Simulator für den Microcontroller PIC16F84

Kurs: TINF18B3

Student: Maximilian Maier

& Marcel Sommer

Vorlesung: Systemnahes Programmieren 2

Dozent: Stefan Lehmann

Inhalt

1. Grundsätzliche Arbeitsweise eines Simulators

In einem Simulator werden komplexe System simuliert. Gewöhnlicherweise ist die Simulation dieser Systeme für die theoretische oder formelmäßige Behandlung zu komplex. Deshalb wird ein Simulator entworfen. Der Simulator wird dabei mit der Struktur, Funktion und Verhalten des Systems bestückt. Anschließend werden in den Simulator echte Werte eingegeben, um ein Ergebnis zu erhalten. Im Falle des PIC Simulators werden Programme aus Assamblercode als Werte in den Simulator eingegeben. Diese Programme führt der Simulator aus und gibt ein oder mehrere Ergebnisse zurück. Auf diese Weise müssen die Programme nicht per Hand berechnet werden. Der Simulator wurde extra für die Ausführung solcher Programme entworfen, sprich die Befehle der Programme und viele weitere Funktionen sind im Simulator implementiert.

2. Vor- und Nachteile einer Simulation

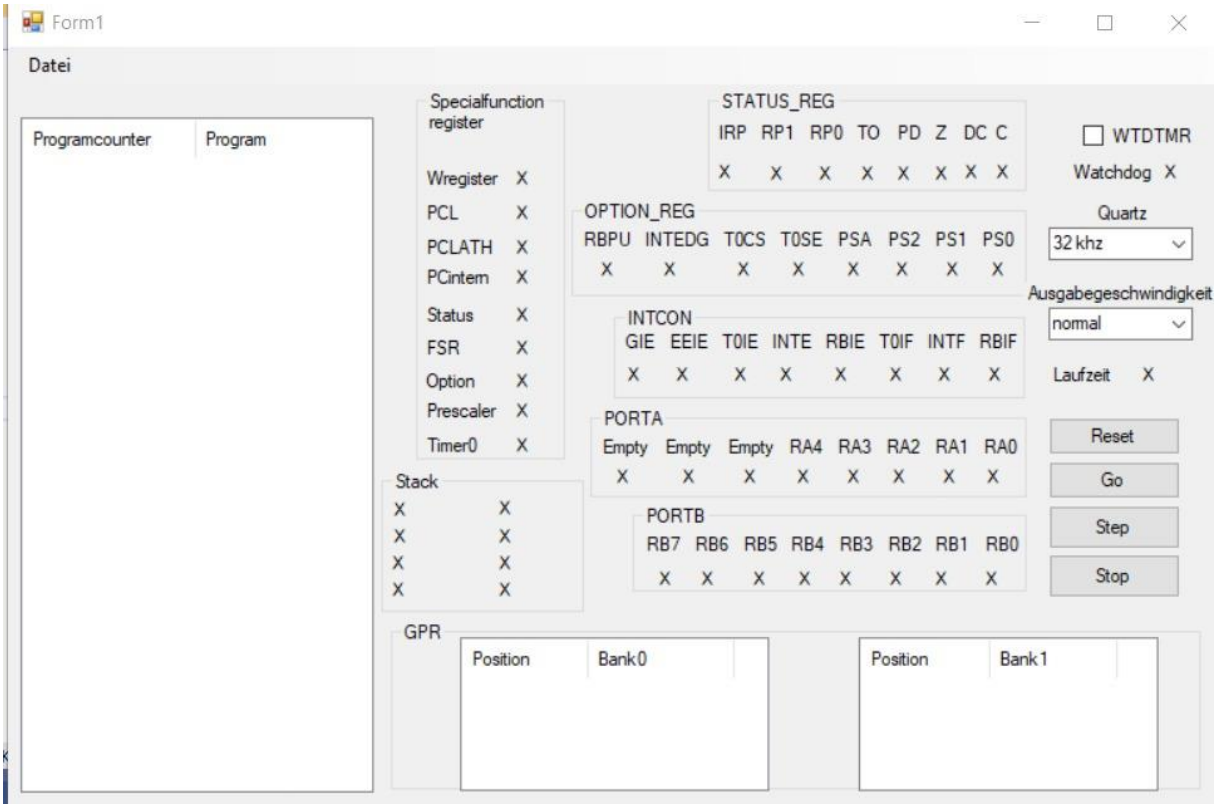
Vorteile:

- Ergebnisse können vor der Realisierung erlangt werden
- Konsequenzen vor einer Realisierung bekannt
- Für komplexe Systeme können einfacher Ergebnisse erlangt werden
- Schnelleres Erlangen von Ergebnissen
- Aufwendige Rechnungen müssen nur einmal implementiert werden
- Wiederverwendung von Simulatoren spart Aufwand
- Anpassung und Erweiterung von einer Simulation einfach

Nachteile:

- Falsche Ergebnisse bei fehlerhafter Implementierung des Simulators
- Berücksichtigung von verschiedenen Faktoren nötig
- Fehler bleiben oft unbemerkt
- Unterschiede zur Realität möglich

3. Programmoberfläche und deren Handhabung



Die Beschreibung der Oberfläche beginnt links oben und endet rechts unten.
Beim Klick auf „Datei“ lässt sich eine beliebige Datei öffnen. Das Öffnen von .LST Dateien ist jedoch vorgesehen. Der Inhalt der .LST Datei wird anschließend im darunter stehenden Feld abgebildet. In der Spalte Programcounter, wird der Programmzähler abgebildet, in der Spalte Programm, der Befehlscode des Programms. Im Specialfunctionregister werden verschiedene Variablen abgebildet. Die gesamte GUI und die dazugehörigen Werte, werden nach jedem Schritt im Programmablauf aktualisiert. Im Stack, wird der Stack des Programms abgebildet. Hier wurde nicht mit einem Array gearbeitet, sondern mit dem Datentyp „Stack“ von C#. Deshalb werden nicht alle 8 Werte des Stacks angezeigt, sondern nur das oberste Element. Dieses wird je nach Position im Stack auf einem der 8 „X“ angezeigt. Die Register STATUS, OPTION, INTCON, PORTA und PORTB, sind ein Abbild der Register des PICSIM. Auch diese werden mit jedem Schritt im Programm aktualisiert. Nach dem Laden des Programms werden die „X“ durch 0 oder 1 ersetzt, je nach Inhalt des Registers. Bei der Checkbox „WTDTMR“ kann der Watchdog durch Anklicken des Feldes aktiviert werden. Dann wird direkt darunter der Watchdog berechnet bzw. gezählt. Bei „Quartz“ kann, falls gewünscht, die Quartzfrequenz eingestellt werden. Hierbei handelt es sich um ein Dropdownmenu. Es sind mehrere Vorauswahlen möglich. Standardmäßig ist die Quartzfrequenz auf 32 khz eingestellt. Bei der Ausgabegeschwindigkeit handelt es sich um die Geschwindigkeit, mit welcher die Befehle abgearbeitet werden und somit die Ausgabe erfolgt. Standardmäßig wird mit normaler Geschwindigkeit ausgeführt. Hierbei wird zwischen jedem Befehl 0,5 Sekunden pausiert. Bei „langsam“ wird zwischen den Befehlen 2 Sekunden pausiert und bei „schnell“ wird keine Pause eingelegt. Bei „Laufzeit“ wird die Laufzeit des Programms ausgegeben, welche mit der Quartzfrequenz berechnet wird. Bei „reset“ werden alle Register, Variablen etc. gelöscht, bzw. zurückgesetzt. Das Programm beginnt von Anfang. Bei „Go“ wird das Programm automatisch ausgeführt und läuft so lange, bis „Stop“ gedrückt wurde. „Step“ ist ein Einzelschritt. Es wird nur der nächste Befehl ausgeführt. „Stop“ beendet die automatische Ausführung, die mit „Go“ gestartet

wurde. Bei „GPR“ wird der Inhalt der Bank0 und der Bank1 zwischen 0CH und 4FH angezeigt. „Position“ gibt die Speicherstelle an, „BankX“ den Inhalt dieser Speicherstelle.

4. Beschreibung des Grundkonzepts[1]

Dieses Projekt wurde entwickelt um das Verständnis für die Funktionsweise eines Mikroprozessors oder Mikrocontrollers zu vertiefen. Gleichzeitig bot sich die Möglichkeit die im Studium gelernte Hochsprache (i.d.R. Java oder C) anzuwenden und zu verfestigen. Durch den großen Umfang dieses Projekts kommen sehr viele Kenntnisse der Hochsprache auf den Prüfstand. Die Studenten erkennen selbst, wo noch Defizite sind bzw. sind gezwungen diese aufzuarbeiten um das Projekt erfolgreich abschließen zu können.

Dieses Projekt wird seit über 15 Jahren an verschiedenen Fakultäten durchgeführt. Dabei sind die Ergebnisse zum Teil hervorragend, denn dieses Projekt hat für viele Studenten einen Wettkampfcharakter. Sei es mit anderen Gruppen, wer schafft was oder der Wettkampf gegen sich selber nach dem Motto „Das schaffe ich auch!“.

5. Beschreibung der Gliederung

Im ersten Schritt der Implementierung des Simulators wurde das Einlesen von den .LST Programmdateien realisiert. Dabei wurde auch die Konvertierung von Hex, Strings und Integer behandelt. Da die Implementierung einer Programmoberfläche erst zu späterem Zeitpunkt geplant wurde, wurde auch eine Ausgabe auf der Console implementiert. Um die eingelesenen Inhalte abspeichern und weiterverwenden zu können wurden als nächstes die Register und die Variablen implementiert. Hierbei wurden 2 Klassen erstellt. In der Klasse Register wurde die Bank0 und Bank1 initialisiert. In der Klasse Globals wurden globale Variablen erstellt, wie zum Beispiel der Programmspeicher. Diese Klasse erweiterte sich im Laufe des Projektes ständig. In der Klasse Register wurde auch zwei Reset Methoden implementiert, welche den Inhalt der Bänke auf die vorgegebenen Standardwerte zurücksetzt. Als nächstes wurde die Klasse Commands erstellt, in welcher gleichzeitig die Switch case Funktion implementiert wurde. Anhand dieser Methode wird entschieden welcher Befehl aufgerufen wird. Danach wurden alle Befehle in dieser Klasse implementiert. Für manche Befehle wurden Hilfsmethoden benötigt, welche bei der Implementierung der Befehle erstellt wurden. Anschließend wurde die Klasse Buttons erstellt, in welcher zunächst der Stepbutton und der Gobutton implementiert wurden. Diese Übergaben den nächsten Befehlscode an die Switchcase Anweisung und führten somit das Programm auf. Der Aufruf der Buttons erfolgte in der Main Methode. Nach erfolgreicher Implementierung aller Befehle und der zugehörigen Hilfsmethoden wurden die ersten Programme zum Test ausgeführt. Hierbei wurden einige Fehler erkannt und behoben. Vor allem die Hilfsmethoden, die die Flags anpassen, mussten stark überarbeitet werden. Nach dem die ersten Programme fehlerfrei liefen wurde die GUI implementiert. Hierbei wurde das gesamte Projekt von einer .net core Anwendung zu einer .net Framework Windows Forms Anwendung umgestellt. Einige Migrationsarbeiten und Nacharbeiten waren nötig. Zunächst wurde das Einlesen der .LST Dateien in der GUI implementiert. Dafür waren auch Überarbeitungen an der

[1] Dieser Paragraph ist aus der Beschreibung des Projektkonzepts aus dem Themenblatt_Simulator.pdf von Professor Lehmann entnommen

alten Einlesemethode nötig. Anschließend wurde das Abbilden des Inhalts der .LST Dateien auf der GUI implementiert. Im nächsten Schritt wurde der GO, Reset, Stop und Step Button implementiert. Dies machte die ganze Buttonsklasse überflüssig. Nachdem die Ausführung des Programms auf der GUI möglich war, wurde der Timer0, der Watchdog und der Vorteiler implementiert. Anschließend wurden auf der GUI die verschiedenen Register und Variablen abgebildet. Die dauerhafte Aktualisierung der GUI war deshalb erforderlich und wurde somit programmiert. Im selben Zug wurde auch der Watchdogtimer, die Quartzfrequenz und die Laufzeit auf der GUI abgebildet. Anschließend wurde die Logik hinter der Quartzfrequenz und der Laufzeit im Programm implementiert. Zur Fertigstellung der GUI wurde der Stack und der Inhalt der Bänke visualisiert. Zu guter Letzt wurde der Timer0 Interrupt im Programm implementiert.

6. Programmstruktur

Der Quellcode des Simulators besitzt mehrere Klassen, die für verschiedenste Bereiche des Simulators zuständig sind. In der folgenden Auflistung werden die Klassen mit ihren Funktionen dargestellt.

- Commands.cs
  - Innerhalb der Commands.cs Klasse werden die Befehle interpretiert und die Logik hinter den einzelnen Befehlen ist dort implementiert.
- LSTparse.cs
  - Innerhalb dieser Klasse wird das .LST Dokument eingelesen, umgewandelt und an den Befehlsdekoder übergeben.
- Register.cs
  - Diese Klasse beinhaltet die Initialisierung der einzelnen Register bzw. der Speicherbänke des Simulators. Zusätzlich sind die Methoden für den Powerreset sowie Resets durch andere Ereignisse implementiert.
- Variables.cs
  - Die Klasse Variables.cs beinhaltet sämtliche Variablen die Global im gesamten Projekt sichtbar sein sollen und von allen Klassen genutzt werden können.

7. Programmiersprache

Bei der Programmiersprache bestand Wahlfreiheit. Es wurde sich dazu entschieden C# zu verwenden. Es waren Programmierkenntnisse in C und C++ vorhanden, jedoch gestaltet sich die Oberflächenprogrammierung mit C# einfacher. Mit Windows Forms kann nämlich die Oberfläche recht einfach erstellt werden. Mit Java wäre auch eine Oberflächenprogrammierung recht einfach gewesen, jedoch waren keine Java Kenntnisse vorhanden. Aus diesen Gründen wurde C# als Programmiersprache festgelegt.

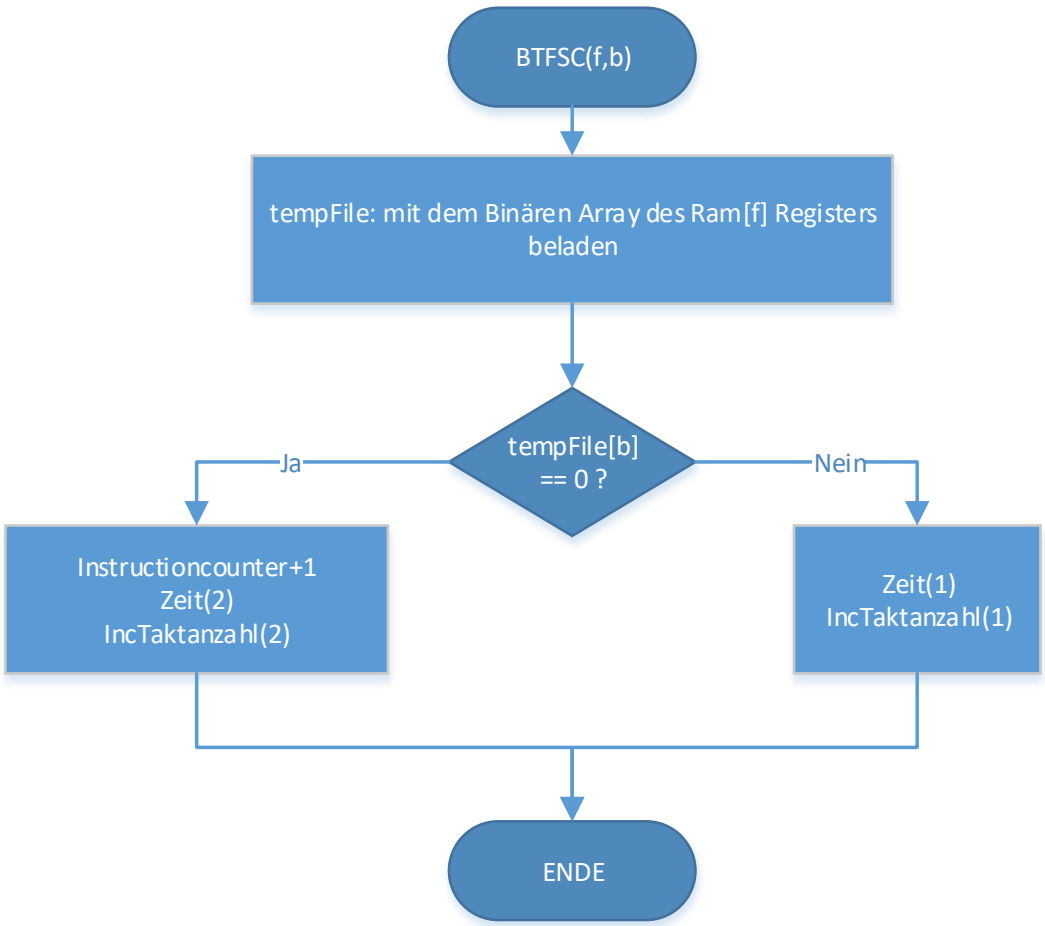
8. Beschreibung von Funktionen

Der Programmmemory wird aus Darstellungsgründen[2] mit Ram abgekürzt.

[2] Die Verwendeten Bilder wurden nicht selbst erstellt, die Beschreibungen wurden auf das selbstimplementierte Programm angepasst

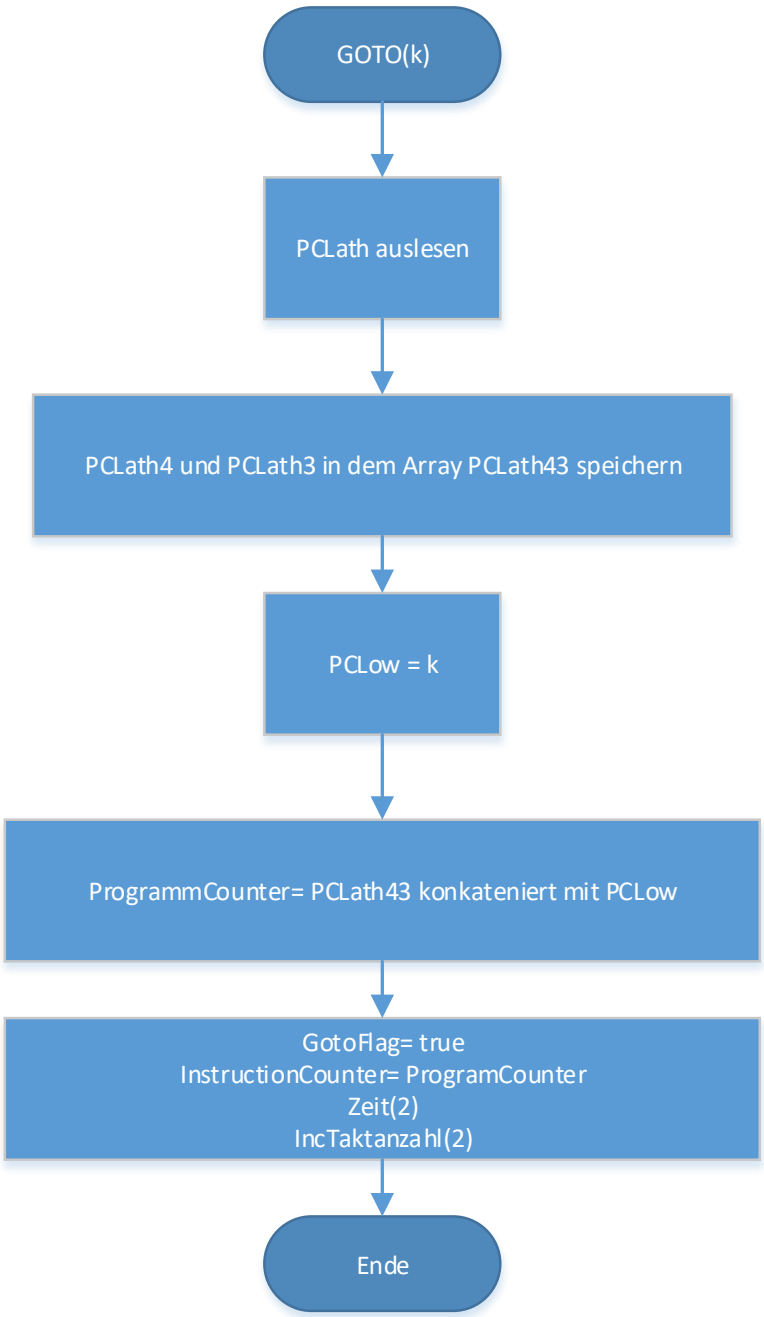
BTFSC (f,d)

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.



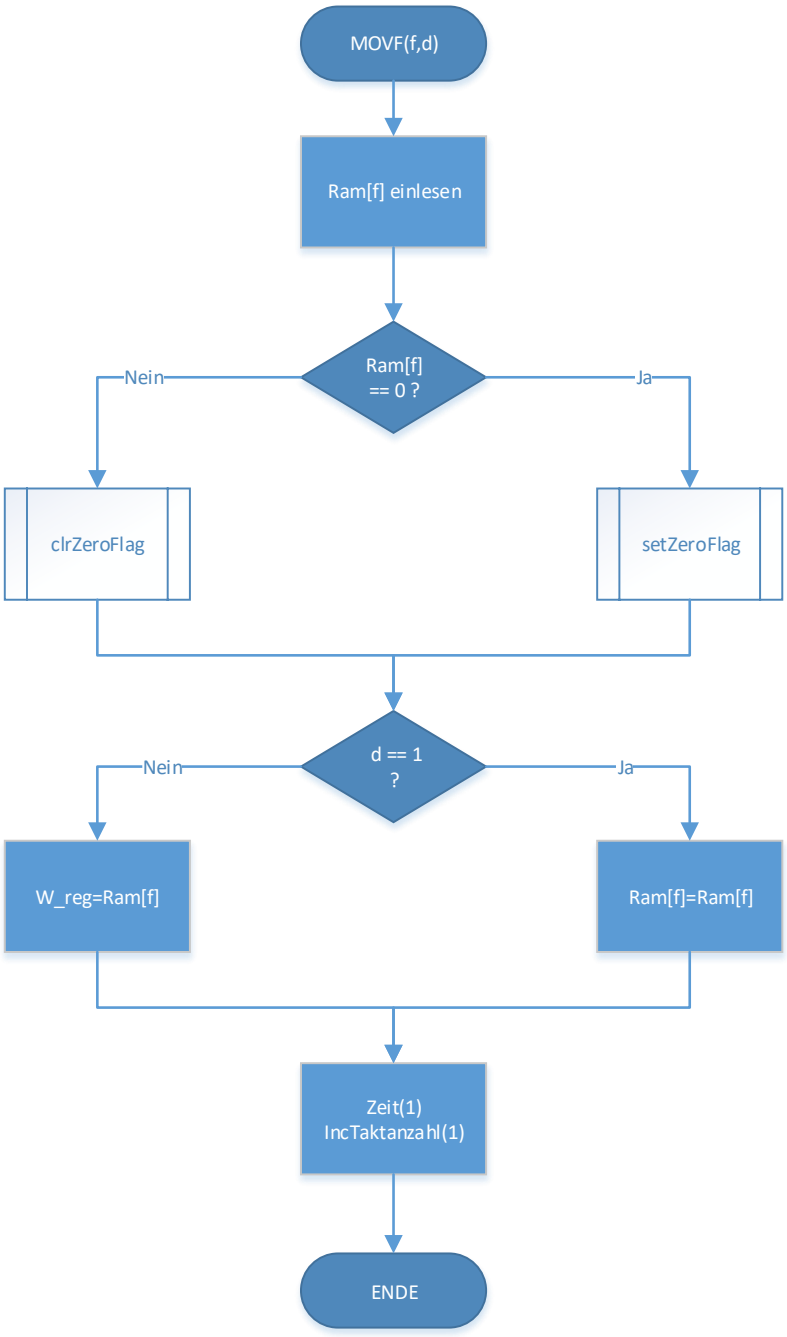
GOTO (k)

Der Goto Befehl verändert den Programcounter. Damit springt der Programcounter statt auf den nächsten Befehl, zu Addresse 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 Programcounter nach jedem Step diesen um 1 erhöht, wird das Gotoflag verwendet, um eine Manipulation des Programcounter anzuzeigen. Bei gesetztem Gotoflag wird der Programcounter nicht erhöht.


GOTO(k)
PCLath auslesen
PCLath4 und PCLath3 in dem Array PCLath43 speichern
PCLow = k
ProgrammCounter= PCLath43 konkateniert mit PCLow
GotoFlag= true
InstructionCounter= ProgramCounter
Zeit(2)
IncTaktanzahl(2)
Ende

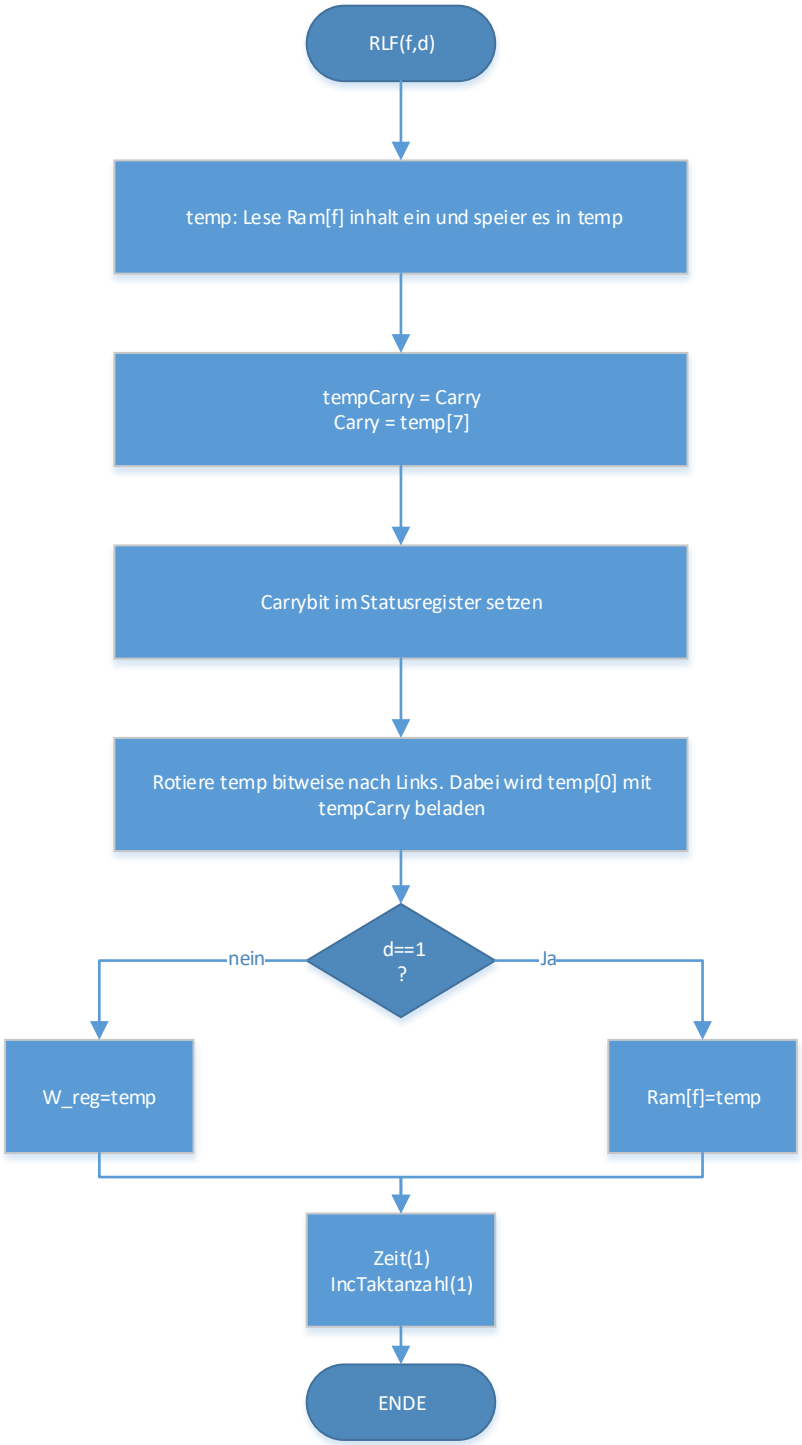
MOVF (f,d)

Der MOVE FILE Befehl liest den Register f wert 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. Im selbstimplementieren Programm wurde die setZeroFlag und die clrZeroFlag Funktion durch die ChangeZ Funktion ersetzt. 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.



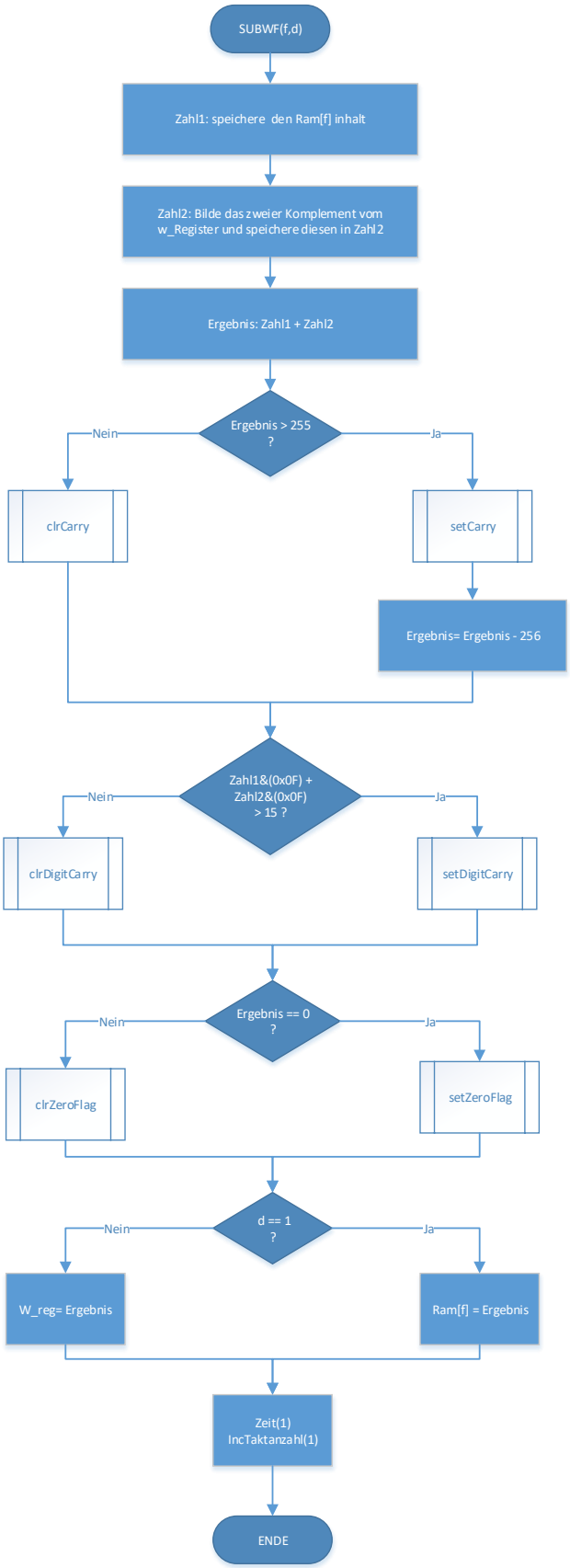
RLF(f,d)

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



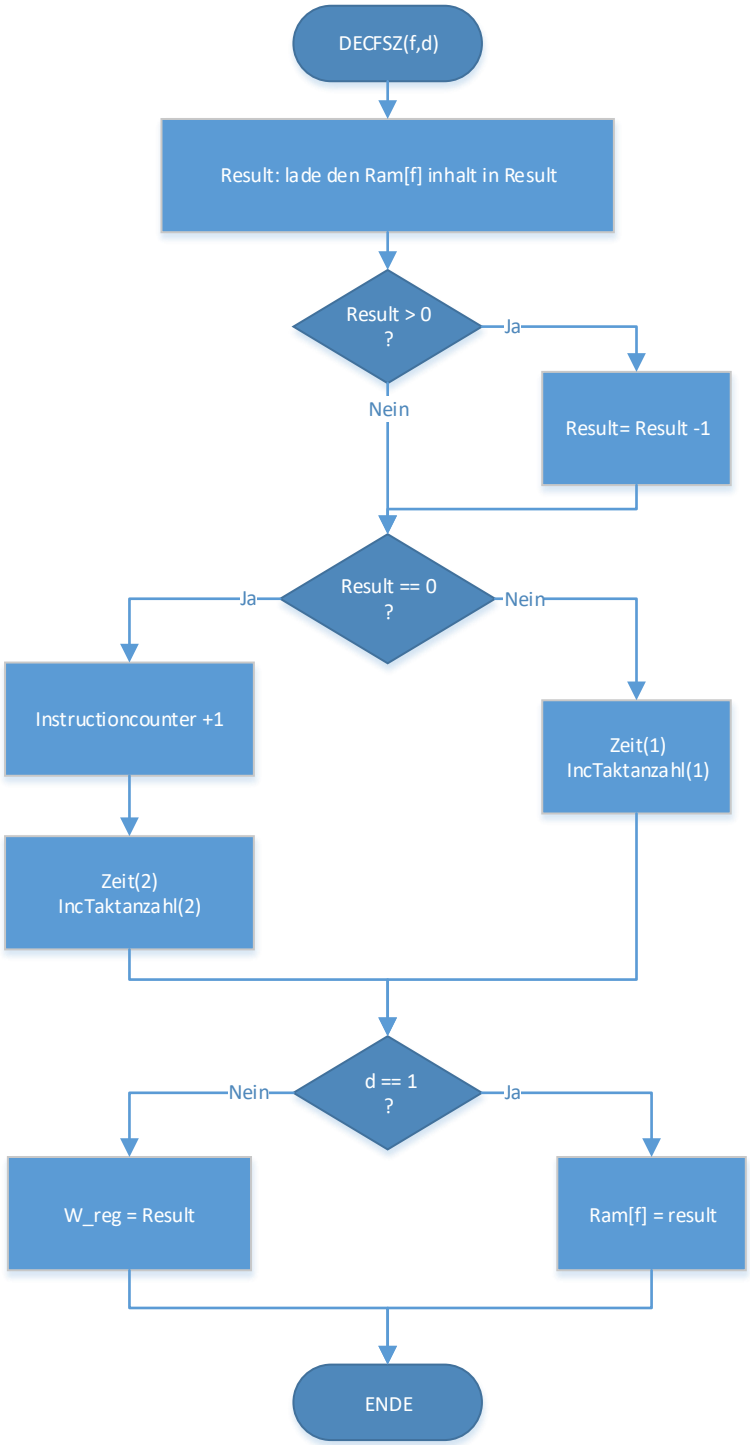
SUBWF (k)

Die Subtraktionsoperation des PIC Simulators nimmt den f- Register Inhalt als Zahl1 und addiert das zweier Komplement des w_Registers zu einem Ergebnis. 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.


SUBWF(f,d)
Zahl1: speichere den Ram[f] inhalt
Zahl2: Bilde das zweier Komplement vom w_Register und speichere diesen in Zahl2
Ergebnis: Zahl1 + Zahl2
Ergebnis > 255 ?
Nein
Ja
clrCarry
setCarry
Ergebnis= Ergebnis - 256
Zahl1&(0x0F) + Zahl2&(0x0F) > 15 ?
Nein
Ja
clrDigitCarry
setDigitCarry
Ergebnis == 0 ?
Nein
Ja
clrZeroFlag
setZeroFlag
d == 1 ?
Nein
Ja
W_reg= Ergebnis
Ram[f] = Ergebnis
Zeit(1)
IncTaktanzahl(1)
ENDE

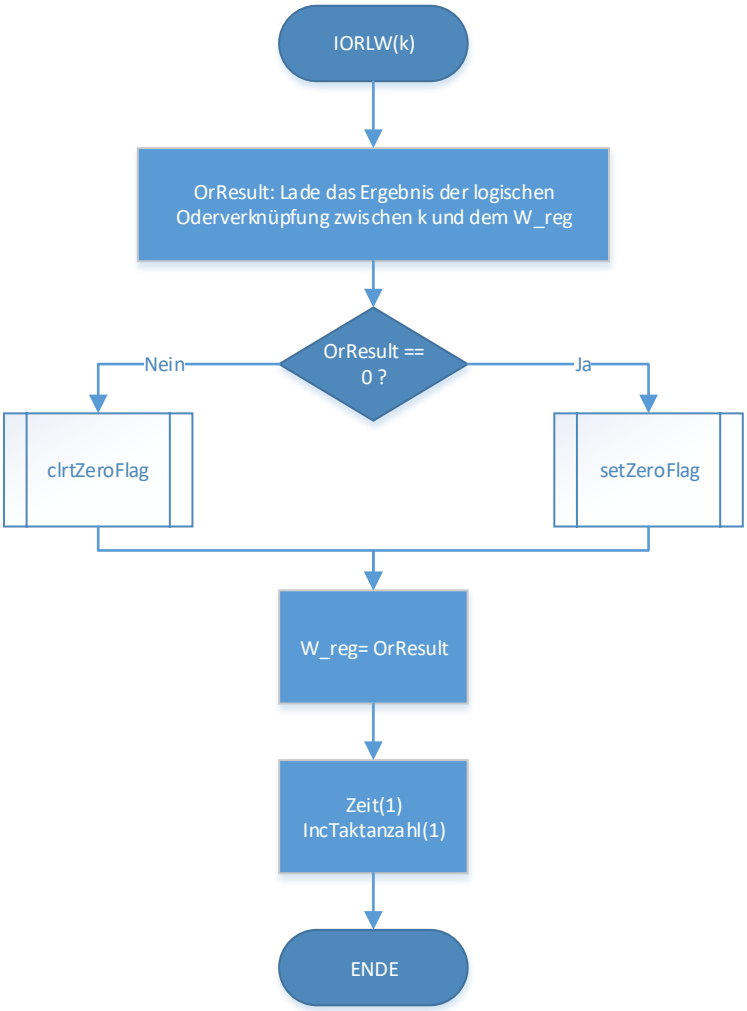
DECFSZ(f,d)

Lade den Speicherinhalt vom f- Register in die Resultat Variable. Falls diese Größer größer 0 ist, ziehe eins davon ab. Wenn das Resultat dannach gleich 0 ist, erhöhe den Programcounter um 1, damit wird der folge Befehl übersprungen. Abhängig vom d Bit wird das Ergebnis entweder ins 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.



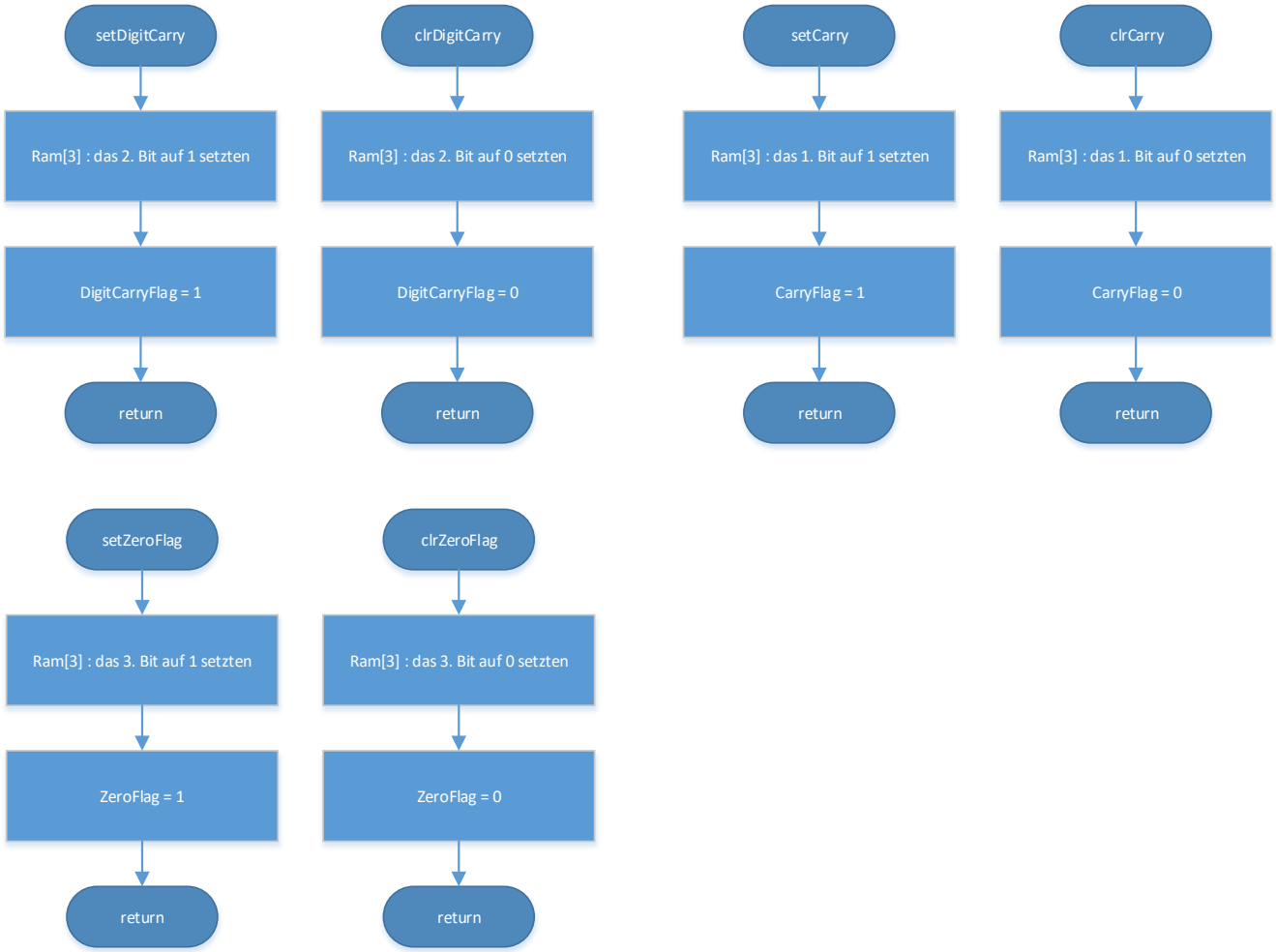
IORLW (k)

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.



9. Realisierung der Flags und deren Wirkungsmechanismen

Um die einzelnen Flags aus den Befehlen herauszufiltern wurden innerhalb der Klasse Commands.cs verschiedenste Hilfsmethoden implementiert. Diese Hilfsmethoden dienten zum einen für die Bestimmung des Wertes eines Flags oder zum Überschreiben eines Flags um dieses zu ändern.

```
public int ExtractCFlag()
{
    return Globals.bank0[3] & 0b0000_0001;
}
```

Beispielhaft an der Methode ExtractCFlag() wird die Funktionsweise einer solchen Hilfsmethode dargestellt. Innerhalb des Status-Registers befindet sich das sog. Carryflag. Dieses ist in unserem Fall in den Arrays bank0 bzw. Bank1 an der 3 Stelle. Um jedoch an das gewünschte Carryflag ohne die restlichen, nicht benötigten Informationen zu kommen muss dieses 8-bit große Register maskiert werden. Hierzu wird der Wert innerhalb des Arrays mit dem binären Wert 0000 0001 logisch UND-Verknüpft. Das Ergebnis ist das Carryflag, welches von der Hilfsmethode zurückgegeben wird.

```
public void ChangeC(int result)
{
    if(result > 255)
    {
        Globals.bank0[3] |= 0b0000_0001;
        Globals.bank1[3] |= 1;
    }
    else
    {
        Globals.bank0[3] &= 0b1111_1110;
        Globals.bank1[3] &= 0b1111_1110;
    }
}
```

Um das setzen eines Flags zu veranschaulichen wird die Hilfsmethode ChangeC() herangezogen. Diese verändert den Wert des Carryflags von 0 zu 1 oder umgekehrt. Dies ist jedoch davon abhängig, ob das innerhalb eines Befehls erzeugte Ergebnis größer 255 ist. Sollte dies der Fall sein findet ein Überlauf statt und das Carryflag wird auf beiden Bänken - bank0 und bank1 – zu 1 verändert. Anderfalls soll das Carryflag auf 0 bleiben.

10. Interrupt Timer0

Bei der Realisierung des Timer0 Interrupts wird überprüft, ob der Timer0 größer 255 wird. Sollte das der Fall sein, wird der Timer0 auf 0 zurückgesetz und das Timer0 Interrupt-Flag(T0IF) wird gesetzt.

```
public bool CheckInterrupt()
{
    int GIE0 = Globals.bank0[11] & 0b1000_0000;
    int T0IF0 = Globals.bank0[11] & 0b000_0100;
    int T0IE0 = Globals.bank0[11] & 0b0010_0000;
    int GIE1 = Globals.bank1[11] & 0b1000_0000;
    int T0IF1 = Globals.bank1[11] & 0b000_0100;
    int T0IE1 = Globals.bank1[11] & 0b0010_0000;
    if ((GIE0 | GIE1) != 0 && (T0IF0 | T0IF1) != 0 && (T0IE0 | T0IE1) != 0)
    {
        return true;
    }
    else
    {
        return false;
    }
}
```

Um zu überprüfen, ob tatsächlich ein Interrupt ausgelöst werden soll wurde die CheckInterrupt() Methode, welche einen bool'schen Wert zurückliefert implementiert. Hierbei werden zuerst die benötigten Flags durch Maskierung in Variablen gespeichert, welche vom Typ int sind. Anschließend wird überprüft, ob die GIE0, T0IF und T0IE Flags gesetzt sind. Dabei muss beachtet werden, dass die Flags auf beiden Bänken vorkommen. Falls die CheckInterrupt()-Methode True zurückliefert, wird der Interrupt ausgelöst, eine Meldung wird in der GUI ausgegeben und die Rücksprungadresse auf den Stack geschoben. Außerdem wird das Programm auf die Adresse vier zurückgesetzt, sowie dass das GIE-Flag zurückgesetzt wird.

```
if(CheckInterrupt() == true)
{
    Globals.stack.Push(Globals.programcounter);
    Globals.programcounter = 3;
    pc = 4;
    Globals.Interrupt = true;
    Globals.bank0[11] &= 0b0111_1111;
    Globals.bank1[11] &= 0b0111_1111; //resets GIE

}
```

Die Variable Globals.Interrupt dient dazu die Meldung auf der GUI zurückzusetzen, damit das Programm normal weiterlaufen kann.

11. Fazit

Die Programmierung eines Simulators für den PIC16F84 Microcontroller war eine gute Übung um die eigene Fähigkeit im Programmieren zu verbessern bzw. neue Programmiersprachen auszuprobieren und dadurch ebenfalls zu lernen. Durch den Einsatz verschiedener Tools wie zum Beispiel Microsoft Visual Studio Community und GitHub war es möglich, dass beide Teammitglieder gleichermaßen an dem Projekt arbeiten konnten, wodurch sich für beide ein tieferes Verständnis mit dem Umgang der Tools sowie für die Erzeugung von Code einstellte.

Im Verlauf des Projekts sind öfter Probleme aufgetreten, die im Voraus schon hätten behandelt werden können. Jedoch sind uns, aufgrund unserer mangelnden Programmiererfahrung, die kritischen Punkte nicht aufgefallen. Beginnend mit den Speicherbänken des PIC haben wir uns für eine Realisierung mittels zweier Arrays entschieden, die im Weiterem Verlauf für viel duplizierten Code sorgte. Des Weiteren haben wir uns bei der Implementierung diverser Befehle schwergetan, da wir noch keine Hilfsmethoden erzeugt haben. Diese Hilfsmethoden dienten dazu Adressen bzw. Einzelne Flags aus den Befehlen des .LST Dokuments zu extrahieren um mit diesen die Befehle korrekt abzuarbeiten. Ein weiterer Punkt, an welchem wir Probleme hatten war die Implementierung des Timers bzw. Des Watchdogtimers. Bei der Implementierung des Timers war zuerst unklar, wie das Zählen überhaupt funktioniert und wie der Prescaler diesen Timer beeinflusst. Jedoch konnten wir mittels des Datenblatts und externer Hilfe den Timer-mechanismus verstehen und anschließend realisieren. Eine weitere Hürde war die Implementierung der Grafischen Benutzeroberfläche (GUI) des Simulators. Zunächst war unklar, wie die Register und die toggle-baren I/O realisiert werden sollten. Als wir diese Hürde überwunden haben stellten wir fest, dass das nächste größere Problem anstand. Das Aktualisieren der Oberfläche. Hierzu mussten wir uns dem Thema “Threading” annehmen, um den Mechanismus der im Hintergrund des Programms abläuft zu verstehen und somit innerhalb der verschiedenen Prozesse die Oberfläche zu aktualisieren.

Zusammenfassend war es eine Spannende Aufgabe, die sehr fordernd aber auch gleichermaßen interessant war. Wenn wir das Projekt nochmals durchführen sollten würden wir einige Dinge anders machen. Zuerst wäre das legen eines roten Fadens ein wichtiger Punkt für die Entwicklung des Simulators, da uns dieser oftmals gefehlt hat bzw. wir nicht wussten, wo wir im Projekt weiter machen müssen. Ein anderer Punkt wäre das Erstellen der GUI bevor die Befehle realisiert werden um das Testen sowie die Übersichtlichkeit innerhalb des Projekts zu verbessern und zu erleichtern.