

Dokumentation zum Simulator des PIC16F84 Mikroprozessors

im Rahmen des Kurses TINF14B3 „Systemnahe Programmierung“

im Studiengang Informatik in der Studienrichtung Informationstechnik

an der

Dualen Hochschule Baden-Württemberg Karlsruhe

von

Nicolas Huentz,

Marie-Kristin Kaiser,

und Daniel Stumpf

Abgabedatum

16.05.2016

Zusammenfassung

Simulatoren helfen nicht nur um Systeme in virtuellen Umgebungen zu testen, sondern können bei Entwicklern solcher Simulatoren auch das Verständnis für das abgebildete System stärken. Aus diesem Grund ist die Anforderung in der Vorlesung *Rechnertechnik II* des Informationstechnik-Studienganges an der DHBW Karlsruhe, die Erstellung einer Simulationsumgebung für den Mikrokontroller *PIC16F84*. Dieses Dokument wird hierbei als Protokollierung des Realisierungsprozesses sowie als allgemeine Hilfestellung für die Anwendung verfasst.

Inhaltsverzeichnis

1	Vorwort.....	1
2	Grundlagen.....	2
2.1	Simulation.....	2
2.1.1	Vorteile.....	2
2.1.2	Nachteile.....	2
2.2	Microcontroller.....	3
2.2.1	PIC16F84.....	3
3	Simulationssoftware.....	4
3.1	Programmoberfläche.....	4
4	Umsetzung.....	7
4.1	Entwicklungsumgebung.....	7
4.2	SWT (Standard Widget Toolkit).....	7
4.3	Modularität.....	8
4.4	Versionsverwaltung.....	9
4.5	Klassendiagramm.....	9
4.6	Befehlsdekodierung.....	11
4.7	Befehlsausführung.....	12
4.8	Funktionsbeschreibungen.....	12
4.8.1	Byteorientierte Operationen.....	13
4.8.2	Bitorientierte Operationen.....	17
4.8.3	Literal- und Kontrolloperationen.....	19
5	Fazit.....	22
5.1	Probleme.....	22

Abbildungsverzeichnis

Abbildung 1: Benutzeroberfläche des PIC16F84 Simulators.....	4
Abbildung 2: Darstellung der Ports A und B.....	4
Abbildung 3: Darstellung der Code-Liste.....	5
Abbildung 4: Darstellung der Timing-Funktion des Simulators	5
Abbildung 5: Darstellung der Register	6
Abbildung 6: Darstellung des gesamten Speicherinhalts.....	6
Abbildung 7: Klassendiagramm	10
Abbildung 8: RRF-Befehl.....	13
Abbildung 9: Ablaufdiagramm RRF-Befehl.....	15
Abbildung 10: Ablaufdiagramm BTFSC-Befehl.....	18
Abbildung 11: Ablaufdiagramm ADDLW	20

1 Vorwort

Das vorliegende Dokument beschreibt die Umsetzung des Projektes der Erstellung einer Simulationssoftware des Microcontrollers des Typs PIC16F84 der Firma Microchip¹. Das Projekt im Rahmen der Vorlesung Systemnahe Programmierung 2 erfordert das Anwenden bereits erlernter Kenntnisse aus den vorigen Semestern wie beispielsweise binäre Schaltoperationen aus der Digitaltechnik (Bool'sche Algebra, binäre Rechenverfahren, Grundgatter, etc.). Des Weiteren sind Vorkenntnisse der Rechnerarchitekturen notwendig, um die Funktionsweise eines Microcontrollers zu verstehen und im Simulator umsetzen zu können.

Die Durchführung der Implementierung erfordert vorab einige Planung. Das Projekt wurde im Team mit drei Personen umgesetzt, weshalb es notwendig ist, im Vorfeld die Aufgaben zu verteilen, damit jeder effizient seine Aufgaben durchführen kann.

Diese Dokumentation erklärt einige grundlegende Begriffe zum Verständnis eines Simulationsprogramms. Anschließend wird näher auf die implementierte Simulationssoftware eingegangen. Hierbei wird die Benutzeroberfläche erläutert, der Aufbau und die Struktur der Klassen Software erklärt sowie einige Assemblerbefehle und deren Umsetzung beschrieben.

¹ <http://www.microchip.com/>

2 Grundlagen

Im folgenden Abschnitt wird auf die Grundlagen für das Verständnis einer Simulationssoftware eingegangen sowie einige Begriffe zum simulierten Microcontroller erklärt.

2.1 Simulation

Eine Simulation ist nach dem Verein Deutscher Ingenieure eine „Nachbildung eines Systems mit seinen dynamischen Prozessen in einem experimentierfähigen Modell, um zu Erkenntnissen zu gelangen, die auf die Wirklichkeit übertragbar sind“ [VDI 3633, (2013)]. Demzufolge wird durch einen Simulator eine Umgebung generiert, deren Eigenschaften denen des Einsatzgebietes entsprechen, das der Simulator simulieren soll.

2.1.1 Vorteile

Simulatoren werden vorteilsweise eingesetzt, wenn die Testungen in realen Systeme zu langsam, zu schnell, zu teuer, zu gefährlich oder auch unmöglich wären. In diesen Fällen oder auch bei komplexeren Systemen, gelangen die Möglichkeiten der Realsysteme schnell an ihre Grenzen. Simulationen können, sofern sie eine geeignete Modellierung zugrunde liegen haben, viele dieser Probleme umgehen. Sie können beliebig beschleunigt oder verlangsamt, sowie einfach reproduziert, analysiert, interpretiert und protokolliert werden. Im Weiteren können, durch das Anpassen der Simulationsvariablen und ausführliche Analysen der Simulation, Systeme optimiert werden bevor sie die reale Welt erreichen. Auf diese Weise werden auch Kostenrisiken gesenkt und damit erhebliche Ersparnisse erzielt.

2.1.2 Nachteile

Um valide Kenntnisse über das abzubildende Realsystem zu erlangen, ist eine genaue Abbildung auf ein Modell nötig. Dies zu erreichen ist jedoch nicht immer machbar. Fehlende oder gar verfälschte Daten können sogar zu vollständig verschiedenen Ergebnissen führen. Eine genaue Datenhaltung ist daher essentiell wichtig. Dazu kommt, dass Simulationsstudien teuer sind. Eventuell kann es also passieren, dass die laut Simulation mögliche Kostenersparnis durch die Simulationskosten bereits wieder aufgewogen wird und selbst dann ist durch die Simulation keine Garantie für die Ergebnisse gegeben. Simulationen liefern also im besten Fall das Optimum für die getroffenen Annahmen in der Abbildung und der getroffenen Zahl an

Alternativversuchen. Auch hier können Möglichkeiten übersehen werden. Da Simulationen mittlerweile immer detailgetreuer werden, steigen auch die nötigen Datensätze und damit der Speicherverbrauch für die Simulationen.

2.2 Microcontroller

Mikrocontroller sind Ein-Chip-Computersysteme, die neben einem Prozessor auch Peripheriefunktionen (BUS-Systemen, Timer,...) besitzen. Meist befinden sich außerdem Arbeits- und Programmspeicher auf demselben Chip. Praktisch sind sie in allen elektronischen Geräten vorhanden, wie zum Beispiel in Uhren, Waschmaschinen, Mobiltelefonen, aber auch alle Computer-Peripheriegeräten wie Maus, Tastatur oder Drucker.

2.2.1 PIC16F84

Der PIC16F84A (PIC = Programmable Intelligent Computer) ist ein 8-Bit RISC Mikrocontroller aus dem Hause *Mikrochip*. Folglich werden auf komplexe Programmierbefehle verzichtet.



Dieser Controller beruht auf der Harvard-Architektur. Dadurch können fast alle Anweisungen durch einen Instruction Cycle verarbeitet werden. Im Stack können bis zu 8 Adressen gespeichert sowie durch je zwei interne und externe Quellen verursachte Interrupts verarbeitet werden. Das Register ist in 2 Bänken unterteilt, welche durch Setzen entsprechender Bits verwendet angesprochen werden können.

3 Simulationssoftware

Im Folgenden wird näher auf die implementierte Simulationssoftware eingegangen. Dabei wird vorab die Benutzeroberfläche sowie deren Bedienung beschrieben. Anschließend erfolgt eine Beschreibung des Grundkonzeptes der Anwendung und eine Erläuterung des Aufbaus der Software mit Erklärungen zur Umsetzung verschiedener Funktionen des PIC16F84.

3.1 Programmoberfläche

Die Benutzeroberfläche des Simulators wurde im Team auf eine einfache, leicht zu verstehende Anordnung und Bedienungsmöglichkeit Wert gelegt. Damit wurde die Oberfläche in vier Abschnitte unterteilt, welche nachfolgend beschrieben werden.

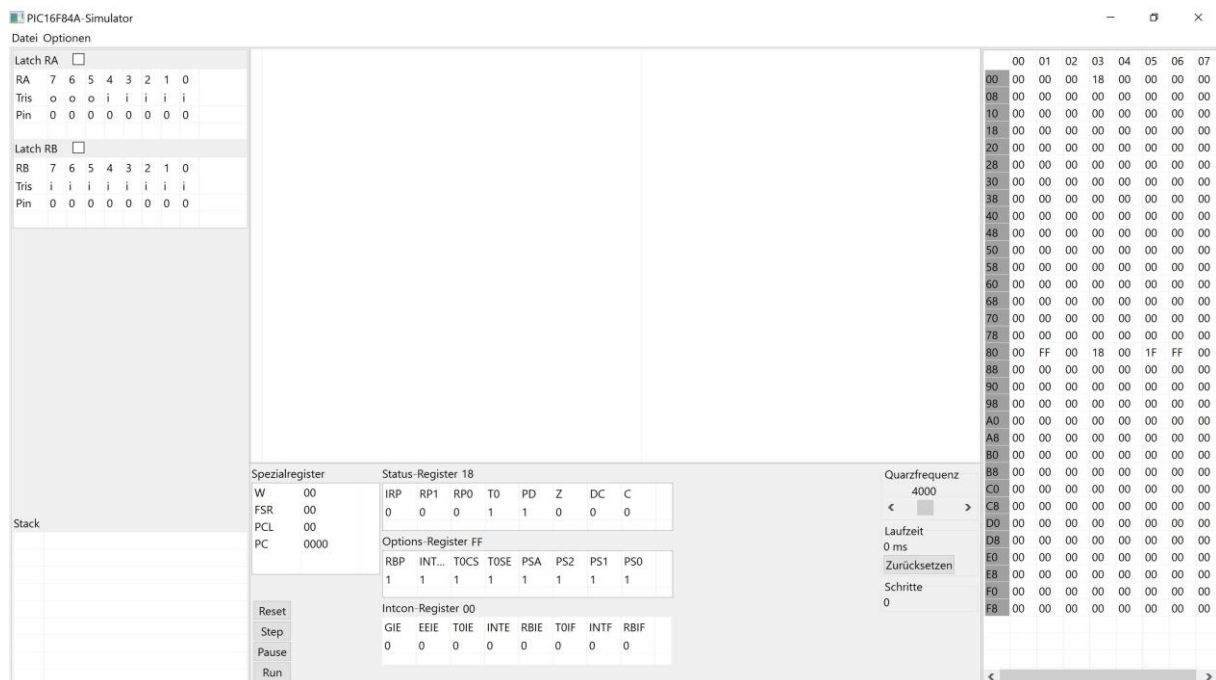


Abbildung 1: Benutzeroberfläche des PIC16F84 Simulators

Auf der linken Seite des Simulation-Programms erscheinen die beiden Ports A und B. Über eine Checkbox ist das Einschalten der Latchfunktion möglich, per Doppelklick auf die einzelnen Bits lassen sich die Werte manuell verändern.

Latch RA		<input type="checkbox"/>						
RA	7	6	5	4	3	2	1	0
Tris	o	o	o	i	i	i	i	i
Pin	0	0	0	0	0	0	0	0
Latch RB		<input type="checkbox"/>						
RB	7	6	5	4	3	2	1	0
Tris	i	i	i	i	i	i	i	i
Pin	0	0	0	0	0	0	0	0

Abbildung 2: Darstellung der Ports A und B


Darunter befindet sich die Visualisierung des Stacks, der beispielsweise die gespeicherten Rücksprungadressen darstellt. Im Stack kann man im Gegensatz zu den Ports keine Werte manuell verändern.

Im zentralen oberen Bereich der Anwendung befindet sich die Codeliste mit dem Dateiinhalt der geöffneten Datei. Hier wird die gesamte Zeile dargestellt und die Position, an der sich das Programm aktuell befindet, wird markiert dargestellt. Über die Checkboxes am linken Rand jeder Zeile kann ein Breakpoint gesetzt werden, um das Programm an beliebiger Stelle unterbrechen zu können.

<input type="checkbox"/>	0000 3011	00024	movlw 11h	;in W steht nun 11h.
	Statusreg. unverändert			
<input type="checkbox"/>	0001 2006	00025	call up1	;beim Call wird
	Rücksprungadresse auf Stack gelegt			
<input type="checkbox"/>	0002 0000	00026	nop	;W = 36h, C=0, DC=0, Z=0
<input type="checkbox"/>	0003 2008	00027	call up2	;in W steht der
	Rückgabewert			
<input type="checkbox"/>	0004 0000	00028	nop	;W = 77h, DC=0, C=0, Z=0;
<input type="checkbox"/>	0005 2800	00029	goto loop	
<input type="checkbox"/>	0006 3E25	00032 up1	addlw 25h	;W = 36h, DC=0, C=0, Z=0
<input type="checkbox"/>	0007 0008	00033	return	
<input type="checkbox"/>	0008 3477	00036 up2	retlw 77h	
<input type="checkbox"/>	0009 2809	00039	goto ende	;Endlosschleife.
	verhindert Nirwana			

Abbildung 3: Darstellung der Code-Liste

Im unteren Bereich befindet sich die Visualisierung der verschiedenen Register, sowie der Einstellung für die Quarzfrequenz. Diese kann über einen Slider variabel eingestellt werden. Zusätzlich wird die aktuelle Laufzeit des Programms sowie die durchgeführten Schritte visualisiert.

Quarzfrequenz
 4000
 <  >

Laufzeit
 0 ms
 Zurücksetzen

Schritte
 0

Abbildung 4: Darstellung der Timing-Funktion des Simulators

Die Bits der Register sind direkt veränderbar. Durch Doppelklick werden Programmintern die Werte abgefragt und entsprechend das Bit auf eins oder auf null gesetzt. Neben der Beschriftung des Registers befindet sich die Anzeige des jeweiligen Registers als Hexadezimalwert. Damit lässt sich auf den ersten Blick der Gesamtwert des Registers erkennen.

Status-Register 18							
IRP	RP1	RP0	T0	PD	Z	DC	C
0	0	0	1	1	0	0	0

Options-Register FF							
RBP	INT...	T0CS	T0SE	PSA	PS2	PS1	PS0
1	1	1	1	1	1	1	1

Intcon-Register 00							
GIE	EEIE	TOIE	INTE	RBIE	TOIF	INTF	RBIF
0	0	0	0	0	0	0	0

Abbildung 5: Darstellung der Register

Auf der rechten Seite im Simulator befindet sich die Abbildung des gesamten Speichers. Alle Werte werden als Hexadezimalwert angezeigt und können auch hier per Doppelklick selektiert und geändert werden. Damit ist es möglich, Speicherinhalte während der Simulation eines Programms zu verändern. Die Werte aus dem Speicher werden stets synchron mit den anderen Darstellungen in der Anwendung gehalten. Damit werden in der gesamten Anwendung stets die aktuellen Werte angezeigt.

	00	01	02	03	04	05	06	07
00	00	00	00	18	00	00	00	00
08	00	00	00	00	00	00	00	00
10	00	00	00	00	00	00	00	00
18	00	00	00	00	00	00	00	00
20	00	00	00	00	00	00	00	00
28	00	00	00	00	00	00	00	00
30	00	00	00	00	00	00	00	00

Abbildung 6: Darstellung des gesamten Speicherinhalts

4 Umsetzung

In diesem Abschnitt wird der Aufbau der Anwendungsstruktur erklärt. Dazu wird vorab die grundlegende Idee erläutert und anschließend anhand einiger Codeauszüge genauer auf die Vorgehensweise bei der Implementierung der Software eingegangen.

4.1 Entwicklungsumgebung

Um die angestrebten Ziele in dieser Arbeit umsetzen zu können, musste eine Oberfläche geschaffen werden, mit der es den Anwendern ermöglicht wird, einfach und intuitiv die gewünschte Simulation durchführen zu können. Dabei war es wichtig, die Anwendung auf die nötigen Funktionen zu beschränken. Die Entwicklungsumgebung des Simulators ist Eclipse Mars.2, als Programmiersprache wurde Java verwendet. Eclipse ist eine quelloffene Entwicklungsumgebung und wurde ursprünglich für die Programmiersprache Java entwickelt. Mittlerweile bietet Eclipse allerdings die Möglichkeit, eine Vielzahl anderer Programmieraufgaben umzusetzen. Dies wird durch ein modellbasiertes System, bei dem Komponenten modularisiert und gekapselt werden können, umgesetzt. Die Modularisierung der Komponenten wird mit OSGi umgesetzt, ein Framework, das es ermöglicht, Anwendungen und deren Dienste zu modularisieren und zu verwalten.

In dieser Arbeit wurde Eclipse, welches selbst auf Java basiert, als Entwicklungsumgebung für Java-Applikationen verwendet.

4.2 SWT (Standard Widget Toolkit)

Die von der Eclipse Foundation² bereitgestellte Programmbibliothek ermöglicht es, grafische Anwendungen mit der Programmiersprache Java zu entwickeln. SWT bedient sich dabei nativen grafischen Elementen des Betriebssystems, wodurch dem Benutzer gewohnte grafische Elemente wie beispielsweise beim Öffnen einer Datei bereitgestellt werden. Dem Entwickler wird es dabei einfach ermöglicht, diese Elemente in die Anwendung einzubinden und damit ein einheitliches Erscheinungsbild von Anwendungen zu entwickeln.

² <http://www.eclipse.org/>

4.3 Modularität

Um eine gewisse Modularität des Simulators erreichen zu können, wurde dieser nach dem Model-View-Controller (MVC) Prinzip implementiert. Dadurch wurde es ermöglicht, die grafische Oberfläche (View) unabhängig von der Programmlogik (Model) zu programmieren. Dies hatte den wesentlichen Vorteil, dass im Team jeder seine Aufgabe nahezu unabhängig von den Partnern umsetzen konnte. Somit wurden das Model und die View, nachdem diese implementiert waren, miteinander durch den Controller verknüpft. Somit greift die Controller-Klasse auf View und Model zu, dagegen haben diese beiden Klassen keine Abhängigkeiten untereinander.

Die GUI (Graphical User Interface) bildet die View und bietet dem Benutzer die Ansicht des Simulators. Diese arbeitet unabhängig von der Programmlogik und dient nur zur Darstellung der Inhalte sowie zur Programmsteuerung.

Das Model beinhaltet die Programmlogik. Hier werden die zu simulierenden Befehle umgesetzt und die entsprechenden Programmvariablen mit Werten befüllt und angepasst.

Der Controller verbindet die GUI mit dem Model. Benutzereingaben wie beispielsweise das Öffnen eines Dokumentes oder das Setzen von Breakpoints werden von der GUI über den Controller an das Model weitergeleitet, um die entsprechenden Schritte beim Programmdurchlauf durchzuführen. Des Weiteren ist es möglich, Speicherinhalte, die auf der GUI dargestellt werden, manuell zu ändern. Auch hier reagiert der Controller und ändert die Werte entsprechend auch im Model.

Durch die Umsetzung nach dem MVC-Prinzip und der damit verbundenen Modularität ist es möglich, spätere Änderungen des Simulators einfach durchführen zu können. Beispielsweise kann man den Simulator um Funktionen erweitern, ohne Änderungen an der View vornehmen zu müssen. Wichtig war bei der Implementierung die strikte Beachtung der Trennung zwischen Model und View, damit die gewünschte Modularität erhalten bleibt.

4.4 Versionsverwaltung

Eine Online-Versionsverwaltung bietet zum einen den Vorteil, stets eine Datensicherung zu besitzen, sofern die Daten immer aktuell gehalten werden, zum anderen können mehrere Personen an einer Version unabhängig voneinander arbeiten. Zur Erstellung des Simulators wurde im Rahmen dieses Projektes GitHub³ verwendet. Dabei war es möglich, ein Repository zu erstellen und alle Teammitglieder als Entwickler zu integrieren. Durch die integrierte Versionskontrolle konnten testweise erstellte Programmfunktionen später zusammengeführt oder wieder verworfen werden. Zudem kamen zu keinem Zeitpunkt Diskussionen auf, welches die im Moment aktuelle Version ist oder wo man eine Datensicherung ablegen soll.

4.5 Klassendiagramm

Um das angestrebte MVC-Entwurfsmuster umzusetzen, wurde die Anwendung ausgehend von drei Klassen aufgebaut. Diese drei Klassen sind die Model-, die View- und die Controller-Klasse. Die Klasse View bekam mit der Klasse CreateItem eine Hilfsklasse, die dieser Klasse bestimmte Elemente bereitstellt und somit eine Übersichtlichkeit geschaffen wurde.

Der Controller wurde um die Klasse PicSimListener erweitert, welche sämtliche Listener an die View übergibt, um alle Änderungen, die an der View durch den Benutzer vorgenommen werden, registrieren zu können und somit programmintern zu verarbeiten.

Dazu gibt es zwei Thread-Klassen, die jeweils instanziiert werden, wenn ein Programm gestartet wird. Dies hat den Vorteil, dass ein gestartetes Programm unabhängig von der Anwendung läuft und jederzeit pausiert oder gestoppt werden kann. Hier gibt es jeweils eine Klasse für einen einzelnen Befehl sowie für den Start aller Befehle.

Im Model werden sämtliche Daten verarbeitet und über den Controller an die View weitergegeben, um diese dort dem Anwender bereitstellen zu können.

Die Abhängigkeiten werden im nachstehenden Klassendiagramm visuell dargestellt.

³ <https://github.com/>

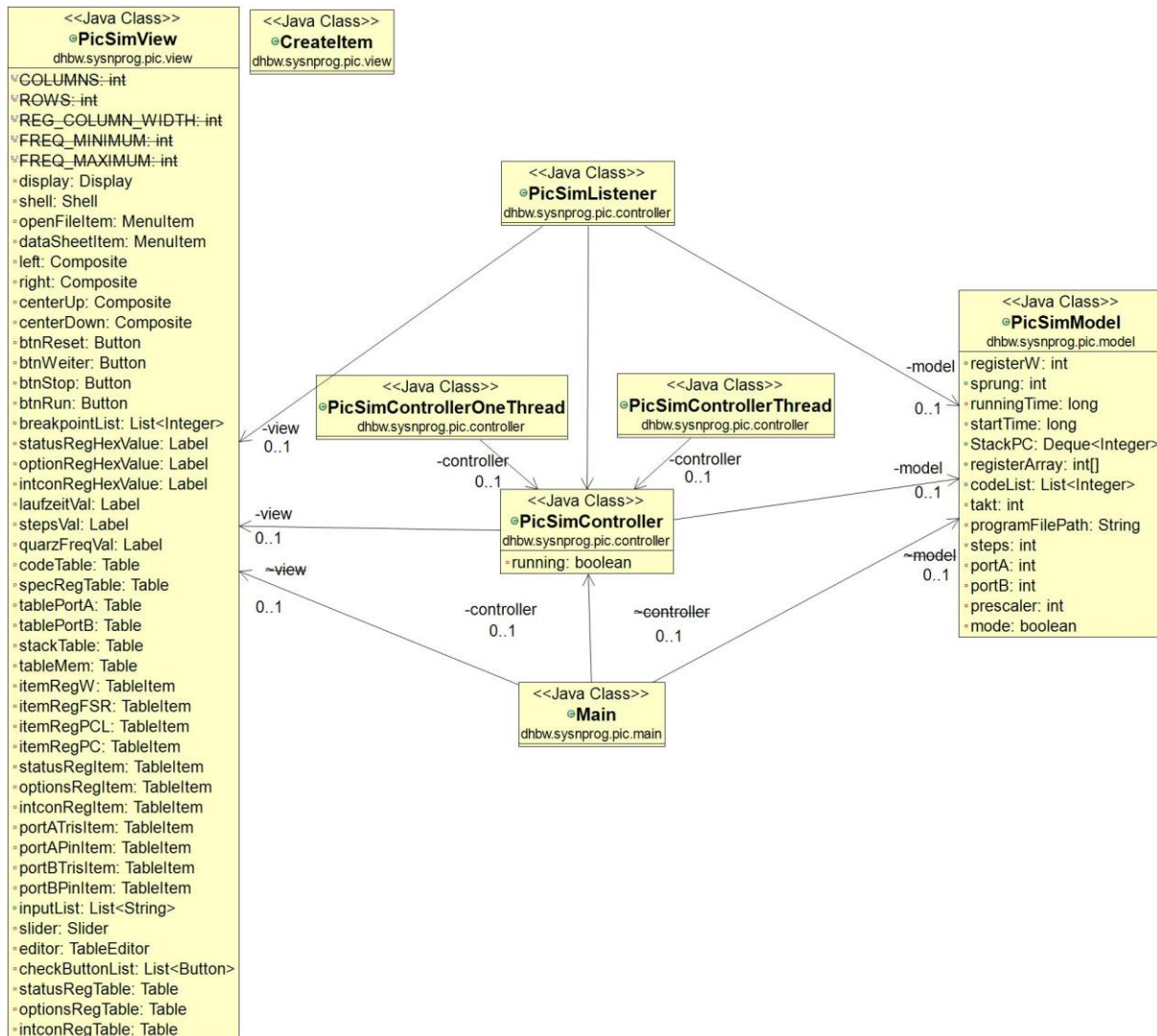


Abbildung 7: Klassendiagramm

Im Model werden sämtliche Daten verarbeitet und über den Controller an die View weitergegeben, um diese dort dem Anwender bereitstellen zu können. Hier werden unter anderem die verschiedenen Befehle dekodiert und die entsprechende Funktion zur Ausführung des Befehls aufgerufen wie im nächsten Abschnitt erläutert wird.

4.6 Befehlsdekodierung

```
public void doAction(int code) throws InterruptedException {
    final int codeTmp = code;

    final int hex16 = codeTmp & 0xFFFF;
    switch (hex16) {
        case 100:
            do_clrwdt();
            break;
        case 9:
            do_retfie();
            break;
        case 8:
            do_return();
            break;
        case 99:
            do_sleep();
            break;
    }
}
```

Quellcode 1: Befehlsdekodierung

Bei der Befehlsdekodierung wird der Methode „doAction“ ein Integerwert übergeben, welcher den gesamten 14-bit Operationscode darstellt. Dieser muss in den eigentlichen Befehlscode sowie in den restlichen Bestandteil des Codes aufgeteilt werden. Dies geschieht anhand einer stufenweise logischen Verundung, damit eine eindeutige Zuordnung des Befehls erreicht werden kann. Quellcode 1 zeigt hierbei die erste Stufe der logischen Operation. Der Operationscode wird mit dem Hexadezimalwert 0xFFFF verundet. Sollte hier kein Befehl zutreffen, wird die nächste Stufe aufgerufen. Hier wird dann mit 0xFC00 verundet, die restlichen Bits stellen dann Literale oder Registeradressen dar. Diese stufenweise Abarbeitung der Dekodierung erreicht letztendlich eine eindeutige Zuordnung des Operationscodes.

Erfolgt eine Übereinstimmung im Switch-Case-Statement, wird hierin die entsprechende Funktion aufgerufen und der Befehl kann gestartet werden.

Durch die Trennung zwischen der Dekodierung und der Befehlsausführung wird im Quelltext eine gewisse Übersichtlichkeit beibehalten und etwaige Änderungen oder Befehlserweiterungen können einfach implementiert werden.

4.7 Befehlsausführung

Anhand des Befehls BTFSC wird die Durchführung eines Befehls beschrieben.

```
/**
 *
 * If bit 'b' in register 'f' is '1' the the next instruction is
executed.
 * If bit 'b' in register 'f' is '0' then the next instruction is
discarded
 * and a NOP is executed instead, making this a 2Tcy instruction
 *
 * @param b_f
 */
private void do_btfsf(int b_f) {
    final int adress = b_f & 0b0001111111;
    final int bit = (b_f & 0b1110000000) / 128;
    if (!checkBitSet(bit, adress)) {
        do_nop();
        setProgramCounter(getProgrammCounter() + 1);
    }
}
```

Quellcode 2: Befehlsausführung BTFSC

Die Befehlsfunktion bekommt hierbei einen Integerwert übergeben, welcher in diesem Fall eine Speicheradresse sowie ein Bit, welches geprüft werden soll, darstellt. Dieser Wert wird auch wieder über eine logische Operation in die beiden Einheiten „Adresse“ und „Bit“ aufgeteilt und anschließend entsprechende Aktionen des Befehls durchgeführt.

4.8 Funktionsbeschreibungen

Beim Microcontroller PIC16F84 wird zwischen drei Arten von Befehlen unterschieden. Diese werden wir bereits anhand der Befehlsdekodierung ersichtlich wurde, anhand des Operationscodes voneinander differenziert.

Die drei Arten der Befehle sind

- Byteorientierte Operationen
- Bitorientierte Operationen
- Literal- und Kontrolloperationen.

Durch die stufenweise logische Verundung bei der Befehlsdekodierung in der Anwendung können diese Operationen zweifelsfrei voneinander unterschieden werden.

Im Folgenden werden für jede Befehlsart eine Operation exemplarisch erläutert.

4.8.1 Byteorientierte Operationen

Nachstehend wird der Befehl RRF (Rotate Right f through Carry) erläutert. Bei diesem Befehl wird der Inhalt des f-Registers ein Bit, also eine Stelle nach rechts durch das Carry-Bit verschoben.



Abbildung 8: RRF-Befehl

In der Simulationsanwendung erhält die Funktion `do_rrf(int f_d)` einen Integerwert übergeben, welcher die Speicheradresse im f-Register sowie ein Destination-Bit enthält. Dieser Wert muss aufgeteilt werden, damit auf den Speicherinhalt zugegriffen und das Destination-Bit ausgewertet werden kann. Das Destination-Bit gibt an, an welche Stelle das Ergebnis später gespeichert werden soll.

Durch eine logische Verundung wird also die Speicheradresse des f-Registers ausgewertet und der Speicherinhalt in eine Variable gespeichert. Zudem wird das Carry-Bit ausgewertet, da dieses auf das spätere Ergebnis auch Auswirkungen hat. Ist dieses bereits gesetzt, wird zum Wert der Variable, in der der Inhalt des f-Registers zwischengespeichert wurde, der Wert 256 hinzuaddiert. Dies geschieht, da das Carry-Bit in dem Fall als neuntes Bit angesehen werden kann.

Anschließend wird das LSB⁴ ausgewertet. Ist dieses auf 1 gesetzt, wird das Carry-Bit gesetzt, da dieses Bit durch die Rechtsverschiebung der Bits an die Stelle des Carry-Bits verschoben wird. Ist dieses Bit null, wird das Carry-Bit gelöscht.

Nun erfolgt die Verschiebung der einzelnen Bits. Hierfür gibt es in Java den „>>“-Operator, welcher genau diese Operation auf eine Variable anwendet. Anschließend steht in der Variable der neue Wert, jetzt muss nur noch das Destination-Bit ausgewertet und der Wert der Variable an eine Speicherstelle geschrieben werden.

Ist das Destination-Bit gesetzt, also „1“, wird der Wert wieder an die Ursprungsstelle im f-Register geschrieben. Ist dieses Bit „0“, wird das Ergebnis in das W-Register geschrieben. Das Abspeichern des Wertes wird mittels einer separaten Funktion durchgeführt. Anschließend ist die Durchführung des Befehls abgeschlossen.

⁴ Least significant Bit

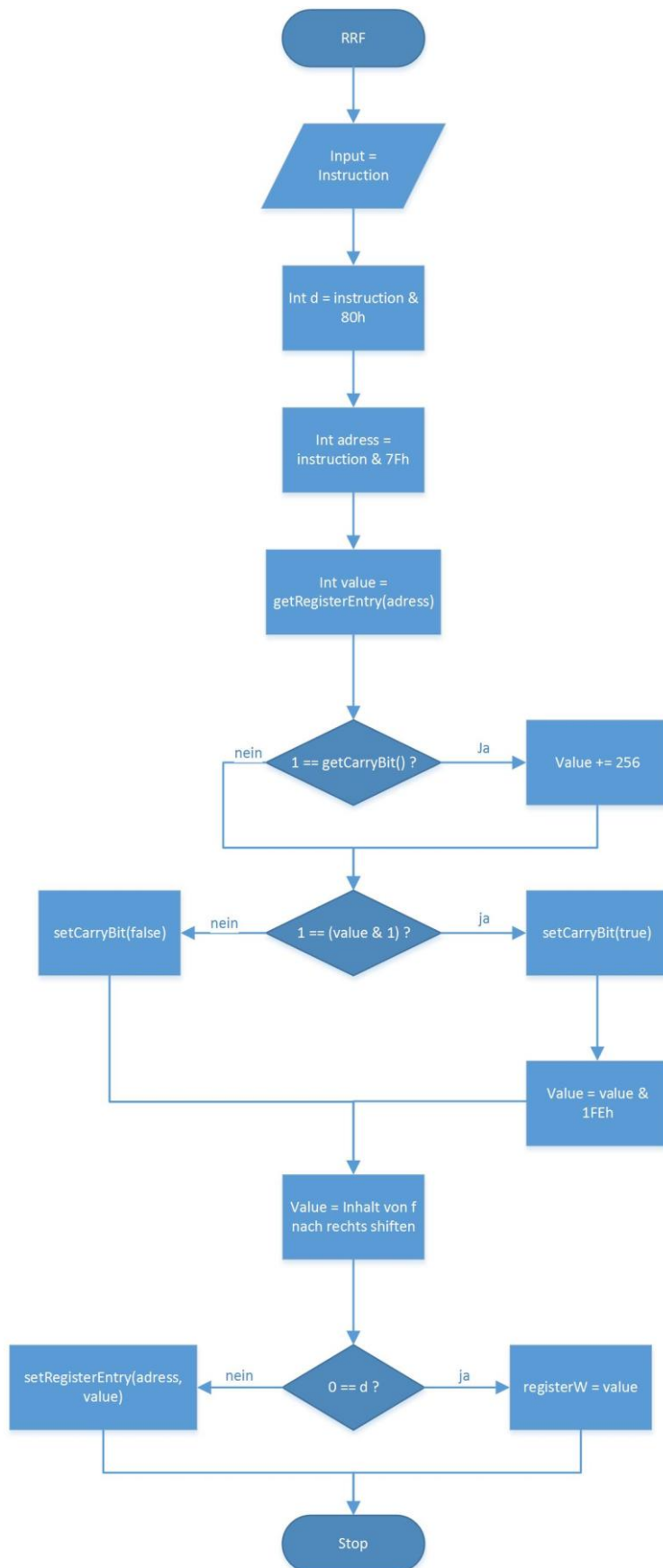


Abbildung 9: Ablaufdiagramm RRF-Befehl

```

/**
 * The contents of register 'f' are rotated one bit to the right through
the
 * Carry Flag. If 'd' is 0 the result is placed in the W register. If
'd' is
 * 1 the result is placed back in register 'f'.
 *
 * @param f_d
 *
 */
private void do_rrf(int f_d) {
    final int d = f_d & 0b10000000;
    final int adress = f_d & 0b01111111;
    int value = getRegisterEntry(adress);
    if (1 == getC()) {
        value += 256;
    }
    if ((value & 0b00000001) == 1) {
        setC(true);
        value = value & 0b11111110;
    } else {
        setC(false);
    }
    value = value >> 1;
    if (0 == d) {
        registerW = value;
    } else {
        setRegisterEntry(adress, value);
    }
}

```

Quellcode 3: RRF-Befehl

In Quellcode 3 wird die Befehlsdurchführung im Simulationsprogramm dargestellt.

4.8.2 Bitorientierte Operationen

Nachstehend wird der BTFSC (Bit Test, Skip if clear) Befehl erläutert. Dieser Befehl prüft, ob ein Bit *b*, welches durch 3 bit im Operationscode dargestellt wird, im f-Register gesetzt ist oder nicht. Ist dieses Bit gesetzt, wird der nächste Befehl im Programm ausgeführt, ansonsten wird der nächste Befehl übersprungen.

Zunächst wird wieder der übergebene Integerwert der „do_btfsf(int b_f) in die 3 bit für das Bit und die Speicheradresse für das f-Register aufgeteilt und in unterschiedliche Variablen gespeichert. Anschließend wird einer ausgelagerten Funktion die Speicheradresse und das entsprechende Bit übergeben. Diese Funktion gibt einen bool'schen Wert zurück, welcher angibt, ob das entsprechende Bit an der Speicheradresse gesetzt ist oder nicht.

Über eine if-Abfrage wird demnach der Programmzähler um eins erhöht, sofern der Rückgabewert „false“ liefert. Damit wird der nächste Befehl ohne eine Abarbeitung einfach übersprungen. Sollte der Wert true liefern, wird der Programmzähler nicht erhöht und der nächste Befehl dementsprechend durchgeführt.

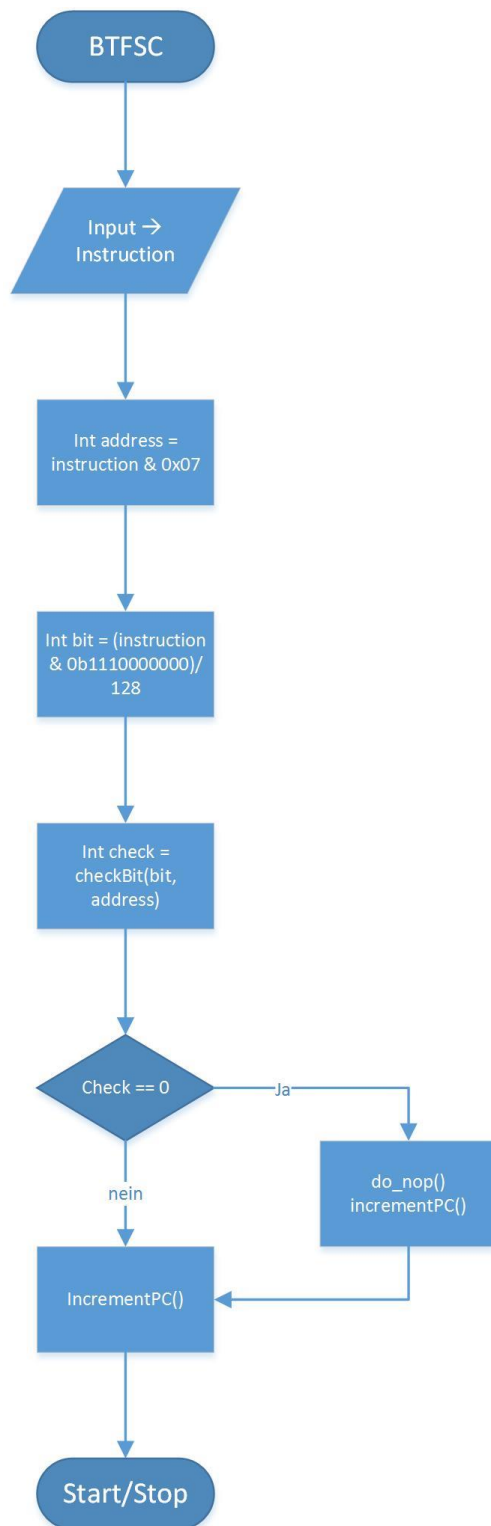


Abbildung 10: Ablaufdiagramm BTFSC-Befehl

Nachfolgend wird deutlich, wie der Befehl programmintern abgearbeitet wird.

```

/**
 *
 * If bit 'b' in register 'f' is '1' the the next instruction is
executed.
 * If bit 'b' in register 'f' is '0' then the next instruction is
discarded
 * and a NOP is executed instead, making this a 2Tcy instruction
 *
 * @param b_f
 */
private void do_btfsc(int b_f) {
    final int address = b_f & 0b0001111111;
    final int bit = (b_f & 0b1110000000) / 128;
    if (!checkBitSet(bit, address)) {
        do_nop();
        setProgramCounter(getProgrammCounter() + 1);
    }
}

```

Quellcode 4: BTFSC-Befehl

4.8.3 Literal- und Kontrolloperationen

Der ADDLW (Add Literal and W) addiert den Wert eines 8-bit Literals, welches der Operationscode beinhaltet, zum Wert des W-Registers und speichert den Wert wieder dort ab.

Zunächst wird das Literal aus dem Operationscode ausgewertet. Dies wird mit einer Verundung mit dem Wert 0xFF erreicht und das Ergebnis wird in einer Variable gespeichert. Anschließend wird dieser Wert mit dem Inhalt des W-Registers addiert und wieder im W-Register gespeichert.

Danach muss überprüft werden, ob verschiedene Bits noch gesetzt werden müssen. Zum einen wird geprüft, ob das Ergebnis größer als 255 ist. Ist dies der Fall wird das Carry-Bit gesetzt und das Ergebnis mit 0xFF verundet, da nicht mehr als 8 bit gespeichert werden können.

Anschließend wird geprüft, ob von Bit 4 auf Bit 5 ein Übertrag bei der Operation stattfindet. Ist dies der Fall, wird das DC-Bit gesetzt.

Die dritte Prüfung beinhaltet den Test, ob das Ergebnis null ergibt. Ist der Test positiv, wird das Zero-Bit gesetzt.

Nach den drei Tests wird das Ergebnis dann im W-Register gespeichert und die Abarbeitung des Befehls ist durchgeführt.

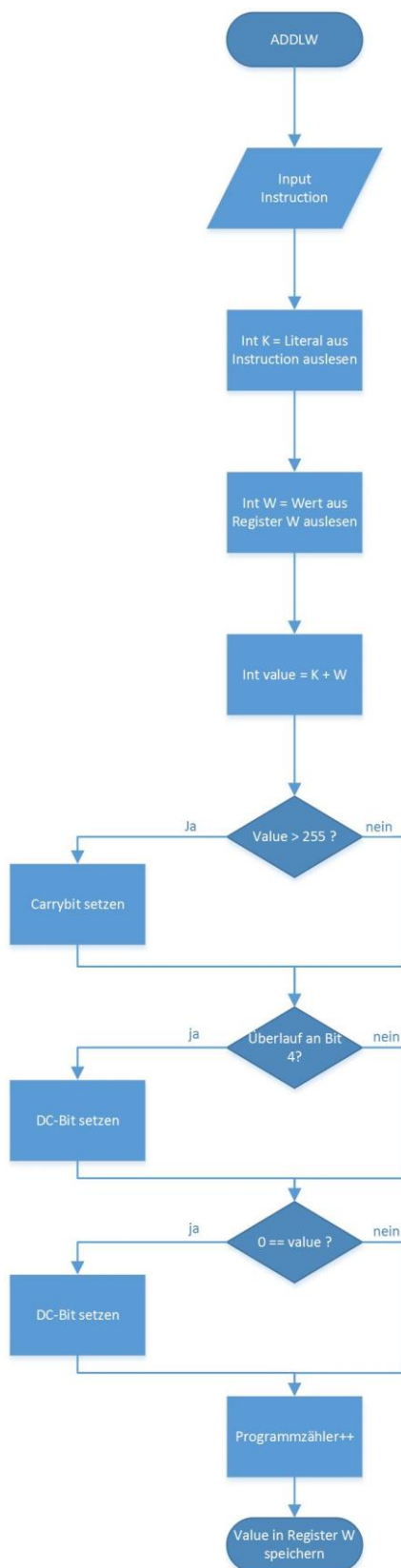


Abbildung 11: Ablaufdiagramm ADDLW

Abbildung 11 zeigt das Ablaufdiagramm des Befehls, nachstehend wird die Umsetzung im Quellcode dargestellt.

```
/**
 * The content of the w-register are added to the eight bit literal
 * 'hex4'
 * and the result is placed in the w-register
 *
 * @param lit_K
 *
 */
private void do_addlw(int lit_K) {
    final int lit_K_temp = lit_K & 0b01111111;
    int value = lit_K_temp + registerW;
    if (value > 255) {
        setC(true);
        value = value & 0xFF;
    } else {
        setC(false);
    }
    if (((lit_K_temp & 0b1000) + (registerW & 0b1000)) == 16) {
        setDC(true);
    } else {
        setDC(false);
    }
    if (value == 0) {
        setZ(true);
    } else {
        setZ(false);
    }
    registerW = value;
}
```

Quellcode 5: ADDLW Befehl

5 Fazit

In diesem Kapitel werden die Ergebnisse reflektiert und mit den ursprünglichen Vorstellungen verglichen. Die Umsetzung des Simulators erforderte einige Grundkenntnisse und führte darüber hinaus zu einigen spezifischeren Problemen. Durch das breite Band und die erfordernten Kenntnisse aus früheren Semestern konnte dieses Wissen angewendet und durchaus erweitert werden. Gerade für Projekte im späteren Berufsleben war dieses Projekt eine durchaus interessante Aufgabe, da hier auch fächerübergreifend gearbeitet werden musste. Die Planung der Anwendung war dabei genauso wichtig, wie die technische Umsetzung oder die gegenseitige Unterstützung im Team. Je größer das Team wird, desto schwieriger wird es die zu bewältigenden Aufgaben zu verteilen oder es kommt der Eindruck auf, dass Aufgaben umfangsmäßig ungleich aufgeteilt wurden.

Letztendlich war das Projekt eine sehr interessante Aufgabe, das zum einen das fachspezifische Wissen erweitern konnte, aber vor allen Dingen auch das Arbeiten im Team mit der dazugehörigen Koordination und Planung des gesamten Projektes.

Durch die nachträgliche Entscheidung, nach dem MVC-Konzept zu entwickeln, musste ein Teil der Arbeit noch einmal umgesetzt werden, was letztendlich durch eine bessere Anfangsplanung vermeidbar gewesen wäre aber dennoch eine wertvolle Erfahrung für das gesamte Team ist.

Durch den modularen Aufbau des Simulators konnten nahezu alle gewünschten Funktionen umgesetzt werden. Durch die 64-bit Architektur der Prozessoren des PCs, auf dem die Serielle Schnittstelle implementiert werden sollte, wurde zwar eine offensichtlich darauf ausgelegte Bibliothek gefunden, allerdings konnte trotz vieler Versuche bis jetzt keine Verbindung zu einem Port aufgebaut werden.

5.1 Probleme

Beim Versuch die serielle Schnittstelle zu implementieren, traten diverse Probleme auf. Zur Verbindung mit der Seriellen Schnittstelle wurde mit verschiedenen Java-Bibliotheken versucht, auf verfügbare Schnittstellen des PCs zuzugreifen. Aufgrund der 64-bit Architektur wurde letztendlich eine Bibliothek (RxCx 2.2 x64) gewählt, die für diese kompatibel ist. Hierzu wurden nach Anleitung für die Serielle und Parallele Verbindung jeweils eine *.dll-Datei in das

Java-Verzeichnis des PCs kopiert, sowie die entsprechende .jar-Datei in die Java-Klasse importiert, in der die serielle Verbindung umgesetzt werden sollte.

Beim Testen der Verbindung wurden allerdings nach diversen Versuchen und verschiedenen Einstellungen keine verfügbaren Ports zurückgeliefert. Damit war es nicht möglich, die Funktion der seriellen Schnittstelle weiter im PIC-Simulator zu implementieren.

Um mit der seriellen Schnittstelle arbeiten zu können, ist die Verfügbarkeit der Schnittstelle in der Java-Bibliothek Bedingung. Um evtl. eine spätere Umsetzung der seriellen Schnittstelle möglich zu machen, wurde die Klasse im Java-Projekt belassen. Besteht die Möglichkeit auf eine Schnittstelle zugreifen zu können, kann im Simulator dann ein neuer Thread erzeugt werden und Daten über eine geöffnete Verbindung gesendet werden. Die zu sendenden Daten können dabei an von der Bibliothek bereitgestellte Funktionen übergeben werden und somit über die serielle Schnittstelle versendet werden.