Alexander Sinicyn und Harm-Christian Schweizer

PIC Simulator

Erstellung eines PIC Simulators mit JavaScript

Inhaltsverzeichnis

Vorwort

Im vierten Semester des Studiengangs Informationstechnik an der DHBW Karlsruhe sollte ein Simulator des in der Vorlesung behandelten Microcontrollers PIC 16F84 programmiert werden. Es sollen die wichtigsten Funktionen des PICs simuliert werden. Während des Programmablaufs werden alle wichtigen Speicherbänke, Register und Ports auf einer grafischen Benutzeroberfläche dargestellt. Die Arbeit erfolgte in zweier Teams.

Entwicklungsumgebung

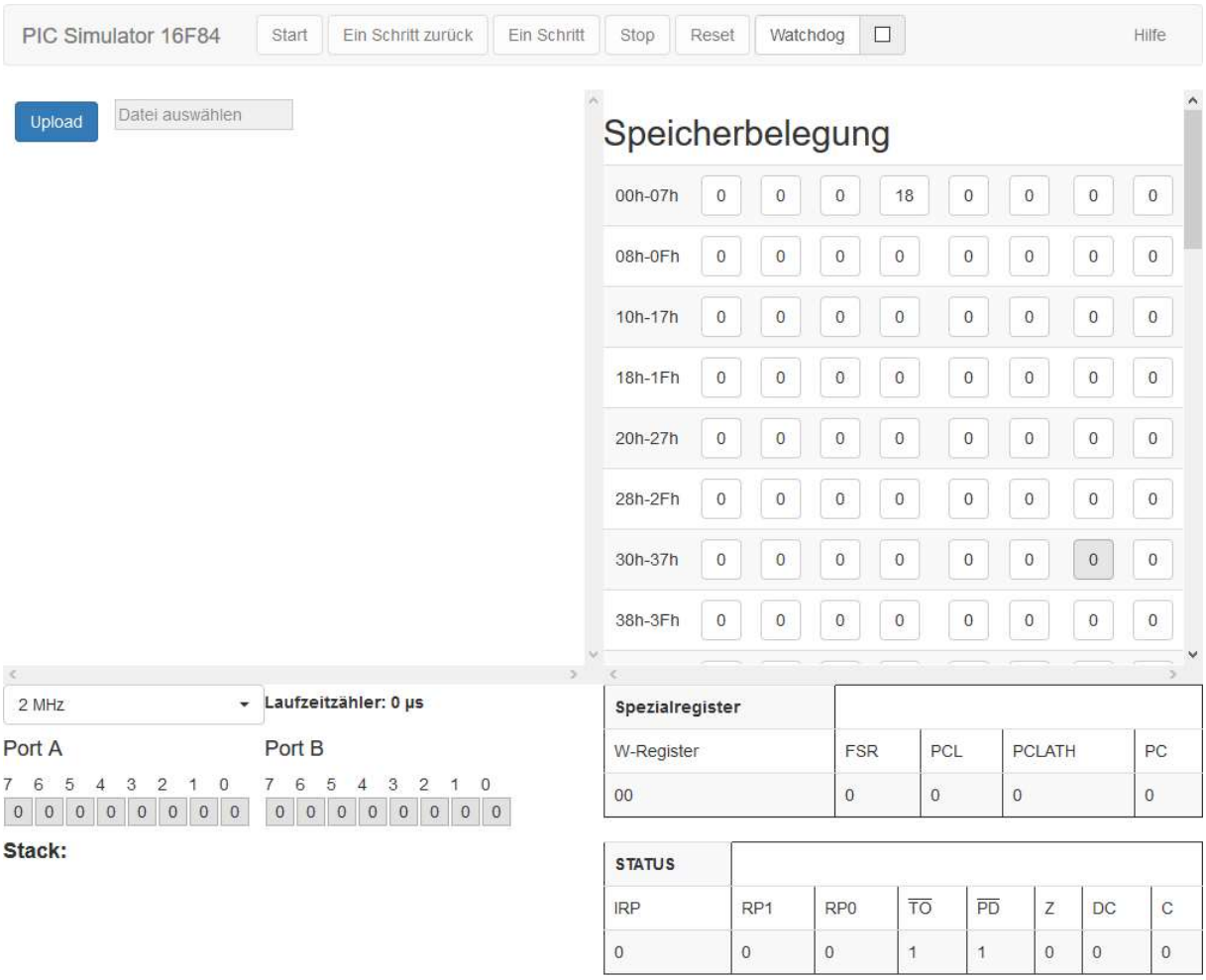
Als Programmiersprache wurde JavaScript und HTML/CSS gewählt. Als Frameworks wurde AngularJS, jQuery und Bootstrap verwendet.
Die Sprachen wurden gewählt da bereits in einem anderen Projekt Erfahrungen damit gemacht wurden.
Ein weiterer Vorteil von HTML und JavaScript ist, dass man relativ komfortabel eine Ansehnliche Oberfläche gestalten kann.
AngularJS ermöglicht es einige ansonsten komplexe Funktionen vereinfacht zu schreiben.
Bootstrap wurde für eine angenehm lesbare Darstellung gewählt.
JQuery wurde für den Button benötigt, der den Programmcode einließt um einen (attraktiveren) Bootstrap-Button verwenden zu können und im nebenstehenden Label den Dateinamen ausgeben zu können.

Als Entwicklungsumgebung wurde WebStorm von JetBrains verwendet was bereits einen integrierten Webserver beinhaltet und so das schnelle betrachten von Webseiten ermöglicht. Ein weiterer Punkt der für WebStorm sprach war, dass GitHub als Codeverwaltung integriert ist. Durch GitHub war es möglich im Team zu programmieren und bei Fehlern, durch einen automatische Versionierung, auf ältere Stände zuzugreifen. Auch ein Editieren der gleichen Datei ist möglich. Eventuelle Überschneidungen werden dann durch WebStorm einzeln abgefragt und es kann Änderung für Änderung entschieden werden, welche übernommen werden soll.

Die Benutzeroberfläche

Damit die Benutzeroberfläche gut bedienbar bleibt wurde eine Navigationsleiste am oberen Bildschirmrand eingefügt. Alle Steuerungsbuttons wie "Start", "Schritt zurück", "ein Schritt", "Stop", "Reset" und "Watchdog" sind darin zu finden. Auch ein Hilfebutton über den man diese Dokumentation aufrufen kann ist hinterlegt.
Der Bereich unterhalb der Navigationsleiste ist zu je 50% (in der Breite) aufgeteilt zwischen dem Programmtext und der Anzeige des Speichers. Um ein übermäßiges Scrollen zu verhindern wurden beide Fenster in der Höhe begrenzt und ein zusätzlicher Scrollbalken aktiviert.
Unterhalb der Programmtextanzeige ist ein Button zur Geschwindigkeitsauswahl zu finden und ein Laufzeitzähler, der die Laufzeit in Mikrosekunden anzeigt. Darunter wurden die Ein-/Ausgänge PORTA und PORTB gesetzt mit Buttons zur Auswahl. Der visualisierte Stack wurde unterhalb davon positioniert.
Unterhalb der Speicheranzeige werden die in Tabellenform Spezialregister und das STATUS-Register angezeigt.

Auf eine Auswahl für externe Taktgeneratoren wurde verzichtet, da hierzu ein zusätzliches Framework benötigt worden wäre. Da die Einarbeitungszeit dafür als zu lang eingeschätzt wurde, wurde auf diesen Teil komplett verzichtet.



Realisierung

Klassenübersicht

Da es in JavaScript/ HTML keine echten Klassen gibt, dienen die einzelnen JavaScript Files als Ersatz für die Klassen. Diese Files enthalten nach der Model View Controller Architektur
Je einen Controller. Einige der hier dargestellten Controller werden zur besseren Übersichtlichkeit in eine JavaScript Datei geschrieben.

Anzeige.html

Die eigentliche Darstellung des Simulators erfolgt über die HTML-Seite.
Sie dient als View und enthält keine Logik. Hier wird der Assemblercode angezeigt, der Takt ausgewählt, Breakpoints gesetzt und der Speicherinhalt visualisiert.

RamCtrl.js

Das Ramarray des RamControllers repräsentiert den Speicher des PIC Microcontrollers. Dazu gehört der *Instructioncounter* und das Arbeitsregister (*W_reg*). Damit mehrere Controller auf den Ram, *Instructioncounter* und das Arbeitsregister zugreifen können, werden diese in einer Factory, einem AngularJS spezifischen Model, gespeichert. Diese bilden einen eigenen *Scope*, auf dem dafür ausgewählten Controller zugreifen können.

BefehlsspeicherCtrl.js

Im Befehlsspeicher wird die eingelesene Datei, die durch die Uploadfile Funktionalität geliefert wird, zu einzelnen Linien verarbeitet und in das *Content* Array gespeichert. Des Weiteren werden Befehlszeilen ausgefiltert und in das *Operations* Objektarray gespeichert.
Das dadurch entstandene Array *Content* wird in der Anzeige.html als LST- Quelltext ausgegeben.

UploadCtrl.js

Damit der Nutzer weiß welche Datei er in den Simulator geladen hat, wird der Name mit diesem Controller ausgefiltert und im View angezeigt.

DropdownCtrl.js

Die Taktauswahl wird mit Hilfe des DropdownControllers realisiert. Sie enthält die vorgegebenen Frequenzen und leitet die vom Nutzer ausgewählte Frequenz an die Ablaufssteuerung weiter.

DataFactory.js

Die Factory ist eine AngularJS spezifische Funktion um Daten zwischen verschiedenen Controllern zu teilen und gleichzeitig zu synchronisieren. In dieser wird ein *PicData* Objekt mit Variablen und Funktionen beladen und in den einzelnen Controllern eingebunden.

AblaufsCtrl.js

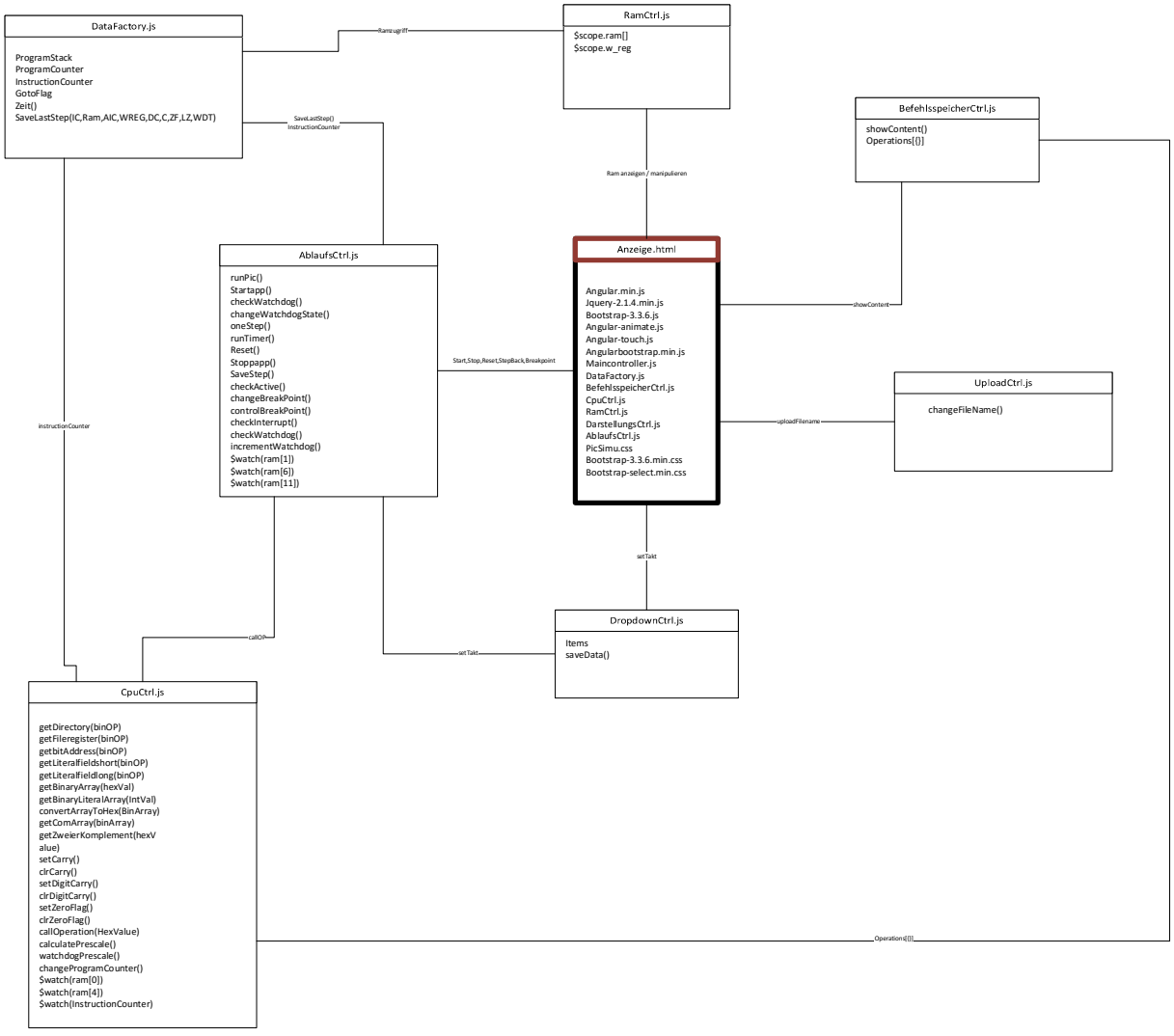
Steuert die Abläufe in dem Simulator. Diese beinhalten sowohl den Start durch den Nutzer als auch die Tests auf Interrupt, Timer und Breakpoints. Die *$watch*-Funktionen sind Angularspezifische Echtzeitüberprüfungen, ob sich der Wert der Zielvariable verändert hat.

CpuCtrl.js

Dieser Controller führt alle Picspezifischen Operationen durch. Einige Funktionen dienen nicht zur Ausführung von Assemblerbefehlen, sondern zum besseren Umgang mit JavaScript. Die Hauptfunktion für die Ausführung der Assembleraufrufe geschieht über die callOperation Funktion. Hier wird als Übergabeparameter ein Hex Wert mitgegeben. Der durch den PIC-Assembler-Compiler erzeugte LST-Code enthält vor jeder programmrelevanten Zeile zwei Hex Werte. Der erste stellt die Programmzeile und der zweite den Befehlsaufruf dar. Durch Maskierung wird der richtige Befehlsaufruf aus dem Hex Wert ausgelesen. Diese Prozedur eignet sich auch zur Auslese der Befehlsparameter.

Gesamtübersicht

Auf der folgenden Seite werden alle verwendeten JavaScript und HTML Dateien mit ihren Verbindungen dargestellt. Frameworks wie JQuery und Bootstrap wurden nicht beachtet, da diese Funktionalitäten liefern, aber nicht verändert wurden.



Funktionalitäten

Code einlesen

Sobald eine Datei über den Upload-Button ausgewählt wird, wird die Funktion "showContent" aufgerufen. In der Funktion wird der gesamte Text eingelesen und an den Stellen an denen ein "\n" enthalten ist wird ein neues Element in einem Array gespeichert.

```
var befehlssatz = newArray();
befehlssatz = $fileContent.split('\n');
```

Der Dateiinhalt wird danach Zeilenweise in der "Programmanzeige" ausgegeben.

```
$scope.content = befehlssatz;
```

Eine Schleife wird durch AngularJS in HTML möglich. Die Schleife läuft solange durch "content" solange noch Elemente vorhanden sind.

```
<p ng-repeat="lineincontent"
ng-class="{highlighted:checkActive(line)}">
<input type="checkbox" ng-click="changeBreakPoint(line)
"ng-show="checkBreakpoint(line)">{{line}}
</p>
```

Über eine Regular Expression werden die Zeilen identifiziert, welche mit zwei vierstelligen Blöcken beginnen, die Hex-Codiert sind. Alle Zeilen werden durch eine Schleife überprüft und in ein Objekt-Array abgespeichert welches die Zeile und den eigentlichen Befehl beinhaltet.

```
If(/[0-9a-fA-F]{4}\s*[0-9a-fA-F]{4}/.test(befehlssatz[i])){

tempbefehlsarray=befehlssatz[i].split('');
```

Effektiv abgearbeitet wird letzten Endes nur das Objektarray, da hier die Befehle einzeln vorliegen.

Befehlsabarbeitung

Die Befehlsabarbeitung basiert auf der Funktion "oneStep". Die Funktion "runpic", die ausgeführt wird, wenn auf "Start" geklickt wird, ruft letzten Endes lediglich automatisiert immer wieder die Funktionen "startapp"und "checkwatchdog" auf. Die Funktion "startapp" ruft dann die Funktion "oneStep" auf und führt diese, wenn der Watchdog dies zulässt aus. Zur Umsetzung des Watchdogs wird die Startapp nicht direkt durch den Start-Button aufgerufen, sondern erst durch die "runPic" Funktion ausgeführt. In dieser wird vor dem Programmstart geprüft, ob der Watchdog aktiv ist oder nicht.
Die Funktion "oneStep" speichert im ersten Schritt alle Zustände ab und ermöglicht so den Schritt zurück. Danach wird überprüft ob ein Interrupt vorliegt und gegebenenfalls werden durch die Funktion "checkInterrupt" die notwendigen Werte verändert.

```
$scope.SaveStep();

$scope.checkInterrupt();
```

Anschließend beginnt die eigentliche Befehlsausführung. Die Funktion "callOperation" wird aufgerufen und der Befehl der aktuellen Zeile wird übergeben.

```
$scope.callOperation($scope.operations[DataPic.Instruction
counter].befehl);
```

Die Variable DataPic.Taktanzahl enthält den aktuellen Stand der Takte. Die Funktion calculatePrescale berechnet den Prescalerwert, der im OPTION_REG eingestellt wurde und ermittelt mittels Modulo Division, ob die Taktanzahl ganzzahlig durch den Prescaler teilbar ist. Außerdem muss die Taktanzahl größer oder gleich 4 sein, damit der erste Durchgang keine Timerincrementierung auslöst. Die runTimer Funktion erhöht den Stand des TMR0 Registers um 1. Falls es zum überlauf kommt, wird das TI0F -Flag gesetzt und der TMR0 beginnt von neuem zu zählen.

```
if(((DataPic.Taktanzahl%$scope.calculatePrescale())==0)
&&DataPic.Taktanzahl>=4){
$scope.runTimer();}
```

Ist das Goto-Flag nicht gesetzt soll regulär der Programcounter erhöht werden. Ist es gesetzt soll der Instructioncounter nicht erhöht werden

```
if(DataPic.GotoFlag==1){
     DataPic.GotoFlag=0;}
else{
     DataPic.Instructioncounter++;
}
```

Anschließend müssen noch die "neuen" Werte für den Laufzeitzähler und den Instructioncounter in der Factory verfügbar gemacht werden, da sie ansonsten verloren gehen würden.

```
DataPic.AnzeigeIC++;
$scope.Laufzeit=DataPic.Laufzeit;
$scope.Instructioncounter=DataPic.Instructioncounter;
```

Interrupts

Jedes Mal, wenn ein Befehl ausgeführt werden soll wird die Funktion checkInterrupt ausgeführt. In dieser Funktion werden verschiedene Zustände abgeprüft und gegebenenfalls der Instructionpointer manipuliert werden.

```
if(($scope.T0IE && $scope.T0IF && $scope.GIE)||
($scope.INTE && $scope.RB0InterruptFlag && $scope.GIE)||
($scope.RBIE && $scope.RBIF && $scope.GIE))
```

Da ein Interrupt den PIC aus dem sleep-Befehl "aufwecken" kann, wird vorsichtshalber das Sleepflag auf false gesetzt. Dies hat nur eine Auswirkung, wenn der PIC tatsächlich im sleep-Zustand ist.

```
DataPic.Sleepflag=false;
```

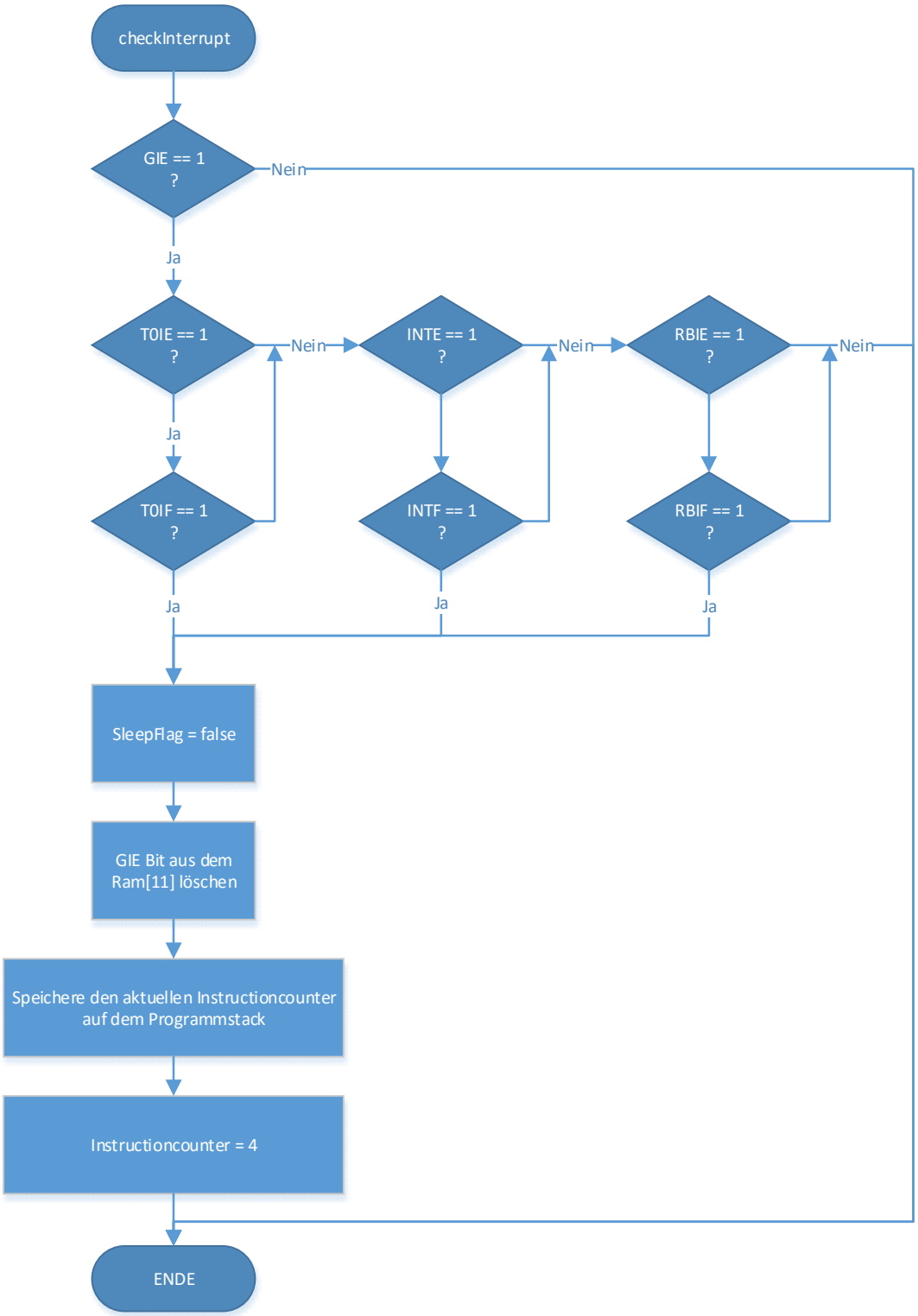
Damit in der Interruptroutine kein weiterer Interrupt ausgelöst werden kann, muss das GIE im Register 11 auf 0 gesetzt werden. Dies geschieht durch die Verundung des aktuellen Zustandes vom Ram[11] und 0x7F. Da das Ergebnis eine Integer-Zahl ist, muss diese vor dem Speichern in eine Hex Zahl umgewandelt werden.

```
$scope.ram[11]=(parseInt($scope.ram[11],16)
&parseInt("01111111",2)).toString(16);
```

Nun wird der aktuelle Instructionpointer im Stack abgespeichert. Die Interruptserviceroutine muss an der Adresse 4 stehen. Damit die Funktion oneStep mit den Interruptbefehlen weiterarbeitet, wird der Wert 4 in den Instructioncounter geschrieben.

```
DataPic.ProgramStack.push(DataPic.Instructioncounter);
$scope.ProgramStack=DataPic.ProgramStack;
DataPic.Instructioncounter=4;
```

Zum Veranschaulichen dieses Vorgangs dient das Flussdiagramm. Da im Programm Ablaufplan keine mehrteiligen Überprüfungen stattfinden können, wurden diese logisch aufgetrennt.

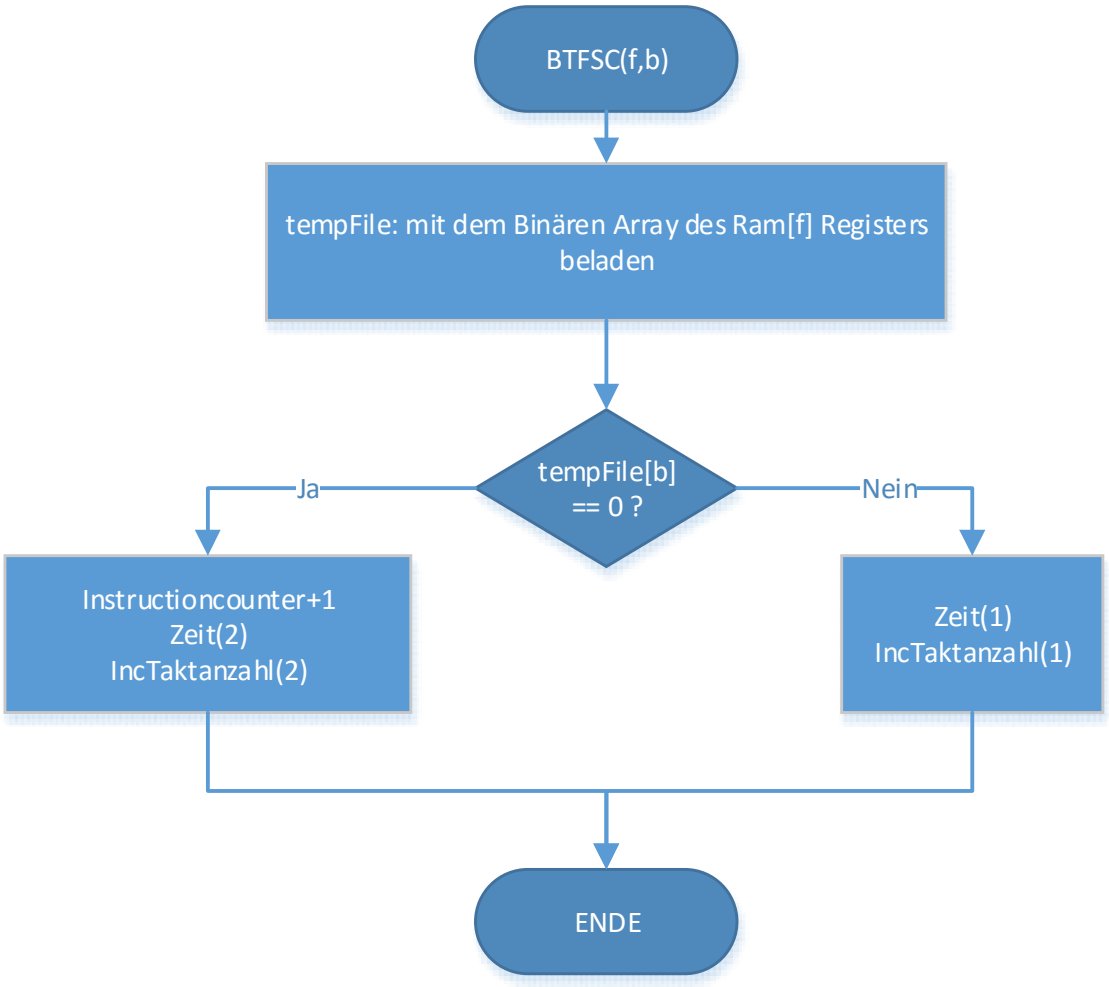


Befehlsbeschreibung

Im folgendem Kapitel wird die Befehle des PIC Registers mit Hilfe von Flussdiagrammen beschrieben. Die Register des PICs werden mit Ram[f] angesprochen. Wobei Ram ein Array ist, dessen einzelnen Zellen mit der "f" Variable angesprochen. Die Funktionsaufrufe "Zeit(x)" und "IncTaktanzahl(x)" erhöhen die Laufzeit und die gesamte Taktanzahl. Da diese keine weitere Bedeutung für die Befehlsausführung haben, wurden sie nicht weiter ausgeschrieben und können im Quelltext nachgelesen werden.

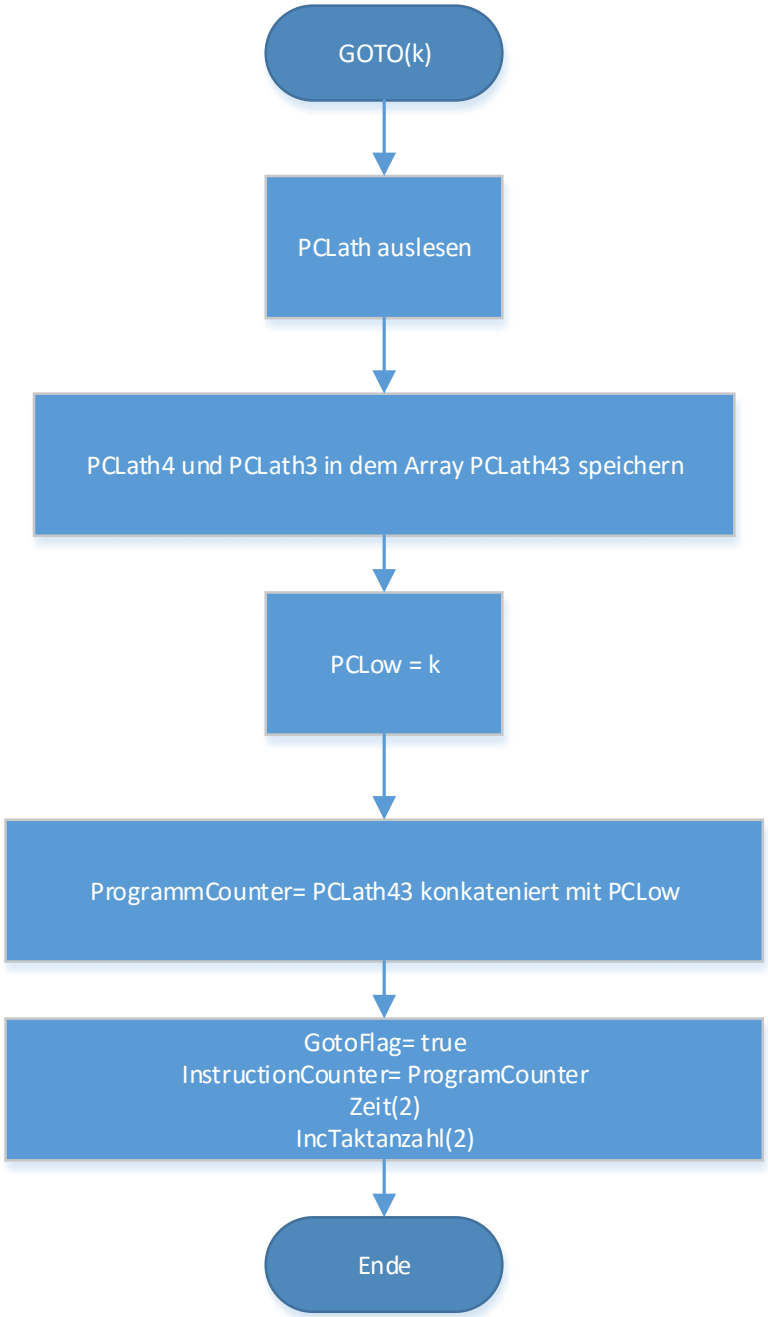
BTFSC (f,b)

Der BitTestFileSkipClear Befehl testet, ob das b - Bit eines Registers f nicht gesetzt ist. Falls es nicht gesetzt ist, wird der darauffolgende Befehl übersprungen. Falls dieser gesetzt ist, läuft das Programm normal weiter.



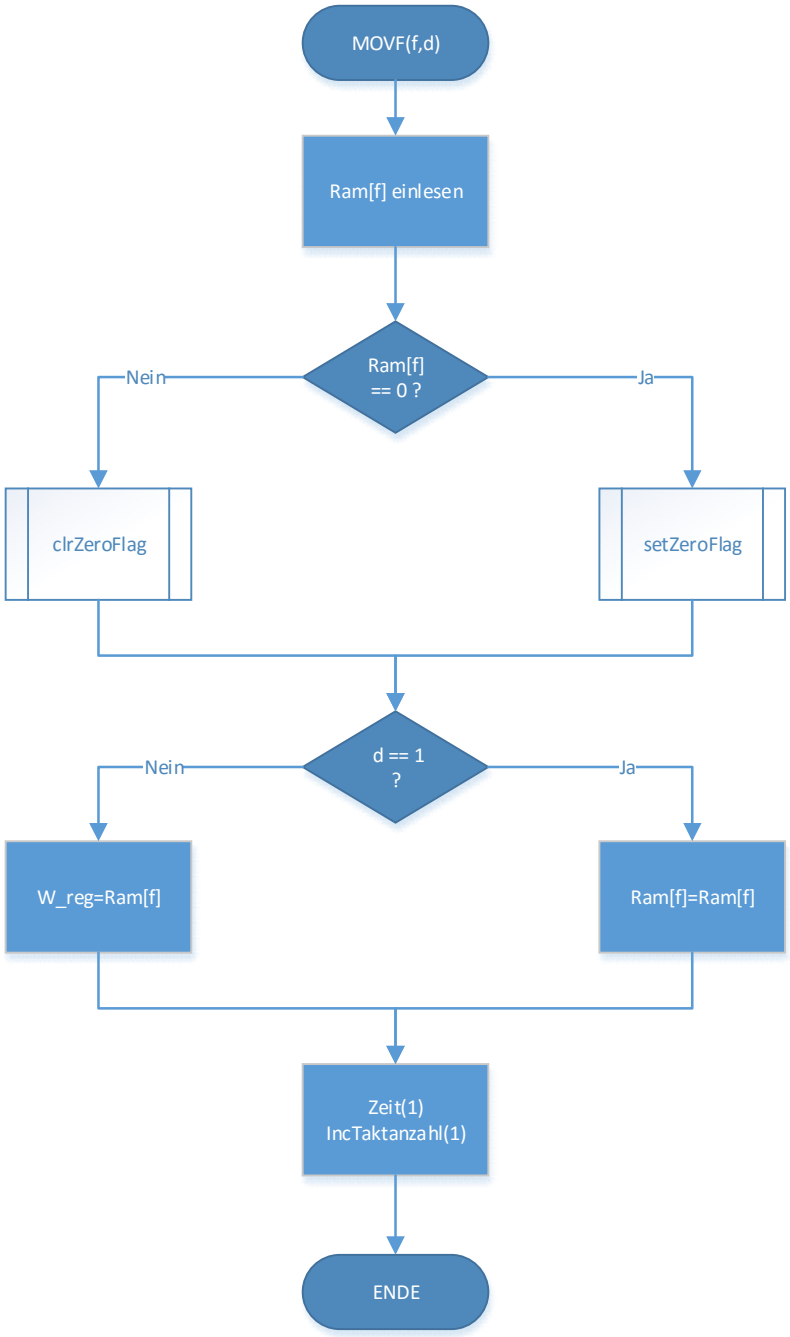
GOTO (k)

Der Goto Befehl verändert den *Instructioncounter*. Damit springt der *Instructioncounter* statt auf den nächsten Befehl, zu Adresse aus der Kombination aus dem *PCLath* und dem *k* Literal. Beim *PCLath* Register werden die Bits 4 und 3 hinzugenommen und an die MSB Position zum *k* Literal konkateniert. Da bei dieser Architektur der *Instructioncounter* nach jedem oneStep diesen um 1 erhöht, wird das *Gotoflag* verwendet um eine Manipulation des *Instructioncounter* anzuzeigen. Bei gesetztem *Gotoflag* wird der *Instructioncounter* nicht erhöht.



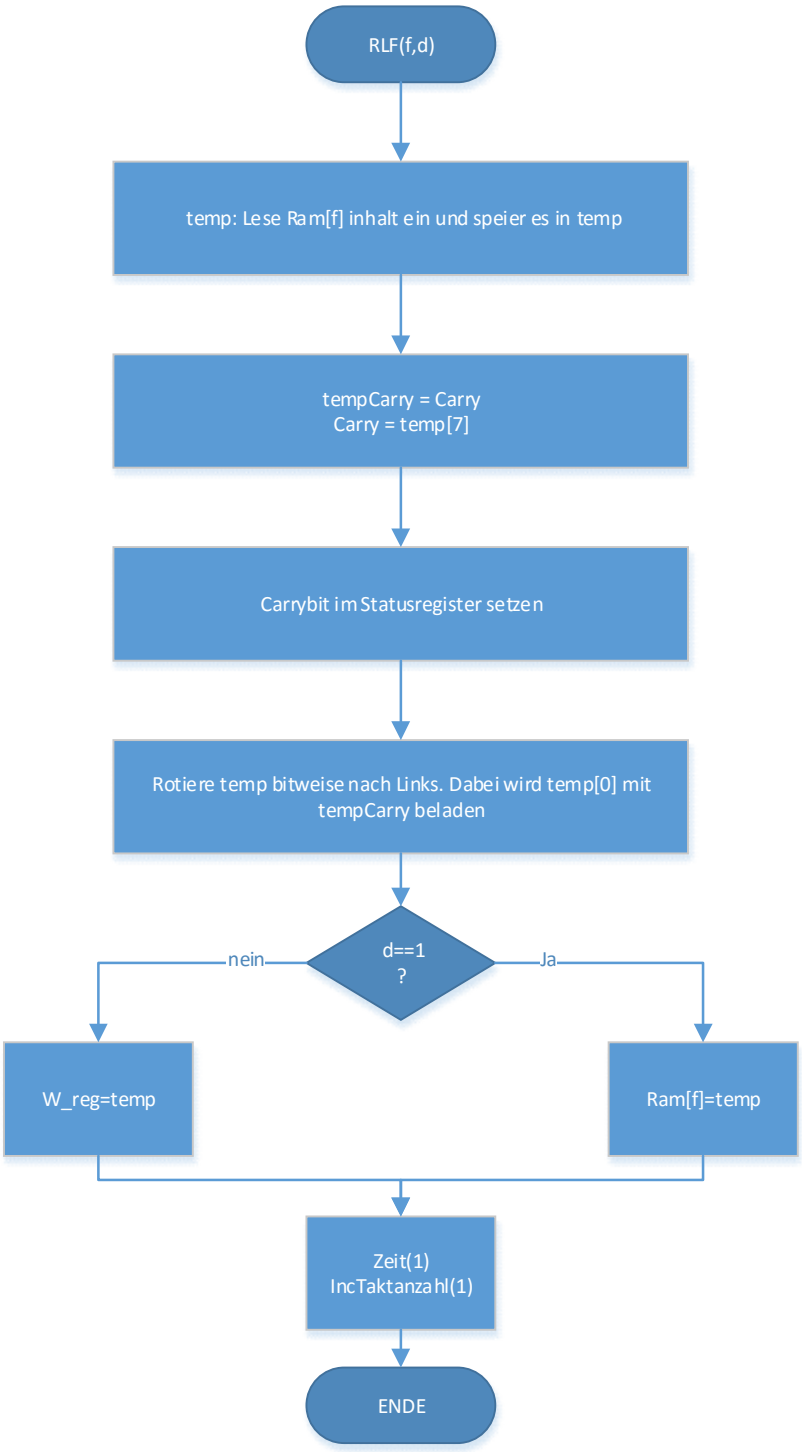
MOVF (f,d)

Der MOVE FILE Befehl liest den Wert des Registers f ein und überprüft, ob dieser gleich 0 ist. Falls dieser 0 ist, wird das Zero Bit im Statusregister durch die Unterfunktion setZeroFlag gesetzt. Bei ungleich 0 wird das *ZeroBit* im Statusregister, durch die clrZeroFlag Unterfunktion, gelöscht. Falls das Direction (d) Bit gleich 1 ist, wird der Registerwert wieder ins F Register gespeichert. Bei d=0 wird es ins Arbeitsregister (*W_reg*) gespeichert.



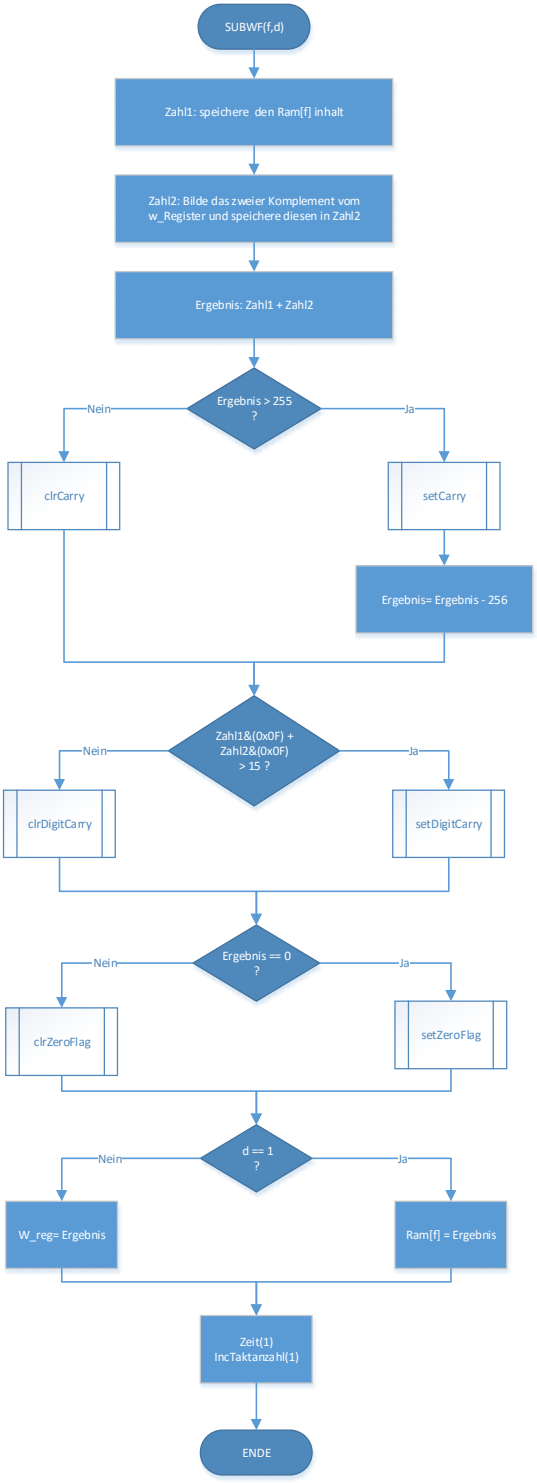
RLF(f,d)

Der RotateLeftFile verschiebt den kompletten f - Register Inhalt um eine Position nach links. Dabei wird das letzte Bit des Registers ins *CarryBit* geschrieben und das *CarryBit* wandert an die erste Position von des f Register. Wie bei allen Programmen bestimmt das d - Bit den Speicherort des Ergebnisses (*temp*).



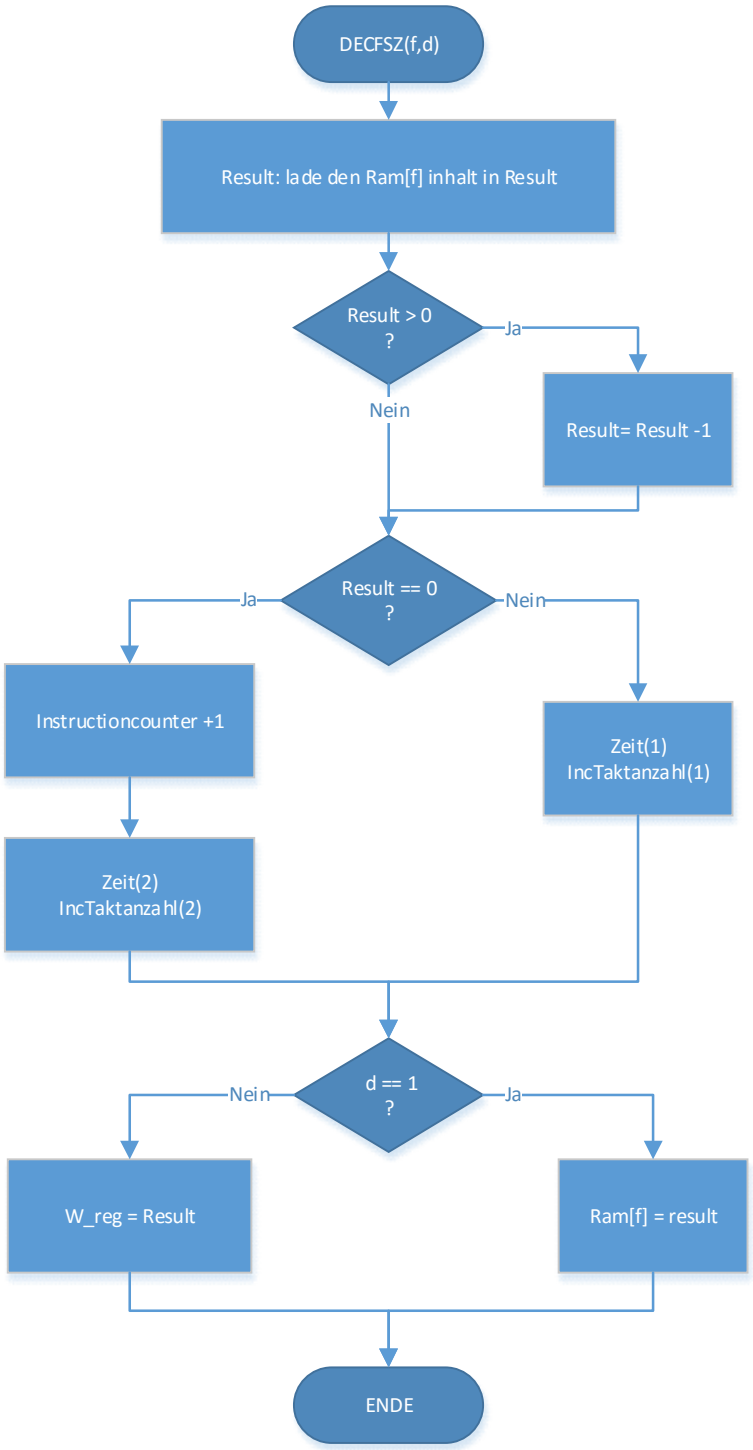
SUBWF (f,d)

Die Subtraktionsoperation des PIC Simulators nimmt den Inhalt des f-Registers als Zahl1 und addiert das zweier Komplement des W_Registers. Falls das Ergebnis größer als 255 ist, wird das *CarryBit* gesetzt und 256 vom Ergebnis abgezogen. Ansonsten wird das *CarryBit* gelöscht. Falls das Ergebnis gleich null ist, wird das *ZeroFlag* gesetzt. Ansonsten wird das *ZeroBit* gelöscht. Zur *Digitcarry* Überprüfung werden Zahl1 und Zahl2 logisch mit 0x0F verknüpft. Falls das Ergebnis der beiden unteren Nibble größer als 16 ist, wird das *DigitCarryBit* gesetzt. Das d-Bit bestimmt den Speicherort



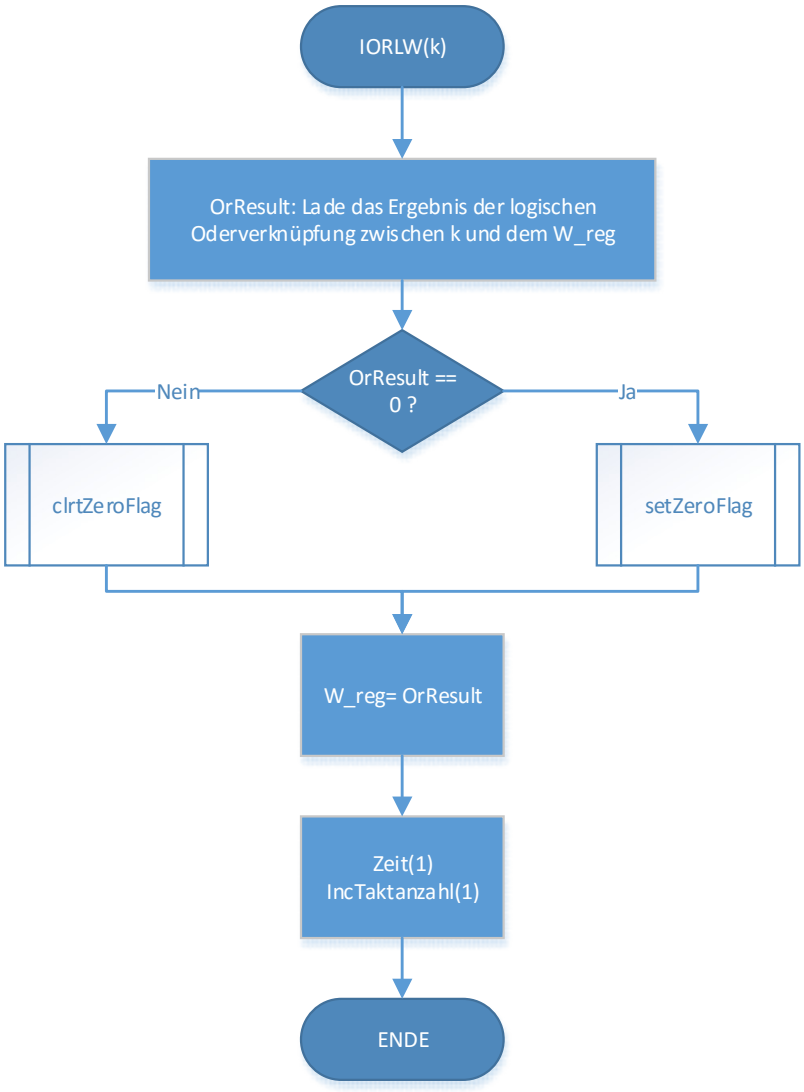
DECFSZ(f,d)

Lade den Speicherinhalt des f- Registers in die Resultat Variable. Falls diese größer 0 ist, ziehe eins davon ab. Wenn das Resultat danach gleich 0 ist, erhöhe den Instructioncounter um 1, damit wird der folge Befehl übersprungen. Abhängig vom d Bit wird das Ergebnis entweder in das W_Register oder in das f-Register selbst gespeichert. Falls ein Befehl übersprungen werden muss, wird sowohl die Zeit als auch die IncTaktanzahlfunktion mit dem Wert 2 ausgeführt. Ansonsten mit dem Wert 1.



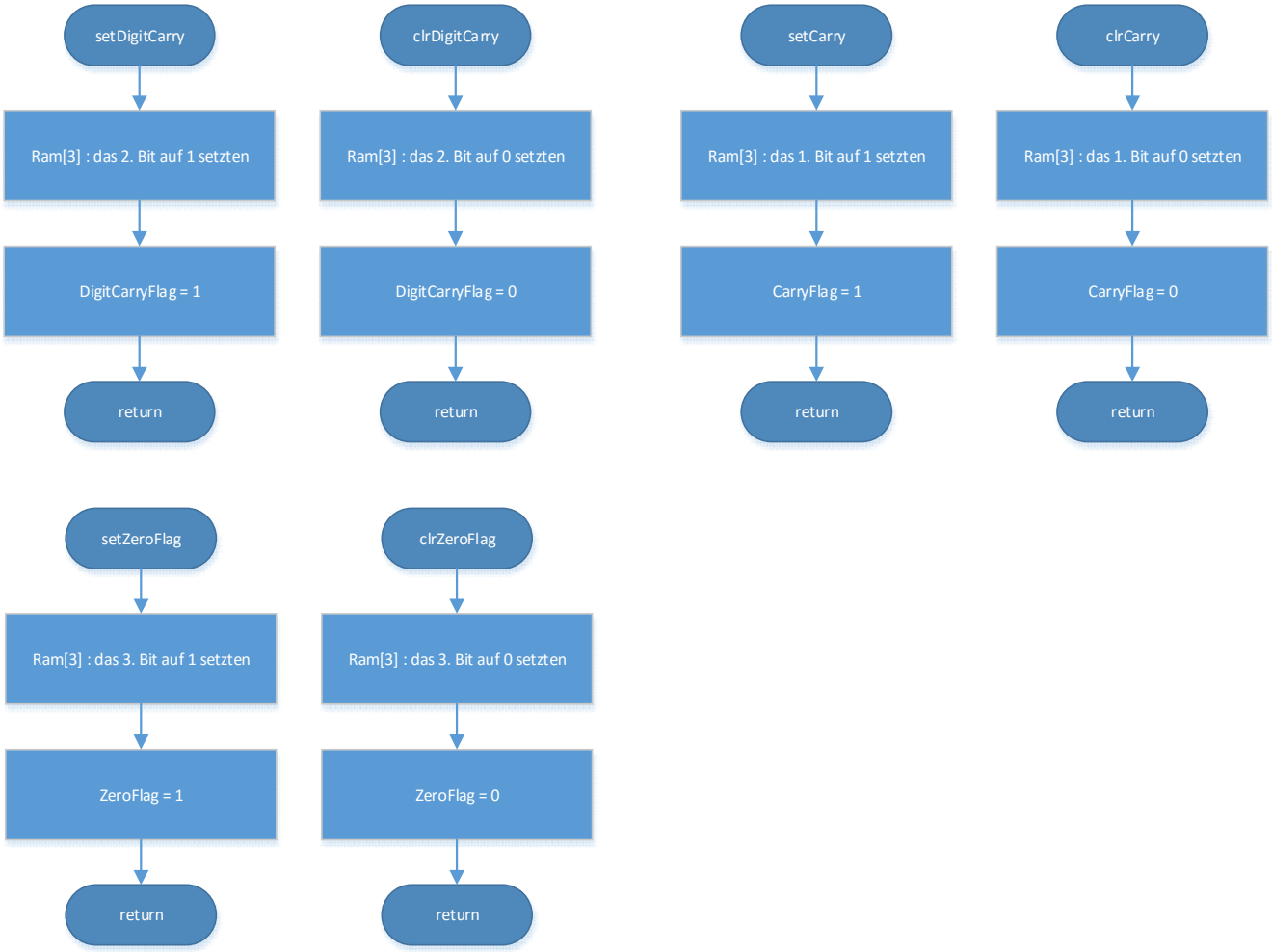
IORLW (k)

Belade das *OrResultat* mit der logischen Verknüpfung aus dem K- Literal und dem W_Register. Falls das OrResultat gleich 0 ist, wird die setZeroFlag Unterfunktion ausgeführt. Ansonsten wird die clrZeroFlag Unterfunktion ausgeführt.



ZeroBit, Digitcarry und Carry Funktionen

Diese Unterprogramme löschen oder setzten die Zerobits, Digitcarrys und Carrys im Statusregister. Das Statusregister wird durch das Ram[3] Arrayelement dargestellt. Das Zerobit ist das Ram[3][2] Bit in der zwei Dimensionalen Arraydarstellung. Digitcarry: Ram[3][1] und Carry: Ram[3][0]. Sie sind in setBIT und clrBIT aufgeteilt. Wobei BIT für das jeweilige Bit steht.



Fazit

Als Fazit ziehen wir, dass eine Umsetzung mit der Sprache JavaScript möglich ist aber nicht empfehlenswert. Da JavaScript nicht direkt mit Binärwerten arbeiten kann muss ständig geparst werden zwischen String und Array. Dies verkompliziert jede noch so einfache Bitänderung. Die Watchdogfunktion führt ebenfalls zu Problemen, da die Ausführung künstlich verlangsamt werden muss damit der Browser nicht abstürzt. Dies hat zur Folge, dass der Watchdog nicht korrekt zählt und so unerwünschte Effekte erzielt werden können. Eine Hardwareanbindung ist möglich aber nicht mit dem von uns gewählten Framework AngularJS. Das Framework NodeJS hätte vermutlich diese Funktionalität geboten konnte jedoch aufgrund von Zeitmangel nicht erarbeitet werden. Die Zeit ist außerdem sowieso sehr knapp da parallel das Fach Software Engineering stattfindet und hier ebenfalls viel Zeit für Programmierungen etc. aufgewendet werden muss.
Vorteilhaft war die Nutzung des Frameworks AngularJS dennoch da es uns ermöglicht hat viele Funktionalitäten vereinfacht umzusetzen. Ebenfalls hilfreich ist die native Implementierung von Databinding. Durch die verwendete Web-Technologie ist der Simulator auch nahezu Plattformunabhängig. Die einzige Voraussetzung ist, dass das Gerät eine Website aufrufen kann und JavaScript verarbeiten kann. Auf Browserinkompatibilitäten sind wir in unserem Projekt glücklicherweise nicht gestoßen.
Alles in allem kann man sagen, dass das Projekt sehr viel für das Verständnis von Assembler und Microcontrollern gebracht hat. Durch die intensive Auseinandersetzung konnten die Zusammenhänge der einzelnen Komponenten besser verstanden werden.