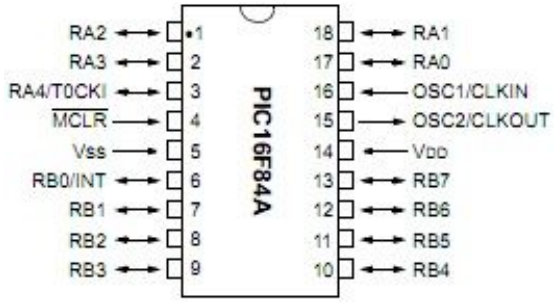
Dokumentation über die Programmierung eines Microcontroller

PIC16F84 Simulators

Bild eines PIC16F84, Quelle: 1)



Pin-Shema eines PIC16F84, Quelle: 2)

Inhaltsverzeichnis:

Allgemein

Worum geht es in dieser Dokumentation?

Für das Modul Systemnahe Programmierung soll ein PIC16F84 Mikrocontroller Simulator programm entwickelt werden. Diese Dokumentation soll festhalten, was ein PIC16F84 ist, wie ein Simulator funktioniert, wie unserer zu bedienen ist und wie wir unseren Simulator umgesetzt haben.
Das Ziel dieses Projektes, ist ein Simulator, der Assembler Code für einen PIC16F84 lesen und interpretieren kann. Dazu soll eine eigen Benutzeroberfläche (GUI - Guided User Interface) erstellt werden.

Was ist ein PIC16F84?

Ein PIC16F84 ist ein Mikrocontroller. Er stammt aus der PIC16CXX-Familie von kostengünstigen, leistungsstarken, CMOS, voll statischen, 8-Bit Mikrocontrollern.

Alle PICmicro ™ -Mikrocontroller verwenden eine fortschrittliche RISC-Architektur. PIC16F8X-Geräte haben verbesserte Kernfunktionen, einen achtstufigen Stack und mehrere interne und externe Interruptquellen. Die getrennten Befehls- und Datenbusse der Harvard-Architektur Erlauben ein 14 Bit breites Befehlswort mit einem separaten 8 Bit breiten Datenbus. Die zweistufige Anweisungspipeline ermöglicht die Ausführung aller Anweisungen in einem einzigen Zyklus, mit Ausnahme von Programmzweigen (für die zwei erforderlich sind). Insgesamt sind 35 Anweisungen (reduziertes Anweisung Set) verfügbar. Zusätzlich ist ein großer Registersatz vorhanden und wird verwendet, um ein sehr hohes Leistungsniveau zu erreichen.
PIC16F8X-Mikrocontroller erreichen normalerweise eine 2: 1-Code Komprimierung und eine bis zu 4: 1 Geschwindigkeitsverbesserung (bei 20 MHz) gegenüber anderen 8-Bit-Mikrocontrollern ihrer Klasse. Der PIC16F8X verfügt über bis zu 68 Byte RAM, 64 Byte RAM Daten-EEPROM-Speicher und 13 E / A-Pins. Ein Timer / Zähler ist ebenfalls vorhanden.

Wie funktioniert eine Simulation?

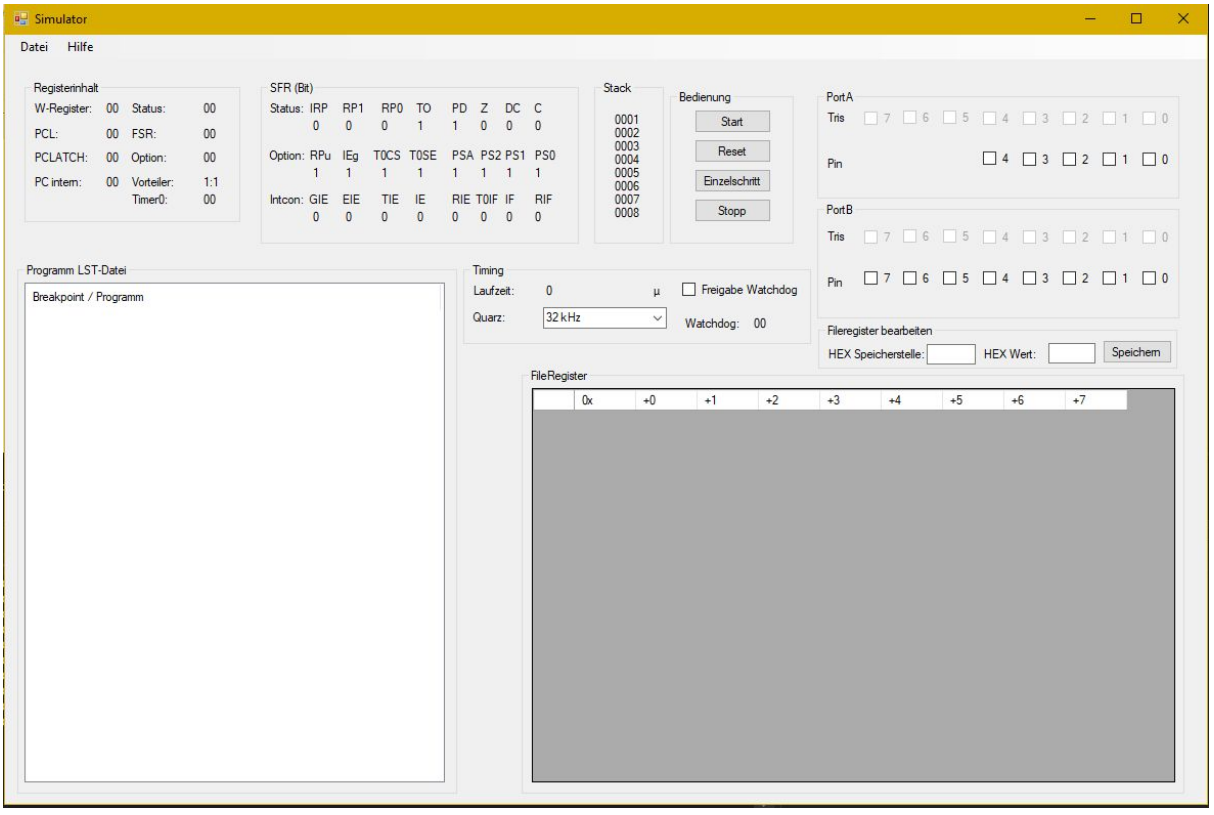
Eine Simulation im Softwarebereich ist eine möglichst originalgetreue Nachbildung von internen Abläufen und Prozessen. Im Gegensatz zu Emulationen ist nicht nur das Ergebnis wichtig sondern die möglichst präzise Nachbildung des Originals.

Jeglicher Form von Simulation sind auch Grenzen gesetzt, die man stets beachten muss. Die erste Grenze folgt aus der Begrenztheit der Mittel, das heißt der

Endlichkeit von Energie (zum Beispiel auch Rechenkapazität) und Zeit. Eine Simulation muss auch wirtschaftlich gesehen Sinn ergeben. Aufgrund dieser Einschränkungen muss ein Modell möglichst einfach sein. Das wiederum bedeutet, dass auch die verwendeten Modelle oft eine grobe Vereinfachung der Realität darstellen. Diese Vereinfachungen beeinträchtigen naturgemäß auch die Genauigkeit der Simulationsergebnisse.

Die zweite Grenze folgt daraus: Ein Modell liefert nur in einem bestimmten Kontext Ergebnisse, die sich auf die Realität übertragen lassen. In anderen Parameter Bereichen können die Resultate schlichtweg falsch sein. Daher ist die Validierung der Modelle für den jeweiligen Anwendungsfall ein wichtiger Bestandteil der Simulationstechnik. Als mögliche weitere Grenzen seien Ungenauigkeiten der Ausgangsdaten (etwa Messfehler), sowie subjektive Hindernisse (zum Beispiel mangelnder Informationsfluss über Produktionsfehler) genannt.

Bedienungsanleitung



Bei unserem Simulator haben wir auf eine eigene GUI gesetzt, die sich intuitiv steuern lassen soll. Dafür wurden alle Kriterien aus dem Bewertungsschema umgesetzt. Im Groben besteht die GUI aus einem Oberen, einem Linken und einem Rechten Bereich. Im Linken wird das Programm angezeigt, welches aktuell simuliert wird, mit einem gelben Balken wird dabei markiert, welche Zeile aktuell bearbeitet wird.
Rechts befindet sich die Anzeige des Speichers des PIC. Hier können alle Speicherzellen in Dezimaldarstellung betrachtet werden und über die Eingabe von Hexadezimalwerten bearbeitet werden. Zudem befinden sich in dem Bereich auch die Timing Einstellungen, bei denen angezeigt wird, welche Laufzeit das Programm hat und ob der Watchdog gesetzt wurde. Die Quarzfrequenz kann man dabei selbständig bestimmen und der Watchdog lässt sich manuell freigeben.
Im Oberen Bereich befinden sich die Anzeigen der Register, der SFR(Status Flag Register) , des Stacks und Port Register für beide Speicher Bausteine. Ebenfalls befinden sich dort die Bedienelemente für das geladene Programm. Dieses lässt sich über **Start** komplett ausführen und durch **Stopp** jederzeit beenden. Über **Einzelschritt** lässt sich ein Programm Schritt für Schritt abarbeiten. Über **Reset** lässt sich die Ausgabe eines Programmes löschen.
Um ein Programm zu laden muss im oberen Reiter Datei gewählt werden. Dies öffnet ein Dropdown Menü. Über dieses kann mit der Option “Laden” ein Explorer Fenster geöffnet werden in dem man sein Assemblerprogramm auswählen kann. Die Hilfe funktion ist aktuell noch nicht verfügbar.
In unserem Simulator lassen sich nur bereits assemblierte PIC16F84 Programm ausführen.
Beenden lässt sich der Simulator über das Schließen des Fensters.

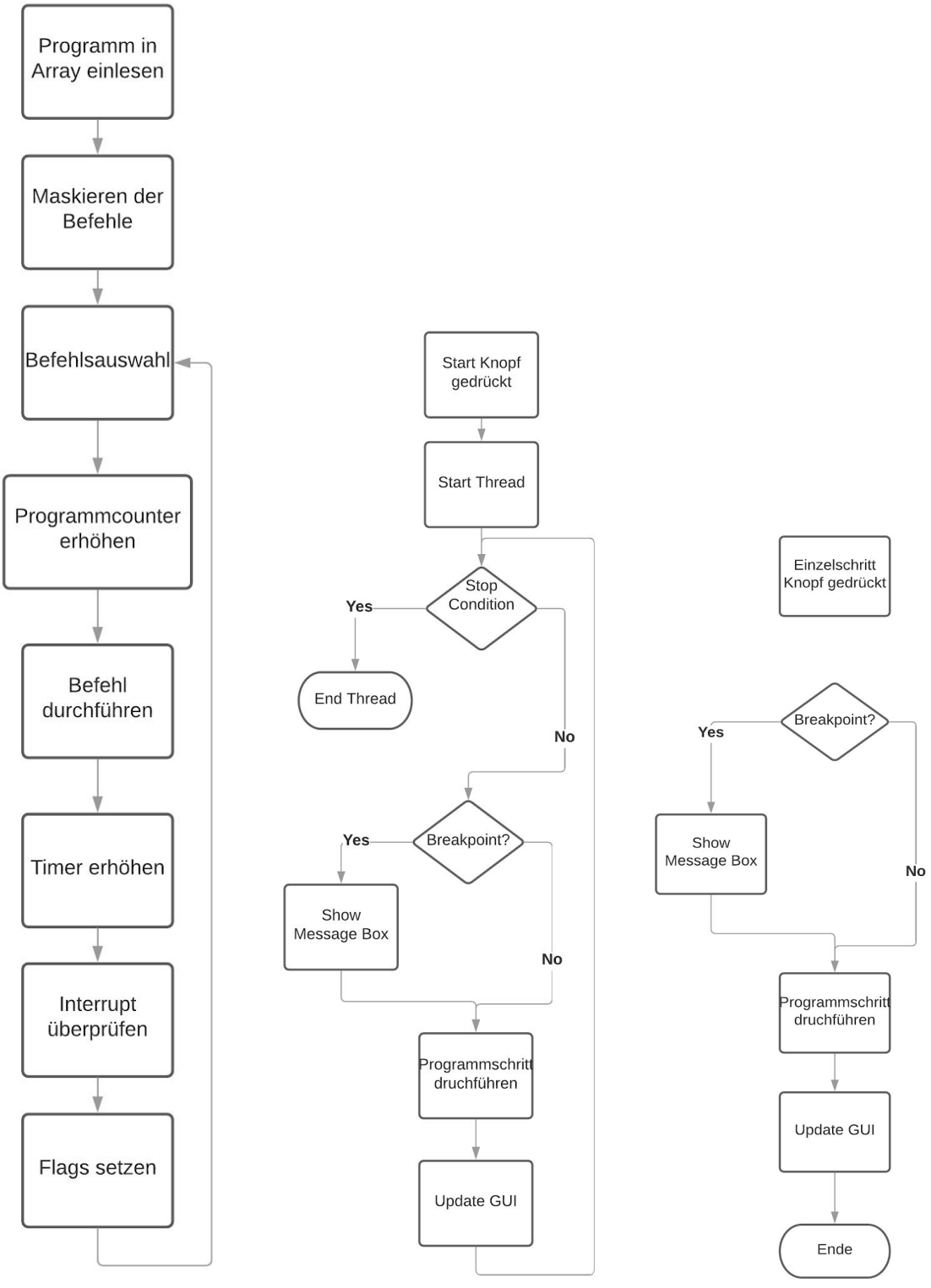
Umsetzung

Zu Beginn des Projektes haben wir die Funktionen des Mikrocontrollers in Methoden aufgeteilt. Pro Funktion haben wir eine Klasse erstellt, die für diese Funktion zuständig ist. So haben wir eine logische Gliederung des Projektes erstellt. In

unserem Konzept soll jedes Programm zeilenweise abgearbeitet werden. Jede Zeile wird nach dem Lesen maskiert um zu bestimmen mit welcher Methode sie ausgeführt werden muss.
Zur Umsetzung haben wir die Entwicklung wie folgt gegliedert:

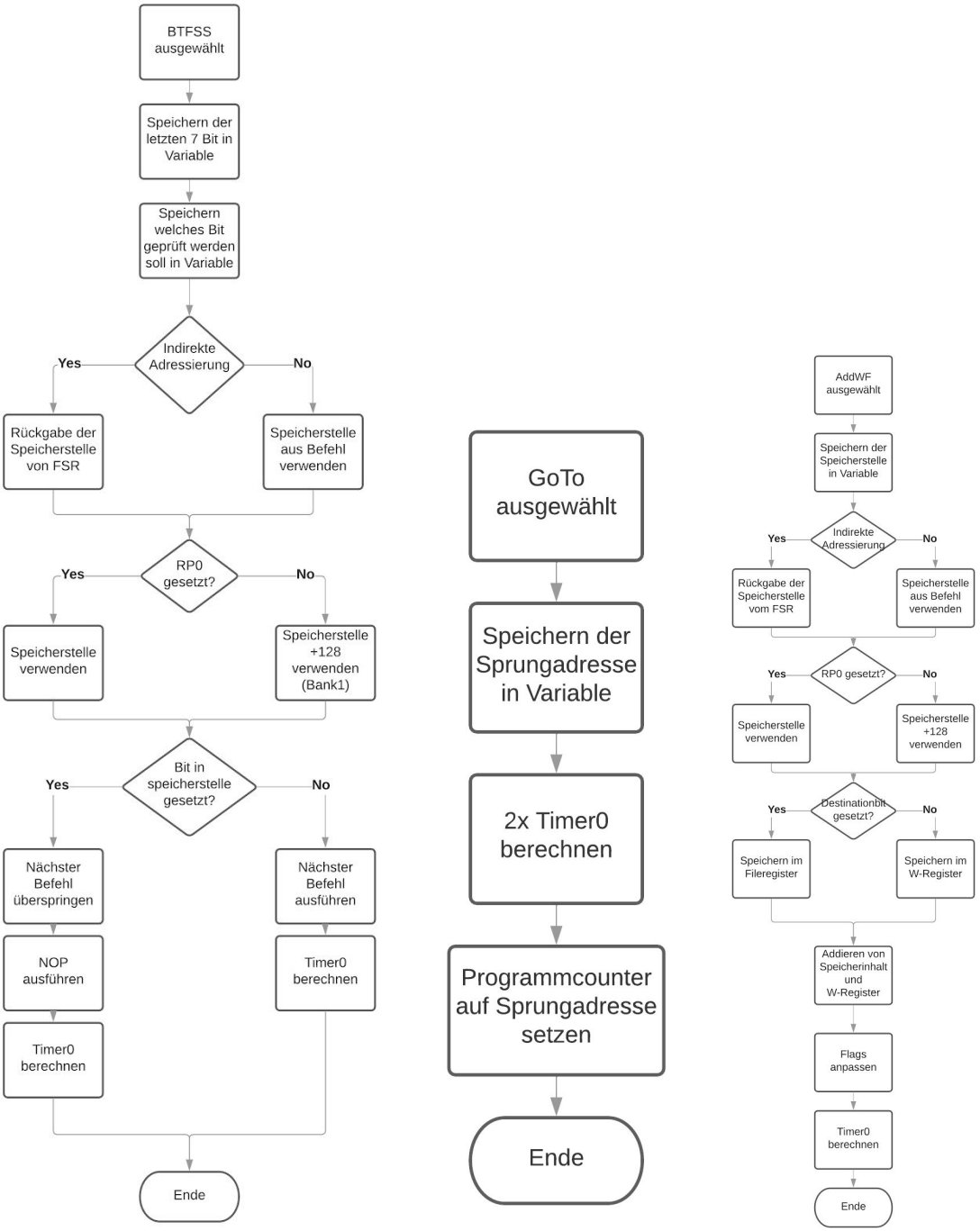
1. Lesen und Parsen der Programme
    1.1. Die Programme sollen aus dem Quelldokument geladen werden und nur der für den Simulator relevante Teil in ein Array geschrieben werden.
2. Maskierung der Programme
    2.1. Die im Array befindlichen Programme sollen Zeile für Zeile analysiert und jeweils nach vorgegebenen Mustern bearbeitet werden. Dabei sollen nach den 35 verschiedenen Kommandos des PIC unterschieden werden.
3. Methoden zur behandlung der Programme
    3.1. Die Kommandos werden nach der im Datenblatt des PIC gegebenen Beschreibung bearbeitet und die angegebenen Operationen ausgeführt.
4. Setzen der Flags
    4.1. Die im Datenblatt beschrieben Flags werden anhand der Operationen gesetzt und können gelesen werden.
5. Watchdog und Interrupts

Der Watchdog und die Interrupts können gesetzt werden und lösen automatisch aus wenn gewünscht.
Unser Simulator durchläuft nacheinander diese Abfragen.


Programm in Array einlesen
Maskieren der Befehle
Befehlsauswahl
Programmcounter erhöhen
Befehl durchführen
Timer erhöhen
Interrupt überprüfen
Flags setzen
Start Knopf gedrückt
Start Thread
Stop Condition
Yes
End Thread
No
Breakpoint?
Yes
Show Message Box
No
Programmschritt druchführen
Update GUI
Einzelschritt Knopf gedrückt
Breakpoint?
Yes
Show Message Box
No
Programmschritt druchführen
Update GUI
Ende

Ablauf eines Programmes in unserem Simulator und Strukturdiagramme der Funktion unserer Knöpfe.

Für die Programmierung haben wir uns entschieden C# zu verwenden, da C# unseren Klassenaufbau unterstützt und durch Windows Forms sich einfache GUIs entwickeln lassen. Zudem hatten wir bereits Erfahrung mit dieser Programmiersprache. Als IDE verwendeten wir Visual Studio, da wir ebenfalls Erfahrung mit ihr haben und Zusammenarbeit über Internet möglich ist.
Die meisten Methoden ließen sich über logische Operatoren einfach umsetzen. besonders fordernd waren hingegen die Befehle, BTFSx, GoTo und ADDWF. Diese haben wir folgendermaßen umgesetzt:



Umsetzung BTFSS, GoTo und MOVWF.

Wir haben den Interrupt Timer0 wie folgend realisiert.

```
if (Global.Bank1[1] >= 255)
{
    //TimerInterrupt
    Global.Bank1[11] = Global.Bank1[11] ^ 0b0000_0100;
    Global.Bank1[139] = Global.Bank1[139] ^ 0b0000_0100;
    Global.Bank1[1] = 0;
    Global.push(programmcounter+1);

    //wenn GIE dann pc-> adresse 4
    if (getGIE() == 1)
    {
        return 4;
    }
```

Findet ein Overflow des Timer0 statt, wird im INTCON register das zweite Bit (T0IF) und der Timer wird auf Null zurückgesetzt. Ist das Global Interrupt Enable Bit (GIE) im INTCON Register gesetzt. Der Programmcounter wird dann auf den Stack gepusht und die Interrupt Routine an Programmcounterstelle 4 aufgerufen.

```
public bool RB0Interrupt(int portA, int portB, int programmcounter)
{
    int eingang = Global.Bank1[134] & 0b0000_0001;
    if (eingang == 1)
    {
        if (getGIE() == 1)
        {
            if (getIE() == 1)
            {
                Console.WriteLine("RB0/RB Interrupt");
                int interrupt = Global.Bank1[6] & 0b0000_0001;
                if (interrupt == 1)
                {
                    Console.WriteLine("RB0 Interrupt");
                    //RB0 Interruptflag
                    Global.Bank1[11] = Global.Bank1[11] ^ 0b0000_0010;
                    Global.Bank1[139] = Global.Bank1[139] ^ 0b0000_0010;
                    Global.push(programmcounter);
                    return true;
                }
            }
        }

    }

    return false;
}
```

Der RB0 Interrupt wird ausgelöst, wenn das Global Interrupt Enable Bit (GIE) und das Interrupt Enable Bit (INTIE) gesetzt ist. Wenn es eine Flankenänderung am RB0 Pin gibt, wird der Interrupt ausgelöst. Dabei wird dann das RB0 Interrupt Flag gesetzt, der Programmcounter auf den Stack gepusht und die Interrupt Routine an der Stelle 4 im Programmcode aufgerufen.

Der RB Interrupt der oberen vier Bits von Port B wird gleich wie der Interrupt von RB0 aufgerufen. Dabei wird auf jede Änderung eines der vier Bits reagiert, das auf Eingang geschalten ist. Es müssen um einen solchen Interrupt ausführen zu können das Globale Interrupt Enable Bit (GIE) und das RB Port Change Interrupt Enable bit (RBIE) gesetzt sein. Der Interrupt setzt das RB Port Change Interrupt Flag bit (RBIF) Bit, pusht den Programmcounter auf den Stack und ruft die Interrupt Routine an Stelle 4 im Programmcode auf.

Die Trisregister sind in der GUI durch Checkboxen realisiert. sie können nicht bearbeitet werden. Die Trisregister werden durch die Speicherstellen im Fileregister beeinflusst. Die Pins von PortA und PortB können durch klicken angesteuert werden.

Fazit

Der Simulator des PIC16F84 kann die wichtigsten funktionen des Mikrocontrollers abbilden und simulieren. Hätten wir mehr Zeit könnten wir theoretisch alle Funktionen des Mikrocontrollers darstellen und per Software nachstellen. Problematisch sind Flags und Interrupts, da diese oft abgefragt werden müssen.
Durch die Zusammenarbeit in der Gruppe, sowie mit anderen Gruppen konnten die meisten Probleme, auf die wir gestoßen sind schnell behoben werden. Unsere vorgehensweise war dabei Schritt für Schritt und hat geholfen dieses Projekt voran zu bringen. Dieses Projekt hat unsere Programmierkenntnisse stark verbessert und

dabei geholfen die internen Abläufe eines Mikrocontrollers besser zu verstehen. Besonders Interruptmechaniken wurden dadurch verständlich, dies hat auch dazu geführt, dass die Funktionsweise eines Computers, der Harvard Architektur, detaillierter verstanden wurde.

Wenn wir dieses Projekt erneut realisieren müssten, würden wir am gleichen Standort arbeiten, da durch die Zusammenarbeit per Internet öfters Probleme in der Kommunikation entstehen können. Abstürze und Netzeinbrüche blieben dabei keine Seltenheit und verzögerten das Projekt. Zudem würden wir vorher eine klare Strukturierung verwenden und Features zum Refactoring verwenden um unseren Code übersichtlich und strukturiert zu halten.

Eins unsere Probleme ist der unübersichtliche Code, der Aufgrund des sehr hohen Umfangs dieses Projektes entstanden ist. Größere Probleme gab es bei unserem Simulator nicht, das häufigste Problem waren Missverständnisse über die Arbeitsweise des PIC und schlechte Programmierkenntnisse. Aber auch diese ließen sich meist einfach beheben.

Während der Entwicklung des Projektes merkten wir das C# Ideal für die Programmierung von Hardware-Simulatoren ist, da es sehr hardwarenahe Befehle unterstützt und sich schnell eine übersichtliche GUI erstellen lässt.

Quellen

Bilder:

1) https://www.amazon.ca/PIC18F84J90-I-PT-FLASH-80TQFP-PIC18F84/dp/B015SXPODQ
2) https://spiratronics.com/images/techim/PIC16F84A%20Pin%20Out.jpg