

# Dokumentation - PIC16F84 Simulator

Vorgelegt von Dominik Lange und Nico Rahm

<b>Allgemeines</b>	<b>2</b>
Grundsätzliche Arbeitsweise eines Simulators	2
Vor- und Nachteile einer Simulation	2
Programmoberfläche und deren Handhabung	3
Die Toolbar	3
Der Programmcode	4
Der Datenspeicher	4
SFR	5
Timing und Steuerelemente	6
Realisation	7
Grundkonzept und Gliederung	7
Programmiersprache und Framework	7
Konzept	7
Gliederung	7
Programmstruktur und Ablauf	9
Interrupts	11
Timer0	11
RB0	11
RB4-7	11
TRIS-Register	11
<b>Zusammenfassung</b>	<b>12</b>

# Allgemeines

## Grundsätzliche Arbeitsweise eines Simulators

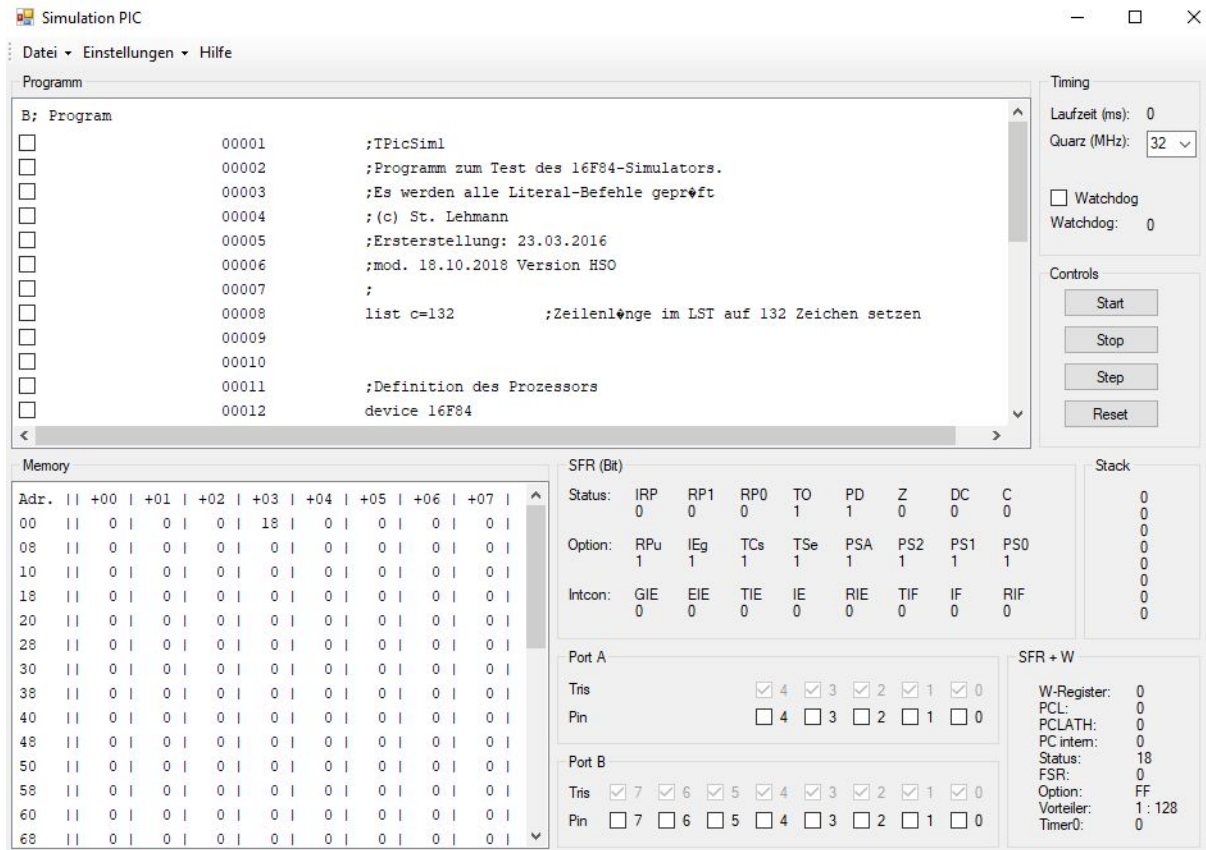
Ein Simulator dient Grundsätzlich dazu eine bestimmte Hardware in mit allen Funktionen in Software abzubilden. Der PIC-Simulator wird zum entwickeln und testen von Assembler Programmen genutzt. Dazu wird der Assembler Code in einer Hochsprache emuliert. Simulatoren stellen meist zusätzliche Funktionen zur Verfügung um den geschriebenen Code zu testen. Dazu gehören eine grafische Benutzeroberfläche mit Register- und Speicher Visualisierung, Schleifen orientiertes oder Schrittweises Ausführen, Breakpoints und Interaktion mit den I/O Ports des Mikroprozessors.

## Vor- und Nachteile einer Simulation

Eine Simulation ist und bleibt lediglich eine Nachbildung der Realität. Daraus resultiert, dass Umwelteinflüsse nur sehr schwer oder gar nicht Simuliert werden können. Auch physikalische Fehler, wie ein falsch übertragenes Bit oder sonstige beeinträchtigungen wie z.B. Verschleiß können hier nicht auftreten. Dadurch kann ein verzerrtes Bild der Funktionalität auf der Hardware entstehen. Zudem benötigt ein Simulator, mit entsprechenden Sonderfunktionen weitaus mehr Ressourcen als ein Mikroprozessor selbst.

Jedoch ist klar, dass eine Simulation auch viele Vorteile bieten kann. So ist der wohl auffälligste Unterschied das einfache Laden des zu testenden Programms. Denn beispielsweise ein EPROM zu beschreiben, bzw. neu beschreiben ist relativ aufwendig. In einem Simulator kann das Programm einfach geladen oder neu geladen werden sobald Änderungen an dem Code vorgenommen wurden. Auch die Tatsache, dass man nach jedem Befehlstakt die Veränderung der Register und Flags beobachten kann macht die Fehlersuche bedeutend einfacher und damit auch effizienter. Über die Möglichkeit die I/O Ports auch manuell zu manipulieren ist man zuzüglich nicht von anderer Hardware abhängig die mit dem Mikroprozessor kommuniziert. So kann man mit idealisierter Eingabe arbeiten und Fehler in anderer Hardware ausschließen.

# Programmoberfläche und deren Handhabung



Die graphische Benutzeroberfläche des PIC16F84 Simulators besteht grundsätzlich aus den folgenden Faktoren:

- ein Fenster zur Visualisierung des Programmcodes (oben links),
- eine Visualisierung des gesamten Datenspeichers (unten links),
- Visualisierungen einzelner Register mit den jeweiligen Bits (unten rechts), sowie
- der Laufzeit und Steuerelemente zum Starten, Stoppen und Resetten (oben rechts).

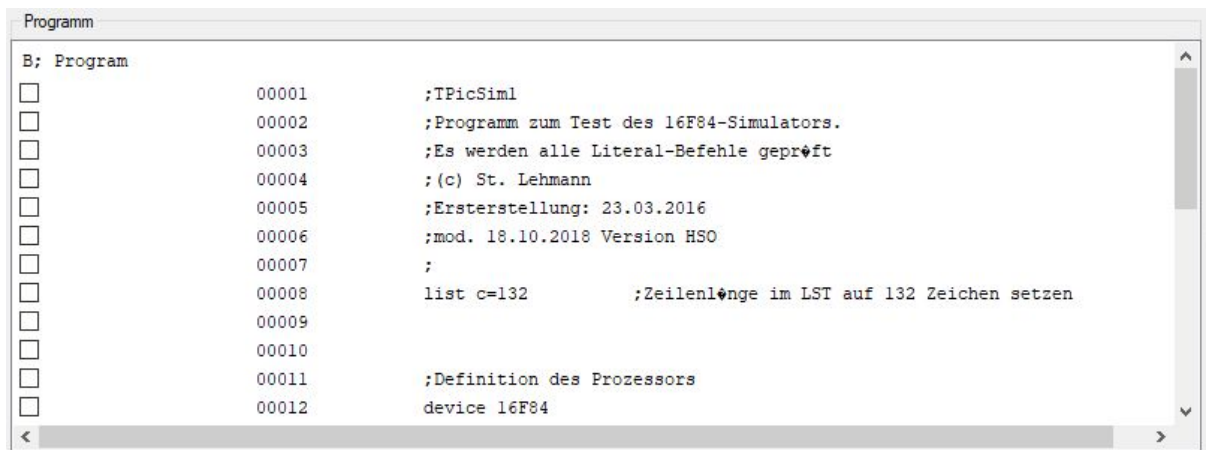
Übernommen wird die Visualisierung von der Form-Klasse namens "Gui\_Simu". Diese verfügt über den generierten Designercode, welcher die Darstellung des Fensters mit den Windows-Elementen übernimmt. Darüber hinaus gibt es den logischen Code, welcher den Elementen Werte zuweist, sie aktualisiert, Timer und Programmschleifen verwaltet, und im allgemeinen das Frontend mit dem Backend verknüpft. Von ihm ist zukünftig die Reden wenn es um den Code der Form-Klasse der Simulation geht.

## Die Toolbar



In der obigen Aufzählung nicht aufgelistet ist die Toolbar, da sie keinen direkten Bestandteil der PIC-Simulation darstellt. Die Toolbar bietet Funktionen zum laden einer .LST-Datei, sowie extra Informationen über die Software.

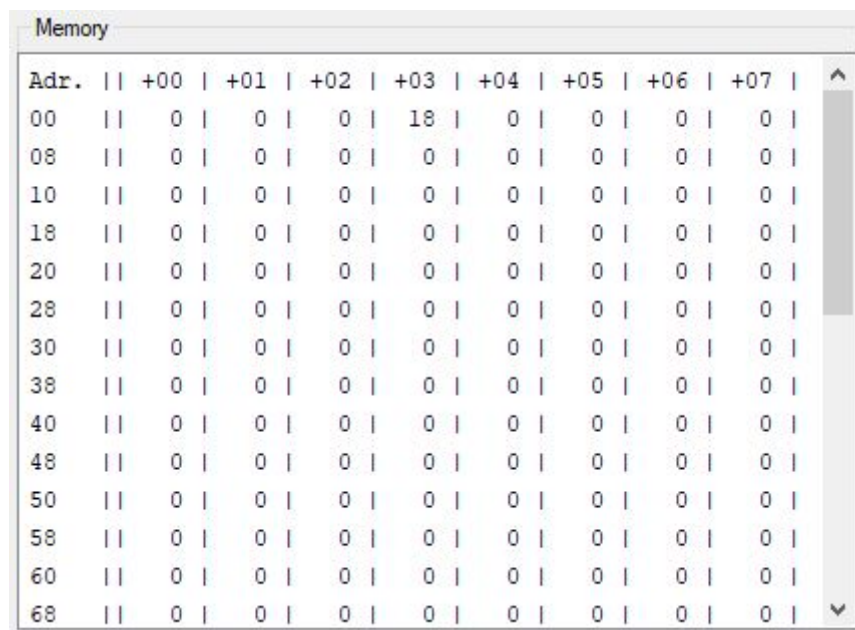
## Der Programmcode



```
B; Program
00001      ;TPicSim1
00002      ;Programm zum Test des 16F84-Simulators.
00003      ;Es werden alle Literal-Befehle geprüft
00004      ;(c) St. Lehmann
00005      ;Ersterstellung: 23.03.2016
00006      ;mod. 18.10.2018 Version HSO
00007      ;
00008      list c=132      ;Zeilenlänge im LST auf 132 Zeichen setzen
00009
00010
00011      ;Definition des Prozessors
00012      device 16F84
```

Die Visualisierung des Programmcodes ist in einem Listview-Element realisiert. Nach dem Laden einer .LST-Datei in der Toolbar wird der Assembler Code in das Listview-Element geladen. Neben der Visualisierung wird hier die Möglichkeit geboten, Breakpoints zu setzen um die Programmausführungen an entsprechenden Stellen zu unterbrechen. Zusätzlich wird die jeweils im nächsten Schritt auszuführende Zeile grau selektiert.

## Der Datenspeicher



Adr.	+00	+01	+02	+03	+04	+05	+06	+07
00	0	0	0	18	0	0	0	0
08	0	0	0	0	0	0	0	0
10	0	0	0	0	0	0	0	0
18	0	0	0	0	0	0	0	0
20	0	0	0	0	0	0	0	0
28	0	0	0	0	0	0	0	0
30	0	0	0	0	0	0	0	0
38	0	0	0	0	0	0	0	0
40	0	0	0	0	0	0	0	0
48	0	0	0	0	0	0	0	0
50	0	0	0	0	0	0	0	0
58	0	0	0	0	0	0	0	0
60	0	0	0	0	0	0	0	0
68	0	0	0	0	0	0	0	0

Der Datenspeicher ist in 2 Partitionen unterteilt, die Special Function Register (SFR) und die General Purpose Register (GPR). Diese werden über einen Adressraum von 00h bis FFh adressiert. Nicht belegte Adressen sind nicht implementiert und liefern den Wert 0.

Zusätzlich ist der Adressraum in 2 Bänke unterteilt. Bank 0 beinhaltet Adressen von 00h bis 7Fh und Bank 1 beinhaltet Adressen von 80h bis FFh.

Im Simulator wird der gesamte Adressraum von 00h bis FFh byteweise visualisiert.

Dabei sind die Adressen zeilenweise von links nach rechts numerisch sortiert. Die beiden Bänke sind nicht weiter gekennzeichnet.

## SFR

SFR (Bit)									Stack
Status:	IRP	RP1	RP0	TO	PD	Z	DC	C	0
	0	0	0	1	1	0	0	0	0
Option:	RPu	IEg	TCs	TSe	PSA	PS2	PS1	PS0	0
	1	1	1	1	1	1	1	1	0
Intcon:	GIE	EIE	TIE	IE	RIE	TIF	IF	RIF	0
	0	0	0	0	0	0	0	0	0

Port A						SFR + W	
Tris	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	W-Register:	0
Pin	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	PCL:	0
	4	3	2	1	0	PCLATH:	0

Port B							
Tris	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
Pin	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
	7	6	5	4	3	2	1

PC intem:	0
Status:	18
FSR:	0
Option:	FF
Vorteiler:	1 : 128
Timer0:	0

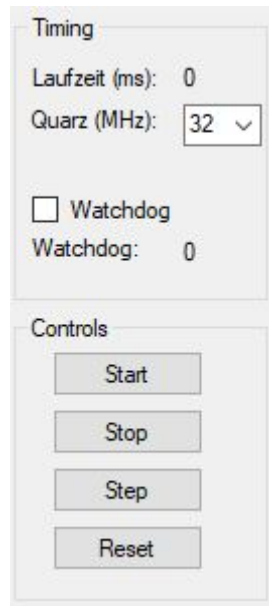
Die oben genannten SFR werden unter anderem von der CPU genutzt, um die Operationen des Gerätes zu steuern. Sie enthalten wichtige Einträge, welche in bitweiser Darstellung deutlich einfacher auszulesen sind.

Im oberen Teil des Bildes ist das Status-, Option- und Intcon-Register bitweise dargestellt. Daneben ist der Stack visualisiert.

Im linken unteren Teil befindet sich PortA und PortB, welche sowohl bitweise angezeigt, als auch manuell getoggelt werden können.

In der rechten Box sind weitere wichtige Register byteweise, so wie der Vorteiler und der Timer0 dargestellt.

## Timing und Steuerelemente



The image shows a software interface with two main sections: 'Timing' and 'Controls'. The 'Timing' section contains a label 'Timing' at the top, followed by 'Laufzeit (ms): 0' and 'Quarz (MHz): 32' with a dropdown arrow. Below this is an unchecked checkbox labeled 'Watchdog' and 'Watchdog: 0'. The 'Controls' section is below the 'Timing' section and contains four buttons: 'Start', 'Stop', 'Step', and 'Reset'.

In der Timing Box wird die Laufzeit angezeigt. Hierbei handelt es sich logischerweise nicht um eine tatsächliche Uhrzeit, sondern um einen simulierten Timer, welcher mit jeder Aktion hoch zählt. Die Quarzfrequenz ist dabei frei wählbar.

In der unteren Box wird die Steuerung des Simulators ermöglicht. Der Button *Start* führt das geladene Programm fortlaufend durch.

Über den Button *Stop* kann die Ausführung unterbrochen werden um sie abzurechnen, oder an einem späteren Zeitpunkt wieder zu starten.

Mit dem Button *Step* kann der Programmcode schrittweise ausgeführt werden, dabei muss der Button für jeden Schritt gedrückt werden.

Mit *Restart* wird die Ausführung unterbrochen und zurückgesetzt. Das bedeutet der Speicher initialisiert und der Program Counter wird zurückgesetzt. Diese Funktion simuliert den Power-Reset des Prozessors.

# Realisation

## Grundkonzept und Gliederung

### Programmiersprache und Framework

Aufgrund der gegebenenheiten ein Programm für Windows zu entwickeln, welches über ein GUI verfügen muss haben wir uns für das .NET Framework mit der Programmiersprache C# entschieden. Hier kann über den GUI Designer mit relativ geringem Aufwand eine grafische Oberfläche erzeugt und mit Funktionen belegt werden.

### Konzept

Angestrebt war eine MVC ähnliche Architektur. Dabei ist das GUI und die restliche business Logik voneinander getrennt. Da für den Simulator keine Persistenz benötigt wird, konnte dieser Faktor außer Acht gelassen werden. Zudem sollte nach dem SOFA Design Prinzip vorgegangen werden. Das heißt alle Methoden sollten kurz sein, nur eine Sache erledigen, dabei möglichst wenig Argumente brauchen und auf einem einzigen Abstraktionslevel bleiben.

### Gliederung

Der Simulator wird in einzelne Komponenten unterteilt, die sich jeweils um eine Funktionalität kümmern. Es gibt diverse Hauptkomponenten ohne die gar keine Simulation möglich ist. Zudem sind weitere Hilfskomponenten angeschlossen, welche bestimmte Funktionen des Simulators realisieren. Somit ist auch ein weiterer ausbau mit weiteren Funktionalitäten gut möglich.

Hauptkomponenten (im folgenden Kapitel beschrieben):

- Parser
- Decoder
- Executer
- ROM
- Memory
- Controller
- GUI

Zusätzliche Komponenten:

- **Interrupt Controller**  
Der Interrupt Controller erkennt und steuert die 3 Interrupts ( T0IF, INTF, RBIF)

- **EEPROM**

Der Halbleiterspeicher kann durch Spannungsimpulse beschrieben und gelöscht werden.

- **Stack**

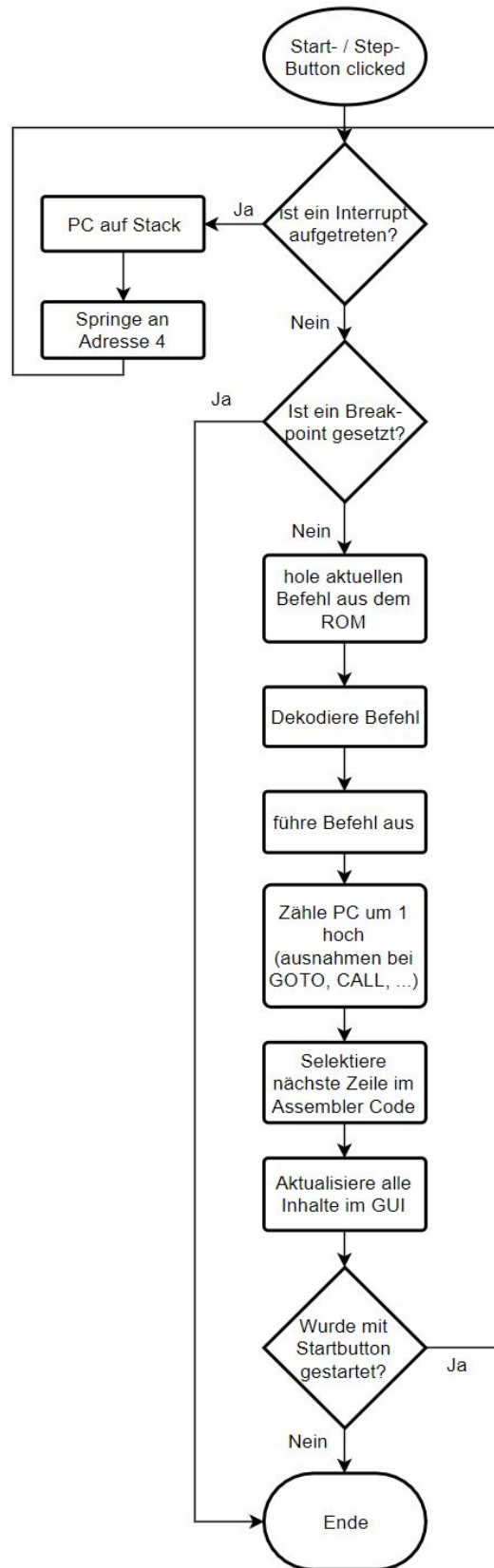
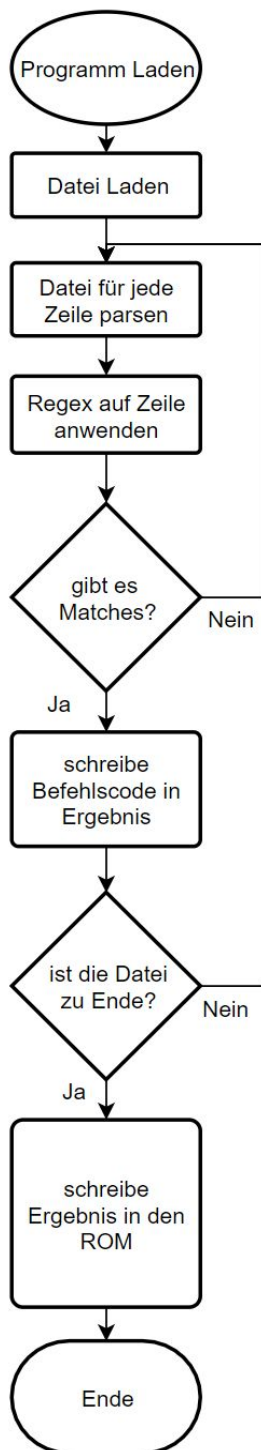
Der Controller schiebt im Falle eines Interrupts oder eines Sprungbefehle seinen aktuellen PC auf den Stack. Diese ist 8 Elemente tief mit je 13 Bit. Er unterstützt die push()-, pop()- und peer()-Operation. Wenn ein neuntes Element auf den Stack geschrieben wird, überschreibt dieses das erste Element.

- **Prescaler**

Der Vorteiler erzeugt den angezeigten Wert des Timers, indem er den tatsächlichen Timerwert durch einen in den entsprechenden Registern gesetzten Wert teilt.



## Programmstruktur und Ablauf



Die Form-Klasse "Gui\_Simu", oder auch "Simulationsklasse" befindet sich auf einem übergeordneten Abstraktionslevel. Sie empfängt Eingaben des Benutzers und verknüpft die Logik im Backend mit dem Frontend und der Visualisierung. Die "Simulationsklasse" erzeugt Instanzen aller notwendigen Klassen und stellt den Ursprung jedes eingeleiteten Vorgangs dar.

Der Programmcode wird zu Beginn über das Frontend in den Parser geladen, wo er eingelesen und geparkt wird. Das bedeutet, der Parser schneidet die enthaltenen Kommandos aus. Diese übergibt der Parser einer Instanz der Klasse Rom, wo sie in einem 1024 Elemente großen integer-array gespeichert werden. Die Rom Instanz kann die Kommandos anschließend mit einem Aufruf der Funktion *fetchCommand(pc)* und dem übergebenen Program Counter (PC) wieder ausgeben.

Um die Ausführung des Assembler Codes einzuleiten, muss einer der oben genannten Buttons gedrückt werden. In diesem Falle wird die Funktion *step()* der Controller-Instanz durchgeführt, welche alle relevanten Aktionen einer Befehlsabarbeitung durchführt. Bevor allerdings ein Befehl abgearbeitet werden kann, muss überprüft werden ob ein Interrupt vorliegt. Dies übernimmt der InterruptController, welcher im folgenden Kapitel noch genauer erläutert wird. Anschließend wird überprüft, ob die Abarbeitung über einen gesetzten Breakpoint gestoppt werden soll. Ist auch das nicht der Fall, werden folgende Punkt abgearbeitet:

- Über die Speicherklasse (Memory) holt der Controller sich den aktuellen Wert des PCs.
- Zusammen mit dem PC wird der Befehlscode von der ROM-Instanz geholt.
- Der Befehlscode wird dem Decoder übergeben, welcher eine Instanz der Command-Klasse liefert.
- Die Command-Instanz kann nun dem Executer übergeben werden, um es auszuführen.
- Anschließend wird der PC inkrementiert

Abschließend aktualisiert die "Simulationsklasse" die Visualisierung. Dazu gehört:

- die Selektierung der nächsten Zeile
- das Aktualisieren der Laufzeit
- das Aktualisieren des Datenspeichers
- das Aktualisieren der einzeln aufgelisteten Register
- das Aktualisieren des Stacks

Nach Abarbeitung der Routine wird auf eine erneute Eingabe der Nutzers gewartet. Wird der Button *Start* gedrückt, so startet dieser einen Timer, welcher die beschriebene Routine regelmäßig aufruft.

## Interrupts

Interrupts werden über eine Instanz der Klasse "InterruptController" verwaltet. Bei jedem "Step" wird vor der Ausführung geprüft, ob ein Interrupt vorliegt. Wird kein interrupt erkannt, so folgt die Ausführung dem normalen Schema.

Wenn jedoch die Funktion "checkInterrupt()" einen Interrupt erkennt, so wird vom InterruptController die Funktion "interruptOccured()" des Executors aufgerufen. Diese speichert den aktuellen PC auf dem Stack, löscht das GIE-Flag aus dem INTCON Register und setzt den PC auf 4.

Danach folgt wieder die Ausführung nach normalem Muster. Es wird der Befehl der aktuellen Adresse, in diesem Fall "4", geholt und normal ausgeführt. Erst wenn ein Return Befehl ausgeführt wird, wird wieder der aktuelle Wert vom Stack geholt und in den PC geschrieben.

Das Setzen der einzelnen Interrupt Flags erfolgt je nach Interrupt unterschiedlich:

### Timer0

Der Timer0 Interrupt wird bei jedem Überlauf ausgelöst. Dabei wird bei jedem Inkrement des Timers geprüft, ob ein Überlauf stattgefunden hat. Ist dies der Fall, wird das T0IF gesetzt.

### RB0

Der RB0 Interrupt hat eine erhöhte Komplexität, da hier eine Unterscheidung nach Art des Inputs realisiert werden muss. Der Interrupt wird entweder bei jeder steigenden oder fallenden Flanke am Pin "RB0" ausgelöst. Dazu wird die "setBit()" Funktion in der Klasse Memory so überwacht, dass bei jeder stimulation des RB0-Pins der Interrupt Controller informiert wird. Dieser stellt fest, ob es sich um eine fallende oder steigende Flanke handelt und welche Konfiguration in diesem Moment eingestellt ist. Stimmen Konfiguration und aktuelle Flanke überein, wird das INTF gesetzt.

### RB4-7

Bei den übrigen RB Interrupts wird ein ähnliches Prinzip angewandt, wie bei dem RB0 Interrupt. Jedoch ist dieser Interrupt nicht an eine bestimmte Bedingung geknüpft. Die "setBit()" Funktion wird überwacht. Sobald sich der Wert von einem der RB4-7 Bits verändert, wird vom Interrupt Controller das Entsprechende RBIF gesetzt.

## TRIS-Register

Mit den Tris Registern können die entsprechenden Port Register bitweise auf Input, bzw. Output gesetzt werden. Wird beispielsweise das 2. Bit des TrisA Registers auf 1 gesetzt, wird das 2. Bit des PortA Registers als Input definiert.

Die Bits der Tris Register können nicht manuell getoggelt, sondern nur über den Programmcode gesetzt werden. Die Port Bits jedoch können sowohl über den Programmcode, als auch manuell gesetzt werden. Dies geschieht stets in einer If-Schleife, wobei die Bedingung ist, dass das entsprechende Tris Bit auf 1 gesetzt ist.

## Zusammenfassung

Das Projekt, eine Simulationssoftware für den PIC16F84 Mikrokontroller zu schreiben war grundsätzlich interessant und führte zu vielen neuen Erfahrungen und Erkenntnissen. Die Bearbeitungsmethoden konnte frei gewählt werden und das Ergebnis war an wenige Vorgaben gekoppelt. Diese beschränkten sich auf die Funktionalität einiger Grundbefehle des Mikrocontrollers, sowie auf vereinzelte Features der Benutzeroberfläche. Hilfsmittel war das gegebene Datenblatt des Controllers.

Diese Freiheit in der Wahl der Programmiersprache, dem Layout der GUI, sowie der Genauigkeit und Funktionalität der Simulation zog einige positive Effekte mit sich. Es floss viel Planungsaufwand und Eigenleistung in das Projekt. Daraus resultierte dann auch eine sehr individuelle und eigens erarbeitete Realisierung. Insgesamt brachte das Projekt viele Erfahrungen im selbstständigen Arbeiten, Planen, Zeitmanagement und in einem zielführenden und übersichtlichen Programmierstil mit sich.

Rückblickend kann festgehalten werden, dass die Funktionen des Mikrocontrollers recht gut über eine geeignete Software nachgebildet werden können. Die angewendeten Methoden, darunter die objektorientierte Programmierung nach dem SOFA Prinzip, halfen dabei den Programmcode extrem strukturiert aufzubauen. Somit werden einzelne Aufgaben immer von den jeweiligen Klassen-Instanzen übernommen, welche untereinander zusammenarbeiten und sich austauschen können. Nachträglich zu realisierende Features oder Aufgaben konnten zielsicher an den entsprechenden Stellen implementiert und hinzugefügt werden. Somit konnten selbst große und komplexe Anforderungen runtergebrochen und einzeln realisiert werden.

Probleme und Unsicherheiten konnten im Datenblatt des Herstellers nachgelesen werden, das hat meistens sehr gut funktioniert, wodurch wenig Risikomanagement erforderlich war. Andere weniger wichtige Komplexitäten, in unserem Fall die Realisierung des Watchdogs, konnten auf Eigenverantwortung ausgelassen werden.