PIC-Simulator für PIC16F84A

Begleitdokumentation für Simulatorprogramm im Fach Rechnertechnik II
des Studienganges Informationstechnik
an der DHBW Karlsruhe

Ersteller:	Martin Mössner
Datum:	6. Juni 2016
Matrikelnummer:	#9541104
Kursbezeichnung DHBW:	TINF14B3
Dozent:	Stefan Lehmann

DHBW Karlsruhe
Erzbergerstraße 121
76133 Karlsruhe

Erklärung

gemäß § 5 (3) der Studien- und Prüfungsordnung DHBW Technik vom 22. September 2011.

Ich habe die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet.

Ort, Datum Unterschrift

Inhaltsverzeichnis

1. Allgemeines

1.1. Grundlegende Arbeitsweise eines Simulators

Die Aufgabenstellung sieht vor, ein Simulatorprogramm zu schreiben, das in der Lage ist, Programmlistings für den PIC16F84 von Microchip auszuführen und zu visualisieren. Im Gegensatz zu einem Emulator steht hier nicht die exakte Nachbildung des Hardwareaufbaus und der entsprechenden internen Abläufe im Vordergrund, sondern viel eher eine komfortable und benutzerfreundliche Nachahmung der generellen Funktionsweise des Mikrocontrollers.
Das Prinzip einer Simulation bietet einige Vorteile: So ist im Falle des PIC die Verwendung eines Programmiergerätes überflüssig und fertig implementierte und assemblierte Programmlistings können schnell und unkompliziert ausgeführt und getestet werden. Darüber hinaus ist es möglich, in Einzelschritten zu arbeiten und damit die Fehlersuche im Code erheblich zu vereinfachen. Hilfreich ist es dabei auch, die verwendeten Speicherzellen und betroffene Register und Flags zu jedem Zeitpunkt im Detail betrachten zu können.
Die Anbindung externer Hardware ist bei einem Simulator jedoch häufig nicht umsetzbar oder zumindest massiv erschwert. Je nach Komplexität des simulierten Systems steigt auch die Wahrscheinlichkeit, dass die Software nicht das exakte Verhalten des Chips nachahmt. Da das Betriebssystem und die zugrunde liegende Programmiersprache erheblichen Einfluss auf die Programmperformance nehmen, gestaltet sich das Einstellen bzw. das Einhalten bestimmter, präziser Taktzeiten häufig schwer.

1.2. Die Programmoberfläche des Simulators

Um dem User eine gewohnte und vertraut wirkende Oberfläche zu bieten, wurde das grafische Interface mit dem MFC-Framework realisiert. Die Verwendung von Checkboxes zur Darstellung bestimmter Speicherzellen dient einerseits der einfachen und schnellen Veränderbarkeit der Informationen durch den User und ist andererseits eine geeignete Möglichkeit, einzelne Bits anschaulich zu visualisieren. Auf diese Art und Weise werden folgende Speicherstellen veränderbar dargestellt:

- Option-Register
- Status-Register
- Intcon-Register

- PORTA
- PORTB
- TRISA
- TRISB

Für die Darstellung des Programmspeichers und des SRAM wurde die Möglichkeit einer listenähnlichen Ausgabe gewählt. Durch die Scrollbarkeit ergibt sich einerseits eine recht kompakte Ausgabe zahlreicher Datensätze, anderersetis ermöglicht MFC beispielsweise, die Zeile mit der momentan ausgeführten Instruktion des Programmlistings automatisch in das Sichtfeld des Benutzers zu scrollen.
Die Gestaltung wurde unter dem Fokus erarbeitet, durch durchdachte Beschriftung und Platzierung sämtlicher Kontrollelemente eine selbsterklärende und schnell verständliche Oberfläche zu schaffen.

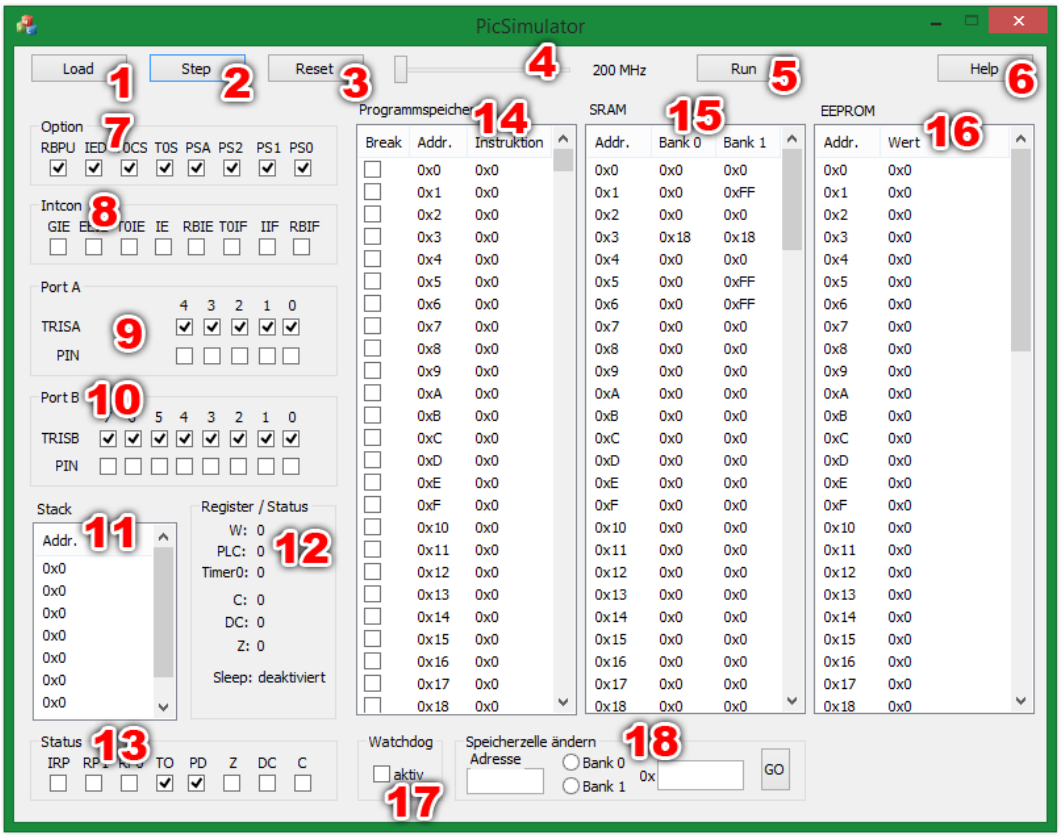

Abbildung 1: Benutzeroberfläche des PIC-Simulators

1) **Load:** Der Klick auf diesen Button öffnet einen Windows-Dialog und ermöglicht das Einlesen eines Programmlistings in den Simulator

2) **Step:** initiiert einen einzelnen Programmzyklus und führt einen Schritt aus

3) **Reset:** setzt den PIC auf die festgelegten Reset-Setting zurück. **Achtung:** Der Programmspeicher bleibt erhalten, um ein erneutes Einlesen des Listings zu vermeiden!

4) **Frequenzwahl:** Die gewünschte Taktfrequenz kann mittels des Wahlschiebers eingestellt werden, rechts daneben befindet sich die Anzeige der momentan eingestellten Taktfrequenz.

5) **Run/Stop:** Startet die automatische Abarbeitung des Listings in der gewählten Taktfrequenz, während des Laufs lässt sich mit Klick auf den Button (der dann mit "Stop" beschriftet ist) die Abarbeitung anhalten

6) **Help:** Öffnet die pdf-Datei der Dokumentation als Hilfestellung

7) **Option-Register:** Stellt mittels Checkboxen veränderbar die Flags des Option-Registers dar

8) **Intcon-Register:** Stellt mittels Checkboxen veränderbar die Flags des Intcon-Registers dar

9) **Port A:** Stellt mittels Checkboxen veränderbar den Status der Pins von Port A sowie das TRISA-Register dar, das der Konfiguration der Ports als Ein-/Ausgänge dient

10) **Port B:** Stellt mittels Checkboxen veränderbar den Status der Pins von Port B sowie das TRISB-Register dar, das der Konfiguration der Ports als Ein-/Ausgänge dient

11) **Stack:** Listet die in das 8-Bit-Stack gepushten Adressen auf, die auf ihre Abarbeitung warten

12) **Register / Status:** Stellt dem Benutzer auf einen Blick die wichtigsten Register und Flags dar, um das Debuggen zu erleichtern

13) **Status:** Stellt mittels Checkboxen veränderbar die Flags des Status-Registers dar

14) **Programmspeicher:** Listet zeilenweise die in den Programmspeicher des PIC geladenen Instruktionen und deren Adresse auf. Die Checkbox vor jeder Instruktion dient dem Setzen eines Breakpoints im Programmlisting.

15) **SRAM:** Listet zeilenweise die Inhalte des SRAM-Speichers und deren Adressen auf

16) **EEPROM:** Stellt in einer Liste die Inhalte des remanenten EEPROM-Speichers dar

17) **Watchdog:** Das Setzen des Hakens aktiviert den Watchdog-Timer

18) **Speicherzelle ändern:** Um Inhalte des SRAM-Speichers zu ändern, können die entsprechenden Kontrollelemente genutzt werden. Es können sämtliche Zellen geändert werden, sowohl auf Bank 0 als auch auf Bank 1. Die Eingabe der Adresse und des Wertes muss hexadezimal erfolgen.

2. Realisation

2.1. Beschreibung des Grundkonzepts

Die Software wurde so konzipiert, dass sie Assembler-Listings nach dem Schema der vorgelegten zwölf Testdateien lesen und verarbeiten kann. Die Software ermöglicht das Verändern sämtlicher Register und Speicherzellen im laufenden Betrieb und bietet die Möglichkeit einer variabel schnellen Ababrbeitung des Listings. Es wurde großer Wert darauf gelegt, alle relevanten Informationen übersichtlich und anschaulich darzustellen und gleichzeitig den Hardwareaufbau des PIC16F84A möglichst realitätsnah zu modellieren. Als Programmiersprache wurde C++ gewählt, Gründe hierfür sind beispielsweise die Verwendung von *unsigned*-Variablen oder eventuelle Vorteile bezüglich präziserer Taktung des Programmablaufs gegenüber der Ausführung in einer Java Virtual Machine.

2.2. Aufbau des Simulators

Am Anfang eines jeden Befehlszyklus werden zunächst die betroffenen Pins von Port A und B auf einen Flankenwechsel überprüft und im entsprechenden Fall werden die jeweiligen Interrupt-Flags gesetzt.
Deren Überprüfung erfolgt im nächsten Schritt. Sind Interrupts global aktiviert, es ist

momentan nicht die Abarbeitung einer Interrupt-Routine der Fall und ein entsprechender Interrupt-Fall ist aktiviert und eingetroffen, wird die momentan ausgeführte Instruktion noch abgearbeitet bevor die momentane Adresse des Programmzählers auf den Stack "gepusht" und als neue Adresse 0x04 festgelegt wird . Um weitere Interrupts zu verhindern, wird das GIE-Bit auf 0 gesetzt.
Im nächsten Schritt wird die an der Adresse des Programmzählers gespeicherte Instruktion ausgeführt und der momentane Status von Port A und B zwischengespeichert, um beim nächsten Befehlszyklus einen Flankenwechsel identifizieren zu können.

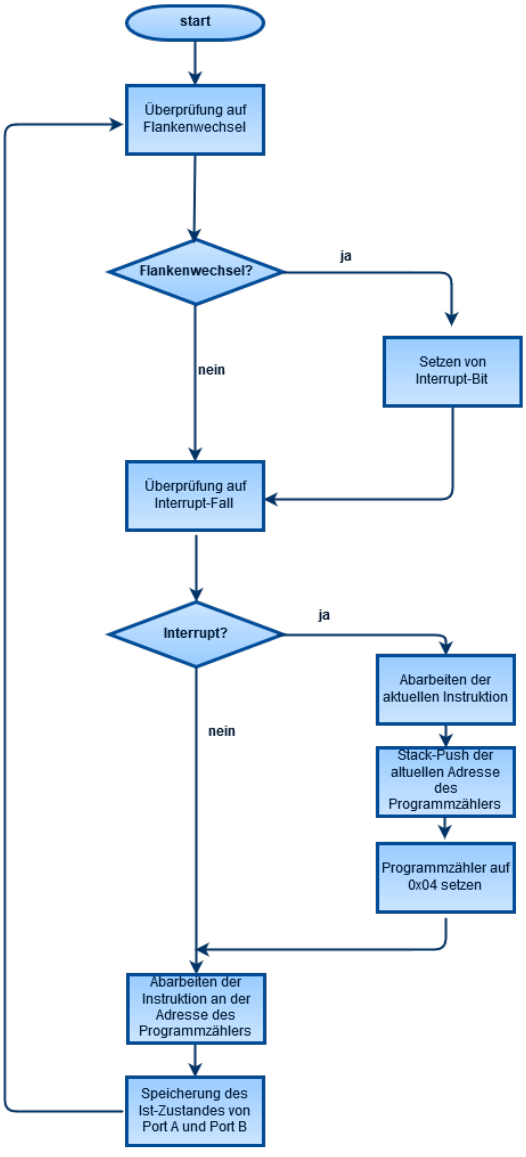

Abbildung 2: Ablaufdiagramm: Befehlszyklus

2.2.1. Befehlsabarbeitung

Die Abarbeitung eines Befehls folgt einem linearen Prinzip. Beim Blick in die Befehlstabelle fällt auf, dass der PIC verschieden lange Opcodes verwendet. Die längsten dienen den parameterlosen Befehlen und umfassen die gesamten 14 Bit der Instruktion, kürzere bestehen aus 7, 6, 5, 4 oder 3 Bit. Um zu gewährleisten, dass für jede Instruktion der passende Opcode ermittelt und die entsprechende Routine gestartet wird, muss zunächst die Übereinstimmung der Instruktion mit einem 14 Bit-Opcode überprüft werden.

```
1 //7-Bit-Opcodes
2 opcode = instruction >> 7;
3 switch(opcode)
4 {
5    case 2: //CLRW
6    {
```

Bleibt dies ergebnislos, wird das Bitmuster der Instruktion wie im Beispiel um 7 Stellen nach rechts verschoben (und von links mit Nullen aufgefüllt), um mittels eines Switch/Case-Statements die Zugehörigkeit zu einem 7 Bit-Opcode zu überprüfen. Wird hier die passende Routine nicht gefunden, wird erneut um eine Stelle nach rechts verschoben, um einen 6 Bit-Opcode zu ermitteln usw.

2.3. Beschreibung ausgewählter Funktionen

2.3.1. BTFSC

Die Funktion *BTFSC* ermöglicht dem Programmierer, in den kontinuierlichen, linearen Programmablauf eines Listings einzugreifen. Hierfür wird das angegebene Bit b der Speicherstelle f überprüft und der nachfolgende Befehl übersprungen, sofern es dem Wert 0 entspricht. Ist das Bit jedoch gesetzt, wird der nachfolgende Befehl ausgeführt.
Der Opcode für diesen Befehl ist nach dem Schema $0\ 1\ 0\ 0\ b\ b\ b\ f\ f\ f\ f\ f\ f\ f$ aufgebaut.

```
1 unsigned short bit = instruction & 896 >> 7; //get bits 7, 8 and 9
2 unsigned short addr = instruction & 127; //get last 7 bits
3 unsigned short f_reg = memoryControl.getMem(addr);
4 unsigned short mask = pow(2.0, bit);
5 if((f_reg&mask) >= 1)
6 {
7    memoryControl.setPLC(plc + 1);
8 }
9 else
```

```
10 {
11     memoryControl.setPLC(plc + 2);
12 }
13 return 2;
```

Nachdem der vorliegende Befehl also als BTFSC-Anweisung identifiziert wurde, liefert eine bitweise Verundung mit 896 (das entspricht binär *00001110000000*) die Wertigkeiten der als *b* festgelegten Stellen der Anweisung. Eine Verschiebung um 7 Stellen nach rechts ermöglicht uns, die drei Bits als Zahl von 0 bis 7 zu interpretieren.
In Zeile 2 folgt das gleiche Prinzip, um die Adresse der zu überprüfenden Speicherstelle herauszufinden. Hierbei wird als Bitmask 127 (binär *00000001111111*) verwendet, um die 7 niederwertigsten Bits als Zahl verwenden zu können. Um nun den Status eines einzelnen Bit der Speicherzelle herauszufinden, muss der Inhalt mit einer Zahl bitweise verundet werden, deren Wert $2^{\text{Bit-Stelle}}$ entspricht. Dies geschieht in den Zeilen 4 und 5. Sofern das entsprechende Bit gesetzt ist, die Verundung also (abhängig von der überprüften Stelle) einen Wert größer oder gleich 1 liefert, wird der PLC um eine Stelle erhöht und ein Überspringen des nachfolgenden Befehls verhindert. Ist das Bit jedoch nicht gesetzt, das Ergebnis der Verundung also 0, wird der PLC auf den übernächsten Befehl festgelegt.

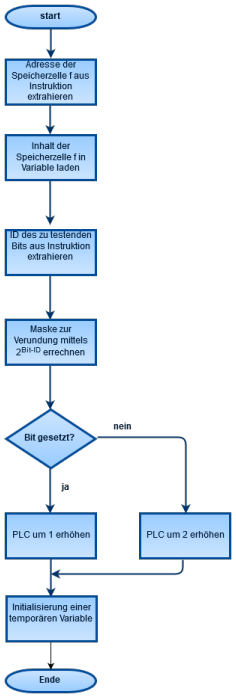

Abbildung 3: Ablaufdiagramm: BTFSC-Befehl

2.3.2. CALL

Der Befehl *CALL* ermöglicht den Aufruf einer Subroutine. Dabei wird zunächst der momentane Status des Programmzählers um einen Befehl erhöht und als Return-Adresse auf dem Stack abgelegt.

```
1  unsigned short k = instruction & 2047;
2  unsigned char PCLATH = memoryControl.getMem(10);
3  if (PCLATH & 8 >= 1)
4     k |= 1 << 12;
5  else
6     k &= ~(1 << 12);
7  if (PCLATH & 16 >= 1)
8     k |= 1 << 13;
9  else
10    k &= ~(1 << 13);
11    memoryControl.stackPush(memoryControl.getPLC() + 1);
12    memoryControl.setPLC(k);
13 return 2;
```

Die neue Adresse des Programmzählers setzt sich aus zwei Elementen zusammen: Die niederwertigen 11 Bit werden direkt aus der Programmanweisung gelesen (indem sie hier in Zeile 1 mit 2047 bzw. *00011111111111* binär verundet werden), die höherwertigen zwei Bit werden aus dem *PCLATH*-Register gelesen. Hier sind Bit 3 und 4 des Registers von Bedeutung, die mittels einer Verundung mit 8 bzw. 16 ausgelesen werden können. Sofern diese gesetzt sind, werden die entsprechenden Bits auch in der neuen Adresse gesetzt (siehe Zeilen 4 und 8). Der Befehl schließt mit dem return-Wert 2 ab, da es sich um eine 2-Cycle-Instruktion handelt.

2.3.3. MOVF

Der Befehl *MOVF* kann je nach Aufruf zwei unterschiedliche Aufgaben erfüllen. Der Opcode für diesen Befehl ist nach dem Schema *0 0 1 0 0 0 d f f f f f f f* aufgebaut. Je nachdem, ob das d-Bit gesetzt ist, wird der Wert der Speicherzelle entweder zurück in selbige oder aber in das W-Register geschrieben. Um das Bit Nummer 8 zu überprüfen, muss die Instruktion mit 128 (binär *00000010000000*) bitweise verundet werden. Das geschieht in Zeile 4.

```
1  unsigned char addr = instruction & 127; //get last 7 bits from ↩
      instruction
2  unsigned char f_reg = memoryControl.getMem(addr);
```

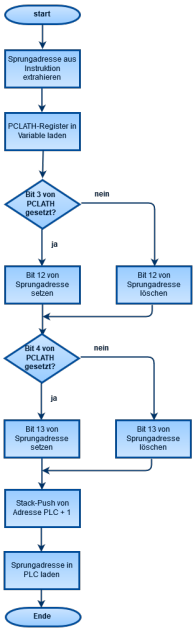

Abbildung 4: Ablaufdiagramm: CALL-Befehl

```
3  unsigned char value = f_reg;
4  if(instruction & 128) //if "d" is set
5  {
6     memoryControl.setMem(addr, value);
7  }
8  else //if "d" isnt set, result is stored in W reg
9  {
10    memoryControl.setW(value);
11 }
12 if(value == 0) memoryControl.setOption(MemoryControl::FLAG_ZERO, 1);
13 else memoryControl.setOption(MemoryControl::FLAG_ZERO, 0);
14
15 memoryControl.setPLC(plc + 1);
16 return 1;
```

In jedem Fall ist das Zero-Flag von der Operation betroffen, der Befehl ist also hilfreich um herauszufinden, ob der Inhalt einer Speicherzelle dem Wert 0 entspricht. Das Flag wird im Beispiel in den Zeilen 12 und 13 gesetzt. Der Programmzähler wird anschließend einen Befehl weiter gesetzt.

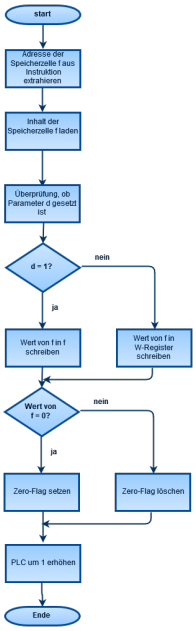

Abbildung 5: Ablaufdiagramm: MOVF-Befehl

2.3.4. RRF

Eine Möglichkeit der Manipulation einer Speicherzelle besteht darin, sämtliche Bits nach links bzw. rechts zu verschieben. Der Befehl RRF verschiebt alle Bits um eine Stelle nach rechts. Hierbei ist auch das Carry-Flag eingebunden, dessen Wert auf Bit Nummer 7 der veränderten Speicherzelle geschrieben wird. Das nach rechts "ausfallende" Bit 0 stellt den neuen Wert des Carry-Flags dar. Um Fehler im Carry-Flag (und in Folge in der

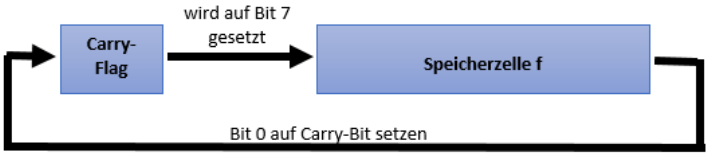

Abbildung 6: Bit-Operationen bei RRF-Funktion

Speicherzelle) zu vermeiden, muss der momentane Status des Carry-Flags zwischengespeichert werden, bevor es mit dem neuen Wert aus der Speicherzelle überschrieben wird. Dies geschieht in Zeile 3. Die eigentliche Verschiebeoperation wird mittels des "»"-Operators in Zeile 6 durchgeführt. War das Carry-Flag vor der Operation gesetzt, wird das höchstwertige Bit nun durch die Addition von 128 zum Wert der Speicherzelle

gesetzt.

```
unsigned short addr = instruction & 127; //get last 7 bits from ↩
    instruction
unsigned short f_reg = memoryControl.getMem(addr);
unsigned short carry_now = memoryControl.getOption(MemoryControl::↩
    FLAG_CARRY);
unsigned short carry_new = 0;
if(f_reg & 1) carry_new = 1;
unsigned short value = f_reg >> 1;
if(carry_now == true) value += 128;
if(instruction & 128) //if "d" is set
{
    memoryControl.setMem(addr, value);
}
else //if "d" isnt set, result is stored in W reg
{
    memoryControl.setW(value);
}
if(carry_new == 0) memoryControl.setOption(MemoryControl::FLAG_CARRY, 1)↩
    ;
else memoryControl.setOption(MemoryControl::FLAG_CARRY, 0);
memoryControl.setPLC(plc + 1);
return 1;
```

Nach Überprüfung des d-Parameters durch Verundung mit 128 wird der neue Wert in das entsprechende Zielregister geschrieben und das Carry-Flag auf den neuen Wert festgelegt.

2.3.5. SUBWF

Die Anweisung *SUBWF* zieht den Inhalt des W-Registers vom Inhalt einer angegebenen Speicherzelle ab. Anzugebende Parameter sind die Adresse der Speicherstelle f sowie das Argument d als 8. Bit der Instruktion. Hier wird festgelegt, ob das Ergebnis der Subtraktion im W-Register oder aber in der Speicherzelle f abgespeichert wird.

```
unsigned short addr = instruction & 127; //get last 7 bits from ↩
    instruction
unsigned short w_reg = memoryControl.getW();
unsigned short f_reg = memoryControl.getMem(addr);
unsigned short value = f_reg - w_reg;
if(instruction & 128) //if "d" is set
```

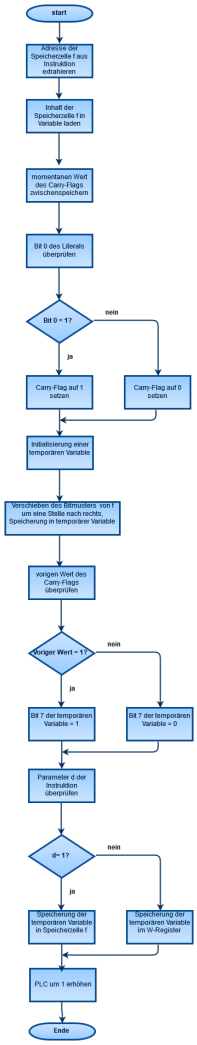

Abbildung 7: Ablaufdiagramm: RRF-Befehl

```
6  {
7      memoryControl.setMem(addr, value);
8  }
9  else //if "d" isnt set, result is stored in W reg
10 {
11     memoryControl.setW(value);
12 }
13 if(value == 0) memoryControl.setStatus(MemoryControl::FLAG_ZERO, 1);
14 else memoryControl.setStatus(MemoryControl::FLAG_ZERO, 0);
15
16 if(w_reg < f_reg) memoryControl.setStatus(MemoryControl::FLAG_CARRY, 1);
17 else memoryControl.setStatus(MemoryControl::FLAG_CARRY, 0);
18
```

```
19 if(value > 15) memoryControl.setStatus(MemoryControl::FLAG_DIGIT, 1);
20 else memoryControl.setStatus(MemoryControl::FLAG_DIGIT, 0);
21
22 memoryControl.setPLC(plc + 1);
23 return 1;
```

Das Ergebnis der Subtraktion ist ausschlaggebend für drei Status-Flags des PIC. Sowohl das Carry-Flag, als auch das Zero-Flag und das Digit Carry-Flag können durch den *SUBWF*-Befehl verändert werden. Eine Besonderheit stellt hier das Carry-Flag dar. Sofern der Wert des W-Registers kleiner als der Wert der Speicherzelle ist, wird das Carry-Flag auf 1 gesetzt; andernfalls auf den Wert 0.

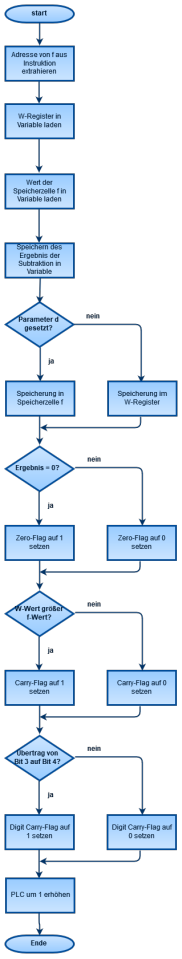

Abbildung 8: Ablaufdiagramm: SUBWF-Befehl

2.3.6. DECFSZ

Der Befehl *DECFSZ* zieht vom Wert einer Speicherzelle f den Betrag 1 ab und überprüft das Ergebnis. Sofern dieses dem Wert 0 entspricht wird das Zero-Bit gesetzt und der nachfolgende Befehl übersprungen, andernfalls wird die folgende Instruktion ausgeführt und das Zero-Bit im Status-Register auf den Wert 0 festgelegt.

```
unsigned char addr = instruction & 127; //get last 7 bits from ↩
    instruction
unsigned char f_reg = memoryControl.getMem(addr);
unsigned char value = f_reg - 1;
if(instruction & 128) //if "d" is set
{
   memoryControl.setMem(addr, value);
}
else //if "d" isnt set, result is stored in W reg
{
   memoryControl.setW(value);
}
if(value == 0) memoryControl.setStatus(MemoryControl::FLAG_ZERO, 1);
else memoryControl.setStatus(MemoryControl::FLAG_ZERO, 0);

if(value == 0) memoryControl.setPLC(plc + 2);
else memoryControl.setPLC(plc + 1);
return 2;
```

Abhängig vom als Parameter d interpretierten Bit 7 wird auch bei diesem Befehl das Ergebnis entweder zurück in die Speicherzelle oder in das Work-Register geschrieben.

2.3.7. XORLW

Der Befehlssatz des PIC bietet zwei Möglichkeiten der Verundung zweier Literale. Neben der inklusiven Verundung besteht die Möglichkeit einer exklusiven Verundung eines Literals mit dem Wert des W-Registers mittels XORLW.

```
unsigned short literal = instruction & 255; //get last 8 bits from ↩
    instruction
unsigned short w_reg = memoryControl.getW();
unsigned short value = w_reg ^ literal;
memoryControl.setW(value);
if(value == 0) memoryControl.setStatus(MemoryControl::FLAG_ZERO, 1);
else memoryControl.setStatus(MemoryControl::FLAG_ZERO, 0);
```

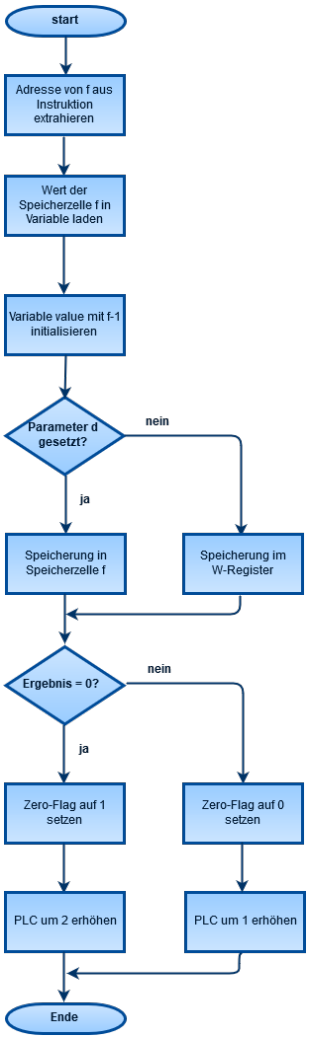

Abbildung 9: Ablaufdiagramm: DECFSZ-Befehl

```
7
8 memoryControl.setPLC(plc + 1);
9 return 1;
```

Die Sprachen der C-Familie bieten zahlreiche Operatoren zur bitweisen Manipulation. So ist eine exklusive Verundung einfach und komfortabel mit dem "^"-Operator zu erreichen. Das Ergebnis wird in einer temporären Variable gespeichert und im Work-Register abgelegt. Sofern das Ergebnis der Verundung 0 ist, wird das Zero-Flag gesetzt und der PLC um einen Wert erhöht.

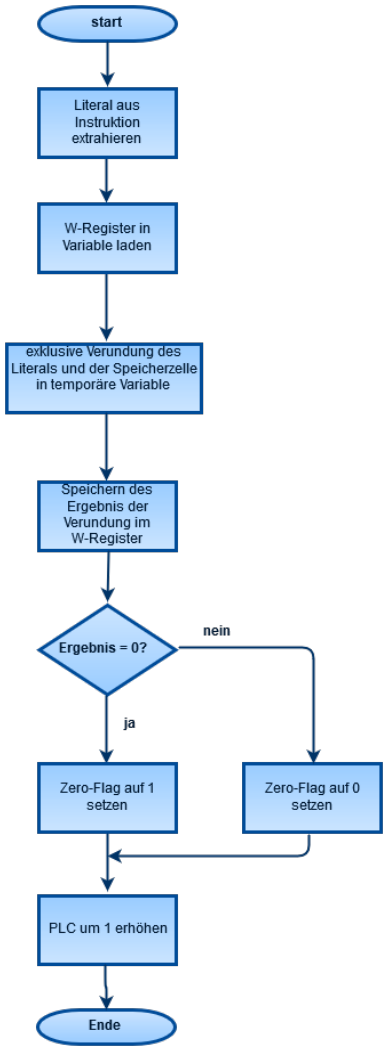

Abbildung 10: Ablaufdiagramm: XORLW-Befehl

3. Reflexion

Zusammenfassend lässt sich sagen, dass es zwar enorm schwer ist, die Funktionen eines Mikrocontrollers möglichst realitätsnah in Software zu simulieren, das Programm allerdings trotz einiger kleiner Einschnitte und (bekannter) Lücken als gelungen betrachtet werden kann.

Die Controller der PIC-Familie sind zwar hinsichtlich ihrer technischen Eigenschaften im Vergleich mit anderen Prozessoren noch von geringer Komplexität, allerdings weist die Architektur vielerorten komplex gestaltete Eigenschaften auf. Dies macht die detailgetreue Simulierung aufwendig, ist allerdings bei genauem Studium des umfangreichen Datenblatts möglich.

Hinsichtlich der gewählten Programmiersprache könnte man bei einer erneuten Umsetzung eines ähnlichen Projekts klügere Entscheidungen fällen. So bietet C zwar großartige Möglichkeiten zum möglichst taktgenauen Timen des Programmes, die Umsetzung der grafischen Oberfläche mittels des MFC-Frameworks birgt allerdings einige Schwierigkeiten. Beispielsweise bringt die Verwendung von MFC eine erhebliche Anzahl Mircosoft-proprietärer Datentypen mit sich, die sich nicht unbedingt immer untereinander oder in gängige Datentypen "typecasten" lassen. Hier wäre es unter Umständen klüger, den Kern des PIC-Simulators in C bzw. C++ umzusetzen und ihn dann mittels einer DLL-Bibliothek in ein Java-Projekt einzubinden, das sich um die Interaktion mit dem Benutzer kümmert. Eine derartige Lösung könnte auch Konflikte unter den verschiedenen Threads (die zur Bereitstellung der GUI bzw. der kontinuierlichen Abarbeitung der Scripts dienen), die bei gleichzeitigem Zugriff auf gleiche Speicherbereiche entstehen, einfacher lösbar gestalten. Auf diese Art und Weise wäre auch eine plattformunabhängige Lösung denkbar.

A. Anhangsverzeichnis

A.1. Abkürzungsverzeichnis

DLL:	Dynamic Link Library
EEIE:	EEPROM Interrupt Enable
EEPROM:	Electrically Erasebla Read Only Memory
GIE:	Global Interrupt Enable
GUI:	Graphic User Interface
IE:	External Interrupt Enable
IED:	Interrupt Edge Select
IIF:	External Interrupt-Flag
MFC:	Microsoft Foundation Classes
PIC:	Programmable Interrupt Controller
PLC:	Program Line Counter
PS0/PS1/PS2:	Prescaler 0-2
PSA:	Prescaler Assignment
RBIE:	Port B Interrupt-Enable
RBIF:	Port B Interrupt-Flag
RBPU:	Port B Pull-Up
SRAM:	Static Random Access Memory
T0CS:	Timer0 Clock-Select
T0IE:	Timer0 Interrupt-Enable
T0IF:	Timer0 Interrupt-Flag
T0S:	Timer0 Select Edge

A.2. Abbildungsverzeichnis

A.3. Quellenverzeichnis

- OMCON24, 2008: Ëmulator"
  `http://wiki.winboard.org/index.php/Emulator`
- Jörg Bredendiek, 2015: "pic: Assembler-Befehle"
  `http://www.sprut.de/electronic/pic/assemble/befehle.html`
- Microchip Technology Inc., 2013: "PIC16F84A Datasheet"
  `http://ww1.microchip.com/downloads/en/DeviceDoc/35007C.pdf`