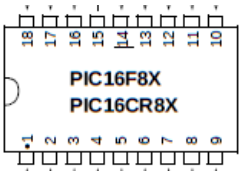


PIC-SIM Dokumentation

des Studienganges Informationstechnik

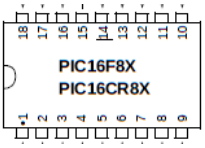
an der Dualen Hochschule Baden-Württemberg Karlsruhe

von

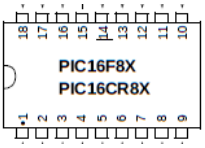
Marcel Fischer & Christian Misch

Abgabedatum 10.07.2023

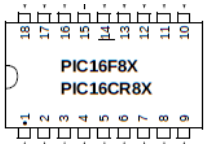
Kurs	TINF21B5
Gutachter	Stephan Lehmann



Inhalt



Abbildungsverzeichnis



1. Einleitung

1.1 Grundsätzliche Arbeitsweise eines Simulators

Simulatoren sind computergestützte Systeme, die dazu dienen, reale oder abstrakte Systeme zu modellieren und nachzuahmen. Sie werden in verschiedenen Bereichen eingesetzt, wie beispielsweise in der Luftfahrt, im Militär, in der Medizin, im Verkehrswesen und in der Forschung.

Ein Simulator ermöglicht es, komplexe Vorgänge und Szenarien zu simulieren, um diese besser zu verstehen, zu analysieren und zu optimieren. Durch die Verwendung von Modellen und Algorithmen werden die Dynamiken und Interaktionen des realen Systems mathematisch abgebildet und in Echtzeit dargestellt.

Die Qualität und Genauigkeit eines Simulators hängen von verschiedenen Faktoren ab, wie der Qualität der verwendeten Modelle, der Richtigkeit der Eingabedaten, der Komplexität des zu simulierenden Systems und der Leistungsfähigkeit der verwendeten Hardware und Software.

Simulatoren bieten eine kostengünstige und sichere Möglichkeit, reale Szenarien nachzubilden und zu erforschen, ohne dabei potenzielle Risiken oder Kosten zu tragen, die mit dem Testen oder der Durchführung von Experimenten im realen Leben verbunden wären. Sie ermöglichen es, verschiedene Szenarien zu analysieren, Hypothesen zu überprüfen und neue Erkenntnisse zu gewinnen.

1.2 Vor- und Nachteile

Eine Simulation bietet verschiedene Vorteile, aber es gibt auch einige Nachteile, die berücksichtigt werden sollten.

Vorteile einer Simulation:

1. Realitätsnahe Darstellung: Eine Simulation kann eine realitätsnahe Darstellung eines Prozesses oder eines Systems bieten, was es ermöglicht, verschiedene Szenarien zu testen und potenzielle Probleme frühzeitig zu identifizieren.
2. Kosten- und Zeitersparnis: Durch die Durchführung einer Simulation können Kosten und Zeit gespart werden, da teure und zeitaufwändige physische Tests vermieden werden können. Stattdessen können virtuelle Tests durchgeführt

werden, um die Auswirkungen von Änderungen oder Entscheidungen zu analysieren.

3. Risikominimierung: Eine Simulation ermöglicht es, Risiken zu minimieren, indem verschiedene Szenarien durchgespielt werden können, ohne dass reale Konsequenzen auftreten. Dadurch können potenzielle Probleme oder Engpässe im Voraus erkannt und entsprechende Maßnahmen ergriffen werden.

Nachteile einer Simulation:

1. Vereinfachte Annahmen: Eine Simulation basiert auf bestimmten Annahmen und Vereinfachungen, um den Prozess oder das System darzustellen. Dies kann dazu führen, dass die Simulation nicht alle Aspekte und Komplexitäten der Realität abbildet, was zu ungenauen Ergebnissen führen kann.
2. Datenbedarf: Eine Simulation erfordert eine solide Datenbasis, um genaue Ergebnisse zu liefern. Wenn die verfügbaren Daten unzureichend oder fehlerhaft sind, können die Ergebnisse der Simulation ebenfalls ungenau sein.
3. Komplexität und Lernaufwand: Die Durchführung einer Simulation erfordert spezifisches Fachwissen und technische Fähigkeiten, um das Modell zu erstellen und die Ergebnisse richtig zu interpretieren. Dies kann einen hohen Lernaufwand bedeuten und zusätzliche Kosten für Schulungen oder die Einstellung von Fachleuten mit sich bringen.

1.3 Programmoberfläche und deren Handhabung.

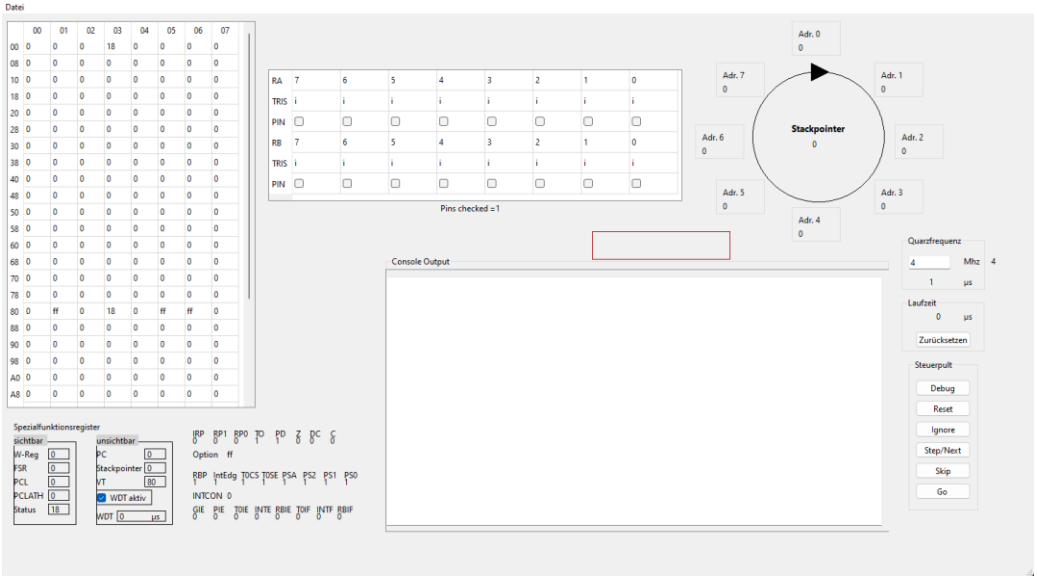


Abbildung 1: Programmoberfläche des Simulators

Abbildung 1 zeigt die gesamte Oberfläche. In unserem Programm sieht man oben links alle unsere Register, diese sind nicht bearbeitbar und zeigen nur die aktuellen Werte, die gespeichert sind. Unter dem DataSpeicher sind die Spezialfunktionsregister abgebildet (Abbildung 2), mit einer Check Box, welche den Watchdog ein- oder ausschalten kann.

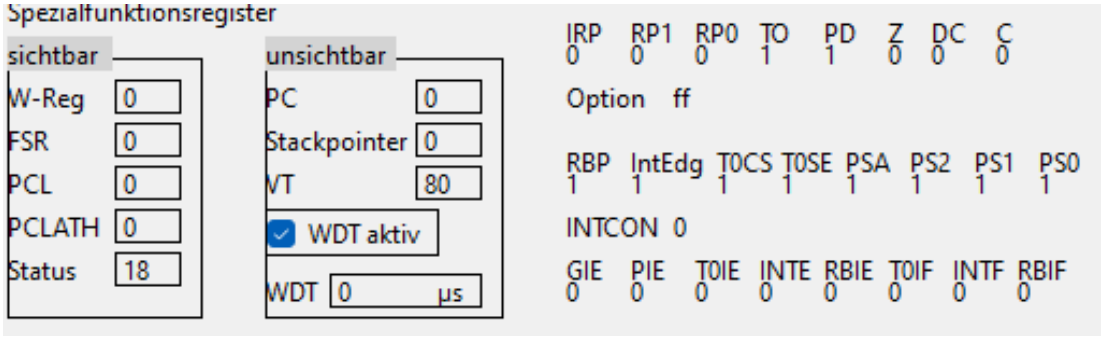


Abbildung 2: Spezialfunktionsregister

In Abbildung 3 ist unser Pin Table zu sehen. Die Checkboxen in der Tabelle zeigen an, ob ein Pin auf 1 oder 0 steht. Rechts daneben ist unsere Visualisierung des Stack-Pointers zu sehen. Das große weiße Feld darunter in Abbildung 4 ist die Programmausgabe, welche das zu abarbeitende Programm anzeigt und den aktuellen Befehl mit einem Pfeil markiert.

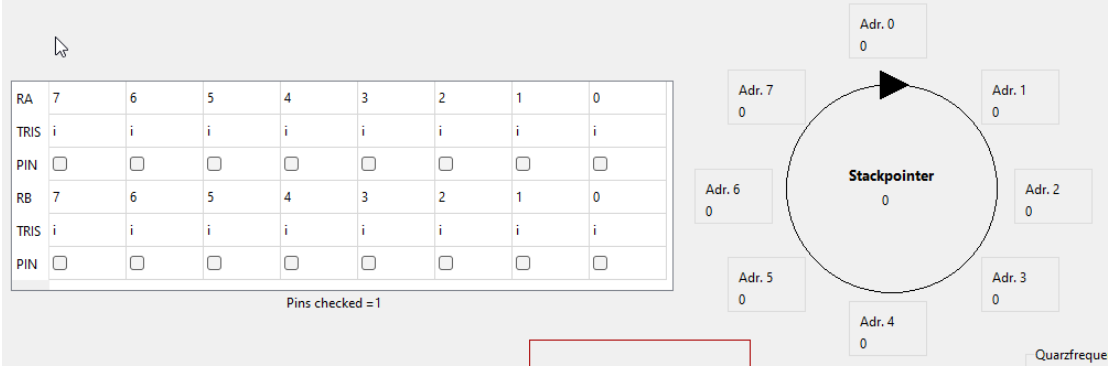


Abbildung 3: IO-Pins und Visualisierung des Stacks

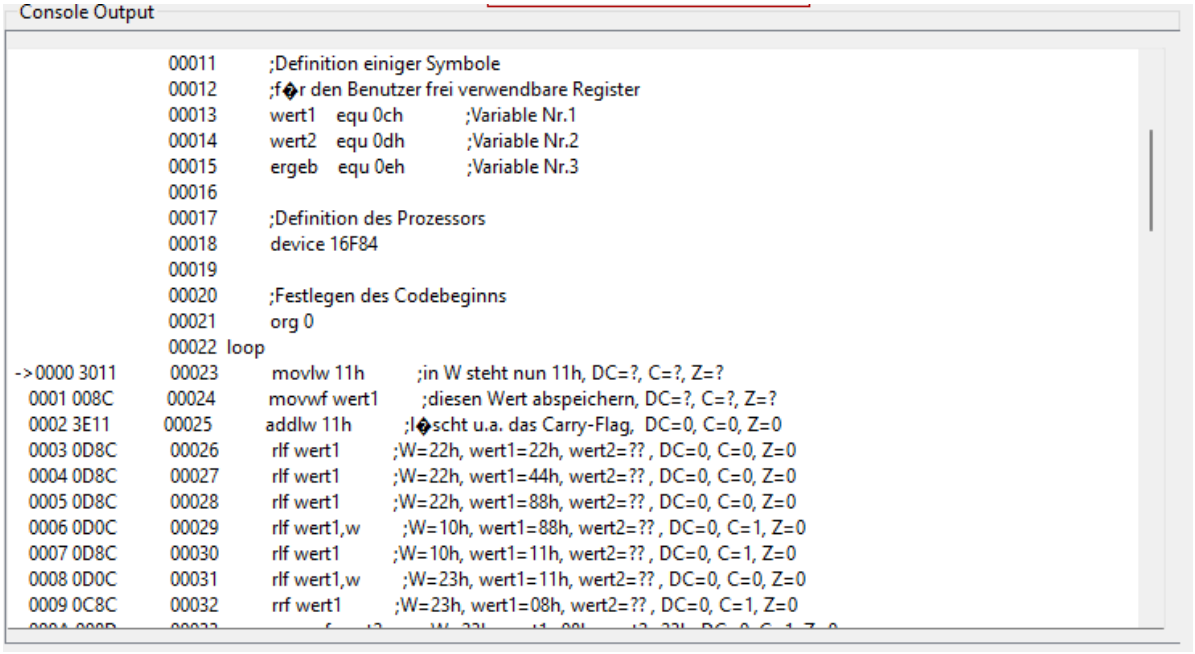
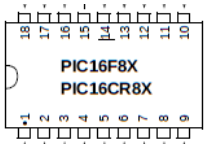


Abbildung 4: Programmausgabe

Abbildung 5: Steuerpult

Ein Programm kann geladen werden, indem man oben links im Menü Band auf Datei klickt und sein Testprogramm im Dateiexplorer auswählt.

Abbildung 5 zeigt den Bereich, in dem wir die wichtigsten Steuerungsbuttons platziert haben. Hier befindet sich oben die bearbeitbare Quarzfrequenz gefolgt von der Laufzeit mit einem Reset Button. Im Bereich Steuerpult sind die benötigten Buttons, um den Simulator durch ein Testprogramm zu führen.



2. Realisation

Die Umsetzung des PIC-Microprozessors als Simulator stellte eine größere Herausforderung da Aufgrund der Größe des Projektes wurde das Vorhaben in verschiedene kleinere Etappen verteilt. Während der Entwicklungsphase wurde primär zwischen der Benutzeroberfläche (GUI) und der Logik unterschieden. Die Logik selbst wurde ebenfalls unterteilt. Die Hauptbereiche waren die Befehle, die Register-Funktionen und das Einlesen eines Programmes. Hierauf wird in Kapitel 2.1.2 Programmstruktur & Variablen genauer eingegangen.

2.1 Grundkonzepts

2.1.1 Welche Programmiersprache wurde gewählt?

Nachdem sich mit den Grundfunktionen des PIC vertraut gemacht wurde musste zunächst die Programmiersprache gewählt werden. Eine gut durchdachte Wahl ist hier sehr wichtig, da die persönlichen Erfahrungen und Kenntnisse bei den verschiedenen Programmiersprachen abweichen. Außerdem musste dies mit den gebotenen Funktionen der Sprachen ins Verhältnis gesetzt werden, da manche Sprachen besser und auch schlechter für das Vorhaben geeignet sind. Die Implementationsmöglichkeiten einer GUI sind von Sprache zu Sprache ebenfalls unterschiedlich.

In unserer Gruppe war grundsätzliches Vorwissen in C, C++, Java und zusätzlich HTML, CSS und Java-Skript vorhanden. Es erschien daher logisch eine dieser Sprachen auszuwählen. Aufgrund der noch nicht langen zurückliegenden Vorlesungen zum Programmieren mit C++, dem daher guten Wissenstandes und den Möglichkeiten im Bezug auf den Umgang mit Strings, globalen Variablen, der Aufteilung auf verschiedene Dateien und weiteren Möglichkeiten, welche sich für das Projekt anbieten, wurde diese als Sprache für das Projekt ausgewählt. Einzig der Implementation einer GUI ist mit C++ mit einigen Problemen verbunden, welche zu Beginn noch nicht gelöst waren. Zunächst wurde Visual C++ (Forms) in Betracht gezogen. Jedoch erwies sich dies in Verbindung mit C++ als wenig intuitiv, sehr kompliziert und umständlich und schlecht dokumentiert. Daher wurde die Auswahl der GUI zunächst verschoben und mit der Implementation der Logik in einem simplen C++ Konsolen-Projekt begonnen.

Im Verlauf des Projektes und nach Betrachtung verschiedener GUI-Implementationen, unter anderem auch „WebView“ stellte sich dann das Framework QT als geeignete GUI für das Projekt heraus und der Simulator wurde mit dessen Hilfe umgesetzt.

2.1.2 Programmstruktur & Variablen

Wie bereits genannt besteht das Programm aus den zwei Hauptteilen GUI und Logik. Die GUI wird mit Hilfe der Mainwindow.cpp und mainwindow.ui Dateien erzeugt. Die Logik ist über verschiedene weitere Dateien aufgeteilt. Folgende Abbildung 6 zeigt die Dateistruktur des Simulators.

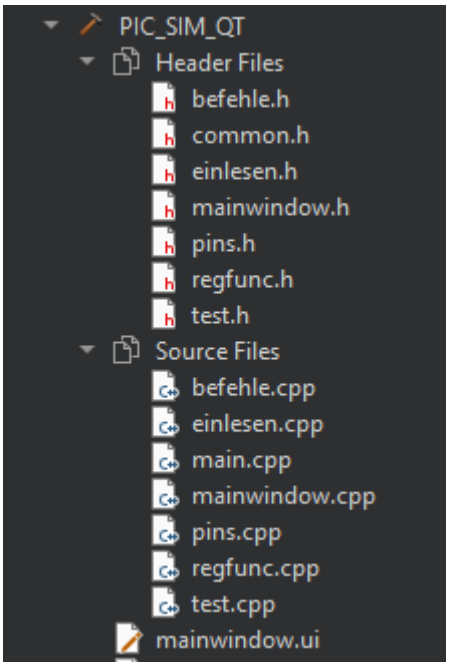
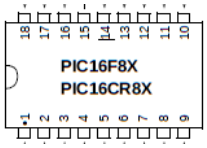


Abbildung 6: Dateistruktur

In C++ ist es üblich zwischen Source- und Header-Dateien zu unterscheiden. Die Headerdateien beinhalten dabei die grobe Struktur der zugehörigen Source-Datei mit den Funktionsdeklarationen aber noch ohne Implementation. Dies ermöglicht das Verknüpfen der unterschiedlichen Dateien. Dadurch können Funktionen auf alle anderen verknüpften Funktionen zugreifen.

Aus der Abbildung ist zu erkennen, dass die Logik des Simulators über eine Vielzahl an Dateien verteilt wurde. Dies dient primär der Übersichtlichkeit. So konnten die verschiedenen Funktionen aufgrund ihrer logischen Aufgabe gruppiert werden.

Unser Simulator ist auf der Basis von globalen Variablen und eigenständiger Funktionen gebaut. Dies wurde gewählt, da der Umgang mit Klassen nicht sehr vertraut war und ein Einarbeiten in diese Thematik das Vorhaben weiter verlangsamt hätte. Die Verwendung von Funktionen in Kombinationen mit globalen Variablen war daher naheliegender, wenn auch nicht die sauberste Lösung. Zusätzlich gibt es für den Zugriff auf einzelne Bits der besonderen Register (Status, Option usw.) jeweils sogenannte Get und Set Funktionen. Dies vereinfacht das Schreiben und Lesen einzelner Bits, da die Logik für das Maskieren (beim Lesen) und verrunden/ordern (beim Schreiben) zentralisiert im wieder verwendet werden kann. Die beiden Speicherbän-



ke sind als ein zweidimensionales Array implementiert. Der Programmspeicher und Stack ist ebenfalls als Array implementiert. Datentyp für alle acht-bit-Werte und somit auch für die Speicherbänke ist uint8_t. Dadurch wird falsches verhalten bei einem überlauf oder dem Bilden des Komplementes verhindert. Bei Rechnungen wird jedoch in Integer umgewandelt, um einen überlauf sicher feststellen zu können,

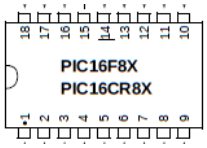
In der main.cpp werden oberhalb der main Funktionen alle globalen Variablen und Arrays deklariert. Diese sind zuvor als „extern“ in der Datei common.h definiert worden. Dies ist notwendig, wenn in C++ globale Variablen verwendet, werden sollen.

Die Main-Funktion des Simulators beinhaltet den Aufruf der Funktionen bootPIC(), welche die internen Variablen auf die beim boot gewünschten Werte setzt. Danach findet sich der automatisch generierte Code zum Starten der GUI.

Die angesprochene bootPIC() – Funktion befindet sich in der Datei einlesen.cpp. Hier befinden sich die ähnlichen Funktionen resetPIC() und wdtResetPIC(). Diese werden aufgerufen, falls ein Reset über den Reset-Button oder ein Reset durch den Watchdog aufgetreten ist. Die folgende Abbildung 7 zeigt beispielhaft die wdtResetPIC()-Funktionen.

```
void wdtResetPIC() {

    qDebug() << "Wdt reset" ;
    wdtReset = 1;
    goLoop = 0;

    setTO(0);

    // Bits im Option reg auf 1 setzen
    dataSpeicher[1][1] = 0xff;

    // Bits im Intcon reg auf 0 setzen
    dataSpeicher[0][0x0B] = 0x00;
    dataSpeicher[1][0x0B] = 0x00;

    // ProgZeiger reset
    setProgZeiger(0);

    // Interne Vorteiler variable auf basis der PS bits setzen
    setPreVar(getPS());

}
```

Abbildung 7: Aufbau der wdtResetPIC()-Funktion



Die Funktion setzt ein Flag über das Auftreten des Watchdog-Resets und auf Basis des Datenblattes werden die geforderten Werte zurückgesetzt. Dabei ist hervorzuheben, dass in diesem Fall das TO-Bit auf 0 und die Option. Register auf 0xff gesetzt werden muss. Nach diesem Prinzip werden im gesamten Projekt die Werte gesetzt und zurückgesetzt.

Zusätzlich zu diesen Funktionen befindet sich in der einlesen.cpp die gesamte Logik zum Einlesen des Programm-Codes als String, das Extrahieren der Befehle und dem Speichern dieser in einem Array als Integer. Außerdem wird hier die globale Zeilennummer extrahiert und mit Hilfe eines Arrays die Befehle auf deren zugehörige globale Zeile gematcht. Dies ermöglicht das Markieren des nächsten Befehls im ausgegeben Programm-String. Dabei wird z.B. die Zeilennummer von Befehl 1 in einem Array matchZeile() an Index 1 eingefügt. Dies ermöglicht ein Abrufen der Zeilennummer für jeden Befehl mit aufwand O(1).

Die wichtigsten Funktionen befinden sich in den Dateien befehle.cpp und regfunc.cpp. Dabei sind die Funktionen zum Interpretieren der Befehle und die Funktionen für die Befehle selbst in der befehle.cpp abgelegt. Dort sind außerdem die Funktionen execBefehl() und die Funktionen für die Interrupts, Timer und Watchdog zu finden.

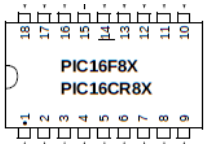
Alle Funktionen, welchen dem Umgang mit den Registern zuzuordnen sind, finden sich in der regfunc.cpp. Dabei handelt es sich um die Get und Set Funktionen für die wichtigen Bits und Register (Status, Option, Intcon, PCL und PCLATH) sowie für das allgemeine Schreiben und Lesen im Datenspeicher, Funktionen zum Synchronisieren der Speicherbänke und kleinere Hilfsfunktionen.

2.2 Implementation

Dieses Kapitel stellt die Implementationen einiger Funktionen genauer vor. Basis dieser befehle sind das Datenblatt und die Themenblätter.

2.2.1 Beschreibung der Funktionen

An alle Befehlsfunktionen wird ein Integer namens Data übergeben. Dabei handelt es sich um den im Programmspeicher abgespeicherten Befehl, welcher beim Einlesen von einem Hex-String zu einem Integer umgewandelt wurde. Diese Befehlsfunktio-



nen folgen alle einem ähnlichen Aufbau welcher im golfenden anhand einiger Beispiele erläutert wird.

BTFSx am Beispiel von BTFSC

Dieser Befehl testet das Bit b im Register der Adresse f und überspringt den nächsten Befehl, falls dieses 1 ist. Dadurch können zum Beispiel schleifen beendet werden. Im Folgenden in Abbildung 8 ein Ausschnitt aus dem Programmcode.

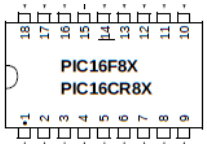
```
void btfsc(int data) {
    cout << "btfsc aufgerufen\n";
    // Testet Bit b an Adr. f und springt, wenn es 0 ist

    uint8_t f = 0x007f & data; // Register Pfad
    int b = 0x0380 & data;
    uint8_t reg = getRegInhalt(f);

    b = b >> 7; // shiften der bits nach ganz rechts um die korrekte zahl zu erhalten
    b = (1u << b);
    //qDebug() << "b" << b << "reg" << Qt::hex << reg << "verundung: " << Qt::hex << ((reg & b) > 0);
    if ((reg & b) > 0) {
        // bit b in reg an stelle f = 1 -> do nothing + next befehl
    }
    else {
        // bit b in reg an stelle f = 0, führe nop() aus und überspringe den nächsten befehl
        incProgZeiger(progZeiger+1);
        setTimer();
        checkWatchdog();
        nop();
    }

    takte += 4;

}
```

Abbildung 8: Befehl BTFSC

Zunächst wird mit Hilfe von Masken die Adresse f und das Bit b aus dem übergebenen integer Data extrahiert. Zusätzlich wird in der variable reg der Inhalt des Registers an der Adresse f lokal abgespeichert. Dies ermöglicht übersichtlicheren Code.

Zunächst müssen die Bits in b um 7 Stellen nach rechts geschoben werden, damit die korrekte Zahl rauskommt, da b als alleinstehende Zahl betrachtet werden muss.

Anschließend wird b mit einer neuen Zahl überschrieben, welche mit Hilfe des „shiften“ Operators eine Zahl darstellt, welche überall 0 ist außer an Bit b, dieses ist 1. Mit diesem b wird eine Verrundung mit reg durchgeführt. Hier findet dann in der if-Anweisung die Prüfung des Bits b im Inhalt (reg) des Registers f statt. Abhängig von diesem Ergebnis, in diesem Fall, wenn das geprüfte Bit = 0 ist, wird dann der Programzeiger mit einer eigenen Funktion inkrementiert. Dadurch wird gleichzeitig das PCL-Register mit erhöht und ein überlauf abgefangen. Außerdem werden mit setTimer() und checkWatchdog() der Timer aktualisiert und der Watchdog angepasst, da



anschließend ein nop() Befehl durchgeführt. Bei einem normalen Befehl mit vier Zyklen werden Timer und Watchdog nach der Abarbeitung des Befehls in execBefehl() abgearbeitet.

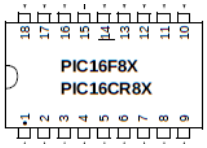
CALL

Der Call Befehl springt mit dem Programmzeiger an eine übergebene Adresse k und berücksichtigt dabei das PCLATH-Register. Die folgende zeigt den Programmcode des Call Befehles (der Wechsel zwischen englischen und deutschen Kommentaren kommt durch das Kopieren der Befehls Beschreibungen aus dem Datenblatt in Kombination mit eigenen Kommentaren).

```
void call(int data) {
    cout << "call aufgerufen\n";
    // Unterprogrammaufruf an Adresse k
    int k = data & 0x7ff;

    qDebug() << "k: " << Qt::hex << k << "\n";

    // First, return address (PC + 1) is pushed onto the stack.
    pushStack(progZeiger);
    cout << "stack[0]: " << stack[0] << "\n";

    // The eleven bit immediate address is loaded into PC bits <10:0>.
    setProgZeiger(k);

    //The upper bits of the PC are loaded from PCLATH.
    int pclath =  getPCLATH() & 0x18;
    pclath = pclath << 8;

    progZeiger = progZeiger & 0x7ff;
    progZeiger = progZeiger | pclath;

    takte += 4;

    // set Timer
    setTimer();
    checkWatchdog();

    takte += 4;
}
```

Abbildung 9: Befehl Call

Zunächst wird k aus Data extrahiert. Anschließend wird die Return-Adresse mit der Funktion pushStack() auf dem Stack abgelegt. Danach wird der Programmzeiger auf den Wert von k gesetzt. Der Inhalt des Registers PCLATH wird lokal gespeichert, das vierte und fünfte Bit werden mit einer Maske isoliert, um 8 Stellen verschoben



und anschließend wir der damit erhaltende Wert mit dem Programm-Zeiger verodert. Da es sich bei diesem Befehl um einen Befehl mit einer Dauer von acht Zyklen handelt wird anschließend wird der Taktzähler um vier Takte erhöht, der Timer und Watchdog abgearbeitet und der Taktzähler erneut um vier erhöhte. Die erneute Anpassung von Timer und Watchdog findet dann wie bei allen anderen befehlen am nächsten Schritt innerhalb von execBefehl() statt.

MOVF

Der Befehl movf verschiebt den Inhalt in Register f abhängig von Bit d in das W-Register oder in das Register f selbst. Abbildung 10 zeigt den zugehörigen Code.

```
void movf(int data) {
    cout << "movf aufgerufen\n";
    /*
    The contents of register f is moved to a
    destination dependant upon the status
    of d.If d = 0, destination is W register.If
    d = 1, the destination is file register f
    itself.d = 1 is useful to test a file register
    since status flag Z is affected.
    */

    uint8_t d = (0x0080 & data) > 0; // d bit
    uint8_t f = 0x007f & data; // Register Pfad
    uint8_t reg = getRegInhalt(f); // Register f Inhalt
    uint8_t result = reg;

    int z = (result == 0);   // Setzt das Zero-Flag basierend auf dem Ergebnis

    if (d == 1)   // Wenn d = 1, wird das Ergebnis in das Register geschrieben
    {
        setRegInhalt(f, (uint8_t)result);
    }
    else   // Wenn d = 0, wird das Ergebnis in WREG geschrieben
    {
        wReg = result;   // speichere wert von Ref in WReg
    }

    setZ(z);
    takte += 4;

}
```

Abbildung 10: Befehl movf

Zunächst werden das bit b, die Adresse f aus data extrahiert. Anschließend wird der Inhalt von f lokal als reg gespeichert. Die lokale Variable Resul wird mit reg befüllt. Außerdem wird eine lokale Variable z abhängig von result auf 1 oder 0 gesetzt. In der folgenden if-Anweisung wird result abhängig von d in Register f oder in das wReg

gespeichert. Anschließend wird das Zero-Flag auf den Wert von z gesetzt und die Takte um 4 erhöht.

RRF

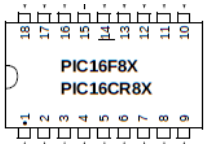
Diese Funktion rotiert den Inhalt in Register f mit Hilfe des Carry-Bits nach rechts. Die folgende Abbildung 11 zeigt den Programmcode.

```
void rrf(int data) {
    cout << "rrf aufgerufen\n";
    uint8_t d = (0x0080 & data) > 0; // d bit
    uint8_t f = 0x007f & data; // Register Pfad
    uint8_t reg = getRegInhalt(f);
    uint8_t result;
    int c = getC();
    int newC;

    if (reg & 0x01) {
        newC = 1;
    }
    else {
        newC = 0;
    }

    result = reg >> 1;

    if (c == 1) {
        result |= 0x80;
    }

    if (d == 1)    // Wenn d = 1, wird das Ergebnis in das Register geschrieben
    {
        setRegInhalt(f, (uint8_t)result);
    }
    else    // Wenn d = 0, wird das Ergebnis in WREG geschrieben
    {
        wReg = result;
    }

    setC(newC);
    takte += 4;
}
```

Abbildung 11: Befehl RRF

Zunächst werden wie bereits zuvor ausgführt die bits d, die Adresse f und der Inhalt von f lokal gespeichert. Außerdem werden die lokalen variablen result, c und newC deklariert und c wird mit dem Inhalt des Carry-Flags befüllt.

Anschließend wird die Rotation durchgeführt. Dabei wird das Bit ganz rechts in die variable newC überführt und alle anderen Bits nach rechts verschoben und in result gespeichert. Danach wird das alte Carry mit einem „oder“ an result angefügt. Das Ergebnis wird dann abhängig von d in f oder dem wReg abgelegt. Anschließend wird das Carry-Flag per Funktion auf den neuen Wert newC gesetzt und die Takte erhöht.



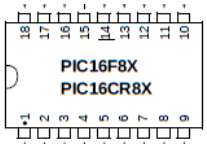
SUBWF

Diese Funktionen subtrahiert den Inhalt des wReg vom Inhalt des Registers an Adresse f. Die Abbildung 12 zeigt den zugehörigen Programmcode.

```
void subwf(int data) {
    cout << "subwf aufgerufen\n";
    uint8_t d = (0x0080 & data) > 0; // d bit
    uint8_t f = 0x007f & data; // Register Pfad

    int c = 0;
    int z = 0;
    int dc = 0;
    int result;
    int reg = (int)getRegInhalt(f); // Reg an Stelle f inhalt

    //DC berechnung
    // DC: bei add: > 15 -> dc = 1, <= 15 -> dc = 0 ; sub: < 0 -> dc = 1, >= 0 -> dc  0

    if ( ((reg & 0x0f) - (wReg & 0x0f)) < 0) {
        dc = 0; // durch "fehler" im PIC eigentlich falsch, müsste = 1 sein
    }

    // Berechnung
    result = reg - wReg;   // subtrahiere wert im register f von wREG

    // Flags festlegen
    if (result < 0) {
        // Setzt das Carry-Flag basierend auf dem Ergebnis
        c = 0; // durch "fehler" im PIC eigentlich falsch, müsste = 1 sein
        result += 256;
    }
    else {
        c = 1;
    }
    z = (result == 0);   // Setzt das Zero-Flag basierend auf dem Ergebnis

    // Speicherort ermittlen
    if (d == 1)    // Wenn d = 1, wird das Ergebnis in das Register geschrieben
    {
        setRegInhalt(f, (uint8_t)result);
    }
    else   // Wenn d = 0, wird das Ergebnis in WREG geschrieben
    {
        wReg = (uint8_t)result;
    }

    // setze Flags
    setC(c);
    setDC(dc);
    setZ(z);

    // Erhöhe Takte
    takte += 4;
}
```

Abbildung 12: Befehl SUBWF

Zu Beginn werden das bit b und Registeradresse f extrahiert und lokale Variablen für die Status Flags, result und dem Inhalt von register f (als reg) deklariert. In diesen Funktionen wird mit Integer-Variablen gearbeitet, damit ein Überlauf erkennbar wird.



Vor der Berechnung wird zunächst mit Hilfe von Masken und einer Hilfsrechnung das DC Flag bestimmt. Hierbei ist zu beachten, dass dies beim PIC invertiert ist. Danach wird die Berechnung durchgeführt und in result abgelegt. Danach folgt durch Abfrage, ob das Ergebnis die 0 durchbrochen hat, die Bestimmung des Carry-Flags, welches ebenfalls beim PIC invertiert. Z berechnet sich anhand von result. Zuletzt wird anhand von d der Speicherplatz bestimmt und das Ergebnis dort abgelegt. Danach werden noch die Flags gesetzt und die Takte erhöht.

DECFSZ

Dieser Befehl dekrementiert den Inhalt in Register f um 1 und überspringt den nächsten Befehl, falls das Ergebnis 0 ist. Folgende Abbildung 13 zeigt den zugehörigen Code.

```
void decfsz(int data) {
    cout << "decfsz aufgerufen\n";

    uint8_t d = (0x0080 & data) > 0; // d bit
    uint8_t f = 0x007f & data; // Register Pfad
    uint8_t reg = getRegInhalt(f); // Register f Inhalt

    uint8_t result = reg-1;

    if (d == 1)    // Wenn d = 1, wird das Ergebnis in das Register geschrieben
    {
        setRegInhalt(f, (uint8_t)result);

    }
    else    // Wenn d = 0, wird das Ergebnis in WREG geschrieben
    {
        wReg = result;
    }

    takte += 4;

    if (result == 0) {
        progZeiger++;
        setTimer();
        checkWatchdog();
        nop();
    }
}
```

Abbildung 13: Befehl DECFSZ

Wie üblich werden Bit d und Adresse f extrahiert und der Inhalt von f in reg abgelegt. Die lokale Variable result wird gleich reg-1 gesetzt. Danach wird anhand von d der Speicherort bestimmt und result dort abgespeichert. Danach folgt die Erhöhung des

Taktes. Falls result = 0 ist wird anschließend der nächste Befehl übersprungen und Timer und Watchdog abgearbeitet, da anschließend ein nop() folgt.

XORLW

Dieser Befehl führt eine exklusive oder Verknüpfung zwischen dem wReg und dem Literal k durch. Abbildung 14 zeigt den zugehörigen Code.

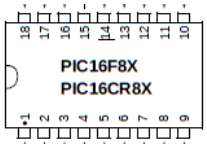
```
void xorlw(int data) {
    cout << "xorlw aufgerufen\n";
    // EXCLUSIV ODER Verknüpfung von W und k

    uint8_t k = 0x00ff & data; // Literal k Pfad

    // Ergebniss ermitteln
    uint8_t result = wReg ^ k;

    int z = (result == 0);   // Setzt das Zero-Flag basierend auf dem Ergebnis

    // Speichern im wReg
    wReg = result;


    // Flags setzen
    setZ(z);

    // Takte erhöhen
    takte += 4;

}
```

Abbildung 14: Befehl XORLW

Zunächst wird das literal k aus Data extrahiert. Anschließend wird das Ergebnis der XOR-Verknüpfung lokal als result abgelegt. Variable z wird anhand des Ergebnisses bestimmt (z = 1,falls result == 0, sonst z = 0) und anschließend wird result im wReg abgelegt. Der Wert von z wird in das Zero-Flag übernommen und danach werden die Takte erhöht.

2.2.2 Realisierung der Flags und deren Wirkungsmechanismen

Im Folgenden wird die Realisierung der Flags am Beispiel des Status-Registers erläutert. Bei Flags handelt es sich um einzelne Bits aus speziellen Registern, welche von besonderer Wichtigkeit sind. Damit die Bits einzeln abgerufen und geschrieben werden können wurde eine getStatus() und setStatus() Funktion entworfen. Die folgende Abbildung 15 zeigt beispielhaft die getStatus() -Funktion:




```
int getStatus(uint8_t maske) {

    // lese status aus dem aktuellen Datenspeicher
    uint8_t status = dataSpeicher[0][3];

    if(maske == 0xff) {
        return status;
    } else {
        return ((status & maske) > 0);
    }
}
```

Abbildung 15: getStatus()-Funktion

Die Funktion bekommt eine Maske übergeben, welche dann auf das lokal abgespeicherte Statusregister gelegt wird. Diese Funktion ist dabei nur dafür gedacht, dass einzelne bits abgerufen werden, da als Ergebnis durch „> 0“ nur 0 oder 1 zurückgegeben wird. Damit in speziellen Fällen bei einer Maske von 0xff trotzdem das gesamte Register zurückgegeben werden kann, wurde die if-Anweisung hinzugefügt.

In weiteren Funktionen für jedes Flag wird diese Funktion dann mit entsprechender Maske verwendet wie folgende Abbildung 16 zeigt.

```
int getC() {
    return getStatus(maskeC);
}
```

Abbildung 16: getC()-Funktion

Dies ist zwar zunächst ein gewisser Programmieraufwand, ermöglicht jedoch später einen sehr einfachen Zugriff auf die Flags.

Zum Setzen der Status-Flags wurde dasselbe Prinzip angewandt. Folgende Abbildung 17 zeigt die setStatus()-Funktion.

```
void setStatus(uint8_t maske, int wert) {

    uint8_t status;

    status = dataSpeicher[0][3];

    //int bit = (status & maske) > 0;

    if (wert == 0) {
        status &= ~(maske);
    }
    else if (wert == 1) {
        status |= maske;
    }

    dataSpeicher[0][3] = status;
    dataSpeicher[1][3] = status;

}
```

Abbildung 17: setStatus()-Funktion

An diese Funktion wird zusätzlich zur Maske noch ein Integer namens „wert“ übergeben. Außerdem wird hier gleich die Synchronisierung auf beide Banken durchgeführt. Auch hier erhält jedes Flag seine eigene kleine Funktion, welche dann diese große Funktion aufruft.

2.2.3 Implementation der Interrupts

Interrupts sind eine wichtige Funktion des PIC und wurden wie folgt gelöst. Die einzelnen für den Interrupt notwendige Flags wurden wie oben anhand des Status-Registers beschrieben zugänglich gemacht. Die folgende Abbildung 18 zeigt die Implementation der Funktion, welche auf Interrupts prüft.

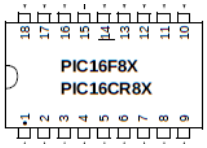
```
void ckeckInterrupt() {

    if(getGIE()) {

        if ((getT0IE() && getT0IF()) || (getINTE() && getINTF()) || (getRBIE() && getRBIF())) {

            pushStack(progZeiger);

            setProgZeiger(0x0004);
            setGIE(0);
        }
    }
}
```

Abbildung 18: Funktion für Interrupts

Diese Funktion wird nach der Abarbeitung eines Befehles in der execBefehl()-Funktion aufgerufen. Hier wird zunächst anhand des GIE-Flags geprüft ob Interrupts erlaubt sind. Danach wird nacheinander anhand der jeweiligen Flags geprüft, ob ein Interrupt-Bedingung aufgetreten ist und das zugehörige enable-bit auf 1 steht. Falls ja, dann löst der Interrupt aus und der Program-Zeiger wird auf die Adresse 0x0004 gerichtet (Interrupt-Service-Routine) und das GIE-Flag auf 0 gesetzt.

2.2.4 Implementation des Watchdog

Die folgende Abbildung 19 zeigt, wie der Watchdog implementiert wurde. Es zeigt wie dieser abgeprüft und unter Beachtung des Vorteilers erhöht wird oder ein Reset ausgelöst wird.




```
void checkWatchdog() {

    if (wdtActive) {
        if (!(wdt < 18000)) {
            // watchdog abgelaufen
            if (getPSA() == 1) {
            //vorteiler bei watchdog
                if (pre > 0) {
                    // Preskaler nicht abgelaufen
                    //erhöhe vorteiler(als  rückwärtszähler)
                    pre--;
                    // setze watchdog zurück
                    wdt = 0;
                }
                else {
                    // Preskaler und wdt abgelaufen, watchdog reset
                    wdtResetPIC();
                }
            } else {
            // vorteiler nicht an watchdog, watchdog abgelaufen -> reset
            wdtResetPIC();
            }
        } else {
            wdt += 4/quarzTakt;
        }
    }
}
```

Abbildung 19: checkWatchdog()-Funktion

2.2.5 Implementation des Timers

Zum Abarbeiten des Timers wurden zwei Funktionen implementiert. Folgende Abbildung 20 zeigt diese zwei Funktionen.

```
void incTimer() {

    if (getPSA() == 0) {
        // vorteiller an timer, erhöhe vorteiler(als    rückwärtszähler)
        if (pre > 1) {
            pre--;
        }
        else {
            // erhöhe Timer wenn pre == 0
            dataSpeicher[0][1]++;
            if (dataSpeicher[0][1] == 0) {
                setT0IF(1);
            }
            // setze prescaler zurück
            setPreVar(getPS());
        }
    } else {
        // vorteiller NICHT an timer
        dataSpeicher[0][1]++;
        if (dataSpeicher[0][1] == 0) {
            setT0IF(1);
        }
    }
}
```

Abbildung 20: Timer-Funktionen

Die setTimer() Funktion prüft, ob der Timer mit dem Programmtakt erhöht, werden Falls ja wird die Funktion incTimer() aufgerufen.

Diese Funktion prüft, ob sich der Vorteiler am Timer befindet, verringert entsprechend die interne Vorteiler-Variable (pre) bzw. erhöht dem Timer, falls pre = 0 oder der Vorteiler gar nicht am Timer ist. Kommt es zu einem Überlauf, d.h. das Timer-Register wird 0, dann wird das Interrupt-Flag T0IF gesetzt.

2.2.6 Implementation des TRIS-Registers

Die Funktionen zur Implementation der IO-Pins, der Feststellung der Flanken und Steuerung über das TRIS-Register sind sehr komplex und verschachtelt. Darum folgt in Abbildung 21 nur der Ausschnitt, welcher die Steuerung über das TRIS-Register betrifft. Die vollständigen Funktionen finden sich in der mainwindow.cpp ab Zeile 362 bis Zeile 530.

```
void MainWindow::gui_set_IO()
{
    int reg = 0;
    for (int i = 1; i < 5; i+=3) {
       for(int j = 0; j < 8; j++) {
           if(i==1) {
              reg = 0x05;
           } else {
              reg = 0x06;
           }
           if(( dataSpeicher[1][reg] & (1u << (7-j)) ) == 0) {
              ui->pin_table->item(i,j)->setText("o");
              for(int k = 0; k < 8; k++) {
                 if(( dataSpeicher[0][reg] & (1u << (7-j)) ) > 0) {
                    ui->pin_table->item(i+1,j)->setData(Qt::CheckStateRole, Qt::Checked);
                 } else {
                    ui->pin_table->item(i+1,j)->setData(Qt::CheckStateRole, Qt::Unchecked);
                 }
              }
           }
           else {
              ui->pin_table->item(i,j)->setText("i");
           }
       }
    }

}
```

Abbildung 21: Funktion TRIS-Register

Die Funktion geht die Pins eines nach dem anderen durch und vergleicht mit dem TRIS-Register. Dabei werden dann die Beschriftungen der PINS auf O für Output oder I für Input gesetzt und im Fall des Outputs wird der Pin abhängig vom zugehörigen Port-Register auf 1 oder 0 gesetzt.

2.3 Der Grundsätzliche Programmablauf mittels State-Maschine

Das folgende Diagramm in Abbildung 22 zeigt den Programmablauf, sobald der Go-Button gedrückt wird. Dabei wird deutlich, welche Befehle der Simulator im normalen Betrieb in welcher Reihenfolge abarbeitet

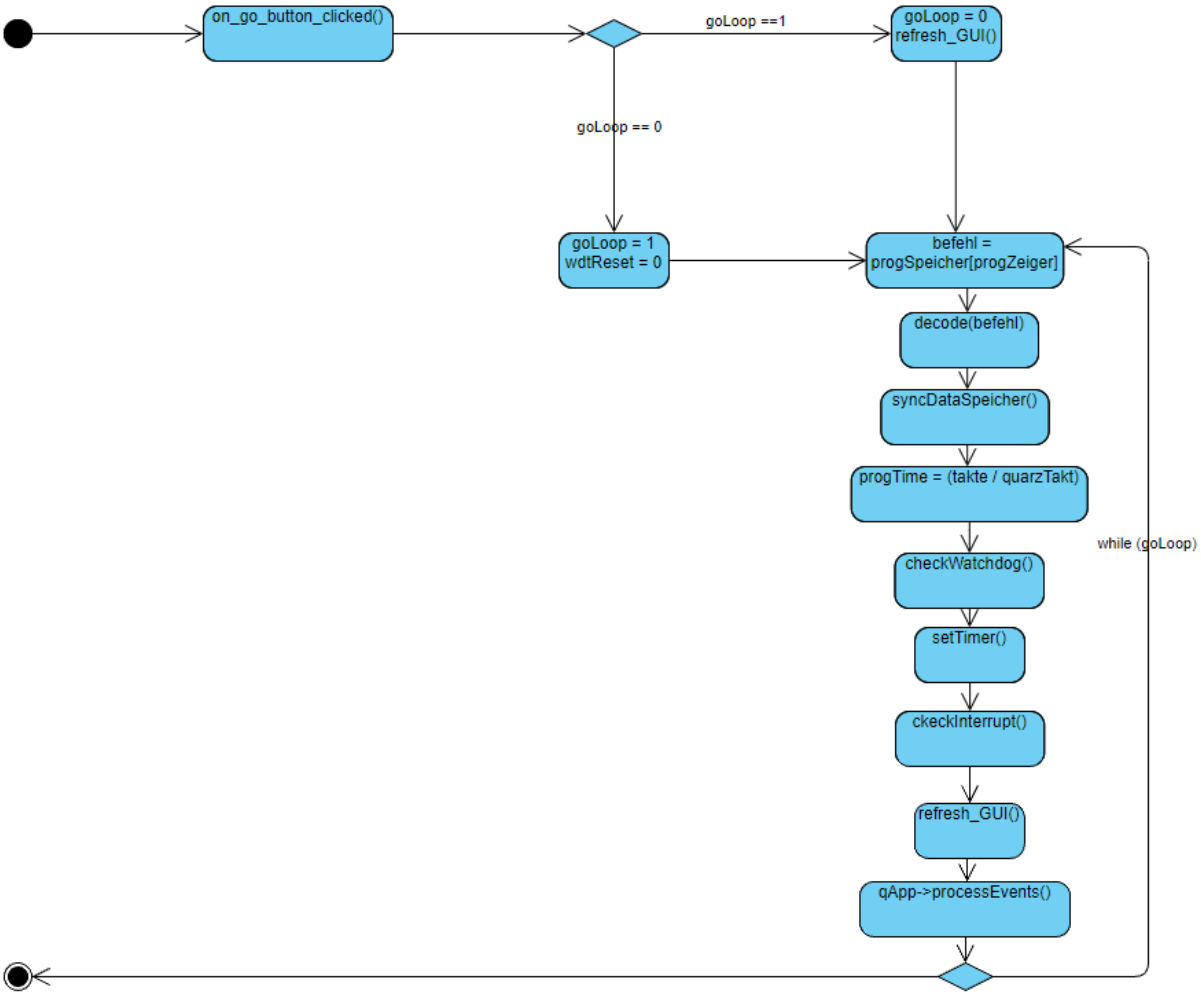
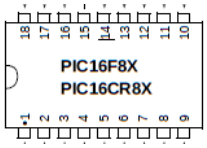


Abbildung 22: Ablaufdiagram GoButton



6. Zusammenfassung und Fazit

6.1 Abdeckung des Bausteins als Simulation

Ein Blick auf das Bewertungsschema in Abbildung 23 zeigt, dass mit Ausnahme von zwei Funktionen alle geforderten Inhaltlichen Funktionen des PIC implementiert wurden. Einige Details im Bereich des Watchdog und Timer wurden dabei vereinfacht, da diese in der Simulation schwer umzusetzen sind. Dies war in den Themenblättern auch so bestätigt. Erweiterungsmöglichkeiten durch mehr Speicherbänke usw. sind auf Grund einiger spezifischer Implementationen ebenfalls nicht ohne weiteres möglich.

Stand: 28.02.2023

Programm	Inhalt
1 (MUSS)	Die einfachen Literalbefehle, u.a. MOVLW, ADDLW, SUBLW
2 (MUSS)	u.a. CALL, GOTO (vereinfacht, ohne Rücksicht auf PCLATH)
3 (MUSS)	u.a. MOVWF, MOVF, SUBWF (nur direkte Adressierung, aber mit d-Bit Auswert.)
4 (MUSS)	u.a. DCFSZ, INCFSZ, RLF, RRF (nur direkte Adressierung, aber mit d-Bit Auswert.)
5 (MUSS)	u.a. BSF, BCF, BTFSC, BTFSS (direkt und indirekt Adressierung)
6 (MUSS)	u.a. Bytebefehle, (direkte und indirekte Adressierung)
7 (MUSS)	Timerfunktion ohne / mit Berücksichtigung der Bits im OPTION-Register (e / o)[2]
8 8 (MUSS) 8	Interrupt für Timer 0 Interrupt für INT (RB0) mind. 1 Interrupt muss realisiert werden Interrupt für RB4-RB7
9	u.a. SLEEP
10 (MUSS)	ADDWF PCL mit Berücksichtigung von PCLATH (theor. Für CALL und GOTO; geprüft an Hand des Codes)
11	Watchdog ohne Vorteiler / u.a. Watchdog mit Vorteiler (e / o)[2]
12	EEPROM ohne Programmierzeit / EEPROM inc. 1ms Programmierzeit (e / o)[2]
13	Lauflicht
14	Leuchtband
15	I/O Ausgangslatch (Wirkung internes TRIS-Register auf Ausgangstreiber)
	Prüfprogramm mit automatischer Punktberechnung

Abbildung 23: Ausschnitt Bewertungsschema

Insgesamt lässt sich Zusammenfassen, dass die meisten wichtigen Funktionen des PIC im Rahmen des machbaren einer Simulation implementiert wurden.

6.2 Fazit

Wir haben uns vor dem Projekt nie mit der Entwicklung einer GUI und einem Programm in dieser Größe beschäftigt. Deshalb setzte zunächst eine völlige Überforderung mit der Thematik ein, vor allem da der PIC-Mikroprozessor für uns völlig neu war. Dies erforderte einen schrittweisen, systematischen Ansatz. Deshalb haben wir uns entschieden die Logik zunächst getrennt zu entwickeln. Dies erwies sich als sehr sinnvoll, da wir in einer Konsolen-Anwendung mehr Erfahrung hatten und das Testen einfach zu handhaben war. Das Nutzen einer Versionsverwaltung über GitHub und das Erstellen von ToDo-Listen auf Basis der Themenblätter war ebenfalls sehr hilfreich, da so die Übersicht behalten werden konnte. So konnte sich das Programm schrittweise aufbauen und ermöglichte ein Aufteilen des großen Projektes in kleine Abschnitte. Nach dem Fertigstellen der Logik war ein vernünftiges Testen erst mit einer GUI möglich. Dies war dann der nächste Schritt

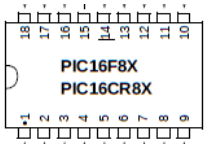
Dabei entstanden Probleme bei der Auswahl, Installation und Bedienung von verschiedenen Frameworks. Es stellt sich die Frage, ob das ausgewählte Framework mit unserem bisherigen Code funktionieren würde. Dies engt die Auswahl stark ein und die finale Entscheidung fiel dabei auf den QT-Creator, da dieser CPP nativ ist und wir somit die GUI ohne große Probleme mit unserem Core Code bespielen können. Ein weiterer großer Pluspunkt für den QT-Creator ist die einfache Implementierung eines Gerüsts für die GUI mit einem visuellen Editor, welcher automatisch den CPP-Code dafür generiert und nur noch minimal manuell angepasst werden muss.

Ein Problem in unserer benutzten GUI sind die eigenen Datentypen (z.B. QString), welche jeweils eine eigene Funktion für die Umwandlung von einem Standardtyp in den speziellen Typ benötigt. Abbildung 24 zeigt eine dieser Hilfsfunktionen.

```
QString MainWindow::string_to_QString(string input){
    QString q_input = QString::fromStdString(input);
    return q_input;
}
```

Abbildung 24: Hilfsfunktion zur Datentypkonvertierung

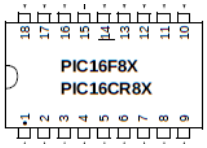
Sowohl während der Entwicklung der Logik als auch der GUI traten immer wieder Fehler auf, welche schrittweise behoben werden mussten. Meistens handelte es sich



dabei um kleinere Fehler im Code oder Missverständnisse mit den Datenblättern. Diese konnten im Einzelnen schnell gelöst werden, häuften sich aber jedoch in Summe zur einem ordentlichen Workload auf.

Im Falle, dass das Projekt erneut durchgeführt werden würde, empfiehlt es sich bei der Planung noch vorsichtiger und genauer vorzugehen. Dies vereinfacht sich enorm, wenn der PIC bereits vorher bekannt ist, wodurch schneller ein besserer Überblick möglich ist.

Zusammenfassend lässt sich feststellen, dass die zu Beginn unmöglich wirkende Aufgabe am Ende doch erfolgreich und sehr viel besser als erwartet abgearbeitet werden konnte. Dabei war die Lernkurve sehr steil und das vorhandene Wissen in C++ wurde das erste Mal richtig auf die Probe gestellt und konnte um ein Vielfaches erweitert werden. Außerdem erscheint das Erstellen einer Anwendung mit GUI nun nicht mehr abschreckende, sondern als machbare Herausforderung. Wir sind mit unserem Simulator sehr zufrieden.



Anhang

Der Source-Code (Programmlisting) befindet sich im Ordner **Code/PIC_SIM_QT**

- ressources
- befehle.cpp
- befehle.h
- common.h
- einlesen.cpp
- einlesen.h
- Header.h
- main.cpp
- mainwindow.cpp
- mainwindow.h
- mainwindow.ui
- PIC_SIM_QT_de_DE.ts
- PicSim.qrc
- pins.cpp
- pins.h
- regfunc.cpp
- regfunc.h
- test.cpp
- test.h