

Handbuch zum Simulator  
für den Mikrocontroller PIC16F84

von

Gabriel Balzer und Michaela Fleig

am

22. Juni 2020

## Inhaltsverzeichnis

1.	Was ist der PIC-Simulator? .....	3
2.	Grundkonzept der Realisierung.....	4
3.	Flags.....	5
	Theorie Flags .....	5
	Praxis Flags .....	5
4.	Befehle.....	6
	Theorie Byte-orientierte Operationen .....	6
	Praxis Byte-orientierte Operationen .....	6
	Theorie Bit-orientierte Operationen .....	7
	Praxis Bit-orientierte Operationen .....	7
	Theorie Literal- und Kontroll-Operationen .....	8
	Praxis Literal- und Kontroll-Operationen .....	8
5.	I/O-Ports .....	10
	Theorie I/O-Ports.....	10
	Praxis I/O-Ports.....	10
6.	Stack .....	11
	Theorie Stack.....	11
	Praxis Stack.....	11
7.	Interrupt .....	13
	Theorie Interrupt.....	13
	Praxis Interrupt.....	13
8.	Fazit .....	14
9.	Anhang.....	15
	Beispiel Jira.....	15
	Link zum GitHub-Projekt .....	15
	Flussdiagramm „DECF“ .....	15
	Flussdiagramm „BTFSS“ .....	16
	Flussdiagramm „CALL“ .....	16
	Datenblatt PIC16F84.....	16

## 1. Was ist der PIC-Simulator?

Ein Simulator ist ein möglichst realitätsnahes Nachbilden von Geschehen der Wirklichkeit. Um Kosten zu reduzieren und Tests zu vereinfachen, werden häufig Problemkreise von der Realität gelöst und in Teilen oder im Gesamten abstrakt in ein Modell überführt, an welchem zielgerichtet experimentiert wird. Die Resultate werden anschließend wieder auf das reale Problem übertragen.

Der PIC-Simulator hat die Aufgabe, Implementierungen in Form von Dateien zu testen und auf seiner Oberfläche auszugeben. Hierbei sollen Teile der Hard- und Software des PIC16F84-Mikrocontrollers realisiert werden, um die Anforderungen der Programme zu erfüllen.

Der PIC-Simulator soll eine einfache Benutzeroberfläche besitzen, die es dem Tester ermöglicht, auf einfache Weise die Programme zu laden und schrittweise oder im Gesamten durchzuführen. Auch eine Ausgabe von verschiedenen Bereichen der Soft- und Hardware sollen möglich sein. Hierzu zählen beispielsweise die Quarzfrequenz und die TRIS-Register A und B.

## 2. Grundkonzept der Realisierung

Zunächst muss die Umsetzung des Simulators für den PIC16F84 Mikrocontroller strukturiert angegangen werden. Für die Planung und Aufgabenzuweisung im Zweierteam wird das webbasierte Projekt-Tool „Jira“ von Atlassian verwendet. Ein Ausschnitt aus dem Tool befindet sich im Anhang „Beispiel Jira“ auf Seite 15. Die Programmiersprache, in der der Simulator geschrieben werden soll, ist das objektorientierte C# von Microsoft, mit WPF-Erweiterung. Dadurch soll die Implementierung von Quellcode sowie die Erstellung und Strukturierung der Benutzeroberfläche vereinfacht über XAML-basierten Code zu realisieren sein. Zusätzlich wurde das Framework „MahApps.Metro“ zur Erstellung der WPF-basierten Oberfläche verwendet. Zur Versionsverwaltung wurde GitHub von Microsoft verwendet. Dadurch lässt sich in kurzer Zeit ein Repository anlegen, Quellcode kann schnell hochgeladen werden, ein Sprung zwischen den Versionen ist immer möglich. Der Link zum Projekt befindet sich im Anhang „Link zum GitHub“ auf Seite 15.

Einige Test-Implementierungen und viele Diagramme helfen, das Konzept zur Umsetzung von Teilbereichen modular aufzubauen. Nachfolgend soll anhand einiger Beispiele das Konzept und seine Umsetzung im Quellcode des Simulators aufgezeigt werden. Abbildung 1 zeigt das Konzept der Benutzeroberfläche.

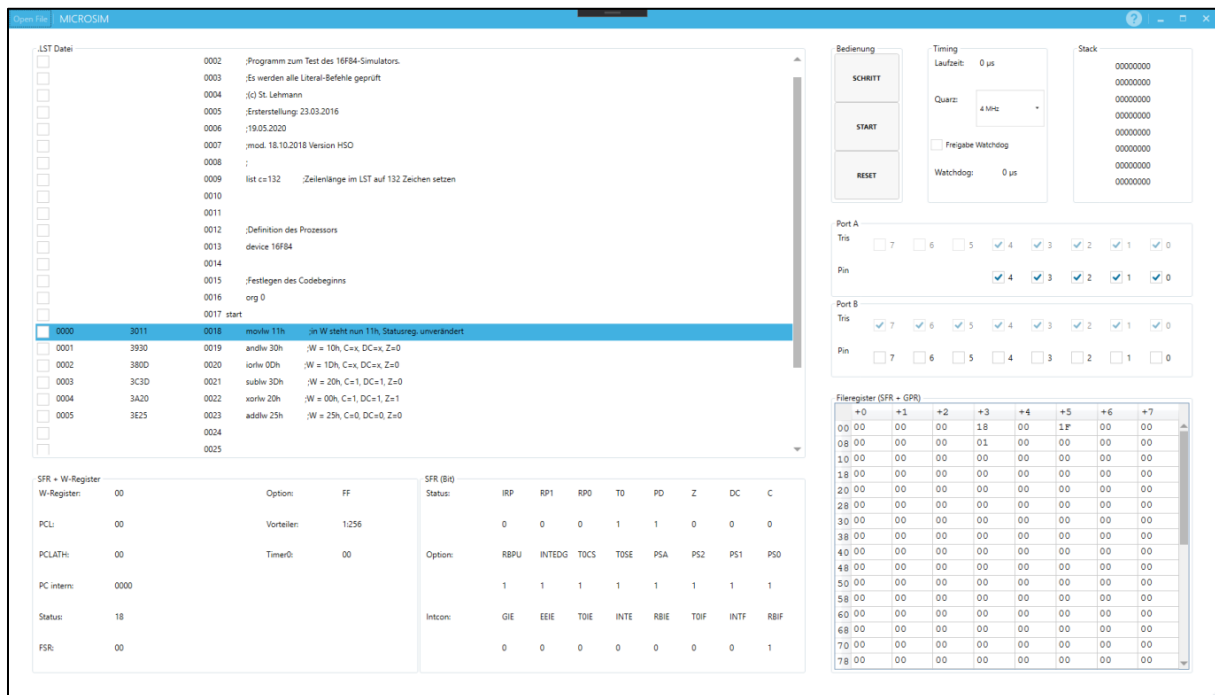


Abbildung 1: Konzept der Benutzeroberfläche des Simulators.

### 3. Flags

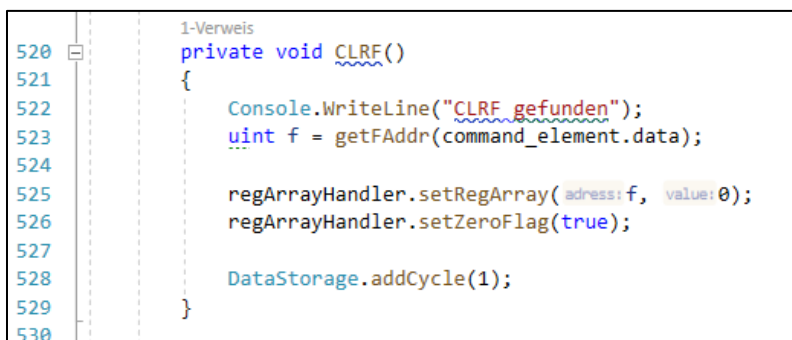
Es existieren im PIC-Mikrocontroller drei wichtige Flags, die in Abhängigkeit des Ergebnisses einer Operation gesetzt oder zurückgesetzt werden können. Anhand ihrer Status werden im Programmverlauf Entscheidungen getroffen.

#### Theorie Flags

Das Setzen der Flags soll beispielhaft anhand des Befehls „CLRF“ (Clear f) gezeigt werden. Hier wird das Zero-Flag (Z-Flag) gesetzt, wenn das Ergebnis der Operation 0 ist.

#### Praxis Flags

Mit dem Befehl „CLRF“ wird der Wert des übergebenen Parameters „f“ in eine Variable geschrieben (Abbildung 2, Zeile 523) und in die Register an die Stelle „f“ geschrieben (Zeile 525). Da der Wert von „f“ 0 wird, wird das Z-Flag gesetzt (Zeile 526).



```
520 1-Verweis private void CLRF()  
521 {  
522     Console.WriteLine("CLRF gefunden");  
523     uint f = getFAddr(command_element.data);  
524  
525     regArrayHandler.setRegArray(address: f, value: 0);  
526     regArrayHandler.setZeroFlag(true);  
527  
528     DataStorage.addCycle(1);  
529 }  
530
```

Abbildung 2: Der Befehl „CLRF“.

## 4. Befehle

Das sogenannte Instruction Set des PIC beschreibt Operanden, die vom Mikrocontroller über 14-bit Opcodes erkannt werden. Dabei werden nicht nur der Operand, sondern auch die Operatoren mit übergeben. Hierbei muss in drei Gruppen unterteilt werden: Byte-orientierte Dateiregister-Operationen, Bit-orientierte Dateiregister-Operationen und Literal- und Kontrolloperationen.

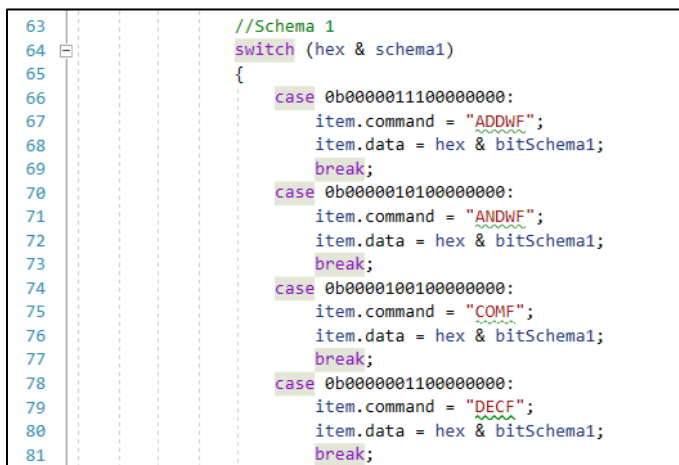
### Theorie Byte-orientierte Operationen

Der Befehl „DECF“ dekrementiert den übergebenen Parameter „f“. Dem Handbuch sind die Grenzwerte von „f“ zu entnehmen. Auch die Operation, die bei Aufruf dieses Befehls durchgeführt wird, ist dort hinterlegt. Der Parameter „f“ wird um einen Wert dekrementiert und in den weiteren, übergebenen Parameter „d“ kopiert, wenn das Ergebnis ungleich 0 ist. Nimmt „d“ jedoch den Wert 0 an, muss das Ergebnis der Operation in das W-Register geschrieben werden und das Z-Flag wird gesetzt.

Die Operation benötigt nur einen Zyklus zur Erfüllung ihrer Aufgabe. Das dazu erstellte Flussdiagramm befindet sich im Anhang Flussdiagramm „DECF“ auf Seite 15.

### Praxis Byte-orientierte Operationen

Der Operator wird zunächst über seine Maske, die im Datenblatt zu finden ist, über einen Switch-Case versucht zu finden, wie in Abbildung 3 in einem Ausschnitt für den Befehl „DECF“ dargestellt ist. Wenn er an der aktuellen Stelle des Programmzählers gefunden ist, wird dieser Befehl wie nachfolgend beschrieben ausgeführt.



```
63 //Schema 1
64 switch (hex & schema1)
65 {
66     case 0b0000011100000000:
67         item.command = "ADDWF";
68         item.data = hex & bitSchema1;
69         break;
70     case 0b0000010100000000:
71         item.command = "ANDWF";
72         item.data = hex & bitSchema1;
73         break;
74     case 0b0000100100000000:
75         item.command = "COMF";
76         item.data = hex & bitSchema1;
77         break;
78     case 0b0000001100000000:
79         item.command = "DECF";
80         item.data = hex & bitSchema1;
81         break;
```

Abbildung 3: Der Befehl „DECF“ wird über sein Encoding-Wert im Switch-Case erkannt.

Wie in der Theorie beschrieben, wird der Operator zunächst auf seine Grenzen geprüft (Abbildung 4Abbildung 3, Zeile 572). Der übergebene Parameter „f“ wird in einer Variablen zwischengespeichert (Zeile 574) und die eigentliche Operation des Befehls wird durchgeführt (Zeile 575).

Es wird geprüft, welchen Wert das Ergebnis annimmt und je nach wird das Z-Flag gesetzt (Zeile 578) und der Wert in das W-Register (Zeile 593) oder an die Speicherstelle von „f“ geschrieben (Zeile 597). Auch der Zyklus wird um eins erhöht (Zeile 599).

```

564 1-Verweis private void DECF()
565 {
566     Console.WriteLine("DECF gefunden");
567     uint f = getFAddr(command_element.data);
568     uint d;
569     int result;
570     int regValue;
571     d = command_element.data & 0b10000000;
572     if (f <= 127)
573     {
574         regValue = (int) regArrayHandler.getRegArray(f);
575         result = regValue - 1;
576         if (result == 0)
577         {
578             regArrayHandler.setZeroFlag(true);
579         }
580         else
581         {
582             regArrayHandler.setZeroFlag(false);
583         }
584     }
585     if (result == -1)
586     {
587         result = 0xFF;
588     }
589     if (d == 0)
590     {
591         DataStorage.w_register = (uint) result;
592     }
593     else
594     {
595         regArrayHandler.setRegArray(address: f, value: (uint)result);
596     }
597     DataStorage.addCycle(1);
598 }
599
600 }
601

```

Abbildung 4: Der Befehl „DECF“.

## Theorie Bit-orientierte Operationen

Der Befehl „BTFSS“ (Bit test f, skip if set) führt abhängig von den beiden übergebenen Parametern „f“ und „b“ den nächsten oder den übernächsten Befehl aus. Hierbei wird das Bit „b“ im Register „f“ betrachtet. Wenn das Bit gesetzt ist, wird der nächste Befehl im Programmverlauf übersprungen, wenn es nicht gesetzt ist, wird der nächste folgende Befehl ausgeführt und der darauffolgende Befehl übersprungen. Die Operation ist mit einer If-Bedingung von höheren Programmiersprachen vergleichbar.

Im Anhang befindet sich ein Flussdiagramm zum Befehl: Flussdiagramm „BTFSS“ 16.

## Praxis Bit-orientierte Operationen

In der Realisierung wird zunächst der Operator, wie im vorherigen Praxis-Beispiel beschrieben, der Operator über einen Switch-Case-Operator gesucht. Dann wird das Bit „b“ aus dem übergebenen Register „f“ ausmaskiert (Abbildung 5, Zeile 1003, 1004).

Zum Überprüfen, ob das Bit in „f“ den Wert von „b“ hat, werden die beiden Werte binär verundet (Zeile 1008) und auf ihren Wert geprüft. Wenn b nicht gesetzt ist, wird nichts unternommen (Zeile 1010, 1011). Wenn b gesetzt ist, wird der Programmzähler um einen Wert erhöht und ein „nop“ wird ausgeführt (Zeile 1017, 1018). Ein „nop“ ist ein leerer Befehl.

```

1-Verweis
996 private void BTFSS()
997 {
998     Console.WriteLine("BTFSS gefunden");
999     uint f = getFAddr(command_element.data);
1000     uint b;
1001     uint result;
1002     uint bCalc;
1003     b = command_element.data & 0b1110000000;
1004     b = b >> 7;
1005     result = regArrayHandler.getRegArray(f);
1006     bCalc = (uint)(Math.Pow((double)2, (double)b));
1007
1008     if ((result & bCalc) == 0)
1009     {
1010         // result[b] = 1
1011         // do nothing
1012     }
1013     else
1014     {
1015         // result[b] = 0
1016         PCL.addToPCL();
1017         NOP();
1018     }
1019     DataStorage.addCycle(1);
1020 }
1021

```

Abbildung 5: Der Befehl „BTFSS“.

## Theorie Literal- und Kontroll-Operationen

Der Befehl „CALL“ ruft eine Unterroutine auf, die an einer anderen Stelle als im fortfolgenden Programmzähler steht. Dabei wird zunächst der Operator über seinen Opcode erkannt. Der übergebene Parameter „k“ wird auf seine Grenzwerte überprüft.

Der Programmzähler wird um einen Wert erhöht und auf den Stack kopiert (push). Die die übergebene Adresse „k“ wird in den unteren Teil des Programmzählers an Stelle 0 bis 10 kopiert. Die oberen Bits des Programmzählers werden durch die Bit-Werte an Stelle 3 und 4 des PCLATH-Registers bestimmt.

Im Anhang befindet sich ein Flussdiagramm zum Befehl: Flussdiagramm „CALL“ 16

## Praxis Literal- und Kontroll-Operationen

Der Befehl wird wie in den Byte-orientierten Operationen über einen Switch-Case-Operator gesucht. Bei Auftreten wird der Befehl wie folgt ausgeführt.



```

1-Verweis
371 private void CALL()
372 {
373     uint pclath;
374     uint pclath3;
375     uint pclath4;
376     Console.WriteLine("CALL gefunden");
377     if (command_element.data <= 2047)
378     {
379         // save pc to stack
380         DataStorage.stack1.SetValueToStck(DataStorage.programCounter + 1);
381
382         pclath3 = regArrayHandler.getRegArray( adress: 0x0A) & 0b00000100;
383         pclath4 = regArrayHandler.getRegArray( adress: 0x0A) & 0b00001000;
384
385         pclath = pclath4;
386         pclath = pclath << 1;
387         pclath = pclath | pclath3;
388         pclath = pclath << 11;
389         pclath = pclath | command_element.data;
390         Console.WriteLine("PCLTEST CALL : " + pclath);
391         PCL.setPCL(pclath - 1);
392         Console.WriteLine("W-reg : " + DataStorage.w_register);
393         DataStorage.addCycle(2);
394     }
395 }

```

Abbildung 6: Der Befehl „CALL“.

Zunächst wird der Parameter „k“ auf seine Grenzwerte geprüft (Abbildung 6, Zeile 377). Wenn er innerhalb der Grenzen liegt, wird der aktuelle Programmzähler auf den Stack kopiert (Zeile 380). Die beiden Bits an der Stelle 3 und 4 des PCLATH-Registers werden ausmaskiert und getrennt gespeichert (Zeile 382, 383). Anschließend werden die getrennten Bit-Werte entsprechend ihrer späteren Wertigkeit in die Variable „pclath“ geshiftet (Zeile 385-389).

Der Programmzähler wird nun auf einen Wert geringer als den ausgerechneten „pclath“-Wert gesetzt. Dies ist der Implementierung geschuldet, da die Routine, die den nächsten Befehl in der geladenen Datei sucht, stets den Programmzähler um einen Wert erhöht. Dadurch wird am neuen Wert „pclath“ als Programmzähler gestartet.

## 5. I/O-Ports

Das TRIS-Register existiert für jeweils Port A und B. Es wird als Richtungsregister verwendet, das die Werte der jeweiligen Bits an seinen Ausgabetreiber meldet. Damit können Port A und B als Eingangs- oder Ausgangsregister geschaltet werden. Port A und B enthalten jeweils Interrupts, die durch Setzen der Bits als Eingang verwendet werden können.

### Theorie I/O-Ports

Das TRIS-Register bestimmt für die Werte, die für Port A oder B eingestellt werden, ob sie Eingang oder Ausgang darstellen. Die Port-Register können bit- oder byteweise geschrieben und gelesen werden. Diese Werte werden von einer Funktion zyklisch überprüft und auf der Benutzeroberfläche ausgegeben.

### Praxis I/O-Ports

Über die Methode „UpdatePin“, die zur Benutzeroberflächensteuerung gehört, werden, wie in Abbildung 7 dargestellt, die aktuellen Werte der TRISA-, TRISB-, PORTA – und PORTB- Register ausgelesen und entsprechend auf der Benutzeroberfläche als Wert einer Checkbox angezeigt.

```
230 private void UpdatePin()  
231 {  
232     var pinvalue = (int)regArrayHandler.getRegArray( adress: 0x05);  
233     var pinaarray :bool[] = Int32Extensions.ToBooleanArray(pinvalue);  
234     View.Pina = pinaarray;  
235  
236     var pinbvalue = (int)regArrayHandler.getRegArray( adress: 0x06);  
237     var pinbarray :bool[] = Int32Extensions.ToBooleanArray(pinbvalue);  
238     View.Pinb = pinbarray;  
239  
240     var trisavalue = (int)regArrayHandler.getRegArray( adress: 0x85);  
241     var trisaarray :bool[] = Int32Extensions.ToBooleanArray(trisavalue);  
242     View.Trisa = trisaarray;  
243  
244     var trisbvalue = (int)regArrayHandler.getRegArray( adress: 0x86);  
245     var trisbarray :bool[] = Int32Extensions.ToBooleanArray(trisbvalue);  
246     View.Trisb = trisbarray;  
247 }
```

Abbildung 7: Die Methode „UpdatePin“.

Diese Werte werden rechts auf der Benutzeroberfläche angezeigt, dieser Ausschnitt ist in Abbildung 8 abgebildet.

Port A								
Tris	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
Pin				<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>

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"/>	<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"/>	<input type="checkbox"/>

Abbildung 8: Anzeige von Port A und B auf der Benutzeroberfläche.

## 6. Stack

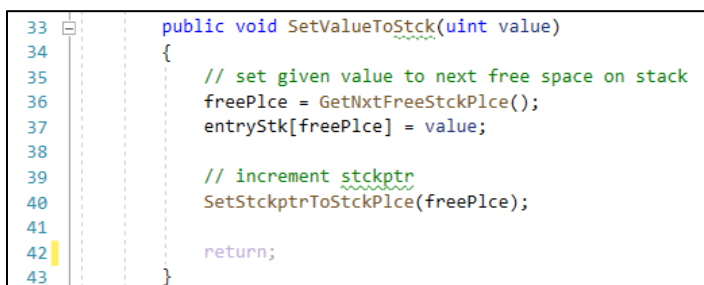
Der Stack kann als Erinnerungsspeicher des Mikrocontrollers angesehen werden. Hier werden alle Daten und Werte gespeichert, die gerade nicht benötigt werden, jedoch für eine spätere Weiterbearbeitung notwendig sind. Beispielsweise beim Aufruf einer Subroutine (CALL, RETFIE, RETURN, etc.) oder beim Aufruf eines Interrupts.

### Theorie Stack

Der Stack ist ein acht Werte tiefer Hardware-Stack für den Programmzähler. Dieser wird vom Simulator jedoch über die Software realisiert. Der Stackpointer kann keinesfalls von außen geschrieben oder gelesen werden. Es ist nur über zwei Funktionen möglich, die Werte aus dem Stack zu lesen oder zu schreiben. Wenn ein Wert geschrieben wird (push), zeigt der Stackpointer auf diesen neuen Wert. Wenn aus dem Stack gelesen wird, wird dieser gelesene Wert gelöscht (pop), der Stackpointer zeigt dann auf den Wert, der nun der zuletzt geschrieben wurde.

### Praxis Stack

Der Stack wird über die Methode „SetValueToStck“ wird ein Parameter übergeben, der auf den Stack an der nächsten freien Stelle (Abbildung 9, Zeile 37) kopiert wird und der Stackpointer auf den neu beschriebenen Wert gesetzt (Zeile 40).

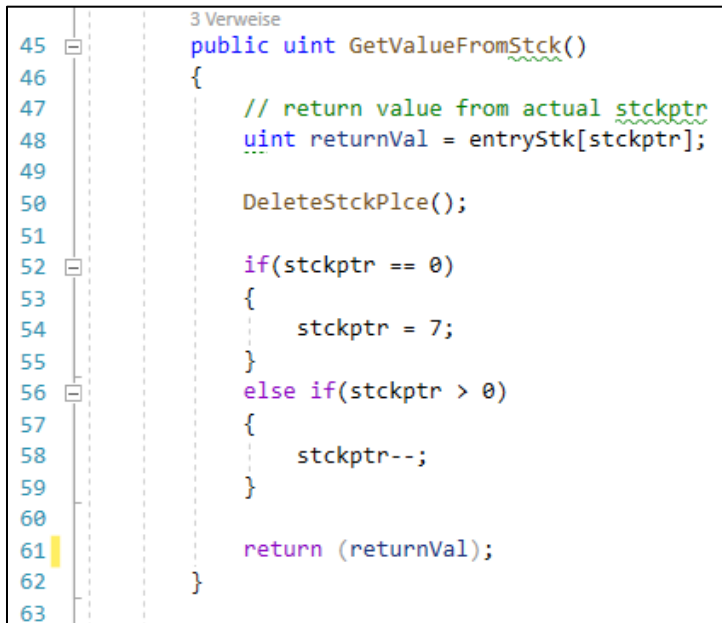


```
33 public void SetValueToStck(uint value)
34 {
35     // set given value to next free space on stack
36     freePfce = GetNxtFreeStckPfce();
37     entryStk[freePfce] = value;
38
39     // increment stckptr
40     SetStckptrToStckPfce(freePfce);
41
42     return;
43 }
```

Abbildung 9: Die Methode schreibt einen übergebenen Wert auf den Stack.

Der Stack wird über die Methode „GetValueFromStck“ ausgelesen. Dabei wird der Wert am aktuellen Stackpointer (Abbildung 10, Zeile 48) gelesen, dieser Platz wird auf einen initialen Wert 0 (Zeile 50) und

der Stackpointer um einen Wert dekrementiert (Zeile 56) falls er größer als 0 ist. Da der Stack als Ringregister angesehen wird, wird der Stackpointer an der Stelle 0 zu 7, dem nächsten Wert.



```
45 3 Verweise
46 public uint GetValueFromStck()
47 {
48     // return value from actual stckptr
49     uint returnVal = entryStk[stckptr];
50
51     DeleteStckPlce();
52
53     if(stckptr == 0)
54     {
55         stckptr = 7;
56     }
57     else if(stckptr > 0)
58     {
59         stckptr--;
60     }
61     return (returnVal);
62 }
63
```

Abbildung 10: Die Methode liest einen Wert aus dem Stack und gibt diesen zurück.

## 7. Interrupt

Ein Interrupt ist eine zeitlich nicht vorhersagbare Programmunterbrechung. Beim PIC16F84 sind keine verschachtelten und keine priorisierten Interrupts möglich. Jede Interrupt-Quelle besitzt ein eigenes Interrupt-Enable-Bit und ein Flag. Exemplarisch für das Erkennen und Behandeln von Interrupts soll nachfolgend der Timer0-Interrupt beschrieben werden.

### Theorie Interrupt

Wenn jeweils das Global Interrupt Enable Bit (GIE) im INTCON-Register, das Timer0 Overflow Interrupt Enable Bit (TOIE) und das Timer0 Overflow Interrupt Flag Bit (TOIF) gesetzt sind, wird ein Interrupt des Timer0 erkannt. Daraufhin soll der aktuelle Programmzähler auf den Stack kopiert und den Programmzähler auf die Interrupt-Einsprungadresse 0x04 gesetzt werden.

Das GIE wird im INTCON-Register gelöscht, bevor die Interrupt-Routine ausgeführt wird. So sind keine verschachtelten Interrupt möglich.

### Praxis Interrupt

Im Simulator wird zu Beginn der Command-Handler-Funktion die Methode „checkInterrupt“ aufgerufen. Diese überprüft zunächst, ob das GIE im INTCON-Register gesetzt wurde (Abbildung 11, Zeile 15). Anschließend wird für Timer0 das TOIE und TOIF überprüft (Zeile 18). Wenn diese beiden auch gesetzt sind, wird der Programmzähler auf den Stack kopiert (Zeile 20), implementierungsabhängig wird der Programmzähler auf 0x03 gesetzt (Zeile 21) und das GIE wird gelöscht (Zeile 22).

```
13 public void checkInterrupt()
14 {
15     if ((regArrayHandler.getRegArray(address:0x0B) & 0x80) == 0x80)
16     {
17         //Timer
18         if (((regArrayHandler.getRegArray(address:0x0B) & 0x20) == 0x20) && ((regArrayHandler.getRegArray(address:0x0B) & 0x04) == 0x04))
19         {
20             DataStorage.stack1.SetValueToStack(Pc1.getPCL());
21             Pc1.setPCL(0x04 - 1);
22             regArrayHandler.setRegArray(address:0x0B, value:(regArrayHandler.getRegArray(address:0x0B) & 0b01111111));
23         }
24     }
25 }
```

Abbildung 11: Die Methode „checkInterrupt“ überprüft, ob und welches Bit gesetzt wurde im INTCON-Register.

## 8. Fazit

Zusammenfassend kann gesagt werden, dass die realisierte Simulationsanwendung erfolgreich umgesetzt werden konnte. Implementierungsabhängige Gegebenheiten wurden mit einem Workaround umgangen.

Mit dieser Programmierarbeit sollte ein möglichst realitätsnahes Nachbilden der realen Funktionalitäten des PIC16F84 realisiert werden. Durch die intensive Beschäftigung mit dem Datenblatt des PIC16F84 konnten viele Teile in der Software implementiert werden. Die einfach aufgebaute Oberfläche stellt wichtige Register und Funktionen dar, die für den Benutzer während der Ausführung der Programme von Nutzen sind.

Die zentralen Implementierungen wurden exemplarisch in diesem Dokument aufgezeigt. Die Theorie aus dem Datenblatt wurde mittels Diagramme in Quellcode der verwendeten Programmiersprache übertragen. Es wurde erkannt, dass Pair-Programming, zu zweit programmieren, den Programmierstil fördert und zeitgleich ein Wissensaustausch stattfindet, der beide Partner bereichert. Es wurden auch einige neue Methoden zur Erstellung einer Benutzeroberfläche und dem Aufbau eines Programms kennengelernt. Den Aufbau des Programms in einem stabilen Konzept zur Gesamtarchitektur zu Beginn des Projektes festzuhalten gehört ebenfalls zu den persönlich gewonnenen Erkenntnissen. Dieses Projekt in einer Gruppe zu realisieren hält nicht nur von unnötigem Aufwand und Ausschweifungen ab, auch das Zeitmanagement und der Arbeitsaufwand wurden fair aufgeteilt.

Während des Entwicklungsprozesses sind nach Implementationen von neuen Befehlen regelmäßig Fehler aufgetaucht. In solch einem Fall musste immer händisch geprüft werden welcher Befehl oder Code-Teil diesen Fehler hervorruft. In Zukunft wäre es sinnvoll neue Implementationen direkt mit einem dazugehörigen Unit-Test zu versehen. Mit diesem Vorgehen kann die Anwendung sich selbst testen und es ist für den Programmierer wesentlich leichter den Fehler zu finden.

Die Programmierung eines Simulators bietet die Möglichkeit, die eigenen Programmier- und Hardware-Kenntnisse zu vertiefen. Anhand dieses Projekts konnte durch ein intensiveres Auseinandersetzen mit der Dokumentation des Mikrocontrollers ein besseres Verständnis für die PIC-Programmierung und –Möglichkeiten erworben werden.

Die vorliegende Programmierarbeit stellt grundlegende Funktionalitäten dar, die der Mikrocontroller bietet. Auf dieser Basis können weitere Implementierungen zur Funktionalität des Simulators wie auch zur benutzerfreundlichen Bedienung hinzugefügt werden.

## 9. Anhang

### Beispiel Jira

<input checked="" type="checkbox"/>	MIC-14
	Reset variables when a file is loaded
<input checked="" type="checkbox"/>	MIC-13
	Show Port A & B in GUI
<input checked="" type="checkbox"/>	MIC-12
	mark active Row in Datagrid
<input checked="" type="checkbox"/>	MIC-11
	Show Stack in GUI
<input checked="" type="checkbox"/>	MIC-10
	Watchdog
<input checked="" type="checkbox"/>	MIC-9
	Enable Breakpoints

Abbildung 12: Ausschnitt aus der Aufgabenverwaltungs-Anwendung „Jira“

### Link zum GitHub-Projekt

Das Projekt befindet sich auf GitHub und ist unter folgendem Link abrufbar:

<https://github.com/GabrielBalzer/microsim> (Stand: 26.06.2020)

### Flussdiagramm „DECF“

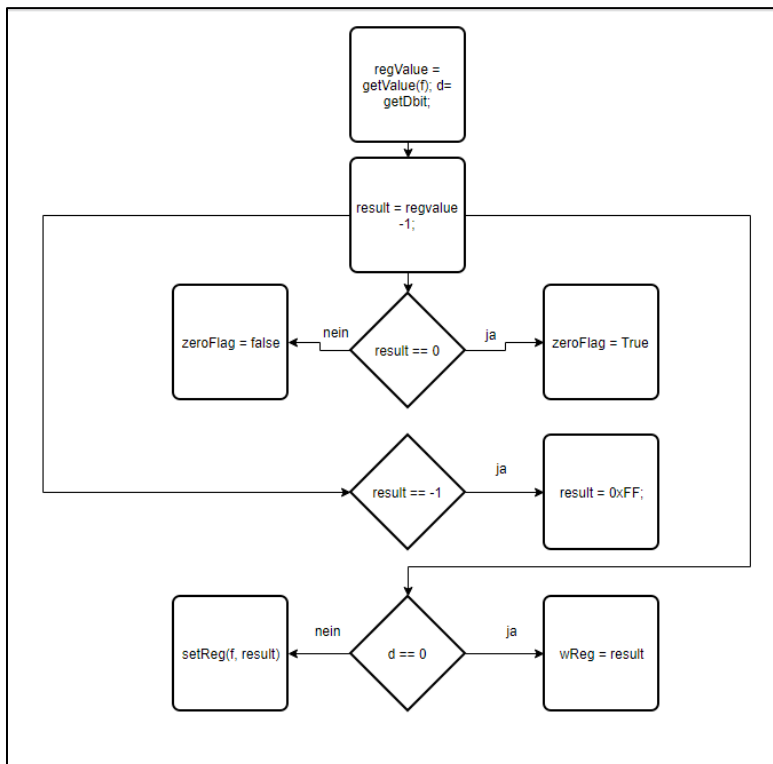


Abbildung 13: Flussdiagramm zum Befehl „DECF“

### Flussdiagramm „BTFSS“

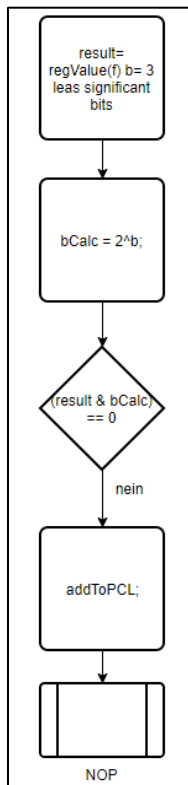


Abbildung 14: Flussdiagramm zum Befehl „BTFSS“

### Flussdiagramm „CALL“

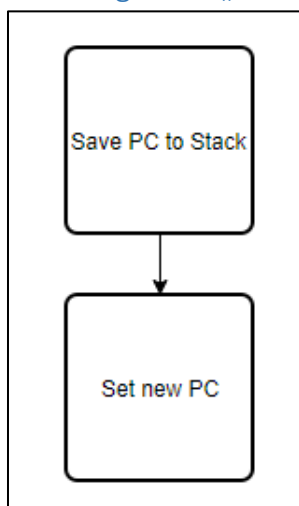


Abbildung 15: Flussdiagramm zum Befehl „CALL“

### Datenblatt PIC16F84

Ein Simulator ist ein möglichst realitätsnahes Nachbilden von Geschehen der Wirklichkeit. Um Kosten zu reduzieren und Tests zu vereinfachen, werden häufig Problemkreise von der Realität gelöst und in Teilen oder im Gesamten abstrakt in ein Modell überführt, an welchem zielgerichtet experimentiert wird. Die Resultate werden anschließend wieder auf das reale Problem übertragen.



Der PIC-Simulator hat die Aufgabe, Implementierungen in Form von Dateien zu testen und auf seiner Oberfläche auszugeben. Hierbei sollen Teile der Hard- und Software des PIC16F84-Mikrocontrollers realisiert werden, um die Anforderungen der Programme zu erfüllen.

Der PIC-Simulator soll eine einfache Benutzeroberfläche besitzen, die es dem Tester ermöglicht, auf einfache Weise die Programme zu laden und schrittweise oder im Gesamten durchzuführen. Auch eine Ausgabe von verschiedenen Bereichen der Soft- und Hardware sollen möglich sein. Hierzu zählen beispielsweise die Quarzfrequenz und die TRIS-Register A und B.