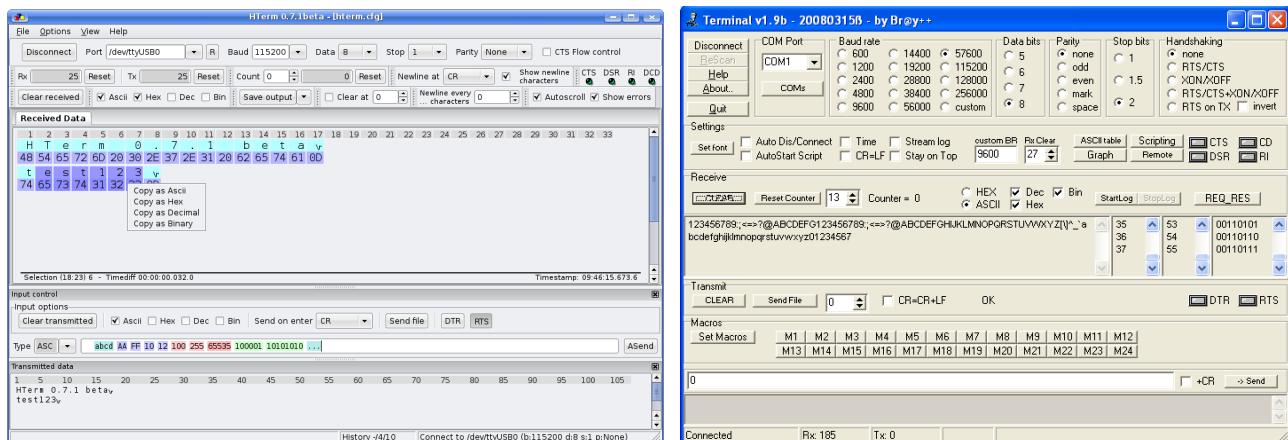
Die fertige Programm kann dann wie folgt aussehen:

```
#include <xc.h>

void __init(void)
{
    OSCCONbits.IRCF = IRCF_4MHZ; OSCTUNEbits.PLLLEN = 0;           // -> 4MHz
    TX_TRI = RX_TRI = INPUT_PIN;
    BAUDCON1bits.BRG16 = 1;
    TXSTA1bits.BRGH = 1;
    TXSTA1bits.SYNC = 0;
    SPBRG1 = 16; SPBRGH1 = 0;
    RCSTA1bits.SPEN = 1;
    TXSTA1bits.TX9 = 0;
    TXSTA1bits.TXEN = 1;
    ENC_BTN_TRI = INPUT_PIN; ENC_BTN_ANS = DIGITAL_PIN;
}

void main(void)
{
    unsigned char character = '0';
    __init();
    while(1){
        if(mGET_BTN_1()){
            if(++character > 'z')
                character = '0';
            if(PIR1bits.TX1IF)           // if() <-> while() ?
                TXREG1 = zeichen;
            while(mGET_BTN_ENC()){};    // ? without this line ?
        }
    }
}
```

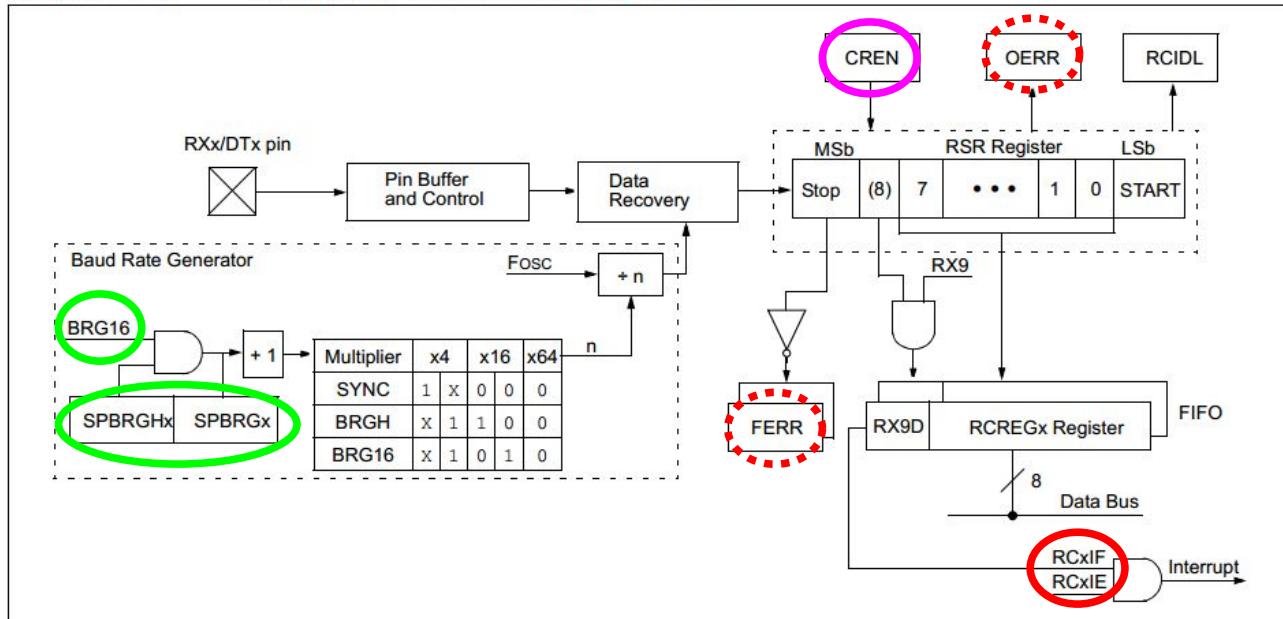
Zur Darstellung der Empfangenen Daten auf dem PC kann ein Terminalprogramm wie z.B. [Br@yTerminal \(www\)](#) oder [HTerm \(www\)](#) verwendet werden.



10.1.5 Empfangen von Daten / Befehlen

Zunächst natürlich wieder der obligatorische Blick ins Datenblatt ...

FIGURE 16-2: EUSART RECEIVE BLOCK DIAGRAM



Die Einstellung für die Baudrate kann aus den vorherigen Kapitel übernommen werden. Der Receiver des USART Moduls wird durch das **CREN** (*Continuous Receive Enable*) Bit aktiviert.

Wenn Receiver_1 ein Datenbyte empfangen hat, dann wird das mit dem Bit **RC1IF** im Register **PIR1** signalisiert. Dieses kann durch Abfragen (*Polling*) überprüft werden. In der Praxis wird dafür aber oft die IR Funktion genutzt, die erst in einem Beispiel weiter unten aufgegriffen wird.

Falls beim Empfang Fehler aufgetreten sind, werden die Error Bits **OERR** und **FERR** gesetzt. **OERR** steht hier für Overflow-Error und bedeutet, dass ein empfangenes Datenbyte nicht im **RC-FIFO** gespeichert werden konnte, da dieses noch vorher empfangene Daten enthielt die nicht abgeholt (*gelesen*) wurden.

FERR steht für Framing-Error und lässt auf eine falsch eingestellte Baudrate schließen. (*das Stopbit war nicht IDLE*)

Aufgetretene Fehler blockieren den weiteren Datenempfang und müssen behandelt werden. Wie schon die IR Steuerung wird auch die Fehlerbehandlung zunächst auf später verschoben.

Der folgende Code zeigt ein sehr rudimentäres Empfangsprogramm ohne Fehlerbehandlung:

```
...
RCSTA1bits.RX9 = 0; //----- _init()
RCSTA1bits.CREN1 = 1;
//----- main()
while(1){
    unsigned char command;
    if(PIR1bits.RC1IF){
        command = RCREG1;
        switch(command){
            case '0': mSET_LED_1_OFF(); mSET_LED_2_OFF();
                        mSET_LED_3_OFF(); mSET_LED_4_OFF(); break;
            case '1': mSET_LED_1_ON(); break;
            case '2': mSET_LED_2_ON(); break;
            case '3': mSET_LED_3_ON(); break;
            case '4': mSET_LED_4_ON(); break;
            default: break;
        }
    }
}
```

Auch dieses Programm kann wieder mit Bray Terminal getestet werden. (*Macro für Funktionstaste*)

10.1.6 MiniRS232 (bidirektional)

Zum Abschluss der Einführungsübungen noch ein kleines Beispiel, welches senden und empfangen kann. Hier wurde das USART Modul mit Funktionen der alten Peripheral Library konfiguriert.

```
#include "plib18fxxk22.h"

#define _XTAL_FREQ 4000000
#define BAUDRATE 19200

void __init(void)
{
    OSCCONbits.IRCF = IRCF_4MHZ; OSCTUNEbits.PLLEN = 0; // -> 4MHz

    mALL_LED_OUTPUT();
    ENC_BTN_TRI = INPUT_PIN; ENC_BTN_ANS = DIGITAL_PIN;
    BTN_L_ANS = BTN_R_ANS = DIGITAL_PIN;
    TX_TRI = RX_TRI = INPUT_PIN; RX_ANS = DIGITAL_PIN;

//#define SPBRG_VAL (((_XTAL_FREQ/BAUDRATE)+32)/64)-1 // BRG16=0, BGH=0 !
//#define SPBRG_VAL (((_XTAL_FREQ/BAUDRATE)+8)/16)-1 // BRG16=0, BGH=1 !
#define SPBRG_VAL (((_XTAL_FREQ/BAUDRATE)+2)/4)-1 // BRG16=1, BGH=1 !

    baud1USART(BAUD_IDLE_RX_PIN_STATE_HIGH &
                BAUD_IDLE_TX_PIN_STATE_HIGH &
                BAUD_16_BIT_RATE &
                BAUD_WAKEUP_OFF &
                BAUD_AUTO_OFF);

    Open1USART( USART_TX_INT_OFF & USART_RX_INT_OFF &
                USART_SYNCH_MODE & USART_EIGHT_BIT &
                USART_CONT_RX &
                USART_BRGH_HIGH & USART_ADDEN_OFF,
                SPBRG_VAL );
}

void main(void)
{
    unsigned char command;
    while(1){
        if(DataRdy1USART()) {
            command = RCREG1;
            switch(command) {
                case '0': mALL_LED_OFF(); break;
                case '1': mSET_LED_1_OFF(); break;
                case '2': mSET_LED_2_OFF(); break;
                case '3': mSET_LED_3_OFF(); break;
                case '4': mSET_LED_4_OFF(); break;
                case '5': mSET_LED_1_ON(); break;
                case '6': mSET_LED_2_ON(); break;
                case '7': mSET_LED_3_ON(); break;
                case '8': mSET_LED_4_ON(); break;
                case '9': mALL_LED_ON(); break;
                case '?':
                    BTN_L_TRI = BTN_R_TRI = INPUT_PIN;
                    __delay_us(10);
                    command = mGET_BTN_L();
                    command += 2 * mGET_ENC_BTN();
                    command += 4 * mGET_BTN_R();
                    if(command < 10) command += '0';
                    else command += 'A' - 10;
                    TXREG1 = command;
                    mALL_LED_OUTPUT();
                    break;
                default: break;
            } //switch(command)
        } //if(DataRdy1USART())
    } //while(1)
}
```

10.1.7 Erweiterte Funktionen des USART Moduls

Bisher blieben einige Einstellungsmöglichkeiten des USART Moduls noch unbeachtet, tauchten aber schon im letzten Beispiel als Parameter für die Funktionen der **Peripheral Library** auf. Mit Hilfe dieser Funktionen wurde die Initialisierung des Moduls durchgeführt.

Bei Verwendung der Bibliotheksfunktionen sollten immer alle Parameter angegeben werden, weil man sich nicht mehr darauf verlassen kann, dass die Default Werte der nicht spezifizierten Registerbits erhalten bleiben.

Generell ist es immer sicherer **ALLE** Bits, welche die Funktion eines Moduls beeinflussen, zu kennen und im Code zu spezifizieren, als sich auf vermeintliche Default Werte zu verlassen!

10.1.7.1 Clock/Transmit Polarity

Im asynchronen Modus kann man mit **BAUDCONbits.CKTXP** die Polarität des Transmitters einstellen. Der Normalfall ist **0** // *Idle state for transmit (TXx) is high*

10.1.7.2 Data/Receive Polarity

Analog zum Transmitter kann man auch für den Receiver die Polarität einstellen. Auch für **BAUDCONXbits.DTRXP** ist der Normalfall **0** // *Receive data (RXx) is not inverted (active-high)*

10.1.7.3 Auto-Baud Detect mode

In diesem Modus wird die Baudrate nicht im Programm gesetzt, sondern das Modul versucht selber die korrekte Einstellung herauszufinden. Normal ist **BAUDCONbits.ABDEN = 0**; // *disabled*. Näheres dazu siehe Data Sheet...

10.1.7.4 Wake-up Enable

Normal **BAUDCONXbits.WUE = 0** // *Receiver is operating normally*

10.1.7.5 Send Break

TXSTAXbits.SENDB = 0; // *See Data Sheet...*

10.1.7.6 Address Detect

Spielt im 8-Bit Modus keine Rolle. → *Data Sheet...*

10.1.7.7 Clock Source Select

Spielt im asynchronen Modus keine Rolle. → *Data Sheet...*

10.1.8 Fehlerbehandlung (Overflow und Framing ERRORS)

Kommt es beim Empfang von Daten mit dem USART Modul zu Fehlern, dann kann das dazu führen, dass ein weiterer Empfang nicht mehr möglich ist weil das Modul komplett blockiert wird.

Deshalb ist es unbedingt erforderlich, Fehler abzufragen und dann evtl. die entsprechenden Aktionen auszuführen, damit das Modul wieder funktionsfähig wird.

Es gibt zwei mögliche Fehler, die vom UART Modul detektiert und angezeigt werden. Beide wurden auch im Text zum [Reciever Block Diagramm](#) weiter oben schon erwähnt.

10.1.8.1 Overflow Error - OERR

Der Überlauf Fehler des Moduls entsteht, wenn der Empfang des dritten Bytes in Folge im Schiebe-Register abgeschlossen ist bevor zumindest ein Byte aus dem Empfangspuffer gelesen wurde. Das dritte Byte müsste jetzt ins Empfangs-FIFO verschoben werden. Dieses enthält aber noch die zwei vorherigen, weil die nicht abgeholt (*gelesen*) wurden und ist somit voll belegt. Das dritte Byte kann nicht ins FIFO geschrieben werden und geht verloren.

Daraufhin wird das OERR Bit im RCSTRAX Register gesetzt und das Modul für den weiteren Empfang gesperrt. Erst nach einer Fehlerbehandlung die im Data Sheet beschrieben wird, kann das Modul wieder genutzt werden.

10.1.8.2 Framing Error FERR

Ein Framing Error wird detektiert, wenn das Empfangssignal zum Zeitpunkt wenn das Stopbit anliegen müsste nicht den Idle Pegel (*hier High*) hat. Die Ursache dafür kann eine falsch eingestellte Baudrate sein.

Wenn der Empfänger schneller abtastet, als der Sender sendet kann er ein Low-Bit der übertragenen Daten erwischen, wenn er eigentlich das Stopbit erwartet.

Tastet der Empfänger langsamer ab als der Sender die Bits überträgt, dann kann nur ein Framing Error erkannt werden, falls der Empfänger ein Stopbit erwartet während der Sender schon weitere Daten überträgt und im Abtastzeitpunkt ein Low-Bit dieser anlegt.

Bei einer bidirektionalen Kommunikation ist es in einer Übertragungsrichtung immer der Fall, dass das schnellere Modul als Empfänger fungiert und Framing Fehler detektiert.

```
While(1) {
    if (RCSTA1bits.OERR) {      // check overflow error
        dummy = RCREG;          // wipe the receiver fifo
        dummy = RCREG;
        RCSTA1bits.CREN = 0;     // clear the OVR flag
        RCSTA1bits.CREN = 1;
    }

    if (RCSTA1bits.FERR) {      // check framing error
        dummy = RCREG;          // wipe the receiver fifo
    }

    if(DataRdy1USART()) {
        command = RCREG1;
        switch(command) {
            ...
        }
    }
}
```

10.1.9 Receiver Interrupt

Die serielle Datenübertragung über UART ist meist relativ langsam im Verhältnis zur Arbeitsgeschwindigkeit des Mikrocontrollers. Oft reicht völlig es aus, wenn einmal pro Durchlauf der Hauptschleife des Programms nachgeschaut wird, ob ein neues Datenbyte empfangen wurde.

Man kann allerdings auch einen Interrupt auslösen lassen, wenn ein Byte empfangen wurde und dieses dann in der Interrupt Service Routine weiterverarbeiten. In der Regel wird das empfangene Byte in der IRS nur abgespeichert (*damit das Empfangsregister wieder frei wird*) und erst später in der Hauptschleife weiter verarbeitet.

Der Receiver IR gehört zu den peripheren IR, deshalb müssen insgesamt die IR-Enable Bits gesetzt werden, damit der IR aktiv werden kann. Natürlich muss auch eine IRS vorhanden sein in welcher der IR behandelt wird.

```
void __init(void)
{
    ...
    OpenUSART( USART_TX_INT_OFF & USART_RX_INT_ON &
    ...
    RCONbits.IPEN = 0;           // disable interrupt priority
    INTCONbits.PEIE = 1;         // enable peripheral interrupts
    INTCONbits.GIE = 1;          // enable global interrupts
} //end system initialization
```

Das IR-Flag des USART Receiver Moduls hat beim PIC18FxxK22 die Besonderheit, dass es **nur lesbar** ist. Man kann es also nicht wie die anderen IR-Flags löschen. Das Flag wird automatisch beim Lesen des RCREG zurück gesetzt. Auch die schon bekannte Fehlerbehandlung wird im IR durchgeführt.

```
void __interrupt(high_priority) SYS_InterruptHigh(void)
{
    if (USART_IR) // ----- USART1_IR (PC)
    {
        if (RCSTAbits.OERR)      // check overflow error
        ...
        if (RCSTAbits.FERR)      // check framing error
        ...
        if (DataRdyUSART()) {   //----- new byte received ##
            rxBuffer = RCREG;  // -> save it in a buffer
            rxFlag = 1;          // and set a flag to signal that
        }
        return;
    } //if (USART1_IR)
    ...
}
```

Das Zwischenspeichern des empfangenen Bytes muss zusätzlich noch irgendwie vermerkt werden, damit die Verarbeitung später in der Hauptschleife des Programms stattfinden kann. Nach der Verarbeitung wird dann das Flag wieder gelöscht.

```
while(1) {
    ...
    if(rxFlag){           // process received byte
        ...
        rxFlag = 0;
    }
    ...
}
```

10.1.10 Receiver Buffer

Die im vorhergehenden Kapitel vorgestellte Zwischenspeicherung empfangener Bytes macht eigentlich erst Sinn, wenn man eine ganze Nachricht zwischenspeichern kann. Also legt man am besten einen Buffer entsprechender Größe an.

Zusätzlich braucht man anstelle eines Flags jetzt einen Zähler, der die empfangenen Daten verwaltet. Buffer und Zähler kann man in einer Struktur zusammen fassen.

```
typedef struct{                                // receiver buffer structure
    uint8_t idx;
    uint8_t bytes[MAX_BUFFER_SIZE]; // maximum usart buffer (or message) size
}usartBuffer_t;

usartBuffer_t rxBufferPC;                      // -> from PC received data variable
```

Beim Befüllen des Buffers muss man darauf achten, dass dieser nicht überläuft und dies auch im Fall der Fälle irgendwo vermerken. Dafür kann man eine weiteres Bitfeld anlegen, in dem man verschiedene Fehler vermerken kann. In diesem Bitfeld können auch die möglichen Fehler des USART Moduls enthalten sein.

```
typedef union{
    struct {
        unsigned ovflError : 1; // Overflow error
        unsigned frmError : 1; // Framing error
        unsigned rxBffOvfl : 1; // user buffer overflow Error
        ...
        unsigned newError : 1;
    };
    uint8_t ALL;
}uartFlags_t;

uartFlags_t uartFlags;
```

Der für die Vorverarbeitung empfangener Datenbytes benötigte Code sieht dann etwa folgendermaßen aus:

```
if (RCSTAbits.OERR)      //----- check overflow error
{
    ...
    uartFlags.ovflError = 1; uartFlags.newError = 1;
}
if (RCSTAbits.FERR)      //----- check framing error
{
    ...
    uartFlags.frmError = 1; uartFlags.newError = 1;
}
if (DataRdyUSART()){    //----- new byte received #####
    if(rxBufferPC.idx < MAX_RX_BUFFER){
        rxBufferPC.bytes[rxBufferPC.idx++] = RCREG;
    }
    else{
        uartFlags.rxBffOvfl = 1; uartFlags.newError = 1;
    }
}
```

In der Hauptschleife kann man dann den Zähler des Empfangsdatenpaketes abfragen und ggf. die Verarbeitung starten, welche je nach Anwendung sehr unterschiedlich aussehen kann.

```
while(1)
//----- new USART bytes
if(rxBufferPC.idx){
    ...
}
```

10.1.11 Transmitter Interrupt und Buffer

Brauchd kä S..

10.1.12 Kommunikation mit zusätzlichem Übertragungsprotokollen

In der Praxis sind alle bisherigen Beispiele eigentlich nicht verwendbar, da Befehle, Nachrichten und Daten meist aus mehreren Bytes bestehen. Um eine sichere Übertragung zu gewährleisten und Übertragungsfehler zumindest zu erkennen wird im Allgemeinen ein Übertragungsprotokoll verwendet. Ein solches Protokoll ermöglicht es Beginn und Ende einer Übertragung zu ermitteln und auch Fehler in einer Datenübertragung zu erkennen.

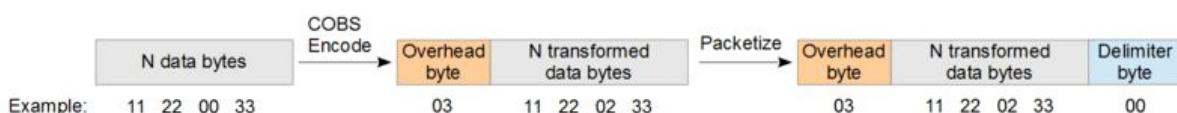
10.1.12.1 Consistent Overhead Byte Stuffing (COBS)

Consistent Overhead Byte Stuffing ist ein einfacher Algorithmus für die Übertragung eines Datenpaketes aus mehreren Bytes. Der Hauptbestandteil von COBS ist der **Delimiter** (z.B. NULL `0x00`), welcher das Ende eines Datenpaketes kennzeichnet. Das Problem besteht jetzt nur darin, dass das Datenpaket selbst diesen Delimiter nicht enthalten darf und man entsprechende Werte bei der Übermittlung durch einen Sender so vermeiden muss, dass der Empfänger in der Lage ist, diese wiederherzustellen.

COBS löst dieses Problem auf sehr einfache Art und Weise:

- Das erste (*Overhead*-) Byte bei Beginn der Übertragung beschreibt die Position des **nächsten** Delimiters in Form eines Offsets. Ob es sich bei dieser nächsten Position um das Endzeichen der Übertragung handelt, ist dem Empfänger zunächst nicht bekannt.
- Nur wenn an einer vorgegebenen Position wirklich ein Delimiter empfangen wird, weiß der Empfänger, dass das Datenpaket vollständig übertragen wurde.
- Steht an der vorgegebenen Position kein Delimiter, dann beinhaltet dieses Byte wiederum den Offset bis zum nächsten Delimiter. Der Empfänger ermittel die nächste Position und ersetzt das empfangene (*Offset*-)Byte mit dem Wert des ursprünglichen Datenpaketes.
- Dieses Spiel geht so lange weiter, bis wirklich ein Delimiter an einer vorgegebenen Position empfangen wird.
- Detektiert der Empfänger einen Delimiter an einer nicht vorgegebenen Stelle, war die Übertragung fehlerhaft und eine Maßnahme zur Fehlerbehandlung ist erforderlich.

Beispiele: (https://en.wikipedia.org/wiki/Consistent_Overhead_Bit_Stuffing)



Example	Unencoded data (hex)	Encoded with COBS (hex)
1	00	01 01 00
2	00 00	01 01 01 00
3	11 22 00 33	03 11 22 02 33 00
4	11 22 33 44	05 11 22 33 44 00
5	11 00 00 00	02 11 01 01 01 00
6	01 02 03 ... FD FE	FF 01 02 03 ... FD FE 00
7	00 01 02 ... FC FD FE	01 FF 01 02 ... FC FD FE 00
8	01 02 03 ... FD FE FF	FF 01 02 03 ... FD FE 02 FF 00
9	02 03 04 ... FE FF 00	FF 02 03 04 ... FE FF 01 01 00
10	03 04 05 ... FF 00 01	FE 03 04 05 ... FF 02 01 00

Der durch die beschriebene Methode erzeugte Overhead umfasst immer die zwei Bytes, welche sich aus dem **Delimiter** (NULL) und dem **Overhead Byte** zu Beginn eines Blocks von max. 254 Bytes, welches die Position des nächsten Delimiters angibt.

Für die Codierung von Datenpaketen, welche größer als 254 Bytes sind, müssen eventuell **mehrere Overhead Bytes** eingefügt werden. (8 und 9)

10.1.12.1.1 COBS Beispielcode

Ein Implementierung des Protokolls mit der Beschränkung auf Datenpakete die max. 253 Datenbytes umfassen, könnte wie folgt aussehen:

Zunächst braucht man für eine bidirektionale Kommunikation jeweils einen Buffer zum Empfangen und Senden der Daten. Dazu kommt noch Zähler, die angeben, wie viele Bytes schon empfangen, bzw. gesendet wurden. Auch für das Kodieren und Dekodieren der Datenpakete werden Variablen angelegt.

```
#define MAX_MSG_SIZE    ???      // max. number of bytes per message +overhead +delimiter
#define COBS_DELIMITER  0x00

uint8_t txBuffer[MAX_MSG_SIZE];
uint8_t rcBuffer[MAX_MSG_SIZE];
uint8_t txCnt, rcCnt;           // message length (receive/transmit)
uint8_t idx_msg;               // index for stepping through the message package
uint8_t off_next_delimiter;    // offset next delimiter position
```

Beim Codieren einer Nachricht vor dem Senden, kann man diese beim Index [1] in den Sendebuffer schreiben und so die erste Stelle [0] frei lassen, weil die ja für das Overhead Byte benötigt wird. Dann kann man das Datenpaket abarbeiten, indem man zuerst den Delimiter anhängt und sich dann von hinten durch die Daten arbeitet, um eventuell vorhandene Bytes, welche dem Delimiterwert entsprechen, mit den entsprechenden Offsets zu ersetzen.

```
void cobsEncode(void)
{
    idx_msg = txCnt + 1;                      // append delimiter
    txBuffer[idx_msg] = COBS_DELIMITER;
    off_next_delimiter = idx_msg;             // process backwards
    while(--idx_msg){
        if(txBuffer[idx_msg] == COBS_DELIMITER){
            txBuffer[idx_msg] = off_next_delimiter - idx_msg;
            off_next_delimiter = idx_msg;
        }
    }
    txBuffer[0] = off_next_delimiter;          // place overhead byte
}
```

Wenn es sich beim Empfang eines Datenpaketes beim zuletzt empfangenen Byte um einen Delimiter handelt, dann ist der Empfang abgeschlossen und das Paket kann decodiert werden

```
if(rcBuffer[rcCnt-1] == COBS_DELIMITER){ // package complete?
    cobsDecode();
}
```

Bei der Decodierung kann man die Nachricht vom Beginn her rekonstruieren, indem man die Daten an den als Offset angegebenen Positionen wieder mit dem Delimiterwert ersetzt. Vor dem Ersetzen muss man sich natürlich den nächsten Offset merken, damit man daraus die nächste Position berechnen kann.

```
void cobsDecode(void)
{
//    if(rcBuffer[idx_msg] > rcCnt-1) -->> ERROR
    if(rcBuffer[0] == rcCnt-1) return;           // no stuffing
    idx_msg = 0;
    while(idx_msg < (rcCnt-1)){
        off_next_delimiter = rcBuffer[idx_msg];
        rcBuffer[idx_msg] = COBS_DELIMITER;      // restore original value
        idx_msg += off_next_delimiter;           // calculate next position
    }
//    if(idx_msg > rcCnt-1) -->> ERROR
}
```

10.1.12.2 Checksumme (Prüfung der Fehlerfreiheit eines Datenpaketes)

Während der Übertragung eines Datenpaketes kann es zu Fehlern kommen, durch die ein oder mehrere Datenbytes verfälscht werden können. Um diese Fehler aufzuspüren kann man aus den Bytes eines Datenpaketes eine Checksumme berechnen anhand welcher der Empfänger die Plausibilität eines Datenpaketes überprüfen kann. Diese Checksumme wird dann einfach noch an das Datenpaket angehängt und mit diesem übertragen.

Im einfachsten Fall kann man die Werte der Paketbytes aufsummieren. Wenn die so ermittelte Checksumme auch nur ein Byte umfassen soll, dann ignoriert man Überläufe bei der Addition einfach und übermittelt quasi nur das Low-Byte der Summe, egal was für einen Umfang die eigentliche Summe hätte.

Der Empfänger muss seinerseits die Checksumme der empfangenen Bytes eines Datenpaketes bilden und diese mit der mitgelieferten Prüfsumme vergleichen. Stimmen die beiden überein, dann kann der Empfänger die Übertragung des Datenpaketes mit einer Bestätigung (*Acknowledge*) quittieren. Bei einer Abweichung signalisiert der Empfänger dies durch eine Fehlermeldung und die Übertragung des Paketes muss wiederholt werden.

//AUSARBEITEN

Die beschriebene sehr einfache Checksumme bildet natürlich nur eine relative Sicherheit, da sich mehrere Fehler innerhalb eines Paketes in der Checksumme neutralisieren könnten. Die Wahrscheinlichkeit dafür ist evtl. so gering, dass dies toleriert und durch weitere Maßnahmen abgefangen werden kann. Falls die Übertragungsqualität dauerhaft schlecht wäre, würde das auf jeden Fall auffallen, da sich in der Realität die Fehler niemals immer kompensieren werden.

In Fällen, wo die durch die beschriebenen Maßnahmen gewonnene relative Sicherheit nicht genügt, müssen natürlich ausgefeilte Algorithmen zur Anwendung kommen. **//LINK?**

10.1.12.3 Maßnahmen zur Fehlerbehandlung

Egal, ob ein Fehler von einem Protokoll wie z.B. COBS, oder über eine Prüfsumme entdeckt wurde, es ist eigentlich immer eine Fehlerbehandlung erforderlich.

//AUSARBEITEN

10.1.13 Kommunikation mit einem GPS Modul

//AUSARBEITEN

Auch GPS Module verwenden oft eine UART Schnittstelle um ihre Daten zu kommunizieren. Dabei werden Zeichenketten anhand der NMEA (*National Marine Electronics Association*) Spezifikationen verschickt.

Ein einfaches Demoprogramm zur Entschlüsselung dieser Zeichenketten findet sich auf der [Hochschulseite](#).

- GPS NMEA (*uCQ_THU_AddOn*)

10.2 SPI

//AUSARBEITEN

10.2.1 Demoprojekte SPI

Auf der [Hochschulseite](#) sind einige Demoprojekte für SPI verfügbar:

- Data-Logger auf microSD Karte
- MPU6000 Beschleunigungssensor und Gyroskop (*uCQ_THU_AddOn*)
- RV-3049 Real Time Clock Module

10.3 I2C

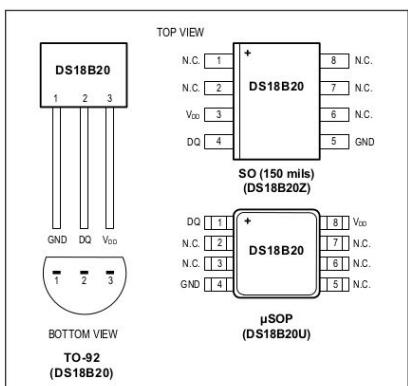
//AUSARBEITEN

10.3.1 Demoprojekte I2C

Auf der [Hochschulseite](#) sind einige Demoprojekte für I2C verfügbar:

- SHT21 Temperatur- und Feuchtigkeits-Sensor (*uCQ_THU_AddOn*)
- ...

10.4 1-Wire



//AUSARBEITEN

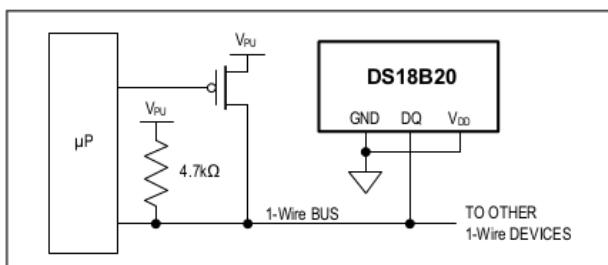


Figure 6. Supplying the Parasite-Powered DS18B20 During Temperature Conversions

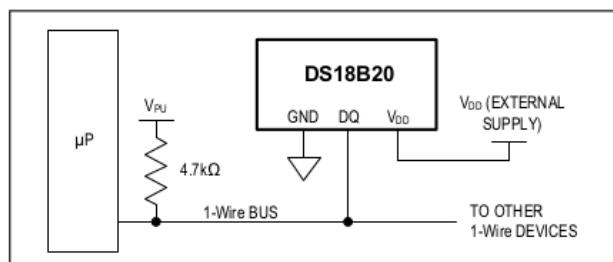


Figure 7. Powering the DS18B20 with an External Supply

//AUSARBEITEN

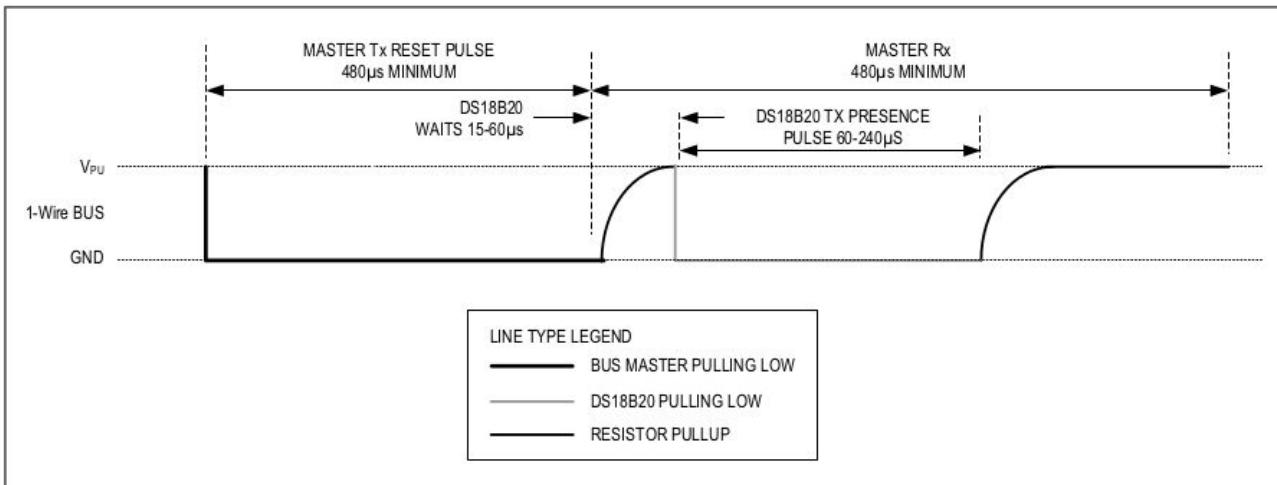
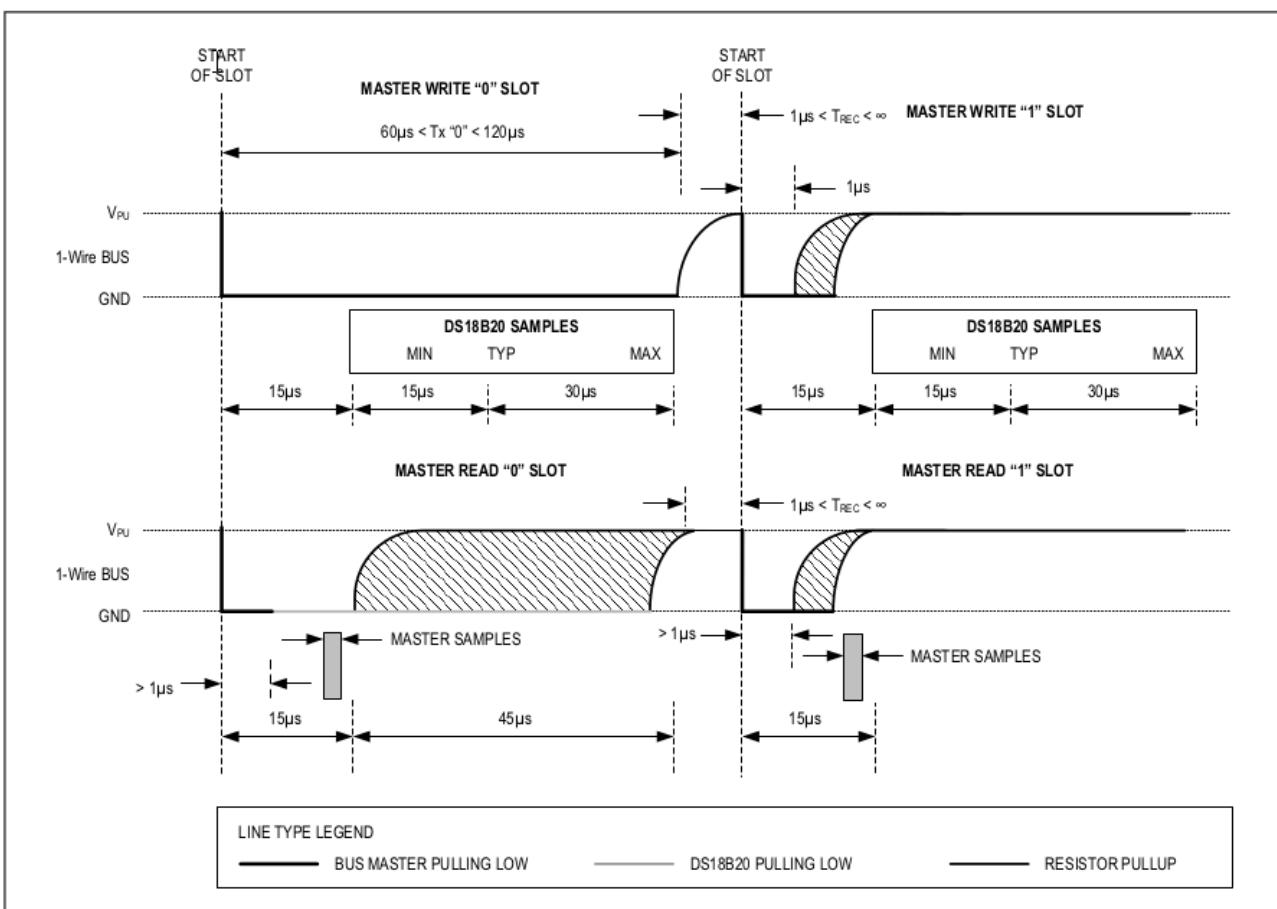


Figure 15. Initialization Timing

//AUSARBEITEN



10.4.1 Demoprojekt DS18x20 1-Wire Temperatur-Sensor

Auf der [Hochschulseite](#) ist auch ein Demoprojekt für einen 1-Wire Sensor vorhanden

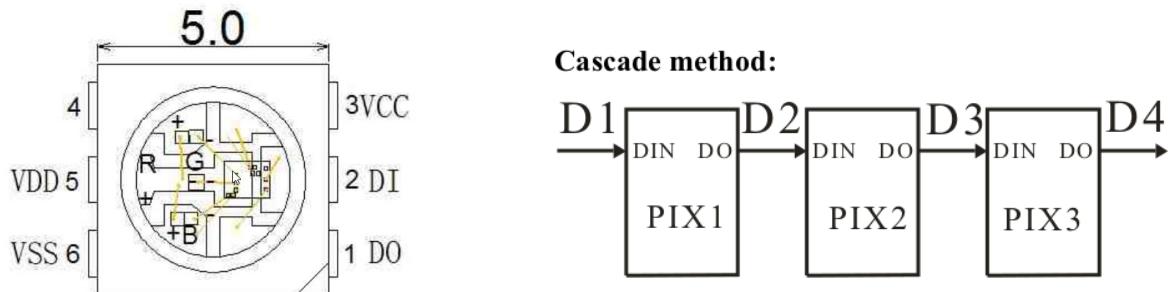
- DS18x20 Temperatur-Sensor (*uCQ_THU_AddOn*)

10.5 Wire Transmission Channel RGB WS2812 und APA102

Populäre RGB LEDs mit integriertem Controller nutzen eine serielle Datenübertragung, bei der sie einen Datenstrom erhalten, ihre eigenen Daten entnehmen und den Rest weiter zur nächsten LED in der Kette schicken. Herausragende Vertreter dieser Art sind die WS2812 und die APA102.

10.5.1 RGB WS2812 (1 Wire; nur Daten)

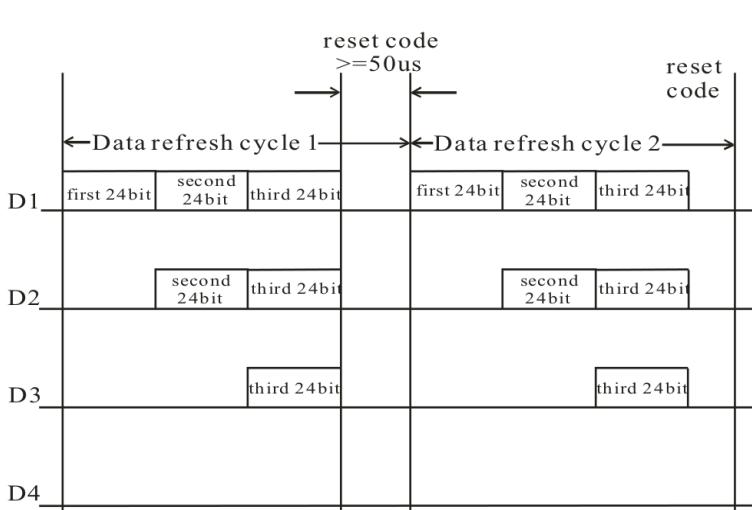
Bei den WS2812 handelt es sich um RGB LEDs, die sich zusammen mit einem intelligenten Controller in einem Gehäuse vom Typ 5050 oder PLCC6 befinden.



Diese LEDs lassen sich kaskadieren, indem man den Dateneingang DI der nächsten mit dem Datenausgang der vorhergehenden verbindet. Über die Datenleitung bekommt der integrierte Controller die Helligkeitsinformation für die drei LEDs (**Rot-Grün-Blau**). Diese umfasst 8 Bit für jede Farbe. Für eine komplette WS2812 also $3 \times 8\text{Bit} = 24\text{Bit}$.

Die Daten für die jeweilige WS2812 sind immer im ersten Paket enthalten, welches sie empfängt. Dieses erste empfangene Paket wird deshalb auch nicht an die nächste weiter geschickt. So wird das zweite empfangene zum ersten gesendeten Paket und damit auch zum ersten empfangenen für die nächste WS2812, die dieses wiederum nicht weiter schickt.

Data transmission method:



Wird nach der Übertragung der Daten für alle angeschlossenen WS2812 eine Pause von $\geq 50\mu\text{s}$ erkannt, dann wird das als **reset code** interpretiert und alle LEDs mit den neu erhaltenen Helligkeitswerten angesteuert.

Es ist auch möglich nur die ersten x WS2812 in einer Kette zu aktualisieren. Die nachfolgenden bekommen dann davon nichts mit.

Nach dem **reset code** kann wieder eine neue Datenübertragung starten.

Die Helligkeitsinformationen innerhalb der 24Bit sind wie in der folgenden Abbildung angeordnet. Die Übertragung beginnt mit dem MSB für die Helligkeit der grünen LED.

Composition of 24bit data:

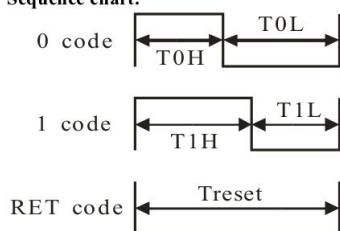
G7	G6	G5	G4	G3	G2	G1	G0	R7	R6	R5	R4	R3	R2	R1	R0	B7	B6	B5	B4	B3	B2	B1	B0
----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----

Note: Follow the order of GRB to sent data and the high bit sent at first.

10.5.1.1 WS2812 Timing

Das anspruchsvollste an der Implementierung eines Datenstromes zur Ansteuerung von WS2812 LEDs ist das für die Codierung einzelner Bits erforderliche Timing.

Sequence chart:



Data transfer time(TH+TL=1.25μs±600ns)

T0H	0 code ,high voltage time	0.35us	±150ns
T1H	1 code ,high voltage time	0.7us	±150ns
T0L	0 code , low voltage time	0.8us	±150ns
T1L	1 code ,low voltage time	0.6us	±150ns
RES	low voltage time	Above 50μs	

Die Übertragung eines Bits (*egal ob „0“ oder „1“*) beginnt immer mit einem High-Pegel. Die Dauer dieses High ist davon abhängig, ob eine „0“ oder eine „1“ übertragen werden soll. Ist der High-Pegel kurz, dann wird eine „0“ übertragen. Bei längerem High eine „1“.

„Kurze“ und „lange“ Pulsdauer für den High-Pegel sind hier allerdings für einen einfachen Mikrocontroller beides extrem kurze Zeiten (*Bruchteile von Mikrosekunden*) und können nicht mehr mit Hilfe von Delays oder Timern realisiert werden. Es benötigt direktes Timing durch Einfügen von **No-Operation** Befehlen, die genau einen Maschinencyklus des Controllers vergehen lassen.

Bei einem PIC18, der mit einer FOSC von 16MHz getaktet wird, dauert ein Befehlszyklus 4/16MHz = 250ns. Das ist die maximale Auflösung die man dann erreichen kann. Diese ermöglicht zumindest theoretisch High Pegel für „0“ von 1 x 250ns Dauer und für „1“ von 3 x 250ns = 750ns, welche gut innerhalb der obigen Spezifikationen liegen. Die Realisierung in „C“ ist leider nicht immer ganz so einfach, da einem der Compiler durch „Optimierungen“ dazwischen funken kann.

Betrachten wir zunächst nur die High-Pegel am Anfang jeder Übertragung eines einzelnen Bits.
Die die Dauer der nachfolgenden Low-Pegel scheint relativ unkritisch zu sein, sonst wäre, wie wir später noch sehen werden, die Ausgabe eines gültigen Datenstromes per Software nicht möglich.

Für die Ausgabe vom „0 code“ muss bei den gegebenen Bedingungen die Datenleitung auf High (1) gesetzt werden und beim nächsten Zyklus sofort wieder zurück auf Low (0).

Beim „1 code“ müssen **zwei Wartezyklen** in Form von Nop() Befehlen eingefügt werden, bevor der Rücksprung auf Low erfolgt. Das führt zum folgenden C Code (*linke Seite*), mit dem resultierenden Maschinencode im Disassambly (*Ausschnitt rechte Seite*).

```

void WS2812_wr(uint8_t * ptrColors,
                 uint8_t nrLEDs)
{
    uint8_t color;

    for(uint8_t i=0; i<(nrLEDs*3); i++) {
        color = *ptrColors++;
        if(!(color & 0x80)){ // bit7 == 0
            WS2812_DATA = 1; // high
            WS2812_DATA = 0; // low
        } else { // bit7 == 1
            WS2812_DATA = 1; // high
            Nop(); Nop(); // wait
            WS2812_DATA = 0; // low
            Nop(); // wait ?
        }
        if(!(color & 0x40)){ // bit6
            WS2812_DATA = 1;
            WS2812_DATA = 0;
        } else {
            WS2812_DATA = 1;
        }
    }
}

```

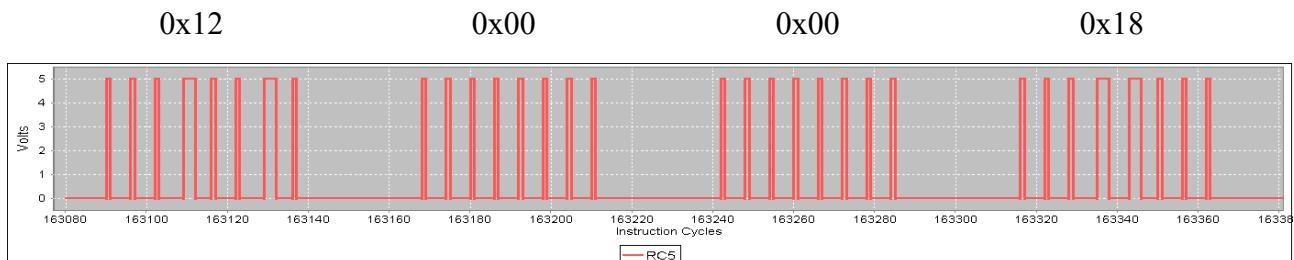
```

        if(!(color & 0x80)){ // bit7 == 0
0x99A: BTFSC color, 7, ACCESS
0x99C: BRA 0x9A4
        WS2812_DATA = 1; // high
0x99E: BSF LATC, 5, ACCESS
        WS2812_DATA = 0; // low
0x9A0: BCF LATC, 5, ACCESS
        } else { // bit7 == 1
0x9A2: BRA 0x9AE
        WS2812_DATA = 1; // high
0x9A4: BSF LATC, 5, ACCESS
        Nop(); Nop(); // wait
0x9A6: NOP
0x9A8: NOP
        WS2812_DATA = 0; // low
0x9AA: BCF LATC, 5, ACCESS
        Nop(); // wait ?
0x9AC: NOP
        }
        if(!(color & 0x40)){ // bit6 == 0
0x9AE: BTFSC color, 6, ACCESS
0x9B0: BRA 0x9B8

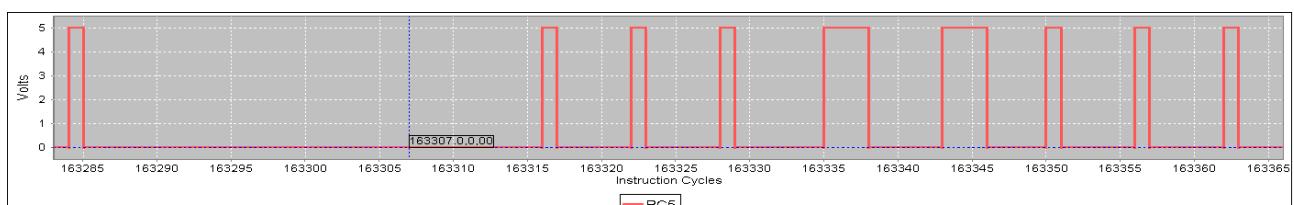
```

Die Dauer der **Low-Phasen** der Datenübertragung wurde bisher total außer Acht gelassen. Sie entsteht in der Hauptsache durch die Sprünge (**BRA**) für die Strukturen der **for-Schleife** und den **if-else** Bedingungen und überschreitet die in der Spezifikation angegebene Zeiten zum Teil deutlich.

Man kann die Dauer anhand des Disassembly und der Zyklenzahl der jeweiligen Befehle bestimmen oder sich den zeitlichen Verlauf an einem Oszilloskop anschauen. Wenn kein Oszilloskop vorhanden ist hilft aber auch der **Simulator** in MPLABX mit seinem **Logic-Analyzer** Fenster. Im unten abgebildeten Logic-Analyzer Fenster ist die Übertragung für 4 Bytes dargestellt.



Anhand der X-Achse kann man sehr einfach die Zyklen für die unterschiedlichen Phasen ermitteln. Die High-Phasen entsprechen genau den berechneten Werten und somit der Spezifikation. Die Low Phasen sind recht unterschiedlich.



- zwischen zwei Bytes sind es	31 Cycles	→ 7,75us (bei 16MHz Takt)
- zwischen zwei kurzen High (0 code)	5 Cycles	→ 1,25us
- zwischen „0“ und folgender „1“	6 Cycles	→ 1,50us
- zwischen „1“ und folgender „0“	4 Cycles	→ 1,00us
- zwischen zwei langen High (1 code)	5 Cycles	→ 1,25us

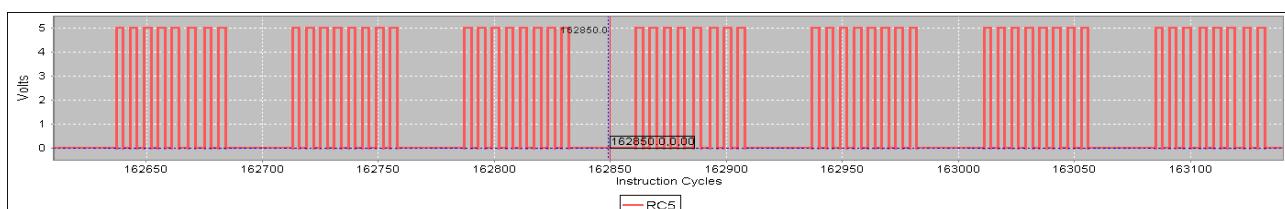
Alle Werte für die „**low voltage time**“ halten die Spezifikationen nicht ein, aber die Übertragung funktioniert trotzdem einwandfrei!

(Wichtig ist natürlich, dass alle deutlich unter den 50us für den reset code bleiben)

10.5.1.2 Vorsicht vor Compiler Optimierungen

Das in der **low voltage time** für die Ausgabe vom **1 code** eingefügte **Nop(); // wait ?** ist eigentlich total unsinnig, da die Zeit auch schon ohne lang genug wäre.

Ohne das **Nop()** schlägt aber die Optimierung des Compilers zu. Egal was man übertragen möchte, das Ergebnis ist immer gleich. Alle High-Phasen (*ob für 0 oder 1 code*) werden drei Zyklen lang.



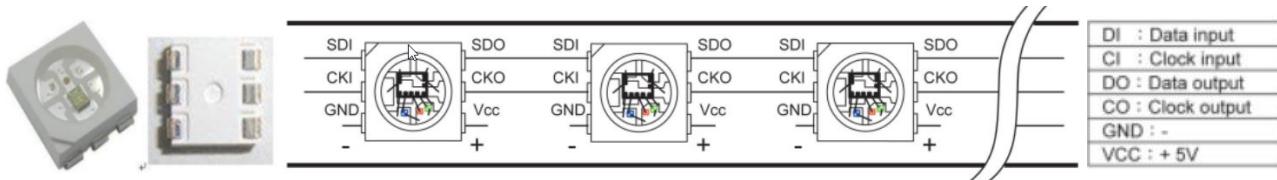
Wer es nicht glaubt, kann es sich im Disassembly anschauen, warum das so ist!

10.5.1.3 WS2812 Demo Projekt

Eine Configuration für eine WS2812 Demo ist innerhalb des *uCQ_THU_AddOn* Projektes enthalten und nennt sich *NEOPIXEL_24*.

10.5.2 RGB APA102 (2 Wire; Daten und Clock)

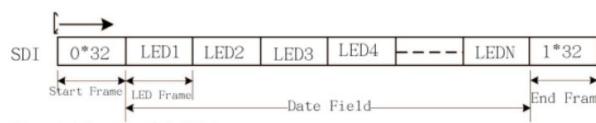
Die APA102 hat die gleiche Bauform wie die WS2812, aber ein zusätzliches Clock Signal, was eine synchrone Datenübermittlung ohne aufwendiges Timing ermöglicht. Das Taktsignal wird genau wie das Datensignal durch die APA102 durchgeschleift und von dieser aufbereitet.



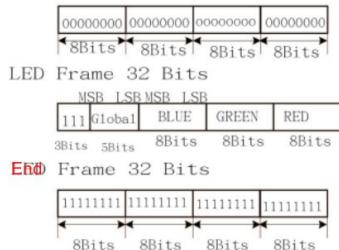
//AUSARBEITEN

(1) .cascading data structure

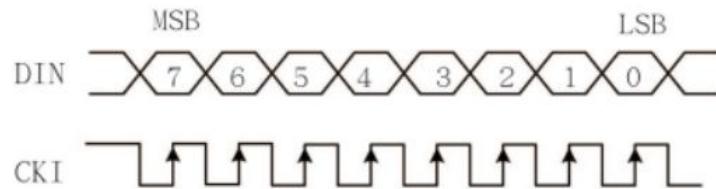
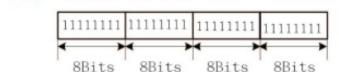
Tabdem N-LED



Start Frame 32 Bits



LED Frame 32 Bits



CKI



//AUSARBEITEN

Data MSB—	Duty Cycle
00000000	0/256(min)
00000001	1/256
00000010	2/256
...	
11111101	253/256
11111110	254/256
11111111	255/256(max)

DATA MSB ↔ LSB	Driving Current
00000	0/31
00001	1/31
00010	2/31
...	
11110	30/31
11111	31/31(max)

10.6 USB

//AUSARBEITEN

Die Kommunikation über USB ist wesentlich komplexer als RS232 und kann von „Newbees“ eigentlich nur bewältigt werden, wenn ein Grundgerüst zur Verfügung steht, das die USB Kommunikation schon beinhaltet und dem Programmierer Schnittstellen zur Verfügung stellt, über die er „seine“ Daten mit dem System austauschen kann.

USB Geräte sind nach ihrem Anwendungsgebiet in Geräteklassen eingeteilt.

- HID - Human Interface Devices
Eigentlich Tastaturen, Mäuse ... , kann aber auch sehr gut für Datenübertragungen von beliebigen anderen Sensoren verwendet werden
- CDC – Communication Device Class
Kann als „Virtueller COM Port“ wie ein RS232 Gerät verwendet werden.
- MSD – Mass Storage Devices
- UAD – USB Audio Device
- UVD – USB Video Device
- ...

Als Einstiegshilfe in die Programmierung von PIC18 USB Controllern kann das „Low-Pin-Count-USB-Demo-Board“ und dessen „User Guide“ benutzt werden. Für den Anfang eignen sich CDC und HID welche mit diesem Board realisiert werden können und die auch im User Guide zum Demo Board vorgestellt werden.

Im Labor sind mehrere dieser Kits vorhanden, die bei Bedarf ausgeliehen werden können. Leider ist die Dokumentation nicht auf dem aktuellen Stand und kann so das eine oder andere Problem verursachen. Eine mit zusätzlichen Kommentaren versehene Version des User Guide ist im auf dem IMM-Server verfügbar im Verzeichnis:

\\hs-ulm\\fs\\org\\Institute\\IMM\\INFO\\LIB\\uC_LIB\\DemoBoards-EvaluationKits\\MCHP__USB-LPCusbDK\\LPCUDK_UG_VSK.pdf

Alle Demo Projekte benutzen die „**Microchip Application Libraries**“, welche auf dem Entwicklungs-Rechner installiert (bzw. kopiert) sein müssen. Bitte immer die neueste Version von der Homepage des Herstellers benutzen.

Achtung:

**Die den Entwicklungskits beiliegende, bzw auf der Internetseite der Kits bereitgestellte Demo-Software ist durch zwischenzeitliche Änderungen der Bibliotheken oft völlig unbrauchbar !!!
Bitte immer die entsprechenden Projekte in den MLA suchen und verwenden !!!**

10.6.1 Basis CDC Projekt

Ein vereinfachtes CDC-Projekt kann man im [Microchip Forum \(www\)](http://www.microchip.com) finden, wenn man nach: *Simplified Microchip USB Demo projects (HID / CDC; PIC18)* sucht.

10.6.2 Basis HID Projekt

Ein vereinfachtes HID-Projekt kann man im [Microchip Forum \(www\)](http://www.microchip.com) finden, wenn man nach: *Simplified Microchip USB Demo projects (HID / CDC; PIC18)* sucht.

Auch ein einfaches Qt-Projekt für die PC seitige Verwendung sollte dort zu finden sein.

10.7 CAN (CANopen)

//AUSARBEITEN

Wie die weiter unten folgende USB Kommunikation ist auch die Kommunikation über CAN Bus etwas komplexer als das relativ einfache Benutzen der USART Schnittstelle.

Auch hier wird spezielle zusätzliche Hardware benötigt die bei Bedarf im Labortrakt T300 ausgeliehen werden kann.

Einfache Demoprojekte welche die Einarbeitung in die Verwendung des CANopen Standards erleichtern können, sind auch auf dem Server des Instituts für Medizintechnik und Mechatronik vorhanden.

Bla...

10.8 Kommunikation mit speziellen Sensoren

10.8.1 Ultraschallsensor SR04 /SR05

//AUSARBEITEN

Im Kapitel [9.2 Pulsdauer-Messungen mit dem Capture Modul](#) wurde eine weitere Art der Kommunikation mit einem Sensor vorgestellt.

Auch diese Beispiele sind auf der [Hochschulseite](#) verfügbar. (*uCQ_THU_AddOn*)

bla...

10.8.2 DCF77

*Auch dieses Beispiel ist auf der [Hochschulseite](#) verfügbar. (*uCQ_THU_AddOn*)*

bla...

11 Verschiedene Grundtechniken

11.1 IPO (EVA) Pattern / Input-Process-Output

Das einfache Entwurfsmuster, welches ein Programm in Eingabe-, Verarbeitungs- und Ausgabe-Prozesse aufteilt, wurde schon in Kapitel [3.4 Hello World III / IPO Pattern](#) eingeführt.

11.2 Time Slots

Das IPO Pattern kann so erweitert werden, dass die Abarbeitung in einem bestimmten Zeitfenster geschieht und dies auch überprüft wird. Die feste Zeitvorgabe für einen Durchlauf der Hauptschleife wird im folgenden Beispiel durch das Warten auf den Überlauf von Timer_0 realisiert, der unabhängig vom Hauptprogramm läuft und der entsprechend der gewünschten Zeit konfiguriert wurde. (*hier ~1/8 s*)

Das Warten auf den Ablauf der Zeit kann wahlweise am Ende, oder am Anfang der Hauptschleife, also vor „*input*“ oder nach „*output*“ erfolgen.

Die Überprüfung ob die Schleife innerhalb der vorgegebenen Zeit abgearbeitet wurde, wird durch die Abfrage des Timer-Überlaufs vor Beginn des Wartes realisiert. Bei Überschreitung der vorgegebenen Zeit wird im Beispiel ein Signal (*alle LEDs an*) ausgegeben und das Programm mittels einer Endlosschleife angehalten.

```
#define _XTAL_FREQ 1000000

#define MASK_TGL_L_LED 0b00000100
#define MASK_...

unsigned char buttons_ago, buttons_now, leds;

void __init(void)
{
    OSCCONbits.IRCF = IRCF_1MHZ;
    ...
    buttons_now = buttons_ago = 0b00010100; // initialize as not pressed
    leds = LATB & MASK_LEDs; // initialize leds variable

    T0CONbits.T0CS = 0; // instruction clock (fosc/4)
    T0CONbits.T08BIT = 1;
    T0CONbits.PSA = 0; // prescaler is assigned
    T0CONbits.T0PS = 0b110; // prescaler 1/128
    T0CONbits.TMR0ON = 1;
}

void main(void){
    while(1){
        // input
        ...
        // process
        ...
        __delay_ms(125); // simulate more code to execute
        // output
        ...
        leds ^= MASK_TGL_M_LEDS; // show timing via blinking middle LEDs
        LATB = leds;
        // timing
        if(INTCONbits.T0IF){ // timing error detection
            mALL_LED_ON(); // show error
            while(1){;} // and stop
        }
        while(!INTCONbits.T0IF){;} // wait (time slot)
        INTCONbits.T0IF = 0;
    }
}
```

11.3 State Machines – Endliche Automaten

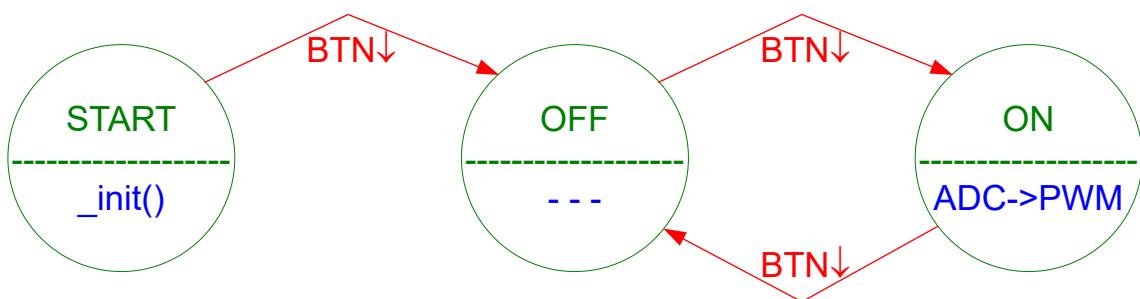
Eine weitere Möglichkeit ein Programm zu organisieren, ist die Einteilung in mögliche Zustände und Zustandsübergänge. Die Anzahl der möglichen **Zustände** muss dabei endlich sein, und alle möglichen **Übergänge** beschrieben werden. Zusätzlich werden **Aktionen** definiert, die während eines Zustands, beim Eintreten in einen Zustand, beim Verlassen eines Zustandes oder abhängig vom Übergang ausgeführt werden.

Beim Entwurf eines solchen Systems hilft ein Zustandsübergangsdiagramm (State diagram).

11.3.1 Einfache ON/OFF oder START/STOP State Machine

Ein einfaches Beispiel, mit nur zwei Zuständen wäre z.B. eine abschaltbare PWM, deren Periode über eine AD-Wandlung gesteuert wird. Bedienhinweise und das Ergebnis der AD-Wandlung sollen zusätzlich auf dem LCD angezeigt werden.

Das Übergangsdiagramm inklusive einer separaten Initialisierung des Systems beim Start (Einschalten der Stromversorgung) könnte folgendermaßen aussehen:



Beim Hochfahren des Systems werden zunächst alle benötigten Komponenten initialisiert und erst beim ersten Drücken der Taste des Encoders geht das System in die eigentliche Start/Stop State Machine über. In der bleibt es dann, bis die Stromversorgung abgeschaltet wird. Die Wechsel zwischen ON/OFF werden durch weitere Tastendrücke ausgelöst.

Zunächst legt man am besten ein Enum mit den möglichen Zuständen an und definiert eine Variable von diesem Typ. Die Deklaration des Enums sollte in umfangreicheren Projekten in einem Header erfolgen. Bei einem einfachen Projekt wie diesem, kann sie aber auch in der Main.c Datei stehen.

Die Variable kann global in der Main-Datei definiert werden, um einen einfachen Zugriff aus allen Funktionen der Datei zu haben. Hier sollte sie auch gleich mit dem passenden Wert initialisiert werden.

```
typedef enum{
    S_START,
    S_OFF,
    S_ON
} States_t;
...
State_t state = S_START;
```

Die Initialisierung des Systems packt man am besten wie gewöhnlich in eine Funktion mit passendem Namen, die man später am Anfang der main() Funktion aufruft.

```
void __init(void)
{
    OSCCONbits.IRCF = IRCF_4MHZ;

    ENC_BTN_TRI = INPUT_PIN; ENC_BTN_ANS = DIGITAL_IN;
//    SPEAKER_TRI = OUTPUT_PIN;-----controlled later in the appropriate states

    ANSELAbits.ANSA0 = 1;
    LATAbits.LATA0 = INPUT_PIN;
    OpenADC(ADC_FOSC_8 & ADC_LEFT_JUST & ADC_12_TAD,
            ADC_POTI & ADC_INT_OFF,
            ADC_TRIG_CCP5 & ADC_REF_VDD_VDD & ADC_REF_VDD_VSS);

// set up PWM out and analog in
    ADCON0bits.GO = 1; while (ADCON0bits.NOT_DONE) {}
    if (ADRESH >= 4) PR2 = ADRESH;
    else PR2 = 4;

    CCPR1L = PR2 >> 1;
    CCP1CONbits.CCP1M = 0b1100; // PWM mode
    CCP1CONbits.P1M = 0; // single
    CCPTMRS0bits.C1TSEL = 0; // CCP1-TMR2
    OpenTimer2(TIMER_INT_OFF & T2_PS_1_16 & T2_POST_1_1);

    LCD_Init();
    LCD_ConstTextOut(0,0," ON/OFF ");
    LCD_ConstTextOut(1,0," -> <- ");
}
```

In der **main()** Funktion wird nach Ausführung von ***__init()*** auf den ersten Übergang gewartet. Ist dieser erfolgt, geht das Programm in die Endlosschleife mit der ON/OFF Funktionalität.

Über die **switch-case** Abfrage springt das Programm in den entsprechenden Block und führt da zunächst (beim Eintritt) einmalige Aktionen aus. In diesem Beispiel wären das die Anpassung der LCD Anzeige und das Ein- bzw. Ausschalten des Ausgangs für den Lautsprecher.

Danach werden, die dem State entsprechenden dauerhaft zu wiederholenden Aktionen, so lange ausgeführt, bis die Bedingung für einen State-Wechsel vor liegen.

Bei Erkennung der Bedingung für einen Wechsel, wird die zum State gehörende Ausführung abgebrochen, der nächste State eingestellt und der momentane State verlassen (*break*). In den Folge-State wird dann beim nächsten Durchlauf der *switch-case* Abfrage innerhalb der *while(1)* Schleife, gesprungen.

```
void main(void)
{
    __init();
    while(!mGET_ENC_BTN()); // wait for button pressed
    state = S_OFF; // next state

    while(1){
        switch(state){
            case S_OFF:
                LCD_ConstTextOut(0,0," start ");
                LCD_ConstTextOut(1,0,"--> <--");
                SPEAKER_TRI = INPUT_PIN;
                while(mGET_ENC_BTN()){} // (ensure button is released)
                do{
                    ;
                }while(!mGET_ENC_BTN()); // until next state transition
                state = S_ON;
                break;
            case S_ON:
                LCD_ConstTextOut(0,0,"ADC      ");
                LCD_ConstTextOut(1,0," >stop< ");
        }
    }
}
```

```

        SPEAKER_TRI = OUTPUT_PIN;
        while(mGET_ENC_BTN()){}; // (ensure button is released)
        do{
            ADCON0bits.GO = 1; while(ADCON0bits.NOT_DONE){;}
            LCD_ValueOut_00(0,4,ADRESH,3);
            if(ADRESH >= 4){
                PR2 = ADRESH;
                CCPR1L = PR2 >> 1;
            }
        }while(!mGET_ENC_BTN()); // until next state transition
        state = S_OFF;
        break;
    default:
        state = S_OFF;
        break;
    }
} //while(1)
}

```

11.3.2 Einfache State Machine mit IPO Modell

Die im vorherigen Kapitel vorgestellte Vorgehensweise funktioniert nur bei sehr einfachen Problemen. Oft laufen in einem Programm mehrere verschiedene Zustandsmaschinen gleichzeitig. Weil ein Mikrocontroller-Programm aber sequenziell abgearbeitet wird, kann das Programm nicht in einem Block bleiben, der ausschließlich einen Zustand von einer dieser Zustandsmaschinen verarbeitet, bis die Bedingung zum Verlassen von diesem eintritt.

Das Programm muss so organisiert werden, dass die den Zuständen entsprechenden Blöcke zyklisch aufgerufen werden. Dabei ist es dann auch vorteilhaft die Verarbeitung der Übergänge und der sich wiederholenden Aktionen im Zustand zu trennen. Dadurch wird die Verarbeitung der Aktionen erleichtert, die nur einmalig beim Eintritt in den Zustand, oder beim Verlassen des Zustandes ausgeführt werden müssen.

```

while(1){
    if(mGET_ENC_BTN()){ //----- state transitions
        switch(state){
            case S_OFF:
                state = S_ON; // next state ON
                SPEAKER_TRI = OUTPUT_PIN; // switch on sound
                LCD_ConstTextOut(0,0,"ADC      ");
                LCD_ConstTextOut(1,0,">stop< ");
                break;
            case S_ON:
                state = S_OFF; // next state OFF
                SPEAKER_TRI = INPUT_PIN; // switch off sound
                LCD_ConstTextOut(0,0," start ");
                LCD_ConstTextOut(1,0,"--> <--");
                break;
            default: break;
        }
        while(mGET_ENC_BTN());
    }
    switch(state){ //----- state actions
        case S_OFF:
            break;
        case S_ON:
            ADCON0bits.GO = 1; while(ADCON0bits.NOT_DONE){;}
            if(ADRESH >= 4){
                LCD_ValueOut_00(0,4,ADRESH,3);
                PR2 = ADRESH;
                CCPR1L = PR2 >> 1;
            }
            break;
        default: break;
    }
} //while(1)
}

```

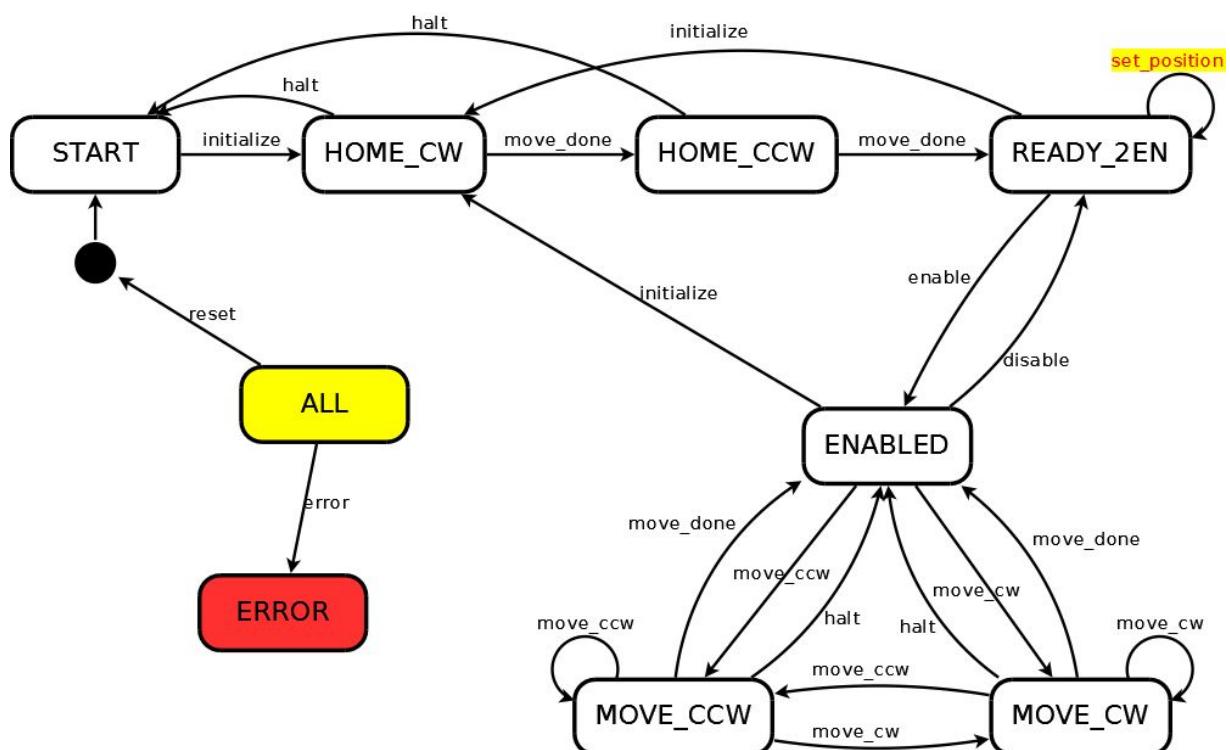
11.3.3 Komplexere (reale) Zustandsautomaten

Die folgende Abbildung zeigt eine Zustandsmaschine für einen Schrittmotorantrieb, mit dem verschiedene Positionen angefahren werden können. Zunächst eine rudimentäre Beschreibung des Ganzen:

Bevor der Motor von der Anwendung benutzt werden kann, muss er auf eine Referenzposition (Home) gefahren werden. Zum Finden dieser Referenzposition sind hier Bewegungen in beiden Drehrichtungen (CW → im Uhrzeigersinn, CCW → gegen den UZS) erforderlich. Ist die Referenzposition gefunden, so ist der Motor generell betriebsbereit (ready to enable), muss aber nochmals über einen Enable-Befehl in den bestromten Zustand gebracht werden (ENABLED)

In EABLED Zustand, können dann Befehle zum Ausführen von Schritten im Uhrzeigersinn, oder gegen den UZS gegeben werden.

Tritt während einer Bewegung ein Fehler auf, geht das System in den Zustand ERROR, der nur durch einen RESET-Befehl aufgehoben werden kann.



Im Gegensatz zu den ersten, einfachen Beispielen, gibt es jetzt nicht nur viel mehr Zustände, sondern auch viele verschiedene Ereignisse mit denen ein Zustandsübergang eingeleitet werden kann. Die Ereignisse können dabei an unterschiedlichen Stellen im Programm auftreten, möglicherweise sogar in einer Interrupt-Routine die bei einem Fehler ausgelöst wird.

Für die vielen möglichen Übergangsmöglichkeiten legt man jetzt genau wie schon vorher für Zustände ein eigenes Enum an.

Weil nicht jede der verschiedenen Anforderungen auf Zustandsänderungen in jedem Zustand erlaubt ist, kann man auch noch ein weiteres Enum mit möglichen Resultaten der Anforderungen auf eine Zustandsänderung anlegen.

...

```

typedef enum { // states for reference/move procedure
    STATE_UNCHANGED = 0, // used as return value
    STATE_START      = 1, // power on state
    STATE_HOME_CW,     // homing clockwise (down)
    STATE_HOME_CCW,    // homing counter-cw(up)
    STATE_READY_2EN,   // ready2enable (homing done)
    STATE_ENABLED,     // enabled
    STATE_MOVE_CW,     // move clockwise (down)
    STATE_MOVE_CCW,    // move counter-cw(up))
    STATE_ERROR       = 0xE0 // ???
} StepperState_t;
extern StepperState_t s1_state;

typedef enum{
    TRES_ILLEGAL_ACTION = -1,
    TRES_STATE_CHANGED = 1
} TransitionResult_t;

```

```

typedef enum{
    T_INITIALIZE      = 0,
    T_ENABLE,
    T_DISABLE,
    T_HALT,
    T_MOVE_CW,
    T_MOVE_CCW,
    T_MOVE_DONE,
    T_SET_POS,
    T_ERROR,
    T_RESET
} StepperTransition_t;

```

In komplexeren Programmen macht es wenig Sinn, die komplette Verarbeitung der Übergänge und der Aktionen in den Zuständen direkt als Code in der Main-Schleife anzulegen. Diese werden vielmehr in Funktionen gepackt, die bei Bedarf (Übergänge) oder zyklisch (Zustands-Aktionen) aufgerufen werden können.

Hier zunächst ein Beispiel für eine Funktion, die für Zustandsübergänge eines Schrittmotors (s1) mit dem Übergabeparameter um welche Art von Übergang es sich handeln soll aufgerufen wird. Als Rückgabeparameter hat die Funktion dann die Information, ob der Statuswechsel eingeleitet wurde.

```

TransitionResult_t s1StateTransition(StepperTransition_t action)
{
    TransitionResult_t result = TRES_STATE_CHANGED;

    if(action == T_ERROR){ // -----
        mS1_DIS(); s1_state = STATE_ERROR;
        return result;
    }
    if(action == T_RESET){ // -----
        mS1_DIS(); s1_state = STATE_START; s1_pos.actual = getS1pos();
        return result;
    }
// else
    switch (s1_state) {
        case STATE_START:
            if(action == T_INITIALIZE){ mS1_EN(); s1_state = STATE_HOME_CW; }
            else{ result = TRES_ILLEGAL_ACTION; }
            break;
        case STATE_HOME_CW:
            if(action == T_MOVE_DONE){ s1_state = STATE_HOME_CCW; }
            else if(action == T_HALT){ mS1_DIS(); s1_state = STATE_START; }
            else{ result = TRES_ILLEGAL_ACTION; }
            break;
        case STATE_HOME_CCW:
            if(action == T_MOVE_DONE){ mS1_DIS(); s1_state = STATE_READY_2EN; }
            ... break;
        case STATE_READY_2EN:
            ...
            ... break;
        case STATE_ERROR:
            result = TRES_ILLEGAL_ACTION; break;
        default:
            result = TRES_ILLEGAL_ACTION; break;
    }
    return result;
}

```

In die Funktion zur Anforderung eines Statuswechsel werden jetzt alle Übergänge für alle Zustände aus dem Zustandsübergangsdiagramm eingetragen. Für alle Übergänge, die im Diagramm nicht existieren wird das Ergebnis zurück gemeldet, dass diese Aktion nicht existiert.

Die Schrittmotoren dieses Beispiels sind nicht sehr dynamisch, und werden über einen Timer gesteuert. Abhängig vom Zustand in dem sich der Motor befindet, wird dann jeweils ein Schritt ausgeführt oder nicht. Bei der Verarbeitung der Schritte kann es, z.B. beim Erreichen einer vorgegebenen Postion, zu Zustandswechseln kommen.

```
StepperState_t s1StateTask()
{
    switch(s1_state){
        case STATE_START:
            s1StateTransition(T_ERROR); // invalid call of function s1Task() ?
            break;
        case STATE_HOME_CW:      // -----length plus
            s1_pos.prior = get_S1_pos();
            mS1_STEP_CW();
            s1_pos.actual = get_S1_pos();
            if(s1_pos.actual > s1_pos.prior + STEP_WIN){
                s1_state = STATE_HOME_CCW;
            }
            else{
                return STATE_UNCHANGED;
            }
            break;
        case STATE_HOME_CCW:     // -----length minus
            ...
        case STATE_READY_2EN:
            return STATE_UNCHANGED; // break;
        case STATE_ENABLED:
            return STATE_UNCHANGED; // break;
        case STATE_MOVE_CW:     // -----length plus
            s1_pos.prior = getS1pos();
            mS1StepCW();
            s1_pos.actual = getS1pos();
            if(s1_pos.actual == s1_pos.prior){ // not moved?
                return STATE_UNCHANGED; //TODO error counter...
            }
            if(s1_pos.actual < s1_pos.prior){ // +/- position?
                s1_pos.actual = -s1_pos.actual;
            }
            if(s1_pos.actual == s1_pos.target){ // target position reached
                s1StateTransition(T_MOVE_DONE);
            }else if(s1_pos.actual > s1_pos.target){ // target position passed
                s1StateTransition(T_MOVE_CCW); // U-turn
            }else{
                return STATE_UNCHANGED;
            }
            break;
        case STATE_MOVE_CCW:    // -----length minus
            ...
        case STATE_ERROR:       // ----- stepper in error state
            return STATE_UNCHANGED; //break;
        default: s1StateTransition(T_ERROR);
    }
    return s1_state;
}
```

11.3.4 Zustandsmaschine mit Funktionszeigern

Anstelle der **switch-case** Verzweigung in der Funktion zur Bearbeitung der State-Aktionen kann man auch für jeden Zustand eine eigene Funktion schreiben, und einen zusätzlichen Pointer anlegen, der auf die für den aktuellen Zustand gültige Funktion zeigt.

Das Enum mit den Zuständen und die Variable, in welcher der momentane Zustand abgelegt ist, werden so überflüssig. In der Funktion welche die Übergänge abhandelt, wird einfach der Funktionszeiger auf die entsprechende Funktion gelegt.

Die Behandlung der erforderlichen Aktionen in den Zuständen, geschieht dann durch den Aufruf der entsprechenden Funktion über den Funktionszeiger.

11.3.5 Übung Menü → Uhr mit Einstelfunktion

Ein Anwendung mit einem Menü könnte eine Uhr sein. Die Einstellung der Zeit kann sehr einfach mit dem Encoder realisiert werden, wenn man die integrierte Taste verwendet um zwischen verschiedenen Menü Einträgen für die Einstellung von Stunden, Minuten und Sekunden sowie der laufenden Uhrzeit umzuschalten.

Als kleine Übung zur Vertiefung der vorangegangenen Kapitel kann man das folgende Programm auf das System umstellen, wie es bei der komplexeren State Machine benutzt wurde, oder sogar die Variante mit Funktionszeiger implementieren.

```
typedef enum{
    CM_HR = 0,
    CM_MIN,
    CM_SEC,
    CM_CLOCK
}CLOCKMENU;

// -----Clock CCP
#define SEC_IR      PIE2bits.CCP2IE && PIR2bits.CCP2IF
#define mSEC_IR_EN() PIE2bits.CCP2IE = 1
#define mSEC_IR_DIS() PIE2bits.CCP2IE = 0
#define mSEC_IR_CLR() PIR2bits.CCP2IF = 0

void setHours(unsigned char* hr);
void setMinutes(unsigned char* min);
void setSeconds(unsigned char* sec);
void Clock(unsigned char* hr, unsigned char* min, unsigned char* sec);
```

```
CLOCKMENU menu;
unsigned char hours, minutes, seconds;

void __init(void)
{
    OSCCONbits.IRCF = IRCF_2MHZ; OSCTUNEbits.PLLEN = 0;           // -> 2MHz

    // encoder setup
    ENC_INT_TRI = INPUT_PIN; ENC_INT_ANS = DIGITAL_PIN;
    ENC_DIR_TRI = INPUT_PIN; ENC_DIR_ANS = DIGITAL_PIN;
    ENC_BTN_TRI = INPUT_PIN; ENC_BTN_ANS = DIGITAL_PIN;

    OpenTimer1(TIMER_INT_OFF & T5_16BIT_RW & T1_SOURCE_FOSC_4 &
               T1_PS_1_8 & T1_OSC1EN_OFF & T1_SYNC_EXT_OFF,
               TIMER_GATE_OFF);

    CCPTMRS0bits.C2TSEL = 0;           // timer <-> ccp module (CCP2 / TMR1)
    CCPR2 = 62500;                   // Fosc/4 / prescaler = 500kHz /8
    CCP2CONbits.CCP2M = 0b1011;       // Compare Mode with Special Event Trigger

    flags.all = 0;
    hours = minutes = seconds = 0;
    menu = CM_HR;                  // Begin with adjusting hours

    LCD_Init();
    LCD_ConstTextOut(0,0,"hrs. +- ");
    LCD_ConstTextOut(1,0,"00:00:00");

    mENC_IR_RST();
    mENC_IR_EN();
    INTCONbits.PEIE = 1;
    INTCONbits.GIE = 1;
}
```

```

void main(void)
{
    __init();

    while(1) {
        switch (menu) {
            case CM_HR:
                setHours(&hours);
                menu++;
                break;
            case CM_MIN:
                setMinutes(&minutes);
                menu++;
                break;
            case CM_SEC:
                setSeconds(&seconds);
                menu++;
                break;
            case CM_CLOCK:
                mENC_IR_DIS();
                mSEC_IR_EN();
                Clock(&hours,&minutes,&seconds);
                mSEC_IR_DIS();
                mENC_IR_EN();
                menu = CM_HR;
                break;
            default:
                menu = CM_CLOCK;
                break;
        }
    }
}

```

```

void setHours(unsigned char* hr)
{
    LCD_ConstTextOut(0,0,"hrs. +- ");

    while(!mGET_ENC_BTN()) {
        if(flags.encUp) {
            if(++*hr >= 24) *hr = 0;
            LCD_ValueOut_00(1,0,*hr,2);
            flags.encUp = 0;
        }
        if(flags.encDown) {
            if(--*hr >= 24) *hr = 23;
            LCD_ValueOut_00(1,0,*hr,2);
            flags.encDown = 0;
        }
    }
    while(mGET_ENC_BTN()) {}
}

```

...

```

void setMinutes(unsigned char* min)
{
    LCD_ConstTextOut(0,0,"min. +- ");
    ...
}

```

...

```

void setSeconds(unsigned char* sec)
{
    LCD_ConstTextOut(0,0,"sec. +- ");
    ...
}

```

```

void Clock(unsigned char* hr, unsigned char* min, unsigned char* sec)
{
    LCD_ConstTextOut(0, 0, "Time:   ");

    while (!mGET_ENC_BTN()) {
        if (flags.newSec) {
            if (++*sec >= 60) {
                *sec = 0;
                if (++*min >= 60) {
                    *min = 0;
                    if (++*hr >= 24) {
                        *hr = 0;
                    }
                    LCD_ValueOut_00(1, 0, *hr, 2);
                }
                LCD_ValueOut_00(1, 3, *min, 2);
            }
            LCD_ValueOut_00(1, 6, *sec, 2);
            flags.newSec = 0;
        }
    }
    while (mGET_ENC_BTN()) { ; }
}

```

```

void __interrupt(high_priority) high_isr(void)
{
    if(ENC_IR){
        if(ENC_DIR == ENC_DIR_UP){
            flags.encUp = 1;
        } else {
            flags.encDown = 1;
        }

        mENC_IR_RST();           // clear flag and toggle edge if necessary
        return;
    }

    if(SEC_IR){
        // if(flags.newSec) -> not yet processed ???
        flags.newSec = 1;
        mSEC_IR_CLR();
        return;
    }
    while(1);                  // (detect unexpected IR sources)
}

```

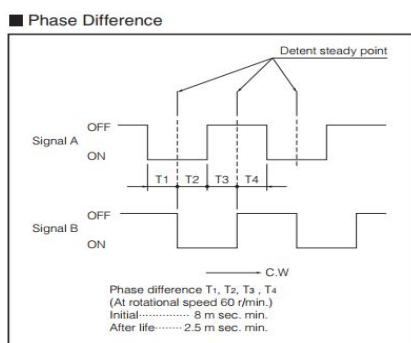
11.4 State Machine zur Auswertung von Drehgebern

Die im Interrupt Kapitel eingeführte Methode funktioniert sehr gut auf der vorhandenen Hardware, ist aber im Allgemeinen nicht immer zuverlässig und auch nicht immer sinnvoll.

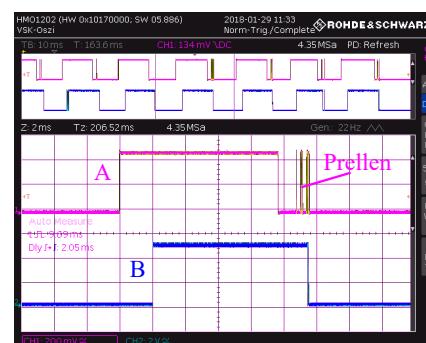
Eine andere Methode der Auswertung beruht auf einer State Machine die von einem Timer-Interrupt aufgerufen wird. Damit keine Änderung verloren geht, muss zunächst der kleinste Zeitabstand für eine Änderung an den Encoder Signalen A und B ermittelt werden.

11.4.1 Ermittlung der erforderlichen Abtastfrequenz

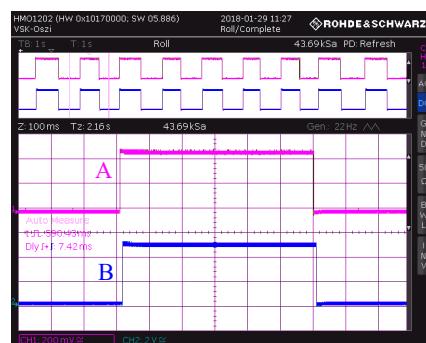
Die Abbildungen in den Datenblättern der Encoder sind dafür nicht immer sehr aussagekräftig, weil die Darstellung idealisiert wurde. Wenn möglich, sollte man den Signalverlauf mit einem Oszilloskop **nachmessen**. Bei der schnellsten Betätigung die zuverlässig detektiert werden soll!



Datenblatt EVEQDBRL416B



Pulsdauer ~10ms (sehr schnell >6r/s)



Pulsdauer ~600ms (sehr langsam ~0,1r/s)

Die Abbildung aus dem Datenblatt gaukelt einem vor, dass die Flanken von Signal B genau auf den Rastpunkten und genau in der Mitte der Pulse von Signal A liegen. Ob die Signalwechsel von Spur B genau im Rastpunkt liegen, kann man leicht ermitteln, wenn man etwas „wackelt“. Hier: **NEIN!**

Die Abbildung im Datenblatt soll für 60 r/min, oder eine Umdrehung pro Sekunde gelten. Dieser Drehgeber hat 32 und Rastpunkte und 16 Pulse (oder 8 Perioden), pro Umdrehung. Bei 1 r/s beträgt die Pulsdauer also $1/16s = 62,5\text{ms}$. (*EVEP Typ → 8 Pulse → 125ms*). Die Phasendifferenzen $T_1..T_4$ können also unmöglich alle 2,5..8ms sein!

Durch die beim EVEQ sehr harten Rasten ist, beim manuellen Drehen in der Realität, auch keine gleichförmige Geschwindigkeit möglich. Die Bewegung wird, bedingt durch die Rastpunkte, eher ruckartig ausgeführt. Das Endresultat davon ist, dass sich auch bei sehr unterschiedlichen Drehgeschwindigkeiten die Phasendifferenzen nicht allzu sehr unterscheiden.

Bei sehr schnellem Drehen (~6r/s), ist der zeitliche Versatz der Spuren ~2ms. Bei sehr langsamer Drehung (1/10 r/s), aber nicht 60 mal höher, sondern auch nur~7,4ms (< 4mal).

Damit auch bei schnellem Drehen eine zuverlässige Detektion der Zustandswechsel möglich ist, kann man innerhalb des minimal gemessenen Wert von 2ms für die Phasenverschiebung zwei mal abtasten und kommt so auf eine Abtastfrequenz von **1/1ms = 1000Hz**.

Wenn kein Oszilloskop zur Verfügung steht, dann kann man für die handbetätigten Drehgeber mit nicht allzu hoher Auflösung auch mit dem groben Richtwert von 1000 Abtastungen pro Sekunde beginnen und testen, ob Abtast- oder Verarbeitungs- Fehler auftreten.

Bei der Verwendung von Encodern als Eingabeelement kann es unkritisch sein, wenn Schritte verloren gehen, nur weil der Benutzer wie ein Irrer am Rad dreht, wenn er den eingestellten Wert als Feedback bekommt. Das Feedback kann dabei beispielsweise eine Anzeige des Wertes, oder die sich verändernde Lautstärke eines Telekommunikationsgerätes sein. Der Benutzer wird einfach so lange drehen, bis er mit dem Ergebnis zufrieden ist. Dreht er allerdings „normal“ und bekommt kein adäquates Feedback, dann wird er mit dem Gerät absolut nicht zufrieden sein!

11.4.2 Signalauswertung und Fehlererkennung

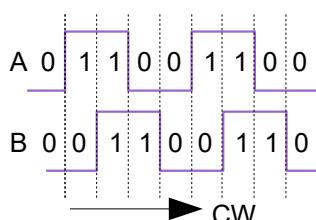
Für die Auswertung und Fehlererkennung gibt es verschiedene optimierte und bewährte Verfahren, die oft auch universell für verschiedene Encoder-Typen einsetzbar sind.

Hier soll ein möglicherweise nicht 100% optimiertes, aber dafür hoffentlich noch einigermaßen nachvollziehbares Verfahren vorgestellt werden.

11.4.2.1 Quadratursignale - Gray-Code

Zunächst haben die Mehrzahl der Encoder mit zwei Signalen (A/B) gemeinsam, dass die Signale um 90° verschobenen Quadratursignalen entsprechen und die Zustandswechsel einem **Gray-Code**. Das bedeutet, dass sich zu einem Zeitpunkt immer nur ein Signal ändern kann.

Die 90° Phasenverschiebung ist bei Encodern mit Rasten, die manuell bedient werden, nicht der Fall, aber das ist für die folgende Auswertung belanglos.



Quadratursignale
Gray-Code

Das Signalpaar der Encoder Ausgänge A und B kann vier verschiedene Zustände annehmen. Betrachtet man die Signale als Binärzahl mit zwei Stellen, dann wäre A 2^1 und B 2^0 und die sich beim Drehen im Uhrzeigersinn ergebende Zustandsfolge 00-10-11-01-00-10-11-01...

In der Signalfolge ändert sich immer nur der Zustand eines Signals. Jeder Zustand hat abhängig von der Drehrichtung nur einen „gültigen“ Folgezustand.

11.4.2.2 Fehlererkennung und Richtungsauswertung beim Graycode

Detektiert man eine Änderung beider Signale, dann liegt ein **Fehler** vor. Die Abtastfrequenz ist zu niedrig gewählt oder die Signale wurden durch äußere Störungen verfälscht.

Die **Drehrichtung** ergibt sich aus zwei aufeinanderfolgenden Zustandsvektoren. Der zukünftige „Folgezustand“ ist natürlich nicht vorhersagbar, aber der momentane Zustand ist natürlich auch der Folgezustand des vorherigen. Man muss also die Zustände bis zum nächsten Wechsel speichern.

Aus den bisherigen Erkenntnissen kann man für alle möglichen Übergänge eine Tabelle erstellen

Zustand vorher		Zustand aktuell		Drehrichtung / Fehler	Zustand vorher		Zustand aktuell		Drehrichtung / Fehler
A-	B-	A	B		A-	B-	A	B	
0	0	0	0	0	1	0	0	0	CCW / down
0	0	0	1	CCW / down	1	0	0	1	ERROR
0	0	1	0	CW / up	1	0	1	0	0
0	0	1	1	ERROR	1	0	1	1	CW / up
0	1	0	0	CW / up	1	1	0	0	ERROR
0	1	0	1	0	1	1	0	1	CW / up
0	1	1	0	ERROR	1	1	1	0	CCW / down
0	1	1	1	CCW / down	1	1	1	1	0

Die Auswertung beginnt man am besten mit einer Abfrage, ob sich die **Zustände geändert** haben.

Auch die Abfrage, ob sich **beide Zustände geändert** haben, zur **Erkennung von Fehlern** ist über eine EXKLUSIV-ODER Verknüpfung sehr einfach zu realisieren und kann als Nächstes erfolgen. (falls sich die Zustände überhaupt geändert haben)

Wenn kein Fehler detektiert wurde, dann kann letztendlich die Richtungsauswertung mittels entsprechender Abfragen durchgeführt werden.

11.4.2.3 Verschiedene Encodertypen

Puls/Detent bla...

11.4.3 Beispielcode

11.4.3.1 States

```
typedef enum {
    ENC_DOWN     = -4,    // ENC_DOWN for 1-pulse/2-detent (toggle)
    ENC_d3        = -3,
    ENC_d2        = -2,    // ENC_DOWN for 1-pulse/1-detent (puls)
    ENC_d1        = -1,    // ENC_DOWN for quadrature encoder
    ENC_IDLE      = 0,
    ENC_u1        = 1,     // ENC_UP for quadrature encoder
    ENC_u2        = 2,     // ENC_UP for 1-pulse/1-detent (puls)
    ENC_u3        = 3,
    ENC_UP        = 4     // ENC_UP for 1-pulse/2-detent (toggle)
} ENC_STATE;
ENC_STATE encState;
```

11.4.3.2 Interrupt Code

```
unsigned char encNow, encAgo;

//-----
if(INP_POLL_IR){                                // timer IR 1000Hz
    if (flags.firstLoop){                         // first loop
        encAgo = 2 * ENC_A + ENC_B;
        flags.firstLoop = 0;
    } else{
        encNow = 2 * ENC_A + ENC_B;
        if (encNow != encAgo){                    // signals changed?
            if((encNow ^ encAgo) == 0b11){        // both signals changed -> ERROR
                flags.encCERR = 1;               // <- breakpoint here
            }
            else {                           // only one signal changed -> OK
                switch(encAgo){                  // up 00-10-11-01-00...
                    case 0b00:                   // down 00-01-11-10-00...
                        if(encNow == 0b10){encState++;}
                        else{encState--;} break;
                    case 0b01:
                        if(encNow == 0b00){encState++;}
                        else{encState--;} break;
                    case 0b10:
                        if(encNow == 0b11){encState++;}
                        else{encState--;} break;
                    case 0b11:
                        if(encNow == 0b01){encState++;}
                        else{encState--;} break;
                }
                if(encState == ENC_UP){       // last action not handled already!!!
                    if(flags.encUp){
                        flags.encOERR = 1; // <- breakpoint here
                    }
                    flags.encUp = 1; encState = ENC_IDLE;
                }
                else if(encState == ENC_DOWN){
                    if(flags.encDown){
                        flags.encOERR = 1; // <- breakpoint here
                    }
                    flags.encDown = 1; encState = ENC_IDLE;
                }
            }
            encAgo = encNow;
        }
    }
    mINP_POLL_IR_CLR();
}
```

```
    return;  
}
```

11.4.3.3 Hauptprogramm

```
void EncoderPolling(void)  
{  
    short value = 0;  
  
    //----- init()  
    OSCCONbits.IRCF = IRCF_16MHZ;  
  
    // use TMR1 and CCP3 for input polling  
    OpenTimer1(TIMER_INT_OFF & T1_16BIT_RW & T1_SOURCE_FOSC &  
               T1_PS_1_1 & T1_OSC1EN_OFF & T1_SYNC_EXT_OFF,  
               TIMER_GATE_OFF);  
    OpenECompare3(COM_INT_ON & ECOM_TRIG_SEVNT & ECCP_3_SEL_TMR12, 16000);  
  
    LCD_Init();  
    LCD_ConstTextOut(0,0,"Enc. Poll");  
    LCD_ConstTextOut(1,0," 0 ");  
  
    ENC_A_TRI = ENC_B_TRI = INPUT_PIN;// encoder setup  
    ENC_A_ANS = ENC_B_ANS = DIGITAL_IN;  
  
    flags.all = 0;  
    flags.firstLoop = 1;  
  
    mINP_POLL_IR_EN();  
    INTCONbits.PEIE = 1;  
    INTCONbits.GIE = 1;  
    //----- main()  
    while(1){  
        if(flags.encUp){  
            value++;  
            LCD_ConstTextOut(1,0,"      "); // delete old value  
            LCD_ValueOut(1,1,value);  
            flags.encUp = 0;  
        }  
        if(flags.encDown){  
            value--;  
            LCD_ConstTextOut(1,0,"      "); // delete old value  
            LCD_ValueOut(1,1,value);  
            flags.encDown = 0;  
        }  
    }  
}
```

11.4.4 Entprellung per Software

Passiert automatisch bla ...

11.4.5 Bisher vernachlässigt ...

Der IDLE State kann von der Ruheposition abweichen, bla ...

11.5 Speicherkarten

AN1045a ...

11.6

11.7 Bootloader (MCHP AN1310)

Aktuelle Mikrocontroller sind im allgemeinen in der Lage ihren Programmspeicher selber zu überschreiben. Diese Funktionalität nutzt man, um die Firmware in den Controllern auch ohne Programmiergerät ändern zu können. Vorwiegend dann, wenn die Firmware von Geräten im Feld, beispielsweise bei oder von einem Kunden auf einen neuen Stand gebracht werden soll.

Ein Teil des Programmspeichers muss dafür einen sogenannten BOOTLOADER enthalten, der die neue Firmware über eine Schnittstelle empfängt und in den Programmspeicher schreibt. Der Bootloader selber muss natürlich in der Regel vorher mit einem „normalen“ Programmiergerät wie z.B. dem PICkit 3 auf den Mikrocontroller geschrieben worden sein.

Beim „Arduino“ wird dieses System auch für die Entwicklung benutzt. Das hat allerdings den Nachteil, dass man sehr eingeschränkte Debug-Möglichkeiten hat und nicht den Umfang, den schon ein einfaches Tool wie das PICkit 3 bietet. Man kann mehr oder weniger nur das Programm blind aufspielen und hoffen, dass es funktioniert. Als Kontrolle werden hier oft LEDs oder Strings benutzt, die bei bestimmten Aktivitäten eingeschaltet oder über die serielle Schnittstelle gesendet werden.

Die Application Note **AN1310 "High-Speed Bootloader for PIC16 and PIC18 Devices"** von Microchip beschreibt einen solchen Bootloader für die USART/COM Schnittstelle. Code für Mikrocontroller und eine passende PC-Software inklusive Sourcecode (*Qt*) können von der Webseite des Herstellers herunter geladen werden. (-><http://lmgtfy.com/?q=AN1310#>)

11.7.1 Bootloader Firmware

Das in der Version v1.05r enthaltene Mikrocontroller Projekt ist leider noch für die alte MPLAB IDE (ohne X). [AN1310_uCQ.zip](#) ([hs-ulm.de/schilling/...](http://hs-ulm.de/schilling/)) enthält eine angepasste Version für die **MPLAB-X IDE** und das uCQ Board.

Vorgefertigte Bootloader- [*.hex](#) Dateien können mit einem Programmiergerät (**PICkit 3**) direkt auf den entsprechenden Controller programmiert werden. Für die Programmierung solcher hex-Files eignet sich die Software **MPLAB IPE** welche zusammen mit der MPLABX IDE installiert wird.

11.7.1.1 Verwendeter Speicherbereich

Die Bootloader Firmware bietet einige Konfigurationsmöglichkeiten. Beispielsweise kann ein Bereich am Anfang oder am Ende des Programmspeichers benutzt werden. In der vorliegenden Konfiguration ist der Bootloader so eingestellt, dass er am Ende des ROMs liegt. Somit bleiben die Interrupt-Vektoren am Anfang des Speichers direkt für die Application verfügbar und es gibt keine Behinderungen/Verzögerungen bei der Interrupt Verarbeitung.

Weil am Resetvektor der Sprung auf den Bootloader-Code stehen muss, kann der der Application nicht an seiner eigentlichen Stelle stehen und muss angepasst werden. Das wird von der Bootloader-Software automatisch erledigt, wenn es sich bei der ersten Anweisung der Application um einen GOTO Sprung handelt. (ist beim C18 und XC8 Compiler gewährleistet) . Der Sprung wird von der Bootloader-Software einfach direkt vor den Bootloader-Code am Ende verschoben.

11.7.1.2 Wie kommt man in den Bootloader-Modus

Beim Start des Programms wird immer zuerst der Bootloader ausgeführt. Dieser überprüft den Zustand der RX Leitung des USART Moduls. Weil der Ruhezustand des RX Signals ein High-Pegel und normalerweise nur bei der Übertragung von Daten Low-Pegel erscheinen, kann ein dauerhafter Low-Pegel als "BREAK" Signal gewertet werden, das den Bootloader-Mode initiiert.

Konkrete Anweisungen dazu enthält das übernächste Kapitel zur Bootloader GUI.

11.7.2 Application Firmware

Bei Firmware (im folgenden Application genannt), die mittels eines Bootloaders auf die Zielplattform gebracht werden soll, müssen bestimmte Grundeinstellungen beachtet werden um die dauerhafte Funktion des Bootloaders nicht zu gefährden.

11.7.2.1 Configuration Bits

Wenn der Loader mehrmals verwendet werden soll, dann muss man darauf achten, dass die Application für die Funktion des Bootloaders wichtige Einstellungen nicht gefährdet. Dies könnte z.B. eine nicht lauffähige Veränderung der Oszillatoreinstellungen in den Config-Bits sein.

Deshalb muss in der PC-Anwendung das Überschreiben der Config-Bits auch nochmals separat aktiviert werden.

11.7.2.2 Reservierung des Bootloader Speicherbereichs

Oft muss schon bei der Erstellung einer Application darauf geachtet werden, dass die Programmspeicherbereiche in denen sich der Loader befindet nicht verwendet werden. Je nachdem welcher Compiler verwendet wird, können dafür unterschiedliche Maßnahmen erforderlich sein.

C18 -> keine Reservierung erforderlich, da der Speicher immer von unten her belegt wird.

XC8 -> Bei diesem Compiler ist die Belegung des Speichers nicht vorhersagbar. Deshalb muss man gegebenenfalls den Speicherbereich des Bootloaders reservieren. Informationen dazu findet man in der Application Note und im User-Guide des Compilers.

Grundsätzlich darf die Applikation nicht größer sein als der „freie“ Speicherplatz. Das kann natürlich von der Entwicklungsumgebung ohne explizite Reservierung von Speicherbereich nicht überprüft werden. Dieser Fall tritt aber wegen des eher geringen Speicherbedarfs des Bootloaders in der Praxis selten ein.

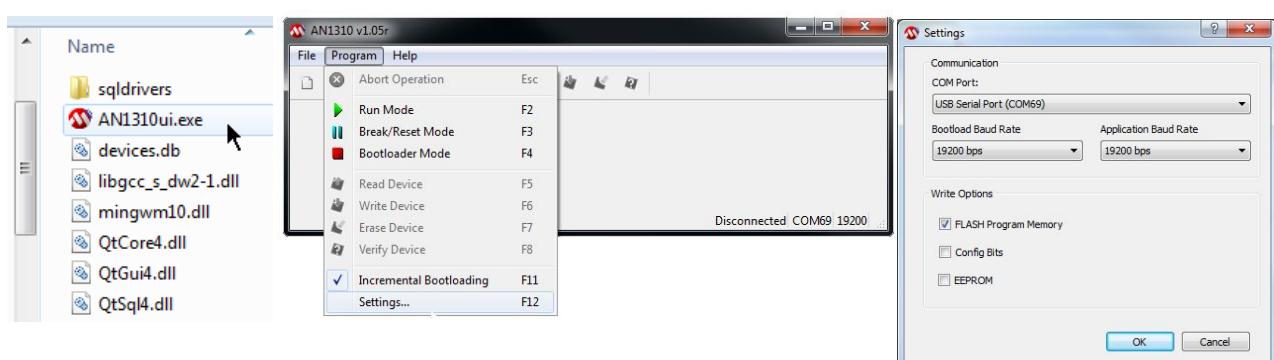
11.7.2.3 Erzeugung des Hex-Files der Application

Zur Erzeugung der Hex-Dateien die über den Loader auf die Zielhardware geladen werden sollen wird **Eport Hex** im **Projekt Kontextmenü** verwendet. (*rechter Mausklick auf Projektnamen*)

Bitte nicht die während des normalen Arbeitens mit der IDE entstandenen Hex-Files suchen und benutzen.

11.7.3 Bootloader GUI

Alle, für die Ausführung der PC-Software unter Windows, unbedingt erforderlichen Dateien sind in [AN1310ui_exe.zip](#) ([hs-ulm.de/schilling/...](http://hs-ulm.de/schilling/)) enthalten. Entpacken und AN1310ui.exe ausführen.



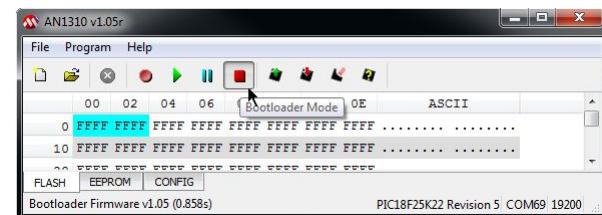
Nach dem Start der Software muss man zunächst die Schnittstelle auswählen an der die Zielhardware angeschlossen ist. Dies geschieht über das Menü **Program→Settings**. Das wichtigste ist die Auswahl des richtigen **COM Port**.

Die **Bootload Baud Rate** ist unkritisch, da der AN1310 Bootloader die Übertragungsrate automatisch anpasst. Soll das im GUI integrierte Terminal benutzt werden um mit der neuen Firmware nach dem Aufspielen zu kommunizieren, dann muss natürlich die **Application Baud Rate** entsprechend eingestellt sein.

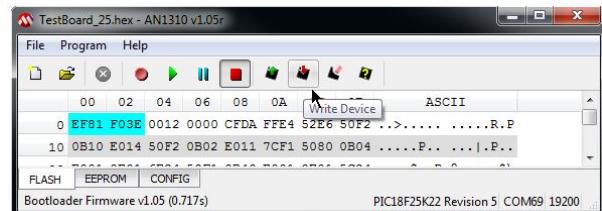
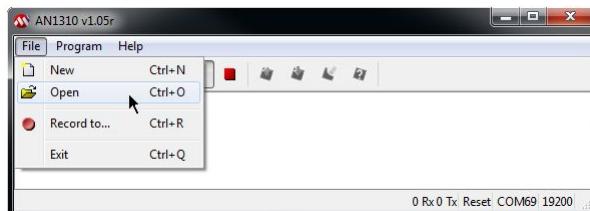
Wenn man sich nicht sicher ist, ob die neuen **Config Bits** nicht den Bootloader außer Betrieb setzen können, dann sollte man das entsprechende Häkchen besser nicht setzen.

Damit der Bootloader auf dem PIC beim nächsten Reset erkennt, dass er im Bootloader Modus bleibt, wird über die Schaltfläche **Pause** die Break-Condition ausgelöst (**Usart-RX Signal am PIC auf Low**). Der Reset kann auf der uCQ-2013 Platine am einfachsten durch Trennung der Stromversorgung des PICs über den Jumper **JP12** unterhalb des PICs erzeugt werden.

Bitte nicht das USB Kabel abziehen, da sonst der USB-Seriell Wandler auf der Platine ...

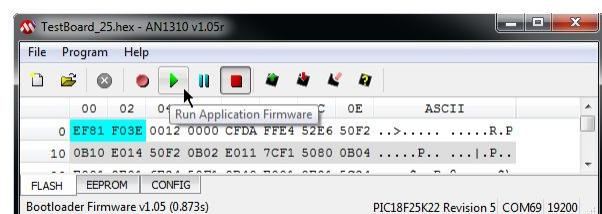
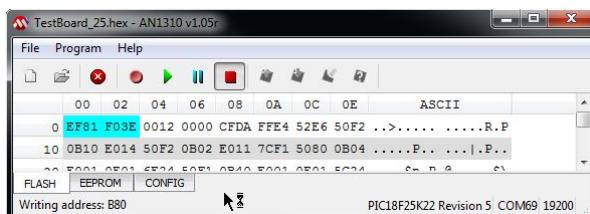


Nach dem Rest kann mit der **Stop** Schaltfläche die Verbindung zum Bootloader hergestellt werden. Bei einer erfolgreichen Verbindung wird in der Statusleiste die Firmwareversion und der Typ des gefundenen PICs angezeigt. (**v1.05 und PIC18F25K22 Revision 5 in der obigen Abbildung**)



Über **File→Open** kann man jetzt die gewünschte neue Application-Firmware importieren und diese dann mittels der Schaltfläche **Write Device** auf die Zielhardware laden.

Während des Schreibens wechselt das Mauszeigersymbol zur Sanduhr und der Fortschritt des Schreibvorgangs wird zusätzlich unten in der Statusleiste angezeigt.



Nach erfolgreichem Schreibvorgang kann man den Bootloader-Modus beenden und die Ausführung der neuen Application über **Play/Run** starten. Alternativ kann man natürlich auch einen weiteren Reset der Hardware ohne vorhandene Break-Condition durchführen

11.7.4 Bootloader konfigurieren für neue PICs

Application Note lesen ?

Hinweise in deutscher Sprache findet man auch auf www.PIC-Projekte.de

12 Weiterführende Informationen zu

Vorsicht: Vor dem Kauf Preise vergleichen !!!

12.1 Empfehlenswerte Microchip Hardware / Einsteiger Kits

12.1.1 MPLAB Snap (PICkit 4 light)

Günstiger Programmer und Debugger für PIC, AVR, ARM...
Keine Bereitstellung der Versorgungsspannung.

12.1.2 PICkit 4

Programmer und Debugger für PIC, AVR, ARM...

12.1.3 Curiosity Development Boards

PICkit on Board...

12.1.4 Xplained Boards

Ähnlich Curiosity für (ehemals) Atmel Controller (AVR, SAM...)

12.1.5 MPLAB Xpress Boards

Für PICs. KEIN DEBUGGER !!!

12.1.6 PICkit 3

Günstig als China Clone, aber nicht mehr so aktuell wie z.B. ein original SNAP zum gleichen Preis.

12.1.6.1 PICKIT 3 Starter Kit

Das **PICKIT 3 Starter Kit** beinhaltet ein PICKIT3, eine kleine Demoplattine mit IC-Sockel für 8/14/20 Pin PICs, jeweils einen PIC16F1829-I/P. / PIC18F14K22-I/P und dem größten Umfang an Beispielprogrammen in „C“ und Assembler.
(Der in „C“ den Beispielen verwendete Compiler ist XC8)

12.1.6.2 PICKIT 3 Debug Express

Auch die **PICKIT 2/3 Debug Express** Kits sind günstigsten Einsteiger Sets inklusive Programmier- und Debugging- Tool, bei denen der verwendete PIC jedoch verlötet ist.

Die User-Guides der Kits sind verhältnismäßig gut, wenn auch manchmal in bestimmten Punkten wie z.B. den Informationen zu „Linkerscripten“ veraltet.

Die Beispielprogramme in den User Guides aller Kits, können natürlich auch auf anderer (eigener) Hardware ausprobiert werden. Die Dokumentation für alle Microchip Demo-Kits ist kostenlos auf der Webseite des Herstellers erhältlich !

12.2 Debuggen (Hardware)

12.2.1 Wie funktioniert ICD ??? (programmieren / debuggen)

Zum Verständnis wie das ICD System funktioniert, kann **Chapter 2. Operation** aus dem **PICkit3 User-Guide (DS52116A)** beitragen. Insbesondere die Abbildungen (+ zugehörige Erläuterungen).

- FIGURE 2-4: STANDARD CONNECTION TARGET CIRCUITRY
- FIGURE 2-5: IMPROPER CIRCUIT COMPONENTS
- FIGURE 2-7: PICkit™ 3 DEBUGGER READY FOR DEBUGGING

12.2.2 Debuggen FAQ – was tun?

Q: Mein Programm tut nicht was es soll.
Wie bekomme ich heraus, was es überhaupt macht?

→ [3.1.8.4 Ausführen, Anhalten und des Programms](#)

Q: Eine bestimmte Funktionalität scheint nicht richtig zu arbeiten oder ausgeführt zu werden.
Wie kann man herausfinden, ob der zugehörige Code überhaupt ausgeführt wird?

→ [3.1.8.7 Program Breakpoint](#)

Q: I/O scheint nicht richtig zu funktionieren.
Wie kann man genau das sehen, was der Controller auch „sieht“?

→ [3.1.8.5 Anzeige von Registern \(PORT, TRIS und ANSEL Register anzeigen...\)](#)

Q: Register oder Variablen haben falsche Werte. Kann man herausfinden, ob diese eventuell von einem Programmteil unerwartet verändert wurden?

→ [3.1.8.7 Data Breakpoint](#)

Q: Kann man es irgendwo einstellen, dass bei einem Breakpoint nicht jedes mal angehalten wird?

→ [3.1.8.8 Breakpoint Pass Count](#)

Q: Die im PIC vorhandenen Breakpoints sind einfach zu wenige, kann man da was tun?

→ [12.2.3 TRAP - undokumentierter Programmhaltepunkt](#)

Q: Kann ein Breakpoint mit bestimmten Bedingungen verknüpft werden?

→ [3.1.8.7 Data Breakpoint](#)

→ [3.1.8.8 Breakpoint Pass Count](#)

→ [12.2.3 TRAP - undokumentierter Programmhaltepunkt](#)

Q: Kann man Register oder Variablen verändern, ohne jedes mal den Code zu ändern, das Programm neu zu übersetzen und eine neue Debug-Sitzung zu starten?

→ [12.2.4 Daten über den Debugger editieren](#)

12.2.3 TRAP - undokumentierter Programmhaltepunkt

Viele PICs verfügen über einen undokumentierten Befehl, welcher es ermöglicht, beliebig viele Haltepunkte zu setzen. Der Befehl muss allerdings wie jeder normale Befehl direkt in das Programm geschrieben werden.

Das Programm hält dann beim Erreichen der Programmzeile automatisch an. Dies ist natürlich nur im Debug-Modus sinnvoll. Ist der Debug-Mode nicht aktiv, dann hat der Befehl keine Funktion.

Ein Beispiel zu einer Makro-Definition für die Benutzung des TRAP Befehls:

```
#ifdef __DEBUG
    #define mDEBUG_STOP()      if (mGET_ENC_BTN()) { asm("TRAP"); }
#else
    #define mDEBUG_STOP()
#endif
```

Das Makro *mDEBUG_STOP()* kann so an beliebigen Stellen im Programm verwendet werden.

Der XC8 Compiler bietet sogar schon eigene Makros zur Nutzung dieses Befehls an. Diese werden in den folgenden Kapiteln vorgestellt.

12.2.4 __builtin_software_breakpoint(void)

Hier dürfte nicht viel mehr dahinter stecken, als:

```
#define __builtin_software_breakpoint()  asm("TRAP")
```

12.2.5 __debug_break(void)

Dieses Makro ist das gleiche, wie *__builtin_software_breakpoint()*, nur wird hier die Erzeugung von Code im Programmspeicher verhindert, wenn das Programm nicht im Debug-Mode erstellt wird. *__debug_break()* wird im Header *xc8debug.h* definiert, der über *xc.h* eingebunden wird.

```
#if defined(__DEBUG)
    #define __debug_break()  __builtin_software_breakpoint()
#else
    #define __debug_break()  ((void)0)
#endif
```

12.2.6 __conditional_software_breakpoint(expression)

Eine Erweiterung von *__debug_break()* mit einer Bedingung. Das Resultat ist das gleiche, wie im Beispiel der Einführung des *TRAP* Befehls. Damit diese Makro benutzt werden kann, muss der Header *assert.h* eingebunden werden. Dies geschieht nicht automatisch im Hintergrund!

```
#if defined(NDEBUG) || !defined(__DEBUG)
    #define __conditional_software_breakpoint(exp)  ((void)0)
#else
    #define __conditional_software_breakpoint(exp)  \
        ((exp) ? ((void)0) : __builtin_software_breakpoint())
#endif
```

12.2.7 Daten über den Debugger editieren

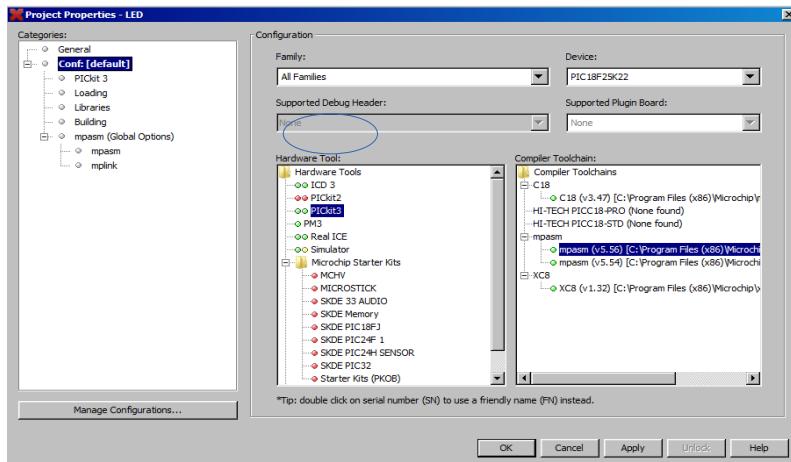
...

12.3 Simulator MPSIM

12.3.1 Genaue Zeitmessungen mit dem Simulator MPSIM

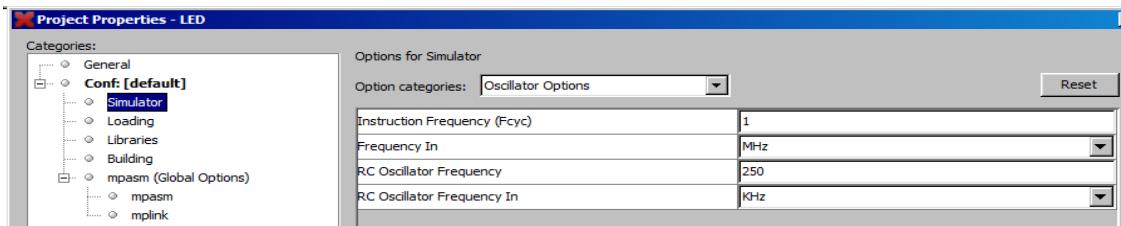
(RB muss noch vernünftig ausgearbeitet werden!!!!!!!!!!!!!!)

Umstellen auf den Simulator über Projekt File/Properties



Einstellen des Stopwatchfensters Window/Debugging/Stopwatch

Stellen Sie die Taxkzfrequenz des Mikrocontrollers ein, sonst stimmen die berechneten Zykluszeiten nicht



Debug Mainprojekt ergibt die Simulation bis zum ersten Breakpoint. Dafür waren 13 Zyklen bei 8MHz Prozessortakt = 6,5 μ s notwendig. Der Schleifendurchlauf benötigte 197121 Zyklen, entsprechend 98 ms

```
Start Page x LED_Main.asm x
38      movlw  B'11111011'; ; linke LI
39      movwf  LATB;        ; Schreibe
40
41      Count             ; Beginn
42
43      rlnacf LATB, 1
44
45      ; Aufgabe 3
46      clrf cnt_LSB;    ; Zähler
47      clrf cnt_MSB;
48      Warteschleife
49      incf cnt_LSB;
50      bnz Warteschleife; ; innere
51      incf cnt_MSB;
52      bnz Warteschleife; ; äußere
53
54      goto Count;       ; Hauptschleife
55
56      END
57

Stopwatch  Target halted. Stopwatch cycle count = 13 (6,5 μs)
Stopwatch  Target halted. Stopwatch cycle count = 197121 (98,5605 ms)

Start Page x LED_Main.asm x
42      Count             ; Beginn der
43
44      rlnacf LATB, 1
45
46      ; Aufgabe 3
47      clrf cnt_LSB;    ; Zähler für
48      clrf cnt_MSB;
49      Warteschleife
50      incf cnt_LSB;
51      bnz Warteschleife; ; innere Schl
52      incf cnt_MSB;
53      bnz Warteschleife; ; äußere Schl
54
55      goto Count;       ; Hauptschleife
56
57

Stopwatch  Target halted. Stopwatch cycle count = 13 (6,5 μs)
Stopwatch  Target halted. Stopwatch cycle count = 197121 (98,5605 ms)
```

12.4 Alternative 8-Bit PICs

12.4.1 Standard Anwendungen

PIC18FxxK42: Der aktuell neueste und modernste PIC18 für Standard Anwendungen

12.4.2 CAN

PIC18FxxK80 / K83: Für CAN Projekte

12.4.3 USB

PIC16F145x / PIC18FxxK50 / PIC18FxxJ53: Für USB Projekte

12.5 PIC Hardware

12.5.1 Befehlstakt

12.5.2 LAT Register & R-M-W

12.6 MPLAB-X Projekte

12.6.1 Konfigurationen

Im **Project Properties** Dialog kann man mit **Manage configurations ...** für ein Projekt verschiedene Konfigurationen erstellen, welche man dann sehr einfach über die Drop-Down Box in der Menüleiste umschalten kann.

Die einzelnen Konfiguration können beispielsweise unterschiedliche Entwicklungstools wie Simulator und PICKIT3 oder verschiedene Controller (25K22 / 24K22) enthalten.

Durch die Möglichkeit Source-Files über das Property Menü (*der Datei*) für einzelne Konfigurationen zu ignorieren, kann man auch größere Unterschiede in mehreren Hardwareplattformen (z.B. *andere Controller Familie*) sehr einfach in einem Projekt integrieren.

12.6.2 Kopieren

Die „**Copy...**“ Funktion des Projekt Kontext Menüs erlaubt das Kopieren eines vorhandenen Projektes an einen neuen Ort und mit neuem Namen. Mit den im Projekt vorhandenen Dateien wird dabei folgendermaßen verfahren.

- Dateien im Projektordner werden an den neuen Ort kopiert.
- Bei Dateien, die sich außerhalb des Projektordners befinden, nimmt die IDE eine gemeinsame Nutzung für mehrere Projekte an und verlinkt diese im neuen Projekt mit den Originalen.

12.6.3 Packen

Mit der Funktion „**Package**“ im Projekt Kontext Menü kann ein Compilier-fähiges Projekt so im Zip-Format gepackt werden, dass es nur noch die wirklich notwendigen Dateien enthält damit das Projekt (*auch an einen anderen Ort*) wieder hergestellt werden kann.

12.6.4 µC-Projekt Dateien (*TODO überarbeiten !!!!!!!!!!!!!!!*)

Bevor das erste Mikrocontroller Projekt begonnen wird, sollten einige Punkte bezüglich der Organisation in verschiedenen Dateitypen kurz angesprochen werden.

Der Aufbau eines Projektes in einer Entwicklungsumgebung und in einer bestimmten Programmiersprache folgt immer bestimmten Konventionen, die unbedingt eingehalten werden müssen. (Diese Konventionen wurden eingeführt um das Ganze zu vereinfachen ;-)

Die hier angesprochenen Punkte können möglicherweise nicht alle auf Anhieb verstanden werden. Dennoch sollte man schon mal davon gehört haben !

Bei Auftreten entsprechender Fragen und Probleme kann man dann nochmals nachlesen, bzw. weitere Informationen im Netz suchen.

12.6.4.1 Source Files

Alles, was eine Reservierung von Programm oder Datenspeicher zur Folge hat sollte in Source-Dateien stehen.

12.6.4.1.1 Assembler (*.asm)

12.6.4.1.2 C (*.c)

- c018x.c
- <mainfile>.c
- <otherfile>.c

12.6.4.2 Header / Include - Files (*.h, *.inc)

Header- (C) oder Include (ASM) -dateien sollten nichts enthalten, was eine Reservierung von Programm oder Datenspeicher zur Folge hat !

Diese Dateien enthalten Beschreibungen (*Deklarationen*) von Objekten und Schnittstellen nicht die Definitionen der Objekte selber.

Meistens gibt es eine Header-Datei, welche global verwendete und wichtige Deklarationen enthält. (siehe auch [5.3.3.1 Variablen \(Flags\) in mehreren Source Dateien verwenden](#))

Eine einer speziellen Source-Datei (*.c; *.asm) zugehörige Include-Datei (*mit gleichem Namensteil*) enthält meist nur solche Informationen die für die „Benutzung“ des Source-Moduls erforderlich sind. Deklarationen die nur innerhalb dieses Source-Moduls verwendet werden können in der Source-Datei untergebracht werden.

12.6.4.2.1 Prozessorspezifische Header/Include -Dateien

Die prozessorspezifische Header-Datei (*p18f25k22.h* oder *p18f25k22.inc*) mit Definitionen für den verwendeten Controller muss immer eingebunden werden.

Die Include-Dateien für die jeweiligen PICs enthalten hauptsächlich Assembler Konstanten, welche die Datenspeicher Organisation der **Special-Function-Register** widerspiegeln. Hier wird als Hilfe für den Programmierer ein **Register-Name** für die entsprechende **Register-Adresse** definiert.

Ein einziger Blick in die Inc-Datei und in die Tabelle „SPECIAL FUNCTION REGISTER MAP...“ des Datenblattes sollte genügen um die Funktionsweise zu verstehen.

Die C18 Header-Dateien (*p18f25k22.h*) erfüllen prinzipiell den gleichen Zweck, nur sind die Zusammenhänge hier etwas schwerer zu erkennen. In der Prozessor-Header-Datei werden Variablen als extern deklariert, welche in einer anderen Datei angelegt wurden (*p18f25k22.asm* ;-), die wiederum in der Prozessor-Bibliothek (*p18f25k22.lib*) enthalten ist. Diese Bibliothek wird über das Linker-Script (*18f25k22_g.lkr*) zum Projekt hinzugefügt.

Damit das Projekt für verschiedene Controller leichter adaptierbar bleibt, wird in C18 Projekten oft eine Datei mit dem Namen "**p18cxx.h**" eingebunden. Innerhalb dieser Datei wird dann über Makrodefinitionen der für den in der IDE ausgewählten PIC passende Header eingefügt.

Achtung:

Wenn man eine Header-Datei im Projektfenster zufügt, wird der darin enthaltene Text nicht dem Projekt hinzugefügt. Dieser Mechanismus dient lediglich dem einfacheren Zugriff auf die Datei. (für den Programmierer, die „In Dateien suchen“ Funktion ...)

Durch die Direktive "#include <p18f25k22.h>" in einer Source- oder auch Header-Datei wird vielmehr die Anweisung gegeben, den Inhalt der Datei (*p18f25k22.h*) an dieser Stelle einzufügen.

12.6.4.2.2 Include Guard für Header Files

Sogenannte "Include Guards" verhindern das mehrfache Einfügen des Textes einer Headerdatei in eine Sourcedatei. Dies würde durch z.B doppelte oder mehrfache Definitionen zu Fehlern bei der Kompilierung führen. Die Anweisungen "#ifndef" in der ersten und "#endif" in der letzten Zeile gehören zusammen. Alles was dazwischen liegt wird nur dann eingefügt, wenn die Definition noch nicht vorhanden ist. Beim ersten Einfügen wird durch das "#define" in der zweiten Zeile die Definition erstellt, und somit ein weiteres Einfügen verhindert.

```
#ifndef DATEINAME_H
#define DATEINAME_H
...
#endif
```

12.6.4.3 Linker Script (*.lkr)

Wird bei aktuellen IDEs automatisch "gefunden".

(Standard Skripte "*_g.lkr" müssen nicht explizit zum Projekt hinzugefügt werden)

- Fügt C18 Projekten weitere Object und Library Files hinzu:
 - den Startup Code (*c018x.o*)
 - die Standartbibliothek (*clib.lib*)
 - die prozessorspezifische Bibliothek (*p18f45K22.lib*)
- Enthält die Beschreibungen von Programm und Datenspeicher des entsprechenden PICs.
(bitte vergleichen mit dem Datenblatt !!!)
- Reserviert gegebenenfalls Programm und Datenspeicher für den Debugger.
- Reserviert gegebenenfalls Datenspeicher für den Software-Stack bei C18 Projekten.

Basierend auf dem Script platziert der Linker Codesegmente und Variablen.

12.7 Eigenheiten der uC-Quick Hardware

12.7.1 Allgemein

12.7.1.1 Gemeinsame Nutzung von Pins durch LEDs, Display und Taster

Die 4 LEDs, die 4 Datenleitungen des Displays (vor Version 2018) und zwei der Taster sind an den selben Pins des Mikrocontrollers angeschlossen. Alle diese Komponenten können bei geeigneter Programmierung trotzdem im selben Projekt verwendet werden.

12.7.2 Version 2018

12.7.2.1 Neue Displayvariante

Das Charakter LCD wurde durch ein Graphikdisplay mit 84x48 Pixeln ersetzt. Das Display hat keine integrierten Zeichensätze.

Das Interface des Graphikdisplays ist seriell, ähnlich SPI, mit einem zusätzlichen Signal zur Unterscheidung von Daten und Kommandos. Es gibt nur Schreibzugriffe.

Die Verwendung eines Character Displays ist weiterhin möglich, das Interface benutzt dann die gleichen Pins wie bisher.

12.7.3 Version 2013

Nichts bekannt ...

12.7.4 Version 2012

12.7.4.1 Probleme durch unbenutztes LCD display

Wenn das LCD angeschlossen ist aber nicht benutzt bzw. initialisiert wird kann dies zu Problemen mit der Tastenabfrage führen. Das Problem besteht darin, dass die Datenleitungen eventuell auf Lesemodus stehen und das Display somit Signale darauf ausgibt.

Dies kann man umgehen indem man die R/W Leitung auf „0“ legt. Der Pin muss natürlich auch als Ausgang konfiguriert sein !

```
ASM
    bcf      LCD_RW_PIN      ; Sicherstellen, dass
    bcf      LCD_RW_TRI       ;      LCD nicht in Lesemodus !!

~~~~~
C
LCD_RW_PIN = 0;           // Sicherstellen, dass
LCD_RW_TRI = 0;           //      LCD nicht in Lesemodus !!
```

Das Problem kann auch hardwareseitig durch einen Pull-Down Widerstand (2k ?) an der entsprechenden Leitung beseitigt werden.

...

13 Anhang

13.1 Fehlermeldungen MPLAB / Linker / C18 / PICKit ...

13.1.1 *Target was not found. You must connect to a target device ...*

Eine der Grundvoraussetzungen ist eine korrekte Stromversorgung des Controllers auf der Zielhardware. Ist diese nicht vorhanden, dann wird er nicht erkannt. Stromversorgung anschließen, oder über PICkit bereitstellen. Dialog siehe [3.1.8.2 Einrichten des Debuggers \(Power\)](#)

13.1.2 *The target device is not ready for debugging ...*

Für das Debuggen muss der PIC mit der IDE kommunizieren. D.h. das Programm auf der Zielplattform muss laufen! Der Oszillatator muss korrekt eingestellt sein und funktionieren!
→ Config Settings überprüfen

13.1.3 *Target Device ID (0x0) does not match expected Device ID (0x5540).*

Keine Kommunikation möglich. Tool ist falsch oder gar nicht mit Zielplattform verbunden.
Falsche Kontrollerfamilie ausgewählt. Ist die „Target Device ID“ nicht 0x0 dann sind eingestellter und eingesteckter PIC nicht identisch.

13.1.4 *PICkit 3 is trying to supply x,yz volts from the USB port, but the target VDD is measured to be ??? volts*

Die Zielplattform kann vom PICkit nicht ausreichen versorgt werden. → Versuchen kleinere Spannung einstellen. Dialog siehe [3.1.8.2 Einrichten des Debuggers \(Power\)](#)

13.2 Assembler Templates

Ein komplettes Template Projekt „_C_ASM_TEMPLATE.zip“ für MPLAB X ist zum Download verfügbar: → http://lmgtfy.com/?q=_C_ASM_TEMPLATE.zip

13.2.1 Assembler Configuration Bits

```
;*****  
;  
; This file is a basic code template for code generation on the  
; PIC18FxxK22. This file contains the configuration bits settings  
;  
; Refer to the MPASM User's Guide for additional information on features  
; of the assembler.  
;  
; Refer to the respective data sheet for additional information on the  
; instruction set.  
;  
;*****  
;  
; Filename:          uCQ_config.asm  
; Date:             13. April 2015  
; File Version:     2.00  
; Author:           VSK  
; Company:          HS-Ulm  
;  
;*****  
  
;  
-----  
; PROCESSOR DECLARATION  
-----  
#ifdef __18F25K22  
    LIST      P=PIC18F25K22      ; list directive to define processor  
    #include <p18f25k22.inc>      ; processor specific variable definitions  
#elif defined(__18F24K22)  
    LIST      P=PIC18F24K22  
    #include <p18f24k22.inc>  
#endif  
-----  
;  
; CONFIGURATION WORD SETUP  
;  
; The 'CONFIG' directive is used to embed the configuration word within the  
; .asm file. The labels following the directive are located in the respective  
; .inc file. See the data sheet for additional information on configuration  
; word settings.  
;  
-----  
;  
;Setup CONFIG1H  
    CONFIG FOSC = INTIO7      ; Oscillator (LP,XT,HSHP,HSMP,RC,RCIO6,ECHP,ECHPIO6  
                            ; ,INTIO67,INTIO7,ECMPIO6,ECLP,ECLPIO6)  
    CONFIG PLLCFG = OFF       ; 4X PLL Enable  
    CONFIG PRICLKEN = OFF      ; Primary clock Enable  
    CONFIG FCMEN = OFF        ; Fail-Safe Clock Monitor Enable  
    CONFIG IESO = OFF         ; Internal/External Oscillator Switchover  
;Setup CONFIG2L  
    CONFIG PWRTE = OFF        ; Power-up Timer Enable  
    CONFIG BOREN = OFF        ; Brown-out Reset Enable (OFF,ON,NOSLP,SBODIS)  
    CONFIG BORV = 190          ; Brown Out Reset Voltage (285,250,220,190) [V/100]  
;Setup CONFIG2H  
    CONFIG WDTE = OFF         ; Watchdog Timer Enable (OFF,NOSLP,SWON,ON)  
    CONFIG WDTPS = 1           ; Watchdog Timer Postscale Select (1,2,4,...,32768)  
;Setup CONFIG3H  
#ifdef __DEBUG  
    CONFIG MCLRE = EXTMCLR   ; MCLR Pin Enable (EXTMCLR,INTMCLR)  
#else  
    CONFIG MCLRE = INTMCLR  
;
```

```

#endif
    CONFIG CCP2MX = PORTB3 ; ECCP2 B output mux (PORTB3, PORTC1)
    CONFIG PBADEN = OFF   ; PORTB A/D Enable
    CONFIG CCP3MX = PORTB5 ; CCP3 MUX (PORTB5, PORTC6)
    CONFIG HFOFST = OFF   ; HFINTOSC Fast Start-up
    CONFIG T3CMX = PORTB5 ; Timer3 Clock input mux (PORTB5, PORTC0)
    CONFIG P2BMX = PORTB5 ; ECCP2 B output mux (PORTB5, PORTC0)
;Setup CONFIG4L
    CONFIG STVREN = OFF   ; Stack Full/Underflow Reset Enable
    CONFIG LVP = OFF      ; Single-Supply ICSP Enable
    CONFIG XINST = OFF    ; Extended Instruction Set Enable
;Setup CONFIG5L
    CONFIG CP0 = OFF      ; Code Protection Block 0
    CONFIG CP1 = OFF
#endif _18F25K22
    CONFIG CP2 = OFF
    CONFIG CP3 = OFF
#endif
;Setup CONFIG5H
    CONFIG CPB = OFF      ; Boot Block Code Protection
    CONFIG CPD = OFF      ; Data EEPROM Code Protection
;Setup CONFIG6L
    CONFIG WRT0 = OFF      ; Write Protection Block 0
    CONFIG WRT1 = OFF
#endif _18F25K22
    CONFIG WRT2 = OFF
    CONFIG WRT3 = OFF
#endif
;Setup CONFIG6H
    CONFIG WRTB = OFF      ; Boot Block Write Protection
    CONFIG WRTC = OFF      ; Configuration Register Write Protection
    CONFIG WRTD = OFF      ; Data EEPROM Write Protection
;Setup CONFIG7L
    CONFIG EBTR0 = OFF      ; Table Read Protection Block 1
    CONFIG EBTR1 = OFF
#endif _18F25K22
    CONFIG EBTR2 = OFF
    CONFIG EBTR3 = OFF
#endif
;Setup CONFIG7H
    CONFIG EBTRB = OFF      ; Boot Block Table Read Protection
END

```

13.2.2 Assembler Main

```

;*****
; This file is a basic template for code generation on the PIC18FxxK22.
; This file contains the basic code building blocks to build upon.
; Refer to the MPASM User's Guide for additional information on features
; of the assembler.
; Refer to the respective data sheet for additional information on the
; instruction set.
;*****
; Filename:          uCQ_main.asm
; Date:             08. April 2015
; File Version:     2.00
; Author:            VSK
; Company:           HS-Ulm
;*****



;-----#
; PROCESSOR DECLARATION
;-----#
    list p=PIC18F25K22      ; list directive to define processor
#include <p18f25k22.inc>    ; processor specific definitions (LATB EQU H'0F8A')

```

```

;-----  

; USER DEFINITION FILE  

;-----  

#include "uCQ_2013.inc" ; project specific (#define nLED_1 LATB,LATB2,ACCESS)  

;  

;-----  

; VARIABLE DEFINITIONS  

; Refer to datasheet for available data memory (RAM) organization  

;-----  

; Example of using GPR Uninitialized Data  

GPR_VAR udata ;-----  

;MYVAR_1 res 1 ; User variable linker places  

;  

; Example of using Access Uninitialized Data Section  

ACS_VAR udata_acs ;-----  

counter_1 res 1 ; variable in ACCESS RAM  

;  

;  

;-----  

; RESET VECTOR  

;-----  

RES_VECT code 0x0000 ; processor reset vector-----  

    goto Init ; go to beginning of program (initialization)  

;  

;-----  

; MAIN PROGRAM      ( goal is to blink LED at RB2 ~5Hz )  

;-----  

MAIN_PROG code ; let linker place main program-----  

Init ; initialization code (start)-----  

    movlw 0x00 ; set internal oscillator to  

    movwf OSCCON ; frequency 31,25 kHz  

    bcf LED_1_TRI ; make PIN B2 (LED) an output  

Main ; main code-----  

    incfsz counter_1 ; 1 cycle (almost)  

    goto Main ; 2 cycles -> loop is 3 cycles  

    btg nLED_1 ; toggle RB2 (LED ON/OFF) 31250/4 / (3*256) /2 -> ~5  

    goto Main  

;  

    end

```

13.2.3 Assembler Interrupt

```

;*****  

;  

; This file is a basic code template for code generation on the  

; PIC18FxxK22. This file contains the basic code building blocks  

; to build interrupt code upon  

;  

; Refer to the MPASM User's Guide for additional information on features  

; of the assembler.  

;  

; Refer to the respective data sheet for additional information on the  

; instruction set.  

;  

;*****  

;  

; Filename:          uCQ_intr.asm  

; Date:              13. April 2015  

; File Version:      2.00  

; Author:             VSK  

; Company:            HS-Ulm  

;  

;*****  

;  

;
```

```

; PROCESSOR DECLARATION
;-----

        list      p=PIC18F25K22          ; list directive to define processor
        #include <p18f25k22.inc>       ; processor specific variable definitions

; Example of using Access Uninitialized Data Section
INT_VAR      UDATA_ACS
W_TEMP       RES      1      ; w register for context saving (ACCESS)
STATUS_TEMP  RES      1      ; status used for context saving
BSR_TEMP    RES      1      ; bank select used for ISR context saving

;-----
; HIGH PRIORITY INTERRUPT VECTOR
;-----
ISRHV      CODE      0x0008

        ; Run the High Priority Interrupt Service Routine
        GOTO      HIGH_ISR

;-----
; LOW PRIORITY INTERRUPT VECTOR
;-----
ISRLV      CODE      0x0018

        ; Run the High Priority Interrupt Service Routine
        GOTO      LOW_ISR

;-----
; HIGH PRIORITY INTERRUPT SERVICE ROUTINE
;-----
ISRH      CODE                  ; let linker place high ISR routine

HIGH_ISR

        ; Insert High Priority ISR Here

        RETFIE  FAST

;-----
; LOW PRIORITY INTERRUPT SERVICE ROUTINE
;-----
ISRL      CODE                  ; let linker place low ISR routine

LOW_ISR

        ; Context Saving for Low ISR
        MOVWF   W_TEMP           ; save W register
        MOVFF   STATUS, STATUS_TEMP ; save status register
        MOVFF   BSR, BSR_TEMP     ; save bankselect register

        ; Insert Low Priority ISR Here

        ; Context Saving for Low ISR
        MOVFF   BSR_TEMP, BSR      ; restore bankselect register
        MOVF    W_TEMP, W          ; restore W register
        MOVFF   STATUS_TEMP, STATUS ; restore status register
        RETFIE

END

```

13.2.4 Assembler EEPROM

```
;*****  
;  
; This file is a basic code template for code generation on the  
; PIC18FxxK22. This file contains the basic code building blocks to build  
; upon.  
;  
; Refer to the MPASM User's Guide for additional information on features  
; of the assembler.  
;  
; Refer to the respective data sheet for additional information on the  
; instruction set.  
;  
;*****  
;  
; Filename:          uCQ_eeprom.asm  
; Date:             13. April 2015  
; File Version:     2.00  
; Author:            VSK  
; Company:          HS-Ulm  
;  
;*****  
;  
;-----  
; PROCESSOR DECLARATION  
;  
list      p=PIC18F25K22           ; list directive to define processor  
#include <p18f25k22.inc>        ; processor specific variable definitions  
;  
;-----  
; EEPROM INITIALIZATION  
; The 18FxxK22 has 256..1024 bytes of non-volatile EEPROM starting at 0xF00000  
;  
;-----  
DATAEE  
    code    0xF00000           ; Starting address for EEPROM for 18F4553  
    DE      "HS-Ulm"          ; Place 'H' 'S' '-' 'U' 'm' 'm' at address 0,1,...,5  
;  
;-----  
END
```

13.2.5 Assembler Include

```
;*****  
;  
; This file is a basic template for USER (signal) definitions  
;  
;*****  
;  
; Filename:          uCQ_2013.inc      (hardware version 2013)  
; Date:             08. April 2015  
; File Version:     1.00  
; Author:            VSK  
; Company:          HS-Ulm  
;  
;*****  
; Files required:  
;*****  
; Notes:  
;*****  
; Revision History:  
;*****  
  
;Singal pins:  
  
; #define SIG_IN           PORTx,Rxy,ACCESS  
; #define SIG_IN_TRI        TRISx,TRISxn,ACCESS  
  
; #define SIG_OUT          LATx,LATxy,ACCESS  
; #define SIG_OUT_TRI       TRISx,TRISxn,ACCESS  
  
; -----push buttons  
#define nBTN_L           PORTB,RB2,ACCESS  
#define BTN_L_TRI         TRISB,TRISB2,ACCESS  
#define nBTN_R           PORTB,RB4,ACCESS  
#define BTN_R_TRI         TRISB,TRISB4,ACCESS  
  
; -----LEDs  
#define nLED_1           LATB,LATB2,ACCESS  
#define LED_1_TRI         TRISB,TRISB2,ACCESS  
#define nLED_2           LATB,LATB3,ACCESS  
#define LED_2_TRI         TRISB,TRISB3,ACCESS  
#define nLED_3           LATB,LATB4,ACCESS  
#define LED_3_TRI         TRISB,TRISB4,ACCESS  
#define nLED_4           LATB,LATB5,ACCESS  
#define LED_4_TRI         TRISB,TRISB5,ACCESS  
  
; -----digital inputs  
#define TTL_IN           PORTA,RA0,ACCESS  
#define TTL_IN_TRI        TRISA,TRISA0,ACCESS  
#define ST_IN             PORTC,RC0,ACCESS  
#define ST_IN_TRI         TRISC,TRISCO,ACCESS  
  
; -----rotary encoder  
#define nENC_BTN          PORTB,RB1,ACCESS  
#define ENC_BTN_TRI        TRISB,TRISB1,ACCESS  
  
#define ENC_INT           PORTB,INT0,ACCESS  
#define ENC_INT_TRI        TRISB,TRISB0,ACCESS  
#define ENC_DIR            PORTA,RA2  
#define ENC_DIR_TRI        TRISA,TRISA2  
  
#define mENC_IR_EN()      bsf INTCON,INT0IE,ACCESS  
#define mENC_IR_DIS()     bcf INTCON,INT0IE,ACCESS  
#define mENC_IR_CLR()     bcf INTCON,INT0IF,ACCESS
```

13.3 C Templates

Ein komplettes gepacktes MPLAB X Projekt „**C_ASM_TEMPLATE.zip**“ mit allen Dateien ist zum Download verfügbar: → http://lmgtfy.com/?q=_C_ASM_TEMPLATE.zip

13.3.1 C Config

```
////////////////////////////////////////////////////////////////////////
//      filename:          uC_config.c
//      configuration bits for uC-Quick projects
//
//#####
//      Author:           V.SchK
//      Company:          HS-Ulm
//
//      Revision:         3.0
//      Date:             April 2016
//      Assembled using  MPLABX
//
//#####
#ifndef __18F24K22
#ifndef __18F25K22
#ifndef __18F26K22
    ERROR No Configuration bits are defined! Double click this message for details."
#endif
#endif
#endif

// PIC18F2xK22 Configuration Bit Settings (24/25/26)
// CONFIG1H
#pragma config FOSC = INTIO67 // Oscillator (LP,XT,HSHP,HSMP,RC,RCIO6,ECHP,
                           // ECHPIO6,INTIO67,INTIO7,ECPMIO6,ECLP,ECLPIO6)
#pragma config PLLCFG = OFF // 4X PLL Enable
#pragma config PRICLKEN = ON // Primary clock enable
#pragma config FCMEN = OFF // Fail-Safe Clock Monitor Enable
#pragma config IESO = OFF // Internal/External Oscillator Switchover

// CONFIG2L
#pragma config PWRTE = OFF // Power-up Timer Enable
#pragma config BOREN = SBORDIS // Brown-out Reset Enable (OFF,ON,NOSLP,SBODIS)
#pragma config BORV = 190 // Brown Out Reset Volt.(285,250,220,190) [V/100]

// CONFIG2H
#pragma config WDTEN = OFF // Watchdog Timer Enable (OFF,NOSLP,SWON,ON)
#pragma config WDTPS = 32768 // Watchdog Timer Postscale Select (1:32768)

// CONFIG3H
#pragma config CCP2MX = PORTB3 // ECCP2 B output mux (PORTB3, PORTC1)
#pragma config PBADEN = OFF // PORTB A/D Enable
#pragma config CCP3MX = PORTB5 // CCP3 MUX (PORTB5, PORTC6)
#pragma config HFOFST = ON // HFINTOSC Fast Start-up
#pragma config T3CMX = PORTC0 // Timer3 Clock input mux (PORTB5,PORTC0)
#pragma config P2BMX = PORTB5 // ECCP2 B output mux (PORTB5,PORTC0)
#ifdef __DEBUG
    #pragma config MCLRE = EXTMCLR // MCLR Pin Enable (MCLR / RE3)
#else
    #pragma config MCLRE = INTMCLR
#endif

// CONFIG4L
#pragma config STVREN = ON // Stack Full/Underflow Reset Enable
#pragma config LVP = OFF // Single-Supply ICSP Enable
#pragma config XINST = OFF // Extended Instruction Set Enable

// CONFIG5L
#pragma config CP0 = OFF // Code Protection Block 0
```

```

#pragma config CP1 = OFF
#ifndef __18F24K22
    #pragma config CP2 = OFF
    #pragma config CP3 = OFF
#endif

// CONFIG5H
#pragma config CPB = OFF           // Boot Block Code Protection
#pragma config CPD = OFF           // Data EEPROM Code Protection

// CONFIG6L
#pragma config WRT0 = OFF          // Write Protection Block 0
#pragma config WRT1 = OFF
#ifndef __18F24K22
    #pragma config WRT2 = OFF
    #pragma config WRT3 = OFF
#endif

// CONFIG6H
#pragma config WRTC = OFF          // Configuration Register Write Protection
#pragma config WRTB = OFF          // Boot Block Write Protection
#pragma config WRTD = OFF          // Data EEPROM Write Protection

// CONFIG7L
#pragma config EBTR0 = OFF          // Table Read Protection Block 0
#pragma config EBTR1 = OFF
#ifndef __18F24K22
    #pragma config EBTR2 = OFF
    #pragma config EBTR3 = OFF
#endif

// CONFIG7H
#pragma config EBTRB = OFF          // Boot Block Table Read Protection

```

13.3.2 C Main

```

//#####
// filename:      uCQ_main.c
//
// main template (for uCQ_2013 board)
//
//#####
// Author:          V.SchK
// Company:        HS-Ulm
//
// Revision:       2.0
// Date:           08. April 2015
// Assembled using MPLABX 5.00+ XC8 v2.00+
//
// todo      - add comments ;-
//           -
//
//#####

#include "uCQ_2013.h"

//### private prototypes ###
void __init(void);

//#####
// Function:      void main(void)
//                 called from the startup code
// PreCondition:  None
// Input:
// Output:
// Side Effects:
// Overview:
//#####

```

```

void main()
{
    __init();

    while(1){
//        ...
    }

//#####
// Function:          void __init(void)
//
// PreCondition:     None
// Input:
// Output:
// Side Effects:
// Overview:
//#####
void __init()
{
    OSCCONbits.IRCF = IRCF_4MHZ;
//    OSCCONbits.IRCF = IRCF_16MHZ;
//    OSCTUNEbits.PLLN = 1;      // →64MHz

//    ANSELA = 0x01;   // all pins except A0 (poti) digital IO
//    ANSELB = 0x00;
//    ANSELC = 0x00;
//    ...
}

```

13.3.3 C Interrupt

```

//#####
// filename:          uCQ_intr.c
//#####
// interrupt functions template
//#####
// Author:            V.SchK
// Company:          HS-Ulm
//
// Revision:         3.0
// Date:             June 2019
// Assembled using   MPLAB-X v5.00+ XC8 v2.00+
//
// todo      - add comments ;-
//           -
//#####

/** INCLUDES *****/
#include "uCQ_2013.h"
//#include "uCQ_2018.h"

/** DECLARATIONS ****/
//#####
// Function:          void high_isr(void)
// PreCondition:     None
// Side Effects:
// Overview:
//#####
void __interrupt(high_priority) high_isr(void)
{
//    if (interrupt_1_enabled && interrupt_1_flagged) {
//        interrupt_1_flag = 0;
//        ...
//        return;
//    }
//    if (interrupt_2_enabled && interrupt_2_flagged) {
//        interrupt_2_flag = 0;

```

```

//      ...
//      return;
//    }
//    ...
while(1){;}      //  (detect unexpected IR sources)
}

#####
// Function:          void low_isr(void)
// PreCondition:     None
// Side Effects:
// Overview:
#####
#endif USE_IR_PRIORITIES
void __interrupt(low_priority) low_isr(void)
{
  while(1){;}      //  (detect unexpected IR sources)
}
#endif

```

13.3.4 C Header Board Version 2013

```

#####
// filename:           uCQ_2013.h
//
// meaningful pin-names (signals) and macro definitions
// for demo projects on uCQ_2013 platform
//
//#####
// Author:            V.SchK
// Company:          HS-Ulm
//
// Revision:         2.0 (XC8 only)
// Date:             May 2019
// Assembled using   MPLABX v5.00+ / XC8 v2.00+
//
// todo   - add comments ;-)
//         -
//
//#####
#ifndef UCQ_2013_H
#define UCQ_2013_H

#include <xc.h>          // XC8 compiler

#define ENCODER_TOGGLE //


//-----internal clock freq.
#if defined(__18F25K22) | defined(__18F24K22) | defined(__18F26K22)
  #define IRCF_16MHZ 0b111
  #define IRCF_8MHZ 0b110
  #define IRCF_4MHZ 0b101
  #define IRCF_2MHZ 0b100
  #define IRCF_1MHZ 0b011
  #define IRCF_500KHZ 0b010
  #define IRCF_250KHZ 0b001
  #define IRCF_31KHZ 0b000
#endif

// -----pin directions
#define INPUT_PIN          1
#define OUTPUT_PIN          0
#define INPUT_REG           0xFF
#define OUTPUT_REG          0x00
#define DIGITAL_PIN         0
#define ANALOG_PIN          1
#define mALL_IO_DIGITAL()  ANSELA = ANSELB = ANSELC = 0x00

```

```

// -----push buttons
#define BTN_L PORTBbits.RB2 // left button
#define BTN_L_TRI TRISBbits.TRISB2
#define BTN_L_ANS ANSELBbits.ANSB2
#define BTN_R PORTBbits.RB4 // right button
#define BTN_R_TRI TRISBbits.TRISB4
#define BTN_R_ANS ANSELBbits.ANSB4

#define BTN_ACTIVATED 0
#define mGET_BTN_L() (BTN_L == BTN_ACTIVATED)
#define mGET_BTN_R() (BTN_R == BTN_ACTIVATED)
//#define mGET_BTN_ENC() mGET_ENC_BTN()

// -----LEDs
#define LED_1 LATBbits.LATB2
#define LED_1_TRI TRISBbits.TRISB2
#define LED_2 LATBbits.LATB3
#define LED_2_TRI TRISBbits.TRISB3
#define LED_3 LATBbits.LATB4
#define LED_3_TRI TRISBbits.TRISB4
#define LED_4 LATBbits.LATB5
#define LED_4_TRI TRISBbits.TRISB5

#define LED_ON 0
#define LED_OFF 1
#define mSET_LED_1_ON() LED_1 = LED_ON
#define mSET_LED_1_OFF() LED_1 = LED_OFF
#define mTOG_LED_1() LED_1 ^= 1
// LED_1 = !LED_1; -> bullshit with xc8
#define mGET_LED_1() (LED_1 ^ LED_OFF)
#define mSET_LED_2_ON() LED_2 = LED_ON
#define mSET_LED_2_OFF() LED_2 = LED_OFF
#define mTOG_LED_2() LED_2 ^= 1
#define mGET_LED_2() (LED_2 ^ LED_OFF)
#define mSET_LED_3_ON() LED_3 = LED_ON
#define mSET_LED_3_OFF() LED_3 = LED_OFF
#define mTOG_LED_3() LED_3 ^= 1
#define mGET_LED_3() (LED_3 ^ LED_OFF)
#define mSET_LED_4_ON() LED_4 = LED_ON
#define mSET_LED_4_OFF() LED_4 = LED_OFF
#define mTOG_LED_4() LED_4 ^= 1
#define mGET_LED_4() (LED_4 ^ LED_OFF)

#define mALL_LED_OFF() LATB |= 0b00111100
#define mALL_LED_ON() LATB &= 0b11000011
#define mALL_LED_OUTPUT() TRISB &= 0b11000011

// -----digital inputs
#define TTL_IN PORTAbits.RA0
#define TTL_IN_TRI TRISAbits.TRISA0
#define TTL_IN_ANS ANSELAbits.ANSA0
#define ST_IN PORTCbits.RC3
#define ST_IN_TRI TRISCbits.TRISC3
#define ST_IN_ANS ANSELCbits.ANSC3

// -----rotary encoder
#define ENC_BTN PORTBbits.RB1
#define ENC_BTN_TRI TRISBbits.TRISB1
#define ENC_BTN_ANS ANSELBbits.ANSB1
#define mGET_ENC_BTN() (ENC_BTN == BTN_ACTIVATED)

#define ENCBTN_IR_FLAG INTCONbits..INT0IF
#define ENCBTN_IR INTCONbits.INT0IE && INTCONbits.INT0IF
#define mENCBTN_IR_DIS() INTCONbits.INT0IE = 0
#define mENCBTN_IR_CLR() INTCONbits.INT0IF = 0

#define ENC_INT PORTBbits.INT0 // A -> interrupt pin int_0/b_0
#define ENC_INT_TRI TRISBbits.TRISB0
#define ENC_INT_ANS ANSELBbits.ANSB0

```

```

#define ENC_DIR PORTAbits.RA2 // B -> direction pin a_2
#define ENC_DIR_TRI TRISAbits.TRISA2
#define ENC_DIR_ANS ANSELAbits.ANSA2
//#define ENC_DIR PORTEbits.RE3 // alternative selection (JP4)
//#define ENC_DIR_TRI TRISEbits.TRISE3

#define ENC_A PORTBbits.RB0
#define ENC_A_TRI ENC_INT_TRI
#define ENC_A_ANS ENC_INT_ANS
#define ENC_B ENC_DIR
#define ENC_B_TRI ENC_DIR_TRI
#define ENC_B_ANS ENC_DIR_ANS

#define ENC_IR_FLAG INTCONbits.INT0IF
#define ENC_IR INTCONbits.INT0IE && INTCONbits.INT0IF
#define mENC_IR_DIS() INTCONbits.INT0IE = 0
//          //
//ifdef ENCODER_TOGGLE // __|__|__|__|__|__ (toggles)
#define ENC_DIR_UP !INTCON2bits.INTEDG0
#define ENC_DIR_DOWN INTCON2bits.INTEDG0
#define mENC_IR_EN() INTCON2bits.INTEDG0 = !ENC_INT; \
INTCONbits.INT0IE = 1
#define mENC_IR_RST() INTCON2bits.INTEDG0 ^= 1; INTCONbits.INT0IF = 0
//      #define ENCODER_PANASONIC // compatibility with old code

#else //          //
#define ENC_DIR_UP 1 // --|--|---|---|---|--- (pulses)
#define ENC_DIR_DOWN 0
#define mENC_IR_EN() INTCON2bits.INTEDG0 = 0; INTCONbits.INT0IE = 1
#define mENC_IR_RST() INTCONbits.INT0IF = 0
#endif

// -----analog input (poti)
#define POTI_TRI TRISAbits.TRISA0
#define POTI_ANS ANSELAbits.ANSA1
#define ADC_POTI ADC_CH0
#define ADC_IR PIE1bits.ADIE && PIR1bits.ADIF
#define mADC_IR_EN() PIE1bits.ADIE = 1
#define mADC_IR_DIS() PIE1bits.ADIE = 0
#define mADC_IR_CLR() PIR1bits.ADIF = 0

// -----speaker
#define SPEAKER LATCbits.LATC2
#define SPEAKER_TRI TRISCbits.TRISC2
#define mTOG_SPEAKER() SPEAKER = !SPEAKER

// -----RS232
#define TX_PIN LATCbits.LATC6
#define RX_PIN PORTCbits.RC7
#define TX_TRI TRISCbits.TRISC6
#define RX_TRI TRISCbits.TRISC7
#define RX_ANS ANSELAbits.ANSC7
#define TX_FREE PIR1bits.TX1IF

// -----BOOTLOADER (TEST)
#if defined(__18F24K22)
#define BOOTLOADER_1310 0x3D0C
#elif defined(__18F25K22)
#define BOOTLOADER_1310 0x7D0C
#elif defined(__18F26K22)
#define BOOTLOADER_1310 0xFD0C
#endif

#define BOOTLOADER_TEST() ENC_BTN_ANS = DIGITAL_PIN; \
if(mGET_ENC_BTN()){ _asm goto BOOTLOADER_1310 _endasm }

// -----debug help signal
#define dbgSIGNAL SPEAKER

```

```

#define dbgSIGNAL_TRI    SPEAKER_TRI
#define dbgPULSE()        mTOG_SPEAKER();mTOG_SPEAKER()

// #define mENCBTN_DEBUG_STOP()
// #define SOFT_BREAK()

#endif /* UCQ_2013_H */

```

13.3.5 C Header Board Version 2018

```

// ##### filename: uCQ_2018.h
//
//      meaningful pin-names (signals) and macro definitions
//      for demo projects on uCQ_2018 platform
//
// ##### Author:          V.SchK
//      Company:         HS-Ulm
//
//      Revision:        3.0 (XC8 only)
//      Date:            May 2019
//      Assembled using MPLABX v5.00+ / XC8 v2.00++
//
//      todo - add comments ;-
//      -
//
// ##### *-----internal clock freq.
#ifndef UCQ_2018_H
#define UCQ_2018_H

#include <xc.h>           // XC8 compiler

#define ENCODER_TOGGLE //


// -----pin directions
#if defined(__18F25K22) | defined(__18F24K22) | defined(__18F26K22)
#define IRCF_16MHZ 0b111
#define IRCF_8MHZ 0b110
#define IRCF_4MHZ 0b101
#define IRCF_2MHZ 0b100
#define IRCF_1MHZ 0b011
#define IRCF_500KHZ 0b010
#define IRCF_250KHZ 0b001
#define IRCF_31KHZ 0b000
#endif

// -----push buttons
#define BTN_L           PORTBbits.RB2           // left button
#define BTN_L_TRI       TRISBbits.TRISB2
#define BTN_L_ANS       ANSELBbits.ANSB2
#define BTN_R           PORTBbits.RB4           // right button
#define BTN_R_TRI       TRISBbits.TRISB4
#define BTN_R_ANS       ANSELBbits.ANSB4

#define BTN_ACTIVATED   0
#define mGET_BTN_L()    (BTN_L == BTN_ACTIVATED)

```

```

#define mGET_BTN_R()          (BTN_R == BTN_ACTIVATED)
//#define mGET_BTN_ENC()      mGET_ENC_BTN()

// -----LEDs
#define LED_1                  LATBbits.LATB2
#define LED_1_TRI               TRISBbits.TRISB2
#define LED_2                  LATBbits.LATB3
#define LED_2_TRI               TRISBbits.TRISB3
#define LED_3                  LATBbits.LATB4
#define LED_3_TRI               TRISBbits.TRISB4
#define LED_4                  LATBbits.LATB5
#define LED_4_TRI               TRISBbits.TRISB5

#define LED_ON                 0
#define LED_OFF                1
#define mSET_LED_1_ON()         LED_1 = LED_ON
#define mSET_LED_1_OFF()        LED_1 = LED_OFF
#define mTOG_LED_1()           LED_1 ^= 1
// LED_1 = !LED_1; -> bullshit with xc8
#define mGET_LED_1()           (LED_1 ^ LED_OFF)
#define mSET_LED_2_ON()         LED_2 = LED_ON
#define mSET_LED_2_OFF()        LED_2 = LED_OFF
#define mTOG_LED_2()           LED_2 ^= 1
#define mGET_LED_2()           (LED_2 ^ LED_OFF)
#define mSET_LED_3_ON()         LED_3 = LED_ON
#define mSET_LED_3_OFF()        LED_3 = LED_OFF
#define mTOG_LED_3()           LED_3 ^= 1
#define mGET_LED_3()           (LED_3 ^ LED_OFF)
#define mSET_LED_4_ON()         LED_4 = LED_ON
#define mSET_LED_4_OFF()        LED_4 = LED_OFF
#define mTOG_LED_4()           LED_4 ^= 1
#define mGET_LED_4()           (LED_4 ^ LED_OFF)

#define mALL_LED_OFF()          LATB |= 0b00111100
#define mALL_LED_ON()           LATB &= 0b11000011
#define mALL_LED_OUTPUT()       TRISB &= 0b11000011

// -----digital inputs
#define TTL_IN                  PORTAbits.RA0
#define TTL_IN_TRI               TRISAbits.TRISA0
#define TTL_IN_ANS               ANSELAbits.ANSA0
#define ST_IN                   PORTCbits.RC3
#define ST_IN_TRI                TRISCbits.TRISC3
#define ST_IN_ANS                ANSELCbits.ANSC3

// -----rotary encoder
#define ENC_BTN                 PORTBbits.RB1
#define ENC_BTN_TRI              TRISBbits.TRISB1
#define ENC_BTN_ANS               ANSELBbits.ANSB1
#define mGET_ENC_BTN()           (ENC_BTN == BTN_ACTIVATED)

#define ENCBTN_IR_FLAG           INTCONbits.INT0IF
#define ENCBTN_IR                 INTCONbits.INT0IE && INTCONbits.INT0IF
#define mENCBTN_IR_DIS()         INTCONbits.INT0IE = 0
#define mENCBTN_IR_CLR()         INTCONbits.INT0IF = 0

#define ENC_INT                  PORTBbits.INT0          // A -> interrupt pin int_0/b_0
#define ENC_INT_TRI               TRISBbits.TRISB0
#define ENC_INT_ANS               ANSELBbits.ANSB0
#define ENC_DIR                  PORTAbits.RA2          // B -> direction pin a_2
#define ENC_DIR_TRI               TRISAbits.TRISA2
#define ENC_DIR_ANS               ANSELAbits.ANSA2
//#define ENC_DIR                 PORTEbits.RE3          // alternative selection (JP4)
//#define ENC_DIR_TRI              TRISEbits.TRISE3

#define ENC_A                    PORTBbits.RB0
#define ENC_A_TRI                ENC_INT_TRI
#define ENC_A_ANS                ENC_INT_ANS
#define ENC_B                    ENC_DIR
#define ENC_B_TRI                ENC_DIR_TRI

```

```

#define ENC_B_ANS      ENC_DIR_ANS

#define ENC_IR_FLAG    INTCONbits.INT0IF
#define ENC_IR         INTCONbits.INT0IE && INTCONbits.INT0IF
#define mENC_IR_DIS() INTCONbits.INT0IE = 0
                           //
#ifndef ENCODER_TOGGLE
#define ENC_DIR_UP     // __|____|____|____|____|__ (toggles)
#define ENC_DIR_DOWN   !INTCON2bits.INTEDG0
#define mENC_IR_EN()   INTCON2bits.INTEDG0
                           INTCON2bits.INTEDG0 = !ENC_INT; \
                           INTCONbits.INT0IE = 1
#define mENC_IR_RST() INTCON2bits.INTEDG0 ^= 1; INTCONbits.INT0IF = 0
//      #define ENCODER_PANASONIC // compatibility with old code

#else
                           //
#define ENC_DIR_UP     1 // ____|____|____|____|____|__ (pulses)
#define ENC_DIR_DOWN   0
#define mENC_IR_EN()   INTCON2bits.INTEDG0 = 0; INTCONbits.INT0IE = 1
#define mENC_IR_RST() INTCONbits.INT0IF = 0
#endif

// -----analog input (poti)
#define POTI_TRI       TRISAbits.TRISA0
#define POTI_ANS        ANSELAbits.ANSA1
#define ADC_POTI       ADC_CH0
#define ADC_IR         PIE1bits.ADIE && PIR1bits.ADIF
#define mADC_IR_EN()   PIE1bits.ADIE = 1
#define mADC_IR_DIS()  PIE1bits.ADIE = 0
#define mADC_IR_CLR()  PIR1bits.ADIF = 0

// -----speaker
#define SPEAKER         LATCbits.LATC2
#define SPEAKER_TRI    TRISCbits.TRISC2
#define mTOG_SPEAKER() SPEAKER = !SPEAKER

// -----RS232
#define TX_PIN          LATCbits.LATC6
#define RX_PIN          PORTCbits.RC7
#define TX_TRI          TRISCbits.TRISC6
#define RX_TRI          TRISCbits.TRISC7
#define RX_ANS          ANSELCbits.ANSC7
#define TX_FREE         PIR1bits.TX1IF

// -----BOOTLOADER (TEST)
#if defined(__18F24K22)
#define BOOTLOADER_1310 0x3D0C
#elif defined(__18F25K22)
#define BOOTLOADER_1310 0x7D0C
#elif defined(__18F26K22)
#define BOOTLOADER_1310 0xFD0C
#endif

#define BOOTLOADER_TEST() ENC_BTN_ANS = DIGITAL_IN; \
if(mGET_ENC_BTN()) { _asm goto BOOTLOADER_1310 _endasm }

// -----debug help signal
#define dbgSIGNAL      SPEAKER
#define dbgSIGNAL_TRI  SPEAKER_TRI
#define dbgPULSE()      mTOG_SPEAKER();mTOG_SPEAKER()

//#define mENCBTN_DEBUG_STOP()
//#define SOFT_BREAK()

#endif /* UCQ_2018_H */

```

13.4 Microchip C18 Compiler / MPASM Assembler (alt)

13.4.1 C18-Compiler → XC8-Compiler migration

13.4.1.1 Delay Funktionen

Diese Funktionen basieren auf dem Zählprinzip. Da sollte man sich gut überlegen ob man die wirklich verwenden möchte und wie man die damit einhergehenden Probleme in den Griff bekommt. Siehe auch [6.2.5 Delay Funktionen](#).

Der XC8 Compiler bietet mit delay_us() komfortablere Makros an, bei denen als Parameter ein Wert für die Zeit angegeben werden kann.

13.4.1.2 Context saving

13.4.1.3 Interrupt – Vektoren – Prioritäten ...

13.4.2 C Bibliotheken und Dokumentation

Die Bibliotheken des C18 Compilers liegen im Verzeichnis

.../Microchip/mplabc18/v3.xx/lib und umfassen

- Startup Code c018.o, c018_e.0, c018i.o, c018i_e.0, c018iz.o, c018iz_e.0
- allgemeine Software Bibliotheken (Mathematikfunktionen ...) lib.lib, lib_e.lib
- prozessorspezifische Bibliotheken (Hardwaremodule ...) z.B. p18f25k22.lib

Die Dokumentation und meist auch den Quellcode der zu Verfügung stehenden Bibliotheken findet man im Installationsverzeichnis des Compilers.

(z.B. .../Microchip/mplabc18/v3.4x/docs und .../Microchip/mplabc18/v3.4x/sources)

13.4.3 Der Startup Code – erste Funktionen

Der C18 Startup-Code ist als Quellcode verfügbar und kann auch verändert (*angepasst*) werden.

Es gibt verschiedene Grundversionen von denen die Standardversion in der Datei

.../Microchip/mplabc18/v3.xx/src/traditional/startup/c018i.c enthalten ist.

```
#pragma code _entry_scn=0x000000
void _entry (void)
{
    _asm go startun_e.dasm
}

#pragma code _startup_scn
void _startup (void)
{
    _asm
        lfsr 1, _stack           // Initialize the stack pointer
        lfsr 2, _stack

        clrf TBLPTRU, 0          // 1st silicon doesn't do this on POR

        bcf __FPFLAGS,RND,0      // Initialize rounding flag for floating point libs
    _endasm

    _do_cinit ();              // MPLAB-C18 initialized data memory support function
loop:
    __init();                 // If user defined __init is not found, the one in lib.lib will be used
    main();                   // Call the user's main routine
    goto loop;
}
```

Im abgebildeten Ausschnitt der Datei kann man Ähnlichkeiten zu den Assembler Templates

erkennen. Die `code` Direktive ist auch hier vorhanden, nur ist die Reihenfolge der Parameter etwas anders und `#pragma` wird vorangestellt. Auch hier gilt, wenn keine Adresse mit angegeben wird, dann darf der Linker den Code beliebig im Programmspeicher platzieren.

Auffällig sind auch noch `_asm ... _endasm`. Diese Codewörter signalisieren dem Compiler, dass es sich bei dem davon eingeschlossenen Code gar nicht um C Code handelt, sondern um Assembler Code. (*Man nennt das auch Inline-Assembler*) Die dort verwendeten Befehle wie z.B. „`clrF`“ findet man deshalb auch nicht in der Doku des Compilers, sondern im Datenbuch des PICs.

Wie Assembler gibt es auch bei C einen absoluten Sprungbefehl „`goto`“, welcher allerdings eher verpönt ist, und nur selten eingesetzt wird. Hier wird damit eine äußere Endlosschleife erzeugt die zwei **Funktionen** (`_init()` & `main()`) immer wieder aufruft.

Damit der Compiler die richtige Verwendung einer Funktion logisch überprüfen kann, muss diese Funktion mit Ihren Parametern vor dem Aufruf der Funktion dem Compiler mitgeteilt werden. Dies wird durch die Prototypen realisiert.

```
/* external reference to __init() function */
extern void __init(void);
/* external reference to the user's main routine */
extern void main(void);
/* prototype for the startup function */
void __entry(void);
void __startup(void);
/* prototype for the initialized data setup */
void __do_cinit (void);
```

Normalerweise würde, wie in Assembler, eine preprocessor-spezifische Header-Datei eingebunden werden (`#include <p18f25k22.h>`), um dem Compiler die Registernamen des PICs bekannt zu machen. Im Falle des Startup Codes wurde dies darauf verzichtet und nur die Register und Variablen als „extern“ deklariert, welche hier auch zur Verwendung kommen.

```
extern volatile near unsigned long short TBLPTR;
extern near unsigned F=0;
extern near char __FPFLAG;
#define RND 6
```

13.4.3.1 Wie kommt der Startup Code in mein Projekt ?

Alles schon wieder vergessen ? → Siehe [1.3.3 Linker](#) weiter oben !

13.4.4 Die benötigten Include-Dateien

Das passende Prozessor-Header File wird beim C18 Compiler über `#include <p18cxx.h>` eingebunden, die ihrerseits die richtige Datei für den im Projekt eingestellten PIC18 einbinden.

```
// uCQ_2013.h

#include <p18cxx.h>                                // C18 compiler

#define IRCF_31KHZ      0b000          // value for oscillator frequency setting
                                         // see data sheet
#define LED_1           LATBbits.LATB2 // LED_1 is connected at port B_2
#define LED_1_TRI        TRISBbits.TRISB2
#define mTOG_LED_1()     LED_1 ^= LED_1 // 

#define OUTPUT_PIN       0             // 0 is for output / 1 for input
```

ACHTUNG: Gemäß dem C-Standard muss jede nicht leere Zeile mit einem „newline“ abgeschlossen sein ! (→ jede Datei muss mit einer leeren Zeile enden)

13.4.5 `__init()` Funktion

Die Funktion `__init()` ist ein Spezialfall des **C18** Compilers. Sie wird automatisch vom Start-up Code vor der Funktion `main()` aufgerufen. In diese Funktion kann/sollte alles geschrieben werden, was nur einmal bei Programmstart / Reset ausgeführt werden soll.

```
// uCQ_main.c

#include "uCQ_2013.h"

void __init(void)
{
    OSCCONbits.IRCF = IRCF_31KHZ;           // ... already defined in <project_name>.h
    LED_1_TRI = OUTPUT_PIN;                 // ...
}

// ANSELA = ANSELB = ANSEL0 = 0x00;        all PINS digital IO (see data sheet)
...
}
```

13.4.6 ~~main()~~ Funktion

Schreibt man den Code für die Initialisierung in die Funktion `__init()`, dann enthält die `main()` Funktion letztendlich nur den Code, der nach der Initialisierung bis zum Abschalten der Stromversorgung oder einem Reset ausgeführt wird. Deshalb ist in `main()` meist gleich zu Anfang eine Endlos-Schleife (`while(1){...}`)

```
void main(void)
{
    unsigned char counter_1 = 0;           // one byte variable, values 0..255

    while(1) {
        if(++counter_1 == 0) {
            mTOG_LED_1();
        }
    }
}
```

13.4.7 C18 Interrupts

//im Moment nur Notizen aus vorhergehenden Versionen!

Für C18-Programme ändert sich die Anweisung in `#pragma code [Label] = Adresse des Vektors:`
Auch an den Vektoren des C-Programms stehen Sprungbefehle die über die „**Inline-Assembler**“
Anweisungen „`_asm ... _endasm`“ eingebunden werden müssen.

Für C-Compiler muss das ganze jetzt noch in eine Funktion eingepackt werden, die allerdings unter
keinen Umständen manuell aufgerufen werden sollte !!!

```
#pragma code high_vector=0x08
void interrupt_at_high_vector(void)
{
    _asm goto high_isr _endasm
}

#pragma code low_vector=0x18
void interrupt_at_low_vector(void)
{
    _asm goto low_isr _endasm
}
```

Bei Verwendung des **C Compilers** wird das Sichern und Wiederherstellen des Kontextes, dem
Programmcode durch den Compiler selber hinzugefügt (*User Guide*).

Allerdings muss dem Compiler auch signalisiert werden, welches die High- bzw. Low-Priority
Interrupt Funktionen sind. Dafür sind die folgenden Zeilen zuständig, welche die entsprechenden
Anweisungen (`#pragma interrupt`) und die frei wählbaren **Namen** (hier `high_isr`) der
entsprechenden Routinen enthalten:

```
#pragma interrupt high_isr
#pragma interrupt low_low_isr
```

Beim C18 Compiler musste man in Gegensatz zum XC8 Compiler die Interruptvektoren manuell
anlegen...

```
#####
// filename:          uCQ_intr.c
// interrupt functions template
#####
// Author:           vSchK
// Company:          HS-Ulm
//
// Revision:         2.0
// Date:             April 2015
// Assembled using   MPLAB-X 1.40+ C18 3.40+
//
#####
/** I N C L U D E S *****/
#include "uCQ_2013.h"

/** P R I V A T E   P R O T O T Y P E S *****/
void high_isr(void);
void low_isr(void);

/** I N T E R R U P T   V E C T O R S *****/
#pragma code high_vector=0x08
void interrupt_at_high_vector(void){ _asm goto high_isr _endasm }

#pragma code low_vector=0x18
void interrupt_at_low_vector(void){ _asm goto low_isr _endasm }
```

```

#pragma code

/** DECLARATIONS *****/
//#####
// Function:           void high_isr(void)
// PreCondition:      None
// Side Effects:
// Overview:
//#####
#pragma interrupt high_isr
void high_isr(void)
{
    if (interrupt_1_enabled && interrupt_1_flagged) {
        interrupt_1_flag = 0;
        ...
        return;
    }
    if (interrupt_2_enabled && interrupt_2_flagged) {
        ...
        while(1); // finde unvorhergesehene interrupts
    }

//#####
// Function:           void low_isr(void)
// PreCondition:      None
// Side Effects:
// Overview:
//#####
#pragma interrupt low_isr
void low_isr(void)
{
    while(1); // finde unvorhergesehene interrupt
}

```

13.4.8 MPASM

13.4.8.1 MPASM Templates

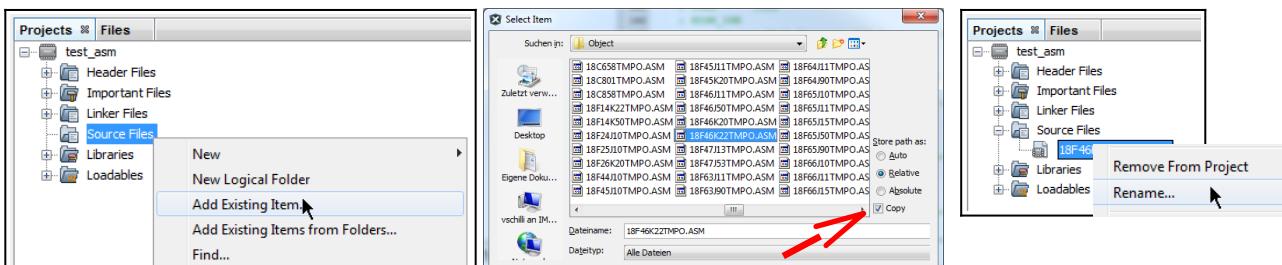
Für Projekte mit MPASM Assembler bietet es sich an die von Microchip bereitgestellte Templates als erste Grundlage zu benutzen. Diese findet man im Installationsverzeichnis mit zum gewählten PIC passenden Namen. z.B.: `../MPLABX/v4.05/mplasmx/templates/Object/<name>.ASM`

Ist kein Template für den geplanten Controller vorhanden, kann man möglicherweise eines aus der gleichen Familie nehmen. (Letzten beiden Ziffern sind gleich/ `PIC18F24K22 → PIC18F46K22`)

Die entsprechende Datei wird am besten **in den Projektordner kopiert** und passend zum Projekt **umbenannt**. Für den weiteren Verlauf dieser Einführung würde sich „**uCQ_main.asm**“ anbieten.

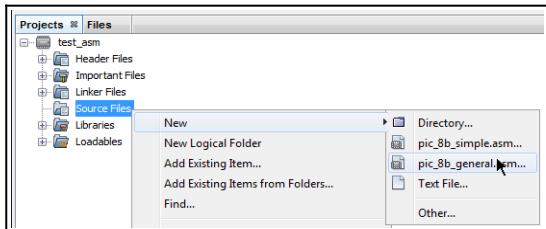
Das Kopieren der Datei kann direkt in der IDE über das Popup-Menü im Projektbrowser erfolgen. Durch Klicken mit der rechten Maustaste auf den logischen Ordner **Source Files** öffnet sich das dazu passende Popup Menu, in welchen man **Add Existing Item** auswählt und dann im weiteren Dialog zur passenden Datei navigiert. Vor dem Bestätigen der Auswahl muss natürlich noch die Checkbox **Copy** aktiviert werden. Die Einstellung, wie der Pfad zur Datei im Projekt gespeichert wird, sollte immer „**Relative**“ sein bzw. bleiben.

Das Umbenennen der Datei kann dann über das Popup-Menü der Datei erfolgen.



13.4.9 Assembler File von der IDE erstellen lassen

Alternativ zum Einfügen einer Template Kopie kann man von der IDE auch eine Datei erstellen lassen, die ein allgemeines Grundgerüst eines Assembler Files für 8-Bit PICs und eine TODO Liste mit Anleitung zur Komplettierung enthält.



In **TODO Step #1 ... TODO Step #4** kann man dann schrittweise das richtige Include-File für den Prozessor, passende Configuration-Bit Settings, Variablendefinitionen und die Interruptvektoren einfügen.

13.4.10 Grundgerüst eines µC Programms in MPASM

Für MPASM Assembler Projekte stehen, [wie weiter oben bereits erwähnt](#), im Installationsordner Vorlagen zur Verfügung, die sich zum Aufbau eigener Basisdateien eignen.
("..../Microchip/MPLABX/v4.05/mpasmx/templates/")

Im folgenden (*bis Kapitel 2.6 Configuration Bits*) wird der Aufbau einer solchen eigenen Basis für einen PIC18FxxK22 Controller beschrieben. Die Resultate sind auch im Anhang zu finden.

Microchip stellt zwei Arten von Templates bereit. Einmal für "absolute" und einmal für "relocatable" angelegte Projekte. Eine Readme-Datei im obengenannten Verzeichnis informiert kurz über die wesentlichen Unterschiede. Weitere Informationen sind im User-Guide zu finden.

Der **"absolute"** Modus, bei dem der Programmierer die volle Kontrolle über jegliche Lokalisation von Programm-Code und Daten übernimmt, wird zwar noch oft verwendet, ist aber für die Praxis **veraltet und untauglich** und wird deshalb hier weder benutzt noch beschrieben.

Projekte oder Programm-Module, die in diesem Modus erstellt wurden, sind nur sehr schwer in anderen Projekten oder für andere Controller weiter zu verwenden.

Als Einstiegstemplate für einen PIC18FxxK22 kann man die Datei „**18F46K22TMPO.ASM**“ aus dem Unterverzeichnis *templates/Object* verwenden. Diese enthält die wichtigsten Abschnitte, welche in einem PIC Programm vorkommen können, aber nicht alle vorkommen müssen.

Während des Durcharbeiten dieses Kapitels ist es notwendig die entsprechenden Dateien in der IDE zu öffnen (*und auch dort hinein zu schreiben*) um ein Grundverständnis für den Aufbau zu bekommen ! Die Templates stammen oft noch aus der „Windows only“ Ära und verursachen auf anderen Systemen Probleme durch nicht beachteten Groß- und Kleinschreibung. (*Dateinamen...*)

13.4.10.1 Kommentare (;)

Ein wichtiger, oft sträflich vernachlässigter Teil eines Programms sind die Kommentare. In den Assembler Templates kann man relativ leicht erraten, dass es sich bei allem hinter einem Semikolon, um einen Kommentar handelt.

Der Kopf jeder Datei sollte einen Kommentar enthalten um was es sich eigentlich handelt und auch innerhalb der anderen Abschnitten sollte nicht aus Faulheit mit Kommentaren gespart werden.

Kommentare helfen dem Programmierer schon beim Schreiben des Programms sich genau darüber klar zu werden, was da eigentlich getan werden soll. Spätestens beim Überarbeiten eines Programms sind sie von sehr großer Hilfe, da meistens nicht mal der ursprüngliche Entwickler auf den ersten Blick erkennen kann was da getan wird und warum genau die jeweiligen Befehle verwendet wurden.

13.4.10.2 MPASM Assembler Direktiven

Neben dem eigentlichen Programmcode braucht der Assembler noch weitere Anweisungen von denen einige in den Templates auftauchen. Eine komplette Liste kann man sich im Handbuch für den Assembler/Linker anschauen. Im folgenden werden die im Template verwendeten Direktiven in der Reihenfolge ihres Auftauchens **kurz erklärt**. Für genauere Erklärungen und weitere Optionen ist der **User-Guide** des Assemblers gedacht.

13.4.10.2.1 list p= (list – LISTING OPTIONS)

list p=18f25k22

Es gibt mehrere List Direktiven. **list p=** setzt den Prozessortyp. Dieser wird allerdings auch von der IDE gesetzt und dem Assembler mitgeteilt. Stimmen die beide nicht überein kommt eine Warnung.

13.4.10.2.2 #include

#include <p18f25k22.inc> und #include "../uCQuick/uCQ_2013.inc"

Spezifiziert eine Datei deren Inhalt an dieser Stelle eingefügt werden soll. Handelt es sich bei der Include-Datei um eine vom System bereitgestellte Datei, dann kennzeichnet man das dadurch, dass der Dateiname in spitze Klammern <System_eigen.inc> eingebunden wird.

Benutzerdefinierte Dateien werden in Anführungszeichen gesetzt "**Benutzer.inc**". Es können auch **Pfade** zu der einzufügenden Datei, ausgehend vom Projektverzeichnis, mit angegeben werden. Müssen vom aktuellen Verzeichnis Ebenen im Verzeichnisbaum nach oben angegeben werden dann geschieht das durch Verwendung des bekannten "./". Dies kann durch Aneinanderreihungen auch über mehrere Ebenen realisiert werden ("../../../meineHeader/meinHeader.inc")

Prinzipiell könnte man jede Art von Textdatei über die #include Direktive einfügen, da hier nur ein Kopieren des (*Text-*) Inhaltes initiiert wird. In nicht total veralteten Systemen, welche über einen sogenannten Linker verfügen, sollte man die #include Direktive nur verwenden um wirkliche Include Dateien einzubinden.

Die Funktion von Include Dateien

Include Dateien enthalten keinen Programmcode sondern hauptsächlich Definitionen, Deklarationen und Makros, die dem Programmierer die Arbeit erleichtern.

Ein Blick in die **prozessorspezifische Include Datei .../MPLABX/mpasmx/p18f25k22.inc** (*Datei öffnen über [Strg Taste] + Doppelklick auf den Namen in der Direktive des Templates*) zeigt viele Zeilen wie LATB EQU H'0F8A'.

Include-Dateien haben unter anderem eine Art Übersetzer Funktion. Der Assembler kann mit Registernamen wie LATB nichts anfangen. Er braucht die Adressen der Register, welche bei unterschiedlichen Controllern auch unterschiedlich sind. Ohne die prozessor-spezifischen Include-Dateien müsste der Programmierer immer die Adressen verwenden. (*auch EQU ist eine Direktive*). Nur die Include Datei ermöglicht die Benutzung von Registernamen wie z.B. LATB.

Projektspezifische Include-Dateien wie z.B. ../uCQuick/uCQ_2013.inc enthalten weitere Definitionen um den Code noch eine Stufe weiter zu abstrahieren.

#define nLED_1 LATB,LATB2,ACCESS (#define ist eine weitere Direktive ...) beispielsweise ermöglicht das Einschalten von LED_1 über ein relativ leicht verständliches bcf nLED_1. Ohne die beiden Include-Dateien müsste der Code bcf H'0F8A', 2, 0 lauten

Für die in den Include-Dateien über die Direktiven EQU oder #define eingeführten programmierer-freundlichen NAMEN werden bewusst GROSSBUCHSTABEN verwendet, um zu signalisierten (*einem anderen Programmierer*) dass es sich hierbei nicht um Variablen oder Unterfunktionen handelt.

13.4.10.2.3 config

Mit der Config Direktive werden die Einstellungen des Konfigurations-Wortes vorgenommen. Diese Konfiguration wird schon zum Zeitpunkt des Aufspielen des Programms auf den Controller und nicht erst bei der Ausführung des Programms eingestellt.

Ein einleuchtendes Beispiel dafür sollte die Konfiguration des Ausführungstaktes **config FOSC = INTIO7** sein.

Wenn kein externer Takt vorhanden ist, dann muss schon vor Ausführung des Programms festgelegt worden sein, dass der interne Takt benutzt werden soll, da sonst keine einzige Befehlszeile ausgeführt werden könnte. (*Also auch keine welche den internen Taktgeber aktivieren könnte*)

13.4.10.2.4 udata / udata_acs (udata_ovr, udata_shr, idata)

GPR_VAR UDATA / ACS_VAR UDATA_ACs sind Anweisungen welche dem Assembler / Linker vorgeben in welchem Bereich des Datenspeichers nachfolgende Variablen angelegt werden sollen und ob diese bei Programmstart initialisiert werden müssen (*udata -> uninitialized, idata ...)*

Die verschiedenen Datenspeicher-Bereiche ergeben sich bei 8bit PICs aus der Aufteilung des RAM in Banks, welche über ein zusätzliches **BANK-Selection-Register** adressiert werden. Dieses „**BSR**“ muss bei jedem Zugriff auf eine RAM Adresse neu eingestellt werden, wenn die Adresse in einer anderen BANK liegt als die Adresse auf welche beim vorhergehenden Befehl zugegriffen wurde.

Der Zugriff auf einen bestimmter Bereich, welcher *Access RAM* genannt wird kann ohne Umwege über das BSR erfolgen. Fast alle „**Special-Function-Register (SFR)**“ des PIC18 liegen deshalb in diesem Bereich um einen schnellen Zugriff zu ermöglichen.

Den „**General-Purpose-Register (GPR)**“ Teil des Access RAM kann man für eigenen Daten nutzen, auf die im Programm häufig zugegriffen wird.

GPR_VAR und ACS_VAR sind für den Assembler bedeutungslose, willkürlich vergebene Label.

13.4.10.2.5 res

Die Anweisung „**MYVAR_1 res 1**“ dient dazu Speicher im RAM in der Größe von einem Byte (**1**) für Daten zu reservieren. Als Name für den Zugriff auf die Daten wurde **MYVAR_1** vergeben.

An welcher genauen Adressen (die BANK und der genaue Ort in der BANK) wurde dabei nicht spezifiziert. Wo genau innerhalb einer Bank ist völlig egal und wird komplett dem Assembler / Linker überlassen. Ob AccessBank oder nicht, ergibt sich aus der udata / udata_acs Direktive.

13.4.10.2.6 code

RES_VECT CODE 0x0000 ist eine Anweisung an den Linker zur Platzierung von Code im Programmspeicher. Es gibt verschiedene Fälle in denen Code genau an einer bestimmten Adresse stehen muss.

Zunächst ist das der Programm-Code der ausgeführt werden soll, wenn der PIC eingeschaltet, oder ein **Reset** durchgeführt wird. In beiden Fällen wird der Zeiger auf die Adresse des auszuführenden Befehls im Programmspeicher auf Null gestellt. Durch „**CODE 0x0000**“ erreicht man das der nachfolgende Code genau an dieser Stelle beginnt.

Der nächste Fall tritt in dem Moment auf, in welchem die Ausführung des Programms durch einen **Interrupt** unterbrochen wird. Der Programmzeiger wird hier wieder auf genau definierte Adressen gestellt, an denen der Interrupt Code stehen muss.

Ein PIC18 verfügt über zwei **Interrupt-Vektoren (Adressen)** die auf **0x0008** und **0x0018** liegen. Weil der Bereich zwischen den Vektoren jeweils nur 8 Programmzeilen entspricht, können dort

meist nur Sprungbefehle stehen.

Ein **CODE ohne nachfolgende Adressangabe** signalisiert (*dem Linker*), dass der nachfolgende Code an einer beliebigen Stelle im Programmspeicher platziert werden kann.

RES_VECT, ISRHV, ISRLV, MAIN_PROG sind hier wieder für den Assembler bedeutungslos.

13.4.10.2.7 end

Die Direktive **END** kennzeichnet einfach das Ende der Programmanweisungen in einer Datei und steht nach der letzten Code-Zeile.

13.4.10.3 Aufsplittung des Assembler Templates in mehrere Dateien

Um die Übersichtlichkeit und Modularität zu erhöhen, kann man das mitgelieferte Template in ein „Config“-Template, ein „Main“-Template, ein „Interrupt“-Template und ein „eeprom“-Template aufsplitten. Die Interrupt und eeprom Templates müssen nur bei Bedarf in das Projekt eingefügt werden und werden auch erst später in den entsprechenden Kapiteln behandelt.

Im Anhang [13.2 Assembler Templates](#) sind entsprechende Dateien zu finden. Diese können als neues Gerüst für eigene Projekte verwendet werden.

13.4.10.3.1 Configuration Template

Die Datei **uCQ_config.asm** enthält nur noch den Abschnitt mit den „CONFIG“ Anweisungen. Diese werden zusätzlich mit Kommentaren und der Angabe der jeweiligen Optionen ergänzt.

13.4.10.3.2 Main Template

Der Code aus dem Template, der für jedes Programm unbedingt erforderlich ist, wird in eine neue Datei namens **uCQ_main.asm** geschrieben. Hier handelt es sich vor allem um den Reset-Vektor mit dem Sprungbefehl (*goto*) zum Startlabel *Init* (*Start*) des Programms. Dazu kommen die Beispiele für Variablendefinitionen und die Befehle für die Frequenzeinstellung des (*internal*) Oszillators.

Aufteilung der Hauptschleife in einmalige Initialisierung und Main-Loop

Der direkt auf das Startlabel folgende Code muss in der Regel nur ein einziges Mal beim Start des Programms durchlaufen werden. Er enthält die **Initialisierung** der zu schützenden Pins und Hardware-Module. Danach folgt eine Endlosschleife in der das **Hauptprogramm** läuft.

Bei **Assembler Projekten** kann dies durch ein zusätzliches Label „**Main**“, nach der Initialisierungsphase, realisiert werden. Der Sprung am Ende wird von „*goto Start*“ auf „*goto Main*“ abgeändert.

```
list      p=PIC18F25K22      ; list directive to define processor
#include <p18f25k22.inc>          ; processor specific definitions (LATB EQU H'0F8A')
#include "../uCQuick/uCQ_13.inc" ; project specific (#define nLED_1 LATB,LATB2,ACCESS)

GPR_VAR    udata             ;-----
;MYVAR_1    res   1           ; User variable linker places
ACS_VAR    udata_acs         ;-----
counter_1  res   1           ; variable in ACCESS RAM
RES_VECT   code   0x0000      ; processor reset vector-----
goto      Init              ; go to beginning of program (initialization)
MAIN_PROG  code             ; let linker place main program-----
Init
        movlw  0x00
        movwf  OSCCON
        bcf    LED_1_TRI
Main
        incfsz counter_1
        goto   Main
        btg    nLED_1
        goto   Main
END
```

Das Template enthält ein kleines Testprogramm, welches eine LED blinken lässt. So kann man sehr leicht erkennen, ob das Programm und der PIC auch richtig funktionieren.

13.4.10.4 Linker C18/MPASM

Die in Assembler und C-Compiler enthaltenen Linker sind dafür zuständig, aus den einzelnen Komponenten des Projekts, bestehend aus verschiedenen Asssembler Files, bzw. den vom Compiler erzeugten Object Files der C Dateien und den Bibliotheken eine einzige, ausführbare Datei zu machen. Hier wird nur kurz auf den Linker von Assembler (*und C18*) näher eingegangen.

13.4.10.4.1 Linker Script

Der Linker des Assemblers verwendet Linker Scripte. (*gilt auch für den alten C18 Compiler*)
Der XC8 Linker kennt keine Skripte und wird etwas anders gesteuert (*User-Guide ?*)

Wenn im Projekt kein spezielles (*eigens für das Projekt erstelltes*) Skript spezifiziert wird, dann benutzt der Linker immer das generische Skript aus dem Installationsordner:

ASM → .../Microchip/MPLABX/mpasmx/LKR

C18 → .../Microchip/mplabc18/v3.xx/bin/LKR

```
// File: 18f25k22_g.lkr - Generic linker script for the PIC18F25K22 processor

LIBPATH .

#ifndef _CRUNTIME
#ifndef _EXTENDEDMODE
FILES c018i.e.o
FILES clib_e.lib
FILES p18f25k22_e.lib
#else
FILES c018i.o
FILES clib.lib
FILES p18f25k22.lib
#endif
#endif

// Spezifikation des Programmspeichers (Vergleich mit Datenblatt)
CODEPAGE NAME=page START=0x00000000 END=0x7FFF PROTECTED
CODEPAGE NAME=idlocs START=0x20000000 END=0x200007 PROTECTED
CODEPAGE NAME=config START=0x30000000 END=0x30000D PROTECTED
CODEPAGE NAME=devid START=0x3FFFFFFE END=0x3FFFFFFF PROTECTED
CODEPAGE NAME=eedata START=0x40000000 END=0xF000FF PROTECTED

// Spezifikation des Datenspeichers (Vergleich mit Datenblatt)
#ifndef _EXTENDEDMODE
DATABANK NAME=gpre START=0x0 END=0x5F
#else
ACCESSBANK NAME=accessra START=0x0 END=0x5F
#endif

DATA BANK NAME=gpr0 START=0x60 END=0xFF
DATA BANK NAME=gpr1 START=0x100 END=0x1FF
...
DATA BANK NAME=sr15 START=0xF38 END=0xF5F PROTECTED
ACCESSBANK NAME=accessssfr START=0xF60 END=0xFFFF PROTECTED

#ifndef _CRUNTIME
SECTION NAME=CONFIG ROM=config
#ifndef _DEBUGSTACK
STACK SIZE=0x100 RAM=gpr4
#else
STACK SIZE=0x100 RAM=gpr5
#endif
#endif
```

NEIN

Die roten Markierungen verdeutlichen die Unterschiede zwischen dem C18 Skript und dem XC8 Skript:

- LIBPATH .**: In XC8 ist dies nicht erforderlich.
- CODEPAGE**: Gilt für C18 Projekte, nicht für XC8.
- DATABANK**: Gilt für Extended Modus, nicht für Normal Modus.
- ACCESSBANK**: Gilt für Normal Modus (nicht Extended).
- DATA BANK**: Gilt für C18 Projekte, nicht für XC8.
- SECTION**: Gilt nur bei C18 Projekten.
- STACK**: Gilt im Debug Modus (Software) Stack für Variablen.
- RAM**: Gilt im Release Modus (Software) Stack für Variablen.

Über das Skript bekommt der Linker die Information wie der Speicher des betreffenden PICs aufgebaut ist. Mit **FILES** werden dem Projekt Bibliotheken und Object-Dateien hinzugefügt.

13.5 ANSI C mini-Guide

Dieses Kapitel ist nicht zum Erlernen der Sprache C geeignet. Es kann lediglich dabei helfen Erinnerungen aufzufrischen oder die richtigen Schlagwörter für eine zielführende Recherche im Internet zu finden!

Eine gute Einführung zum Thema C auf Mikrocontrollern bietet z.B. [Fundamentals of the C Programming Language \(www\)](#) auf der Microchip Wiki Seite.

13.5.1 *Kommentare*

13.5.2 *#include Direktive*

13.5.3 *Variablen*

13.5.3.1 *Data Type*

13.5.3.2 *Identifier*

13.5.4 *Konstanten*

13.5.5 *Operatoren*

13.5.6 *Expressions / Statements*

13.5.7 *Bedingungen (Verzweigungen)*

13.5.8 *Loops (Schleifen)*

13.5.9 *Functions*

13.5.10 *Arrays and Strings (Felder und Zeichenketten)*

13.5.11 *Pointers (Zeiger)*

13.5.12 *Structures, Bit fields, Unions*

13.5.13 *Enumerations*

13.5.14 *Macros (#define)*

14 Notizen

14.1 Dringend erforderliche Korrekturen & Ergänzungen

- ~~UMSTELLUNG auf pic-as Assembler (MPASM Stuff in speziellen Anhang)~~
- ~~UMSTELLUNG auf XC8 Compiler. (C18 Stuff in einen speziellen Anhang)~~
- Kapitel 3.1.6 Assemblieren / Compilieren des Programms
- Link BANK → Befehle einfügen ?
- ~~IO Beispiele in Kapitel 3.2 und 3.3 ausarbeiten (beschreiben)~~
- geändertes Kapitel „Reaktionsspiel MG“ auf 4-LED überprüfen (3 LED war wegen uCQ-2012)
- ~~Templates im Anhang auf 2013er Stand bringen !!!!!!!!!!!!!!! (auch die Kommentare)~~
- ~~Package Funktion in MPLABX erwähnen / beschreiben (12.1.3 Paeken ?)~~
- Stack (Software) des C-Compilers ... ()
- Datentypen (Links)
- Index anfügen ...
- 1.3.4 Object Files und Bibliotheken bla, bla ...
- ~~flags bitfeld von Beispielen erklären !!!~~
- ~~1.2.3.5.2 ff I2C / SPI ergänzen~~
- ~~Encoder Auswertung per Polling (wie Tasten) (auch Literaturhinweise)~~
- ~~Keine Plib ab XC8 v1.35 (muss separat installiert werden)~~
- ~~Microchip Code Configurator (erwähnen)~~
- ~~PICKIT 3 Power muss manchmal kleiner 5V eingestellt werden~~
- ~~3.1.6.2 Fehlermeldungen → Beispiele !!!~~
- 13.1 Fehlermeldungen untergliedern in Compiler, PICkit ...
- ~~Links zu Beispielen auf die HS Ulm Seite umlegen!!!~~

Fehlerhafte Referenzen

- 12.x.x..