PicSimulator Dokumentation

von Damian P. und Emil B.

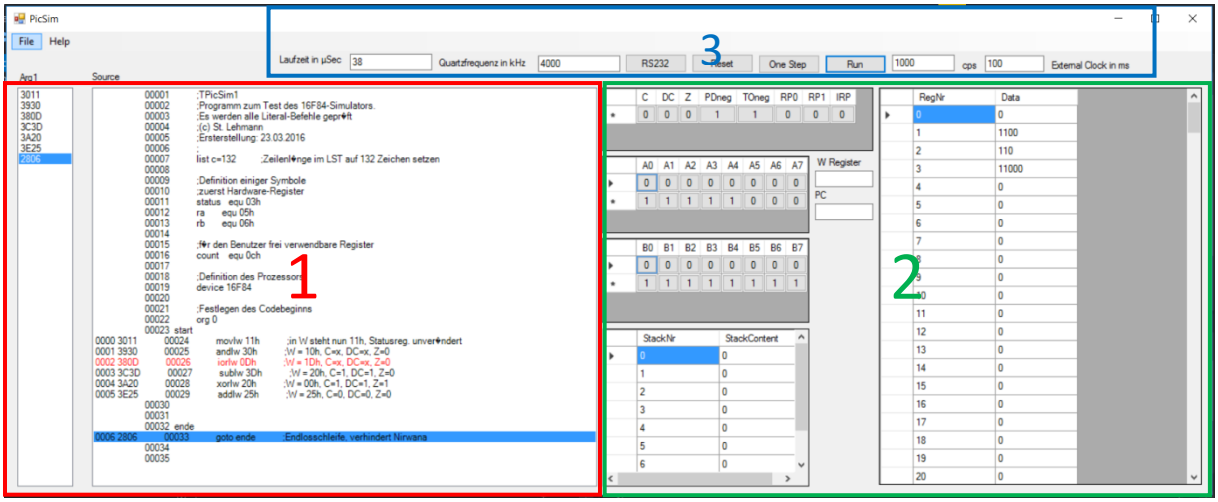
Inhaltsverzeichnis

Vorwort

Im folgenden Dokument soll genauer auf den PIC Simulator eingegangen werden. Bei dem PIC Simulator handelt es sich um einen Simulator für den Mikrocontroller PIC16F84. Dieser entstand im Rahmen eines Testates des Studienganges Informationstechnik an der DHBW Karlsruhe im Fach Rechnertechnik.

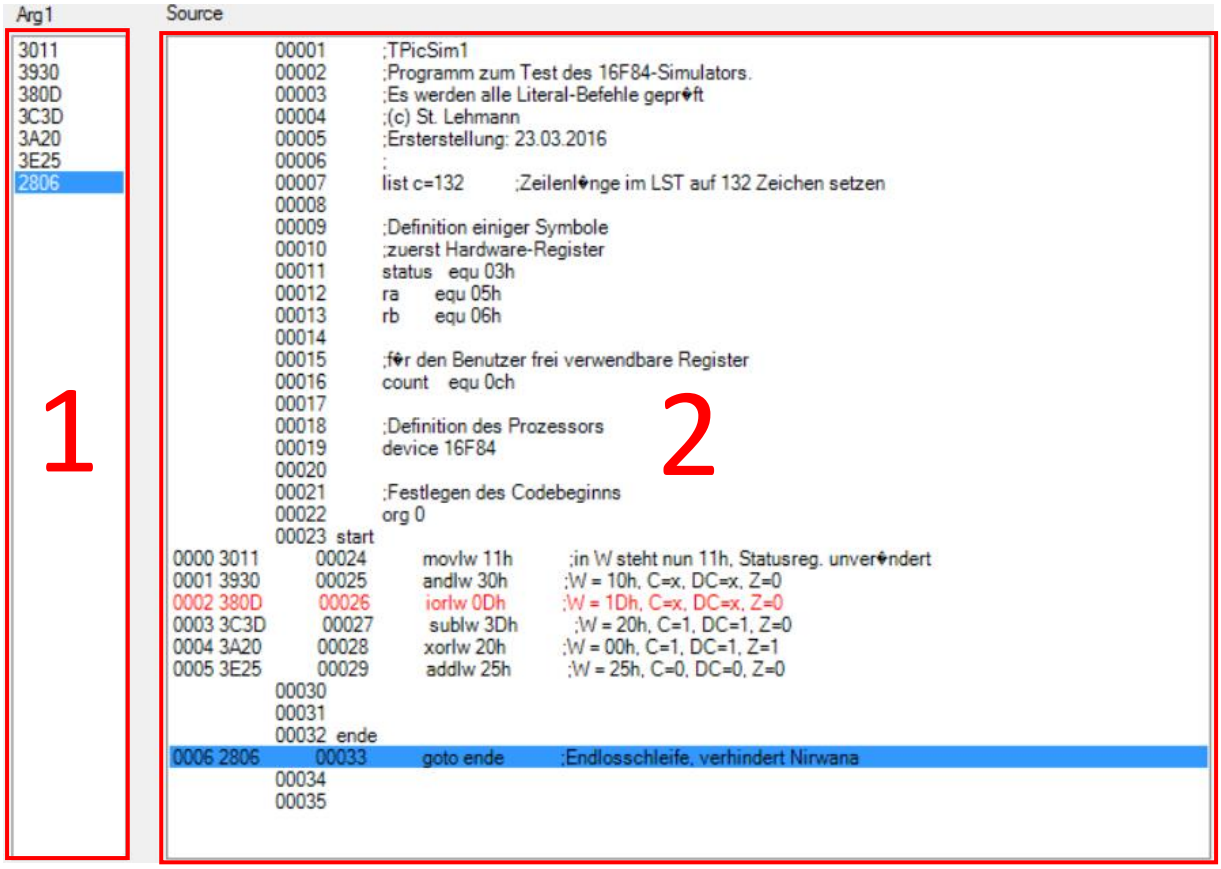
Dabei wurden die Funktionen und Befehle es Mikrocontrollers als Simulator umgesetzt. Von den verschiedenen Funktionen und Befehlen soll im Folgendem die Funktionsweise erklärt werden und der Aufbau dargestellt werden.

Benutzeroberfläche



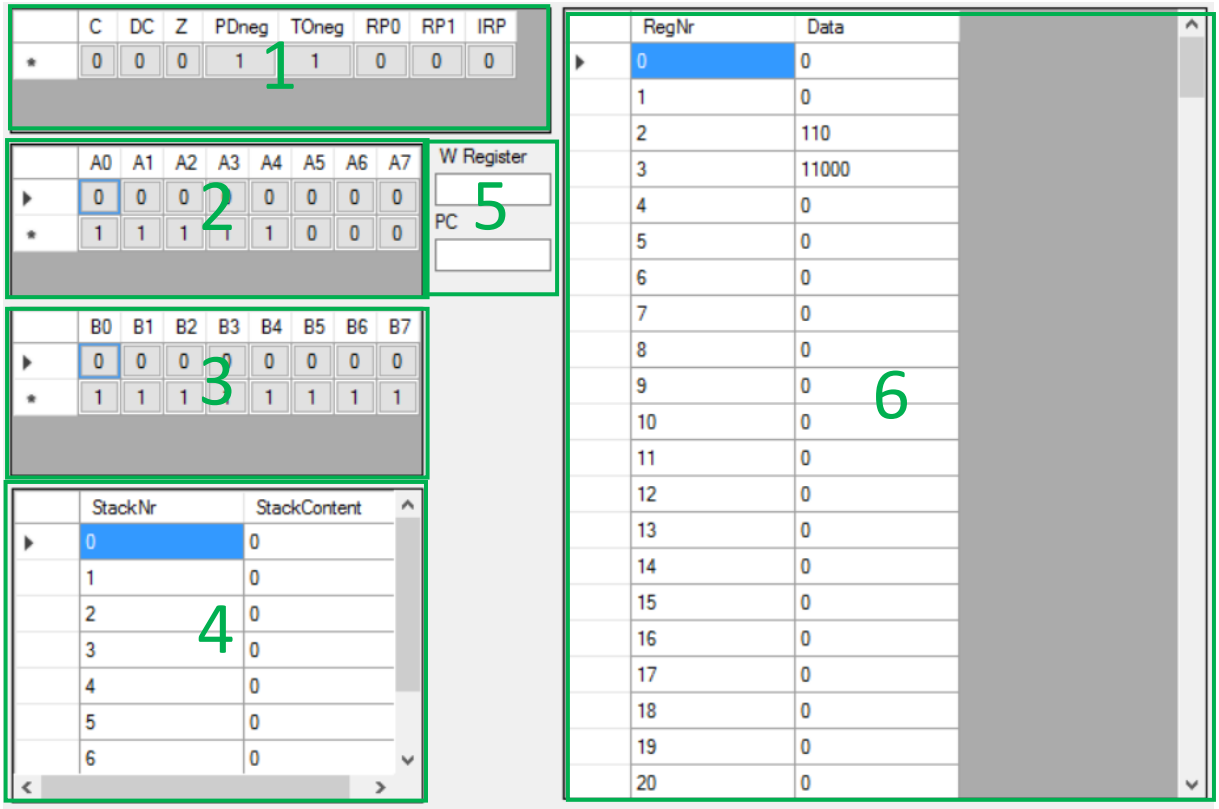
Die Benutzeroberfläche ist in 3 Bereiche unterteilt auf welche im Folgenden genauer eingegangen werden soll.

Der erste Bereich beinhaltet die Programminformationen. In 1 kann man die verschiedenen Befehle in ihrer Hexadezimalen Darstellung erkennen und in 2 ist der erweiterte Programmcode dargestellt. Außerdem erkennt man durch die blaue Markierung welcher Befehl der nächste ist welcher ausgeführt wird. Durch einen Doppelklick in 2 auf eine beliebige Zeile wird dort ein Breakpoint gesetzt, was man an der roten Einfärbung erkennen kann.



Im zweiten Bereich werden die Register und der Stack dargestellt. Im ersten Feld wird das Statusregister angezeigt und in den Feldern 1 und 2 werden Port A und B und darunter die

Passenden Tris Werte angezeigt. Die Werte von Port A und B lassen sich mit einem Klick auf diese ändern. In Feld 4 wird der Stack angezeigt mit der NR und dem passenden Inhalt. In Feld 5 werden die Werte vom W Register und des PCs dargestellt und in Feld 6 sind alle Register in Binärschreibweise vorhanden.



Im dritten Bereich sind die Interaktionsmöglichkeiten dargestellt welche für den Benutzer vorhanden sind. Im Feld 1 kann man die Quarzfrequenz festlegen und dort wird die Laufzeit basierend auf der Quarzfrequenz angezeigt. Feld 2 beinhaltet verschiedenen Buttons um ein Programm zu starten, es Schritt für Schritt durch zu gehen, es zu resetten oder die Schnittstelle RS232 anzusprechen. Im Feld 3 kann man die Geschwindigkeit einstellen mit welcher der Simulator Befehle ausführen soll und in Feld 4 kann man die Externe Clock einstellen.

Aufbau des Programmes

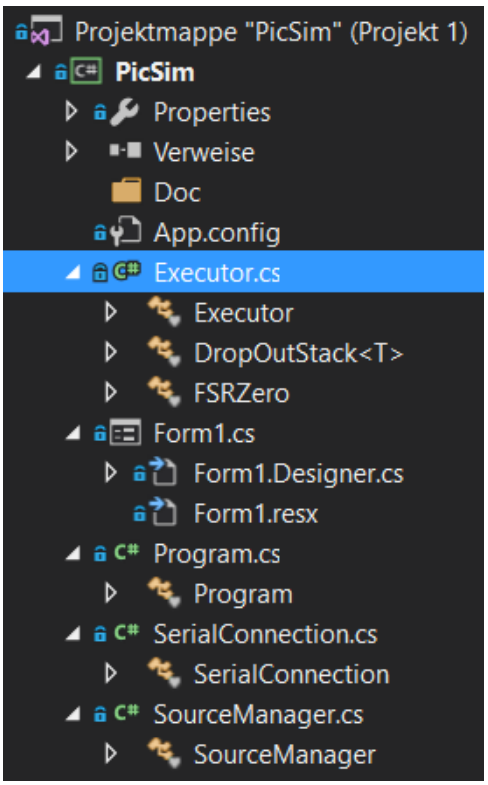
Das Programm ist in verschiedene Bestandteile unterteilt. Auf die genauere Funktionsweise soll im Anschluss eingegangen werden.

So ist die Form1 und Program Klasse dafür Zuständig die Benutzeroberfläche dar zu stellen.

Die SourceManager Klasse ist zuständig den Programmcode eines PIC Programmes ein zu lesen und diesen in ein Array um zu wandeln.

Die Executor Klasse wiederum beinhaltet die Logik des PIC Simulators, da hier alle Befehle und Routinen abgebildet sind.

Und die SerialConnection Klasse ist zuständig für die Verbindung mit der RS232 Schnittstelle und das senden/empfangen der Daten über diese.

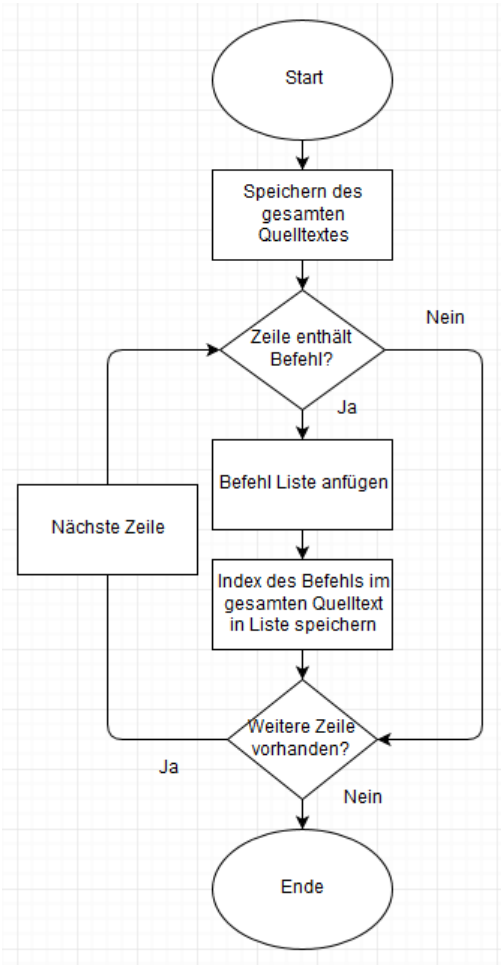


Das Programm wurde in C# in der Entwicklungsumgebung VisualStudio erstellt.

Funktionen

Im folgenden Teil der Dokumentation sollen die verschiedenen Funktionen des Simulators erklärt und aufgezeigt werden.

Code einlesen



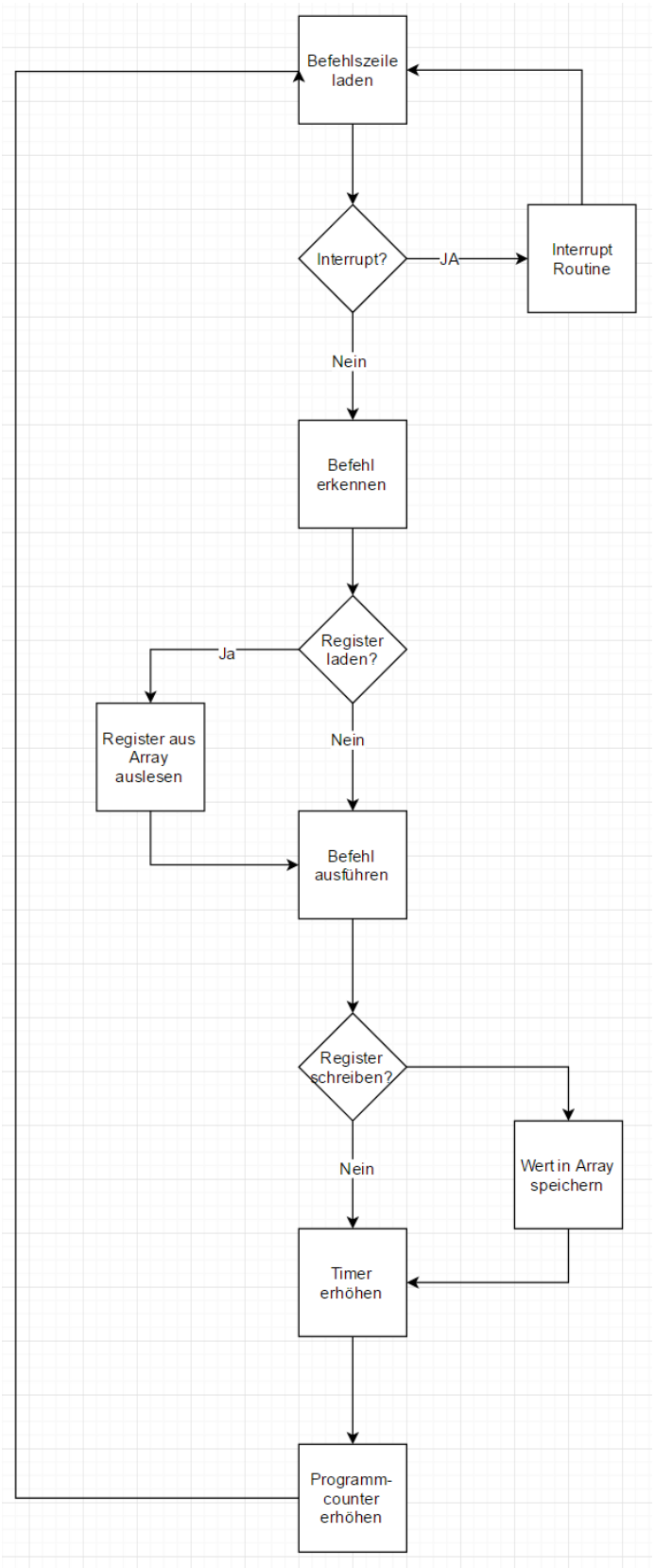
Das Einlesen und Speichern des Quellcodes wird von der Klasse SourceManager übernommen. Der gesamte Quellcode wird durch die Oberfläche (Klasse Form1) an die Methode FillSource übergeben. Die Methode Speichert dann den gesamten Quellcode inclusive Anmerkungen in einer eigenen Liste ab. Anschließend iteriert sie über jede Zeile und überprüft ob der Substring Zeichen 5 bis 9 nicht leer ist. Enthält dieser einen Inhalt wird dieser an eine weitere String Liste angefügt. Zudem wird der Aktuelle Index der Codezeile im gesamten Quelltext in einer weiteren Liste abgelegt. Anhand Dieses Index kann die Oberfläche einem über den gesamten Quelltext inclusive Kommentare vergebenen Breakpoint der Befehlsliste zuordnen. Dies ist Notwendig, damit die GUI die gerade ausgeführte Zeile korrekt im gesamten Quelltext hervorheben kann.

```
public void FillSource(List<string> l)
{
    sourceComplete = l;
    int c = 0;
    foreach (string line in sourceComplete)
    {

        string sArg1 = line.Substring(5, 4);

        if (sArg1 != "    ")
        {
            args1.Add(Convert.ToInt32(sArg1, 16));
            highlightIndex.Add(c);
        }

        c++;
    }
}
```

Befehl bearbeiten

Jeder Cycle und dadurch jeder Befehl durchläuft eine gewisse Prozedur welche bei jedem Befehl grundlegend gleich ist. Zuerst wird aus dem ProgramCounter der Aktuelle wert gelesen. Dann wird der Executor mit der passenden Programmzeile zum ProgrammCounter aufgerufen. Im Executor wird dann zuerst geprüft ob ein Interrupt stattgefunden hat und falls dies der Fall ist wird die Interrupt Routine ausgeführt. Wenn kein Interrupt stattgefunden hat wird im nächsten Schritt der Befehl erkannt. Dazu wird die aus dem Array stammende Zeile verundet und es wird geprüft um welchen Befehl es sich handelt, ein Beispiel zu ein paar Befehlen ist in der Abbildung zu sehen.

```
// Ende Interrupt============================
if ((arg & 0b1111_1111_0000_0000) == 0b0000_0111_0000_0000)
{
    Console.WriteLine("ADDWF");
    ADDWF(arg);
}
else if ((arg & 0b1111_1111_0000_0000) == 0b0000_0101_0000_0000)
{
    Console.WriteLine("ANDWF");
    ANDWF(arg);
}
else if ((arg & 0b1111_1111_1000_0000) == 0b0000_0001_1000_0000)
{
    Console.WriteLine("CLRF");
    CLRF(arg);
}
else if ((arg & 0b1111_1111_1000_0000) == 0b0000_0001_0000_0000)
{
    Console.WriteLine("CLRW");
    CLRW();
}
```

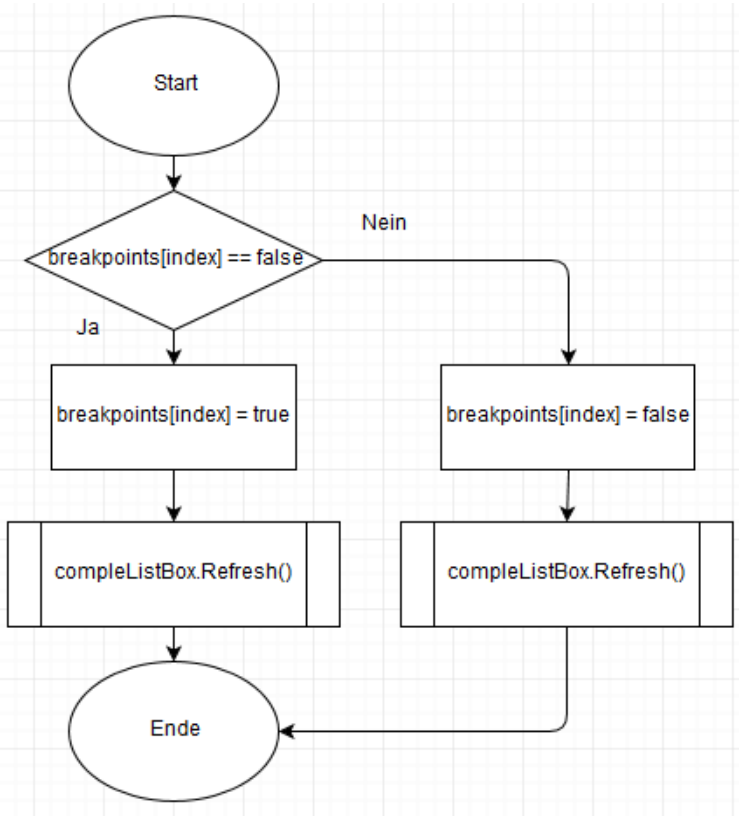


Wenn nun der Befehl erkannt wurde wird als nächstes der Befehlsspezifische Code ausgeführt welcher den Befehl darstellt. Dabei werden auch die Status Bits geprüft und eventuell gesetzt aber dies soll später bei den Befehlen erklärt werden. Wenn der Befehl das lesen oder beschreiben eines Registers beinhaltet wird dazu eine Methode aufgerufen welche anhand der Bank und der Adresse das passende Register aus dem Register Array ausliest oder beschreibt. Am Ende von jedem Befehl wird der Timer aufgerufen. Dabei übergibt der Befehl wie viele Cycels er gebraucht hat. Der Timer Checkt nun ob der Interne Timer aktiviert ist und wird dann abhängig von der Prescaler Einstellung erhöht. Nachdem der Timererhöt wurde wird der Programmcounter noch um 1 erhöt und die Routine läuft von vorne los.

Breakpoints

Breakpoints setzen

Breakpoints werden durch die Klasse Form1 behandelt. Der Status, ob eine Quellcodezeile einen Breakpoint enthält oder nicht wird in einer Bool Liste gespeichert. Ein Actionlistener der zu der Textbox mit dem gesamten Quelltext gehört wird aktiviert, wenn ein Zeilenelement per Doppelklick ausgewählt wird. Bei dem Aufruf dieser Methode ist dann der Index des Ausgewählten Elementes bekannt. Nun wird anhand der Bool Liste geprüft ob der Breakpoint aktiviert ist. Ist er dies nicht, so wird er nun in der Liste aktiviert und umgekehrt. Anschließend wird in beiden Fällen die Textbox neu gezeichnet, damit eine rote Markierung erscheint/verschwindet.




```
private void completeListBox1_DoubleClick(object sender, MouseEventArgs e)
{
    int index = this.completeListBox1.IndexFromPoint(e.Location);
    if (index != System.Windows.Forms.ListBox.NoMatches)
    {
        if (breakpoints[index] == false)
        {
            breakpoints[index] = true;
            Console.WriteLine("activated breakpoint at index: " + index);
            completeListBox1.Refresh();
        }
        else
        {
            breakpoints[index] = false;
            Console.WriteLine("deactivated breakpoint at index: " + index);
            completeListBox1.Refresh();
        }
    }
}
```

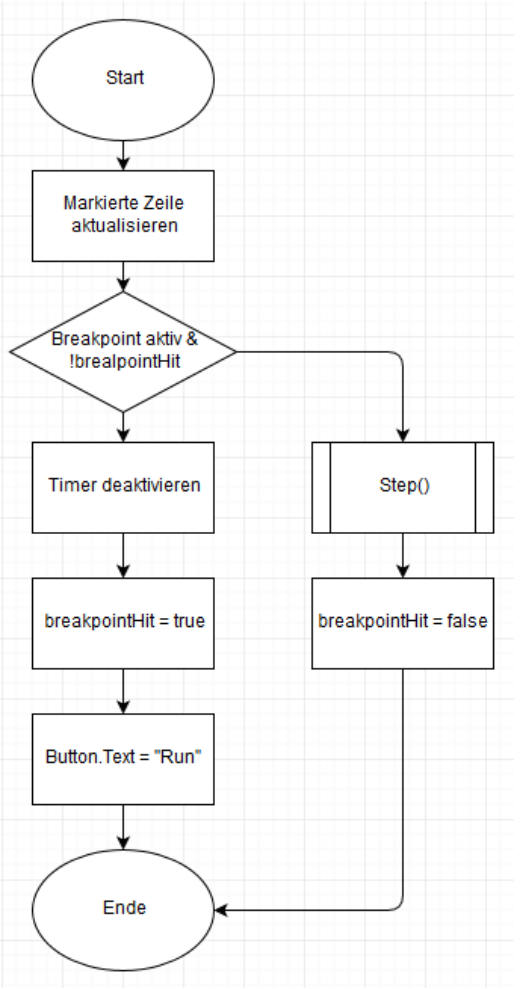
Breakpoints erkennen und Programm anhalten

Ein Timer ist dafür zuständig das geladene Programm automatisch laufen zu lassen. Bei jedem Tick wird anhand des des Programmcounters der aktuelle Index im gesamten Quellcode ermittelt und Hervorgehoben.

Anschließend wird geprüft, ob diese Codezeile einen aktivierten Breakpoint hat und ob direkt vorher ein Breakpoint erreicht wurde. Letzteres dient dazu, damit das Programm weiter läuft da es sonst direkt wieder auf selbigem Breakpoint stoppen würde.

Sollte ein Breakpoint erreicht worden sein, wird der Timer gestoppt, breakpointHit auf true gesetzt und der Text des Entsprechenden Buttons auf „Run“ geändert.

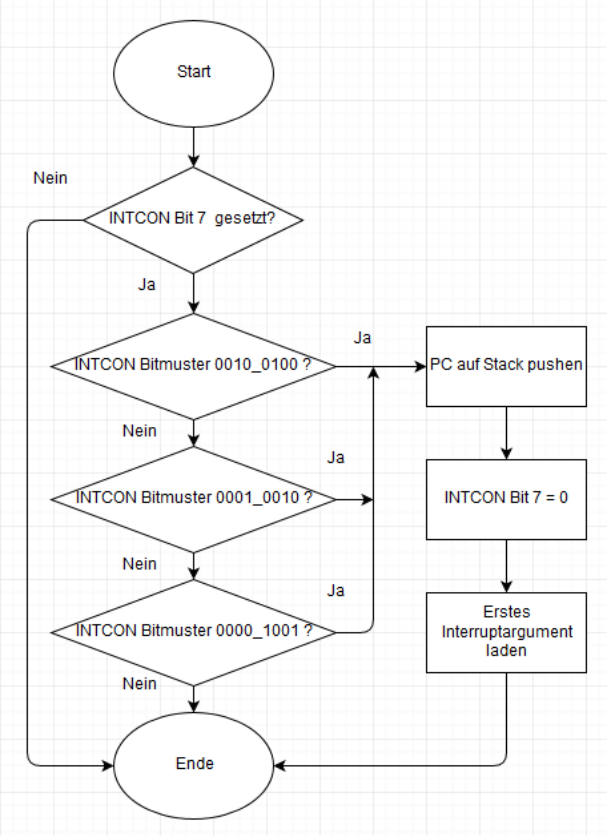
Sollte kein Breakpoint erreicht worden sein, wird der nächste Programmbefehl mit der Methode Step() ausgeführt und breakpointHit auf false gesetzt.




```
private void timerRun_Tick(object sender, EventArgs e)
{
    completeListBox1.SelectedIndex = sourceManager.getIndexInCode(executor.GetPc());
    if (breakpoints[completeListBox1.SelectedIndex] & !breakpointHit)
    {
        timerRun.Enabled = false;
        breakpointHit = true;
        btRun.Text = "Run";
    }
    else
    {
        step();
        breakpointHit = false;
    }

}
```

Interrupts

Bevor ein Befehl ausgeführt wird, wird immer überprüft, ob ein Interrupt aus zu führen ist. Hierzu wird zuerst geprüft, ob das globale Interrupt Bit im INTCON Register gesetzt wurde. Sollte dies der Fall sein so wird weiterführend geprüft, ob im INTCON Register ein Bitmuster steht, welches die Ausführung eines Interrupts erfordert. Wenn die Ausführung eines Interrupts notwendig ist, so wird der aktuelle Programm Counter auf den Stack gepusht. Danach wird der Programm Counter auf 4 gesetzt, das siebte Bit im INTCON Register gelöscht und der zum Interrupt gehörende erste Befehl für die Spätere Ausführung abgespeichert. Anschließend läuft der Executer mit diesen neuen Daten wie gewohnt weiter.s

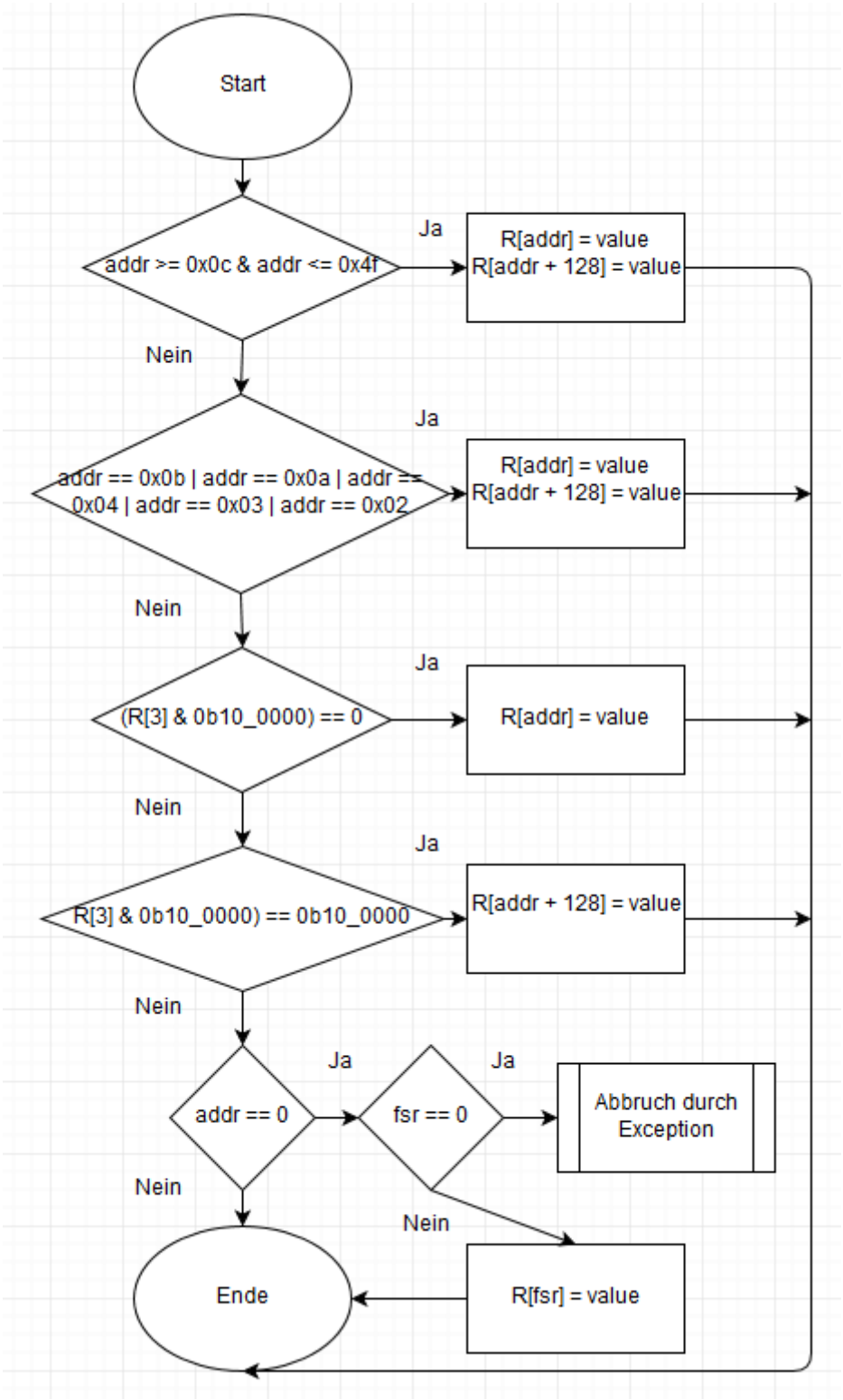


```
int INTCON = readRegister(0x0b);
if ((INTCON & 0b1000_0000) == 0b1000_0000)//Global interrupt bit
{
    if ((INTCON & 0b0010_0100) == 0b0010_0100)
    {
        //Timer interrupt
        Stack.Push(pc);
        pc = 4;
        INTCON &= ~(1 << 7);
        writeRegister(0x0b, INTCON);
        arg = intArg;
        Console.WriteLine("Timer interrupt");
    }
    else if ((INTCON & 0b0001_0010) == 0b0001_0010)
    {
        Stack.Push(pc);
        pc = 4;
        INTCON &= ~(1 << 7);
        writeRegister(0x0b, INTCON);
        arg = intArg;
        Console.WriteLine("RB0 Interrupt");
    }
    else if ((INTCON & 0b0000_1001) == 0b0000_1001)
    {
        Stack.Push(pc);
        pc = 4;
        INTCON &= ~(1 << 7);
        writeRegister(0x0b, INTCON);
        arg = intArg;
        Console.WriteLine("RB Port");
    }

}
```

Register

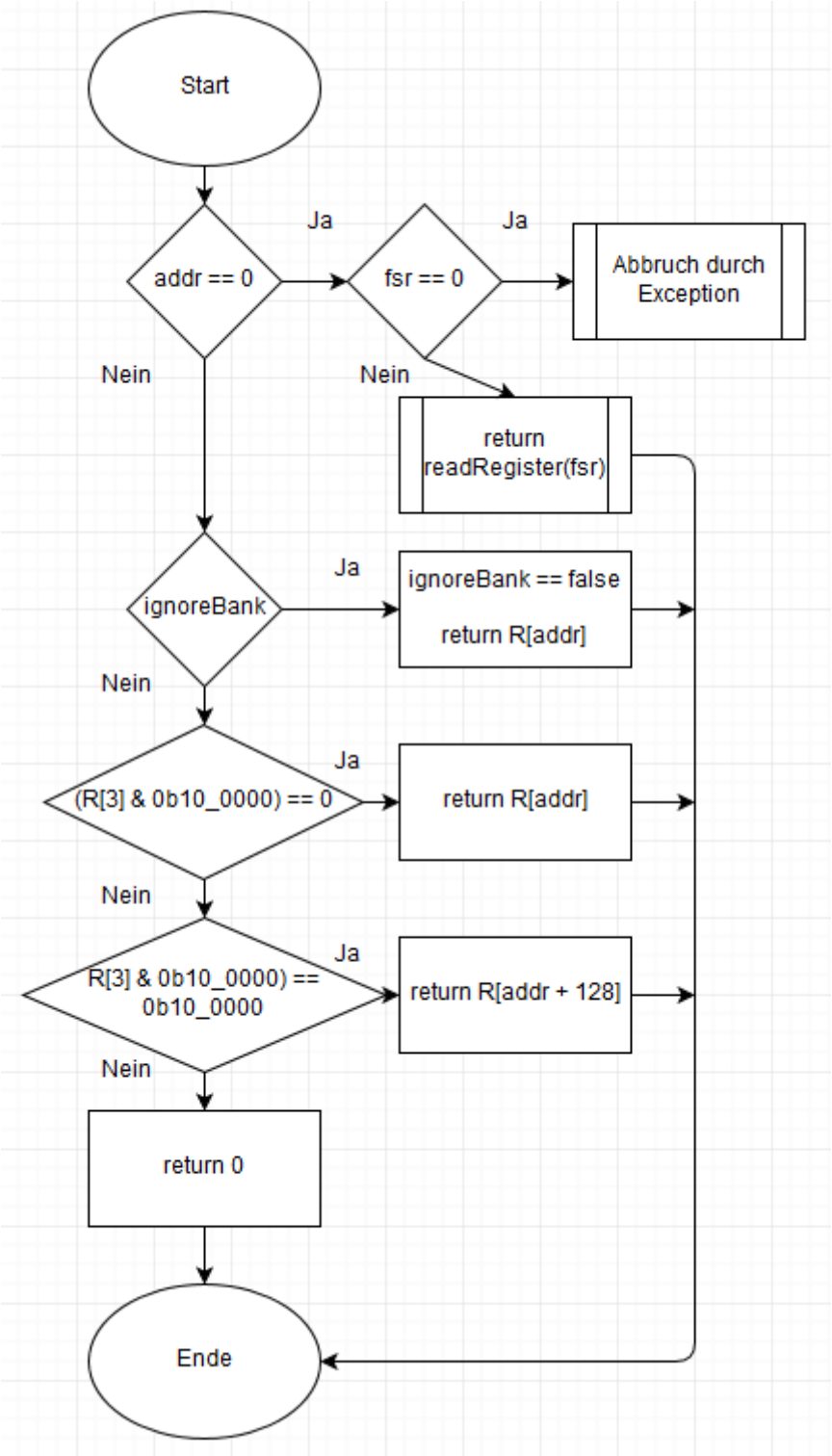
Die gesamten Register werden im Executer mittels eines 255 Stellen langem Integer Arrays R[] realisiert. Da durch Spezialfälle wie z.B. die Indirekte Adressierung und das Wechseln von Bänken nicht jeder Zugriff tatsächlich auf der angeforderten Adresse erfolgen soll wurden diese Ausnahmen in den Methoden readRegister(int addr) und writeRegister(int addr, int value) um gesetzt.

writeRegister



```
private void writeRegister(int addr, int value)
{
    if ((addr >= 0x50 & addr <= 0x7f) | (addr >= 0xd0 & addr <= 0xff))
    {
        //Hier wird nichts gemacht
    } else if (addr >= 0x0c & addr <= 0x4f)
    {
        R[addr] = value;
        R[addr + 128] = value;
    } else if (addr == 0x0b | addr == 0x0a | addr == 0x04 | addr == 0x03 | addr == 0x02)
    {
        R[addr] = value;
        R[addr + 128] = value;
    } else
    {
        if ((R[3] & 0b10_0000) == 0)
        {
            R[addr] = value;
        }
        else if ((R[3] & 0b10_0000) == 0b10_0000)
        {
            R[addr + 128] = value;
        }
    }
    if (addr == 0)
    {
        int fsr = readRegister(0x04);
        if (fsr == 0)
        {
            throw new FSRZero();
        }
        else { R[fsr] = value; }
    }
}
```

readRegister



```
private int readRegister(int addr)
{
    if (addr == 0)
    {
        int fsr = readRegister(0x04);
        if (fsr == 0)
        {
            throw new FSRZero();
        }
        else { return readRegister(fsr); }
    }
    if (ignoreBank)
    {
        ignoreBank = false;
        return R[addr];
    }
    if ((R[3] & 0b10_0000) == 0)
    {
        return R[addr];
    }
    else if ((R[3] & 0b10_0000) == 0b10_0000)
    {
        return R[addr + 128];
    }
    return 0;
}
```

Befehle

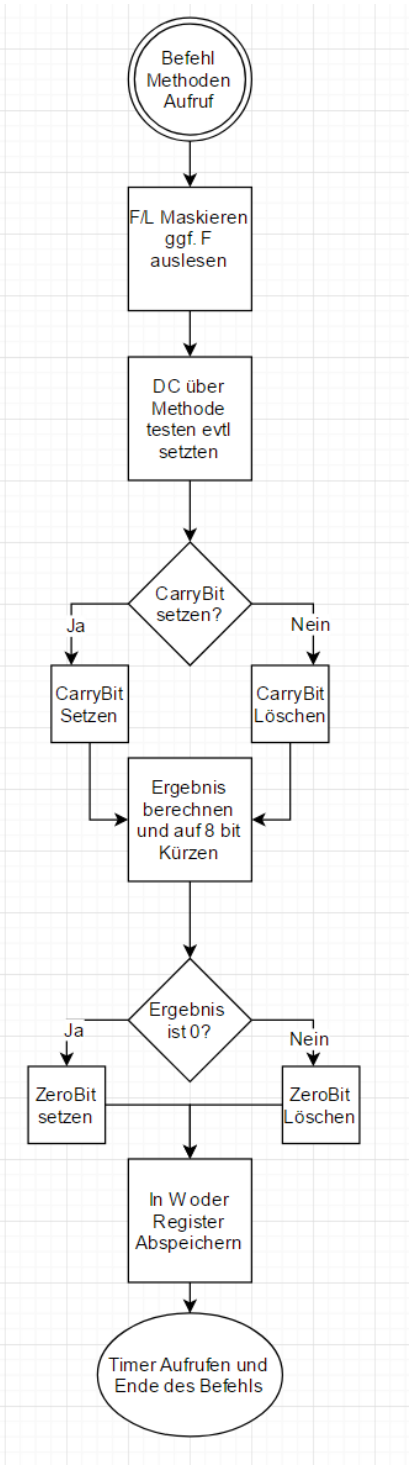
Im diesem Teil der Dokumentation soll genauer auf die verschiedenen implementierten Befehle eingegangen werden. Dabei soll sowohl auf den Ablauf der Befehle als auch auf die Implementierung geachtet werden.

ADDLW/SUBLW/ADDWF/SUBWF

ADDLW, SUBLW und ADDWF, SUBWF sind sehr ähnlich aufgebaut, da es sich jeweils um eine einfache Rechenoperation handelt, welche jeweils alle drei StatusBits beeinflussen können. Zuerst wird aus dem Befehlscode die Zahl (L) oder das Register (F) heraus maskiert. Anschließend wird, wenn nötig der Wert von F aus dem Array, durch die Funktion readRegister,

```
private int readRegister(int addr)
{
    if (addr == 0)
    {
        int fsr = readRegister(0x04);
        if (fsr == 0)
        {
            throw new FSRZero();
        }
        else { return readRegister(fsr); }
    }
    if (ignoreBank)
    {
        ignoreBank = false;
        return R[addr];
    }
    if ((R[3] & 0b10_0000) == 0)
    {
        return R[addr];
    }
    else if ((R[3] & 0b10_0000) == 0b10_0000)
    {
        return R[addr + 128];
    }
    return 0;
}
```

ausgelesen. readRegister ist eine eigene Methode auf die mehrere Befehle zugreifen. In dieser Methode werden Ausnahmen wie die Indirekte Adressierung und die Bank berücksichtigt. Sobald L oder F ausgelesen wurden wird der aktuelle Befehl auf das DigitalCarry geprüft. Dazu werden einer Methode die beiden Operanten der Berechnung übergeben und diese prüft darauf ob das DigitalCarry gesetzt werden soll. Anschließend wird geprüft ob das CarryBit gesetzt oder gelöscht werden muss, wobei das eigentliche setzten oder löschen über eine andere Methode stattfindet. Darauf wird das Ergebnis in eine Variable gespeichert und es wird durch



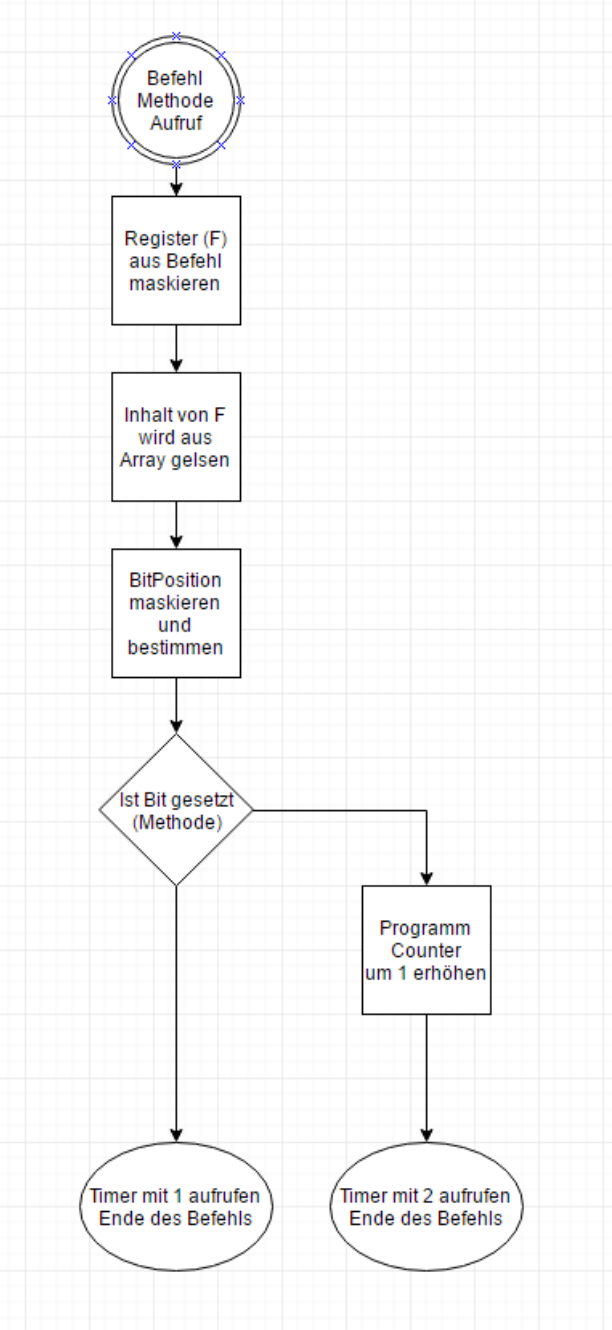
eine weitere Funktion auf ein Byte gekürzt, dass die PIC Register jeweils ein Byte speichern können. Danach wird das Ergebnis der Funktion einer Methode übergeben, welche überprüft ob das Ergebnis null ist und wenn dies der Fall ist das ZeroBit setzt bzw. löscht. Abschließend wird, wenn nötig, geprüft in welches Register das Ergebnis gespeichert werden soll und die Timerfunktion wird mit den passenden Cycle werten aufgerufen

```
private void ADDWF(int arg)
{
    int regaddr = 0b0111_1111 & arg;
    int erg = readRegister(regaddr);
    DigitalCarryPlus(W, erg);
    if((erg + W) > 255)
    {
        SetCarryBit(1);
    }
    else
    {
        SetCarryBit(0);
    }
    erg = erg + W;
    erg = Cut8(erg, false);
    ZeroBit(erg);
    if ((0b1000_0000 & arg) == 128)
    {
        writeRegister(regaddr, erg);
    } else
    {
        W = erg;
    }
    IncTimer(1);
}
```

BTFSC/BTFSS

Die Methoden der Befehle BTFSC und BTFSS beginnen mit dem Maskieren des Registers (F) und dem Auslesen des Wertes aus dem Array zu F. Anschließend wird die Bit Position aus dem Befehl heraus maskiert und bestimmt. Mit der Information des Wertes von F und der Bit Position wird eine Methode aufgerufen, welche prüft ob das Bit an der Position gesetzt ist. Dann wird abhängig vom Ergebnis der ProgrammCounter um eins erhöht und der Timer mit dem Wert zwei aufgerufen oder nur der Timer mit dem Wert eins aufgerufen.

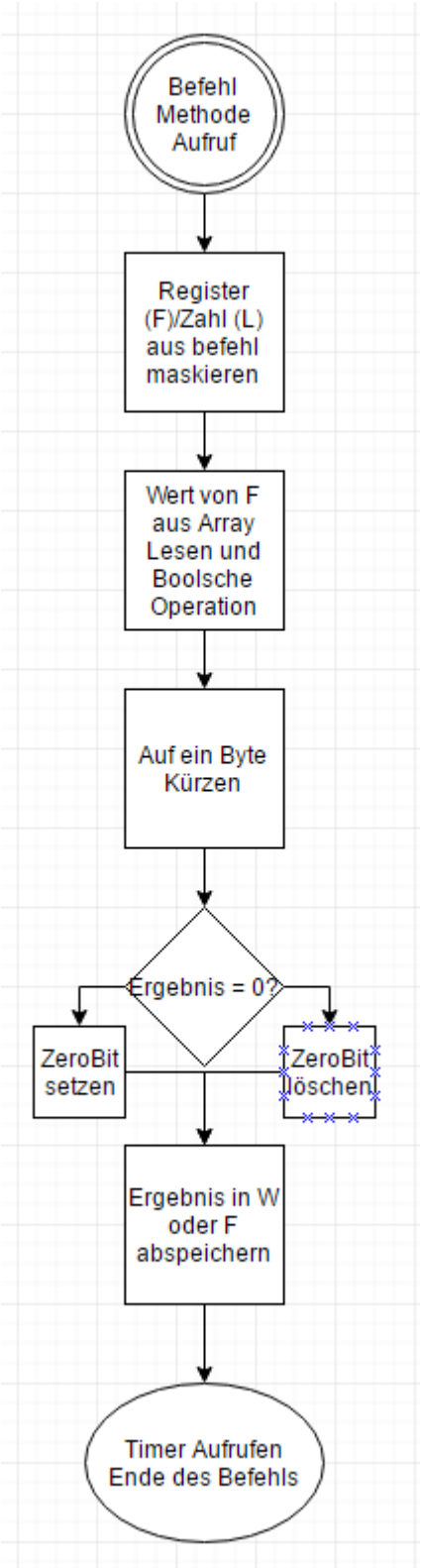
```
private void BTFSC(int arg)
{
    int regaddr = 0b0111_1111 & arg;
    int erg = readRegister(regaddr);
    int bitPosition = 0b0011_1000_0000 & arg;
    bitPosition = bitPosition >> 7;
    if (!IsBitSet(erg, bitPosition))
    {
        pc++;
        IncTimer(2);
    }
    else
    {
        IncTimer(1);
    }
}
```



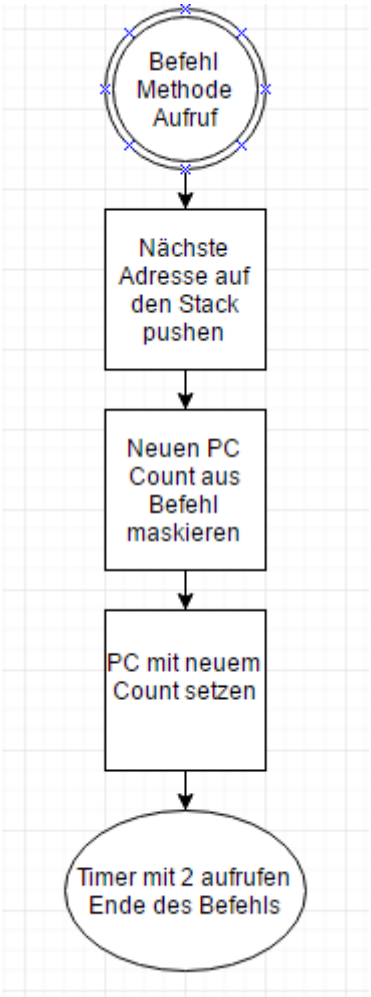
ANDWF/ANDLW / IORWF/IORLW / XORWF/XORLW

Zuerst wird das Register (F) bzw. die Zahl (L) aus. Daraufhin wird die Boolsche Operation mit W und dem Register bzw der Zahl ausgeführt. Um die Länge von einem Byte zu sichern wird danach das Ergebnis auf acht Bit gekürzt. Anschließend wird das Ergebnis der Operation darauf geprüft ob es 0 ist und abhängig davon wird das ZeroBit gesetzt bzw. gelöscht. Abschließend wird geprüft ob wohin das Ergebnis gespeichert werden soll und die Timer Methode wird mit dem Wert Eins aufgerufen.

```
private void ANDWF(int arg)
{
    int regaddr = 0b0111_1111 & arg;
    int erg = readRegister(regaddr) & W;
    erg = Cut8(erg, false);
    ZeroBit(erg);
    if ((0b1000_0000 & arg) == 128)
    {
        writeRegister(regaddr, erg);
    }
    else
    {
        W = erg;
    }
    IncTimer(1);
}
```



CALL/GOTO



Der GOTO und der CALL Befehl sind sehr ähnlich. Bei beiden wird zuerst der neue Wert für den ProgramCounter aus dem Befehl maskiert. Bei der CALL Funktion wird jedoch zuerst der nächste ProgrammCounter Wert auf den Stack gepusht. Dazu wird eine Methode verwendet, welche einen Stack darstellt welcher die Funktionen Pushen und Pullen beinhaltet. Anschließend wird der ProgramCounter auf den vorher ermittelten Wert minus Eins gesetzt, da er noch einmal am Ende eines jeden Befehles erhöht wird. Abschließend wird die Timer Methode mit dem Wert Zwei aufgerufen.

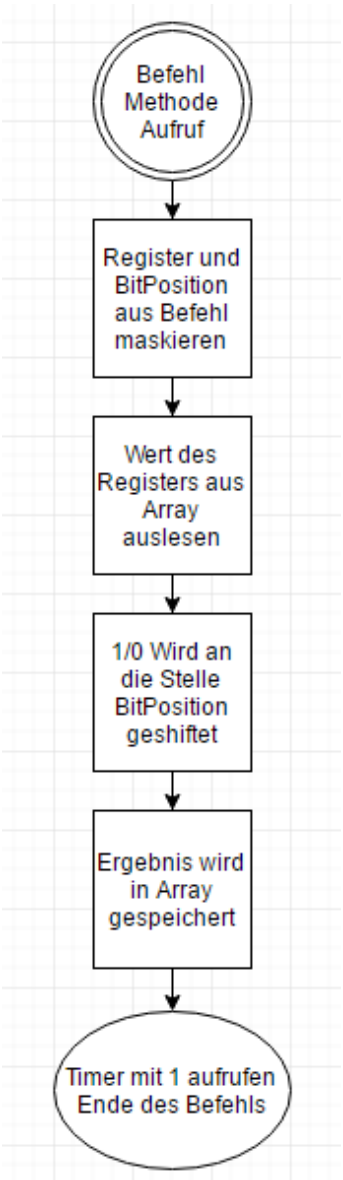
```
private void CALL(int arg)
{
    Stack.Push(pc + 1);
    int addr = 0b0111_1111 & arg;
    pc = addr - 1;
    IncTimer(2);
}

private void GOTO(int arg)
{
    int addr = 0b0111_1111 & arg;
    pc = addr - 1;
    IncTimer(2);
}
```

Bei einem RETURN wird der Eintrag aus dem Stack gepullt. Daraufhin wird der ProgramCounter mit diesem Wert minus Eins überschrieben und der Timer wird mit dem Wert Zwei aufgerufen. Bei einem RETFIE wird zusätzlich noch das Globale Interrupt Enable Bit im INTCON Register gesetzt, bei einem RETLW wird die Zahl aus dem Befehl maskiert und anschließend in W gespeichert.

```
private void RETURN()
{
    pc = Stack.Pop() - 1;
    IncTimer(2);
}
```

BCF/BSF

Zuerst wird die BitPosition und die Register(F) Adresse aus dem Befehl maskiert und in Variablen gespeichert. Anschließend wird der Wert des Registers ausgelesen und die BitPosition wird aus dem Gespeichertem Wert bestimmt. Danach wird eine Eins oder eine Null an die Bit Stelle des Wertes von F geshiftet, welche durch die BitPosition gegeben ist. Abschließend wird das Ergebnis wieder in das Register Array geschrieben und die Timer Methode wird mit dem Wert Eins aufgerufen.

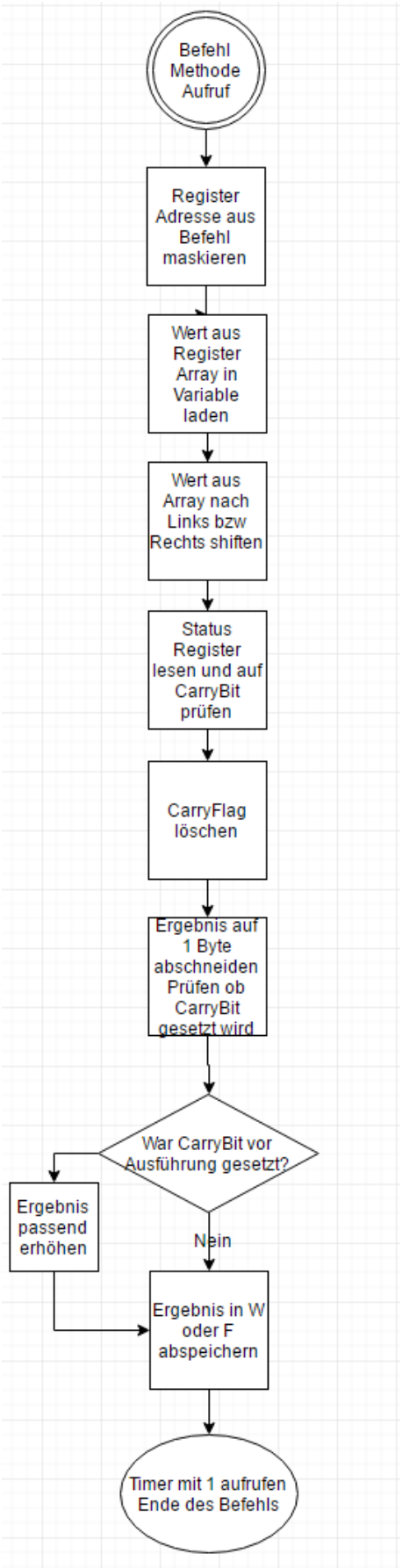



```
private void BCF(int arg)
{
    int regaddr = 0b0111_1111 & arg;
    int bitPosition = 0b0011_1000_0000 & arg;
    int erg = readRegister(regaddr);
    bitPosition = bitPosition >> 7;
    erg &= ~(1 << bitPosition);
    writeRegister(regaddr, erg);
    IncTimer(1);
}

private void BSF(int arg)
{
    int regaddr = 0b0111_1111 & arg;
    int bitPosition = 0b0011_1000_0000 & arg;
    int erg = readRegister(regaddr);
    bitPosition = bitPosition >> 7;
    erg |= 1 << bitPosition;
    writeRegister(regaddr, erg);
    IncTimer(1);
}
```

RLF/RRF

RLF und RRF beginnen damit, dass die Register (F) Adresse aus dem Befehl maskiert wird. Daraufhin wird der Wert von F aus dem Array in eine Variable geladen. Danach wird dieser Wert nach rechts bzw. nach links geshiftet und es wird geprüft ob das CarryBit gesetzt ist. Dazu muss jedoch zuerst das OPTION Register aus dem Array gelesen werden. Die Information über das CarryBit wird gespeichert und es wird auf Null gesetzt. Anschließend wird das Ergebnis auf ein Byte gekürzt und es wird geprüft ob das CarryBit neu gesetzt werden muss. Daraufhin wird abgefragt ob das CarryBit vor dieser Operation gesetzt war und gegebenen Falls wird das Ergebnis so modfiziert, als wäre eine Eins rein geshiftet worden. Zum Schluss wird geprüft ob das Ergebnis in W oder in F gespeichert werden soll und danach wird die Timer Methode mit dem Wert Eins aufgerufen.

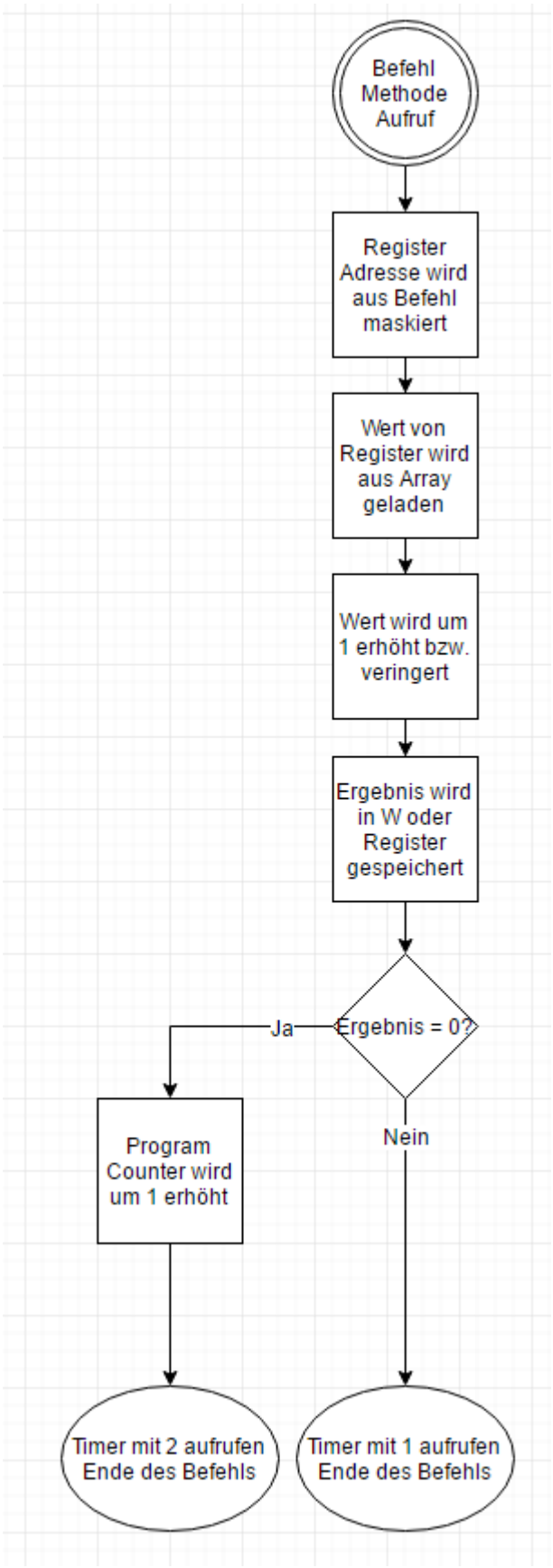
```
private void RLF(int arg)
{
    int regaddr = 0b0111_1111 & arg;
    int erg = readRegister(regaddr);
    erg = erg << 1;
    int statusreg = readRegister(0x03);
    statusreg = statusreg & 1;
    SetCarryBit(0);
    erg = Cut8(erg, true);
    if (statusreg == 1)
    {
        erg++;

    }
    if ((0b1000_0000 & arg) == 128)
    {
        writeRegister(regaddr, erg);
    }
    else
    {
        W = erg;
    }
    IncTimer(1);
}
```



DECFSZ/INCFSZ

Am Anfang des Befehls wird die Registeradresse aus dem Befehl maskiert und der Wert des Registers (F) aus dem Array in eine Variable gespeichert. Anschließend wird dieser Wert um Eins erhöht oder verringert. Dann wird geprüft ob das Ergebnis in W oder in F gespeichert werden soll. An diesem Punkt würden die Befehle INCF bzw. DECF den Timer aufrufen und wären fertig. Bei DECFSC und INCFSC wird jedoch noch geprüft ob das Ergebnis Null ist. Wenn dies der Fall ist wird der ProgramCounter um Eins erhöht und der Timer wird mit dem Wert Zwei aufgerufen. Falls nicht wird lediglich der Timer mit dem Wert Eins aufgerufen.

```
private void DECFSZ(int arg)
{
    int regaddr = 0b0111_1111 & arg;
    int erg = readRegister(regaddr);
    erg--;

    if ((0b1000_0000 & arg) == 128)
    {
        writeRegister(regaddr, erg);
    }
    else
    {
        W = erg;
    }
    if (erg == 0)
    {
        pc++;
        IncTimer(2);
    }
    else
    {
        IncTimer(1);
    }

}
```



Fazit

Die Arbeit am PIC Simulator war sehr vordernt und interessant. Durch die Arbeit hat sich unser Verständnis für die Funktionsweise eines Micro Controllers erheblich verbessert. Das Gesetzte Ziel alle Funktionen des Micro Controllers dar zu stellen konnte dabei bis auf wenige ausnahmen fast vollständig erreicht werden. Das Projekt kann somit als Erfolg gewertet werden.