

Dokumentation

Pic Simulator

Entwickler:	Tobias Bühler, Toni Einecker
Kurs:	TINF18B3
Version:	1.0
Datum:	07.06.2020

Inhalt

1	Einleitung.....	3
2	Was ist ein Simulator.....	3
3	PICSimulator666	4
4	Architektur.....	5
4.1	GUI.....	5
4.1.1	Mnemonic View.....	5
4.1.2	ErrorDialog.....	6
4.2	Controller.....	6
5	Realisierung von Grundfunktionalitäten	7
5.1	Befehle.....	7
5.2	Timer	8
5.3	Interrupts.....	8
5.4	Watchdog	9
5.5	Prescaler	9
5.6	EEPROM.....	9
5.7	TRIS-Register.....	10
6	Hardwareansteuerung	11
7	Fazit	12

1 Einleitung

Dieses Software Projekt wurde im Rahmen der Vorlesung Systemnahe Programmierung 2, innerhalb von 9 Wochen, durchgeführt. Ziel sollte es sein, den Microcontroller PIC16F84 als Software nach zu implementieren. Hierzu standen Lehrmittel der Vorlesung, das Datenblatt des Controllers sowie wöchentliche Treffen mit dem Dozenten zur Verfügung.

Genutzt wurde hierfür Java, da mittels Eclipse eine simple Oberflächen Erstellung möglich ist und Java eine moderne viel genutzte Programmiersprache ist, in der wir unsere Fähigkeiten erweitern wollten. Für eine Versionierung sowie die Möglichkeit zusammen zu Programmieren und den Programmcode auszutauschen haben wir uns für die online Lösung GitHub entschieden.

2 Was ist ein Simulator

Hinter dem Begriff „Simulator“ versteckt sich das Wort Simulation und meint die Analyse eines bestimmten Systems. Ein Simulator ist hier die Implementierungsmethode zur Durchführung von Systemrelevanten Ereignissen. Das in diesem Projekt zugrundeliegende System ist der PIC16F84, welches ein 8-Bit Microcontroller der Firma Microchip Technology ist.

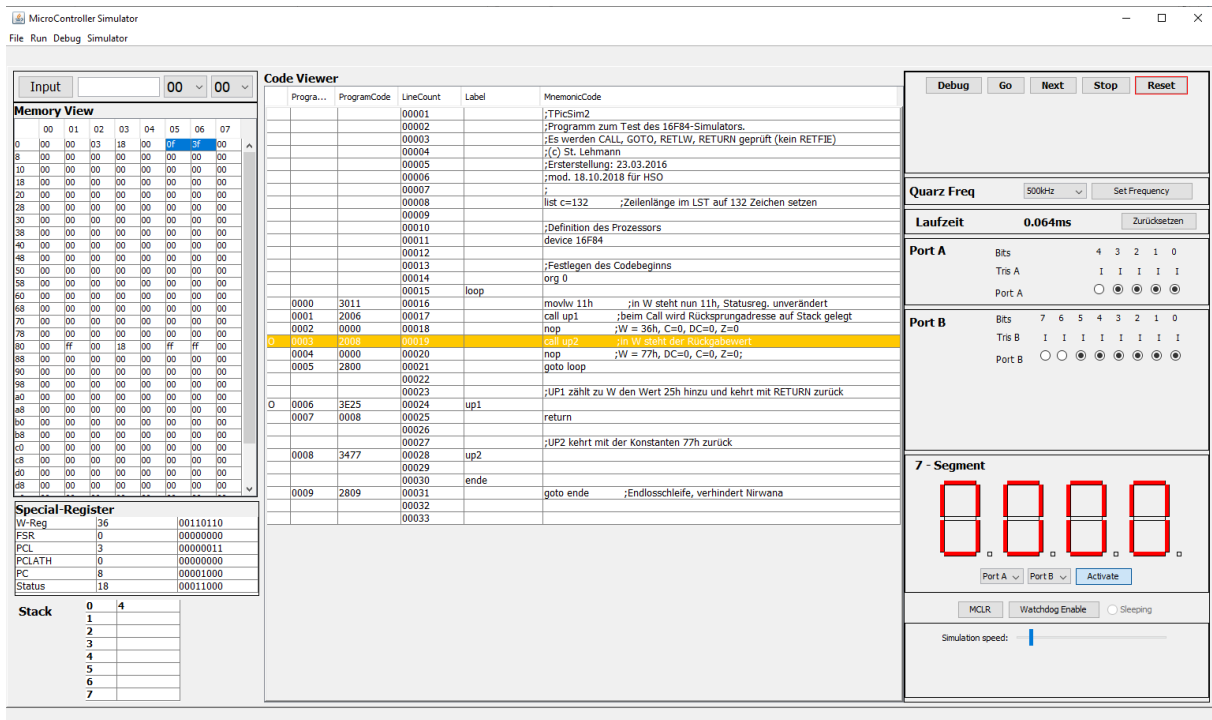
Ein Simulator ist immer dann nützlich, wenn die Untersuchung am realen System zu aufwendig oder zu teuer ist. Ebenfalls können Simulatoren benutzt werden, wenn eine Gefährdung von Mensch, Maschine oder Umwelt vorliegt.

Um beispielsweise ein Assemblerprogramm auf der richtigen Hardware testen zu können, muss der Code zuerst auf die Hardware übertragen werden. Voraussetzung ist allerdings, dass eine Hardware verfügbar ist. Mit einer Simulation der Hardware kann nun das Assemblerprogramm auch ohne Hardware getestet werden.

Als Nachteil ist anzumerken, dass äußere Einflüsse wie Temperatur oder Spannungsschwankungen nicht oder nur bedingt mit in die Simulation einfließen können und dadurch das wahre Verhalten verfälscht dargestellt werden kann.

3 PICSimulator666

Zu sehen ist ein dreigeteiltes Fenster. In der Mitte ist der Code des aktuellen Programms zu sehen. Der Breakpoint wurde durch einen Klick auf die entsprechende Zeile gesetzt, wodurch die gelbe Marke, welche die aktuell auszuführende Programmzeile markiert, bei der Zeile 19 stehen geblieben ist. Bei diesem Befehl (call up2) wurde dementsprechend die darauffolgende Adresse auf den Stack gepusht (links unten).

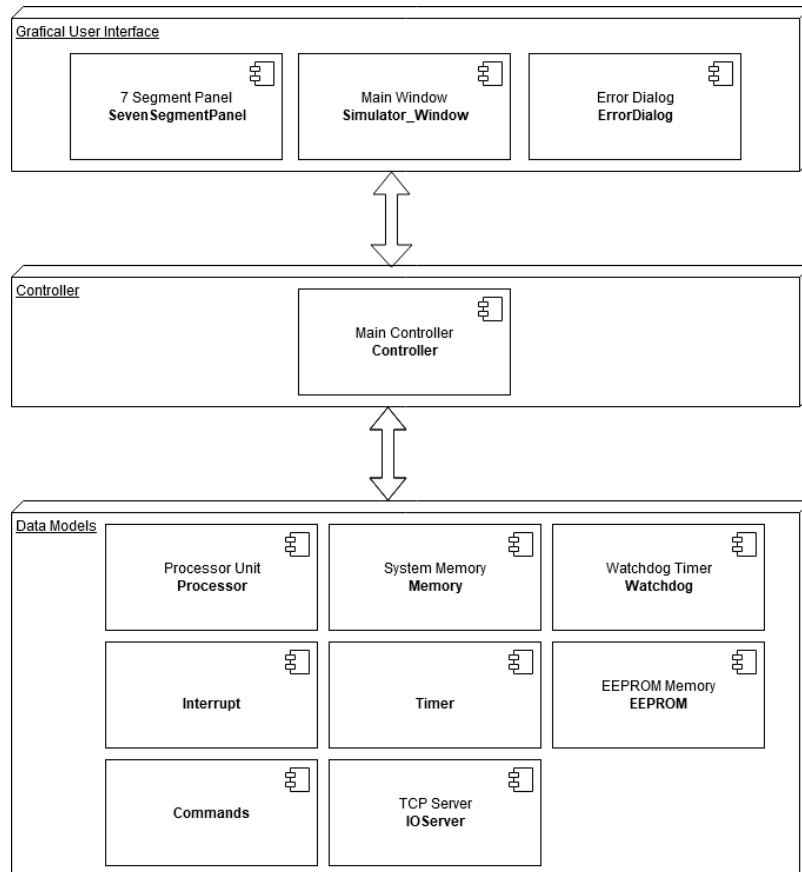


Im linken Teil des Fensters ist der Speicher als Tabelle abgebildet. Änderungen sind in diesem durch den Input Button möglich. Ebenfalls ist hier zu sehen, dass die Änderungen an Port A und B zu einer Änderung in der Memory View geführt hat. Dort sind dementsprechend die zuletzt geänderten Register blau hinterlegt. Eine detailliertere Ansicht wichtiger Register ist darunter abgebildet.

Im rechten Teil befinden alle Schaltflächen zum Interagieren mit dem Simulator. Die wichtigsten Knöpfe zum Starten, Stoppen, Debuggen und Resetten befinden sich ganz oben. Der Reset setzt den Simulator auf den Power-Up zustand zurück. Neben weiteren Kontroll-Schaltflächen und einer Laufzeit Anzeige, befinden sich hier auch die IO-Ports und eine, in diesem Beispiel, aktive Sieben-Segment Anzeige welche den am Port B anliegenden Wert an den durch Port A selektierten Ziffern ausgibt.

4 Architektur

Die Software Architektur basiert auf dem 3-Schichtenmodell. Hierbei bildet die Grafische Oberfläche die erste Schicht. Diese kommuniziert mit der darunterliegenden Controllerschicht. Der Controller bietet dann die Schnittstelle zwischen den Daten und Modellklassen als dritte Schicht.



4.1 GUI

Die Grafische Oberfläche besteht aus einer Single-Window-Application. Dies bedeutet, dass alles im Hauptfenster der Anwendung zu erreichen ist. Einzig Dialoge zum Öffnen oder Speichern von Dateien werden geöffnet, tragen allerdings nicht zur Funktion der Simulation bei.

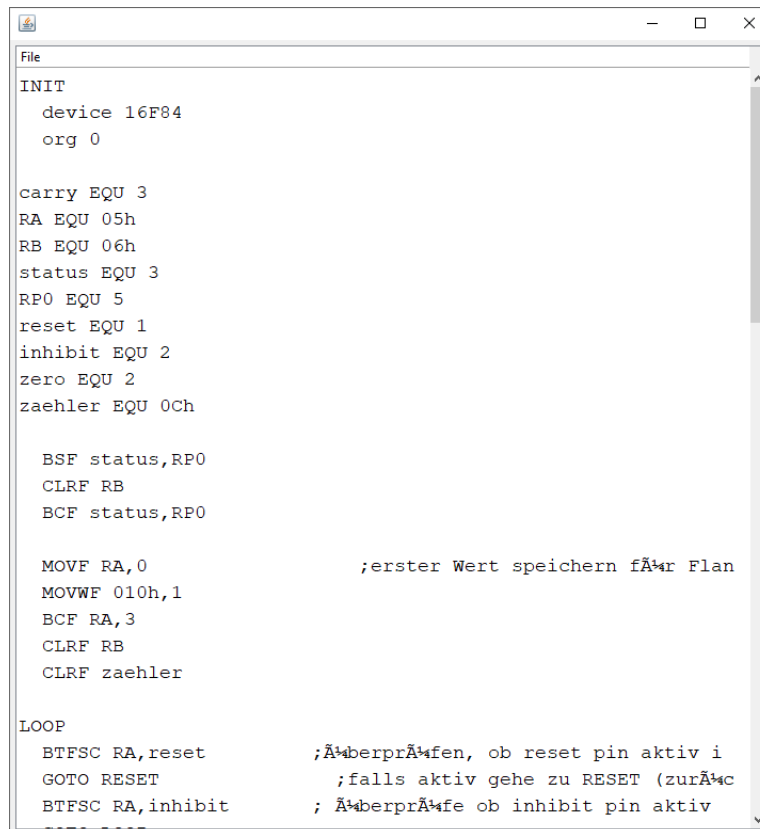
Die GUI lässt sich unterteilen in drei Spalten. Links befinden sich alle Speicherrelevanten Panels, bestehend aus GPR, SFR und Stack. In der mittleren Spalte befindet sich der geladene Code. In dieser Tabelle kann mit Klick auf die linke Tabellenspalte ein Breakpoint gesetzt werden, sofern sich dort valider Code befindet. In der rechten Spalte der GUI befinden sich Buttons zum Starten, Stoppen sowie Nutzen des Debuggers. Ebenfalls befindet sich dort die Auswahl der Quartzfrequenz sowie die Anzeige der bisher vergangen Zeit seit Programmstart. Darunter sind Panels für Port A und Port B sowie eine Sieben Segment Anzeige angeordnet.

Zusätzlich zur Hauptoberfläche werden die im folgenden beschriebenen Unterfenster benötigt.

4.1.1 Mnemonic View

Das Mnemonic View Fenster soll die Möglichkeit bieten, Assembler Code direkt im Simulator zu schreiben. Dieser soll auch direkt zu Binärcode geparkt werden können. Zusätzlich bietet das Fenster die Möglichkeit SRC Files zu speichern sowie zu laden. Das Öffnen, sowie korrekte einlesen des Codes

in das Mnemonic View Fenster funktioniert fehlerfrei. Aufgrund von technischen Komplikationen sowie Zeit Problemen ist das Parsen des Mnemonic Textes zu Code nicht vollständig funktionsfähig.



```
File
INIT
    device 16F84
    org 0

carry EQU 3
RA EQU 05h
RB EQU 06h
status EQU 3
RPO EQU 5
reset EQU 1
inhibit EQU 2
zero EQU 2
zaehler EQU 0Ch

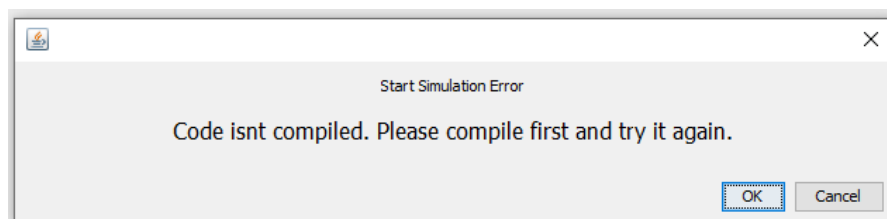
    BSF status,RPO
    CLRF RB
    BCF status,RPO

    MOVF RA,0                ;erster Wert speichern für Flag
    MOVWF 010h,1
    BCF RA,3
    CLRF RB
    CLRF zaehler

LOOP
    BTFSC RA,reset           ;Überprüfen, ob reset pin aktiv ist
    GOTO RESET              ;falls aktiv gehe zu RESET (zurück)
    BTFSC RA,inhibit         ;Überprüfen ob inhibit pin aktiv
```

4.1.2 ErrorDialog

Falls Fehler in der Software auftauchen, soll die Software nicht ohne Fehlermeldung abstürzen. Auch Bedienungsfehler müssen dem Nutzer kenntlich gemacht werden. Hierfür bietet der Error Dialog die Möglichkeit ein Dialogfenster zu öffnen und darin eine Fehlermeldung darzustellen.



4.2 Controller

Der Controller bietet die Schnittstelle zwischen der grafischen Oberfläche und den untergeordneten Komponenten des Controllers.

Die Aufgaben dieser Einheit besteht im Initialisieren der einzelnen Objekte sowie im gegenseitigen Zugriff. Hierbei ist das Prinzip der Datenkapselung zu beachten. Hierfür stehen für alle Zugriffe sogenannte Getter und Setter Funktionen zur Verfügung, um einen direkten Zugriff auf Variablen zu vermeiden.

Wie unter dem Punkt Architektur/Gesamt zu sehen ist, werden Objekte aller Modellklassen im Controller erzeugt. Durch einen angepassten Konstruktor in den Modellklassen kann diesen die

Referenz zur Controller Instanz übergeben werden, wodurch sich dann eine Bidirektionale Assoziation zwischen Controller und den Modellklassen ergibt.

Außerdem behandelt der Controller die Events, die durch einen Buttondruck in der grafischen Oberfläche ausgelöst werden. Zu diesen Events zählen unter anderem:

- Öffnen und einlesen von Dateien
- Aktualisieren der GUI
- Überwachen von Zuständen

Der Controller ist auch für die Initialisierung aller Objekte zuständig, sodass alle Tabellen der GUI richtig beschriftet und mit Werten versehen sind.

5 Realisierung von Grundfunktionalitäten

Die Grundfunktionalitäten des Simulators liegen in der Realisierung von Abläufen wie der Befehlsabarbeitung sowie Timer oder Interrupt Behandlungen.

5.1 Befehle

Jeder Befehl wie er im Datenblatt zu finden ist, wurde als eigene Funktion implementiert. Von diesen werden im Folgenden beispielhaft einige beschrieben.

Die Aufrufe dieser Funktionen erfolgen über die Klasse „Processor“. Dort wird der aktuelle Befehlscode aus dem „Program Memory“ geladen und anschließend an die Funktion „executeCommand()“ übergeben. Dort wird über die ersten Bits der genaue Befehl bestimmt und dementsprechend dessen Funktion aufgerufen.

Die Implementierung der Funktion „movf()“ sieht wie folgt aus:

```
private void movf(int d, int f)
{
    int f_in = ctr.getMemory().get_Memory(f);

    ctr.checkZeroFlag(f_in);
    if(d == 0)
    {
        ctr.getMemory().set_WREGISTER(f_in);
    }else if(d == 1)
    {
        ctr.getMemory().set_SRAM(f, f_in);
    }
    this.ctr.incPC();
}
```

Hier ist zu sehen, dass als Übergabeparameter das Destination-Bit sowie die Speicherstelle übergeben werden. Je nach Wert des Destination-Bits wird entweder (bei d = 0) ein Wert aus dem Register an der Stelle f ins W-Register geladen oder (bei d = 1) der Wert des W-Registers an die Registerstelle f geschrieben.

Der im folgenden gezeigte btfsc Befehl prüft ein Bit in einem Register und überspringt den nächsten Befehl falls dieses eine null ist. Dafür wird zuerst der Wert des Bits aus dem Speicher geladen und auf null geprüft. Falls der Wert gleich Eins ist wird der lediglich der Programmzähler um eins erhöht und das Programm läuft normal weiter. Bei einer Null wird zum Überspringen des nächsten Befehls der Programmzähler ein zweites mal erhöht und ein NOP Zyklus eingefügt.

```
private void btfsc(int b, int f)
{
    int in = ctr.getMemory().get_Memory(f, b);
    if(in == 0)
    {
        this.ctr.incPC();
        ctr.setNopCycle(true);
    }
    this.ctr.incPC();
}
```

Die Funktion Call wiederum beinhaltet das Speichern des aktuellen Programmzählers + 1 auf dem Stack sowie das Laden des Übergabeparameters k in den Programmzähler. Um den zweiten Zyklus und damit eine korrekte Abarbeitungszeit zu gewährleisten wird ein sogenannter Nop Zyklus aktiviert, dieser setzt somit im Prozessor die Abarbeitung des nächsten Befehls um einen Zyklus aus. In folgendem Bild ist dieser Ablauf zu sehen:

```
private void call(int k)
{
    ctr.getMemory().pushToStack(this.ctr.getMemory().programmcounter+1);
    this.ctr.getMemory().programmcounter = k;
    ctr.setNopCycle(true);
}
```

5.2 Timer

Der Timer bietet die Möglichkeit ein Register hochzuzählen, um bei Überlauf eine Aktion auszulösen.

Dies wurde in einer eigenen Klasse implementiert. In dieser sind Variablen für CLKOUT sowie den RA0 Pin, um deren vorherigen Wert mit dem aktuellen Wert zu vergleichen und anschließend auf Flanken zu überprüfen. Ebenfalls befindet sich dort eine Variable preScaler die hochgezählt wird bis der gewählte PreScaler Wert erreicht ist und anschließend das TMRO Register inkrementiert wird.

Das Prozessor-Objekt ruft nach jedem Zyklus die Funktion updateSources auf, um mögliche Signalwechsel zu erkennen. Anschließend ruft das Prozessor-Objekt die Funktion checkTMRIncrement auf. Diese prüft je nach gewähltem Modus eine Änderung und dessen Flankentyp, um gegebenenfalls das Timer Register zu inkrementieren.

5.3 Interrupts

Auch für die Funktionalität eines Interrupts wurde eine eigene Klasse implementiert. Der Ablauf ist ähnlich der des Timer allerdings wird kein Register inkrementiert, sondern der Programmzähler im Falle eines eingetretenen Interrupts auf die Adresse 0x0004 gesetzt. Ob ein Interrupt eines Pins, Timers oder EEPROM ausgeführt wird entscheiden die Interrupt Enable Bits.

5.4 Watchdog

```
public void update(double timeNow) {
    if(ctr.getMemory().get_WDTE() == 1 && (timeNow - this.timeStamp) >= 18.0) {
        incrementWatchDog();
        this.timeStamp = this.ctr.getOperationalTime();
    }
}

protected void incrementWatchDog()
{
    System.out.println(this.preScaler);
    this.preScaler++;
    int preScalerActive = ctr.getMemory().get_MemoryDIRECT(0x81, 3);
    if((preScalerActive == 1) && this.preScaler >= (Math.pow(2.0, ctr.getPrescaler()))) || preScalerActive == 0)
    {
        System.out.println("Watchdog time-out occurred at time: " + this.ctr.getOperationalTime());
        this.ctr.getMemory().set_TO(0);
        this.ctr.getMemory().set_SRAMDIRECT(0x03, 0);

        if(this.ctr.getProcessor().isInSleep()) {
            this.ctr.wakeUpSleep();
        }else {
            this.ctr.reset();
        }
        this.preScaler = 0;
    }
}
```

Der Watchdog erfüllt den Zweck einen möglichen DeadLock also eine Endlosschleife durch einen Reset aufzulösen. Hierfür ruft das Prozessor-Objekt in jedem Zyklus die update Funktion auf und übergibt abhängig von der gewählten Frequenz die aktuelle Zeit seit Programmstart. Falls die Zeitspanne 18ms überschreitet, wird entweder der zugeordnete Prescaler erhöht oder direkt ein wakeUp bzw. ein Reset ausgeführt.

5.5 Prescaler

Der Prescaler befindet sich sowohl im Timer als auch im Watchdog. Er wird über das PSA Bit im Register 0x81 an den Timer zugewiesen falls das Bit nicht aktiv ist. Andernfalls ist der Prescaler dem Watchdog zugeordnet. Eine prescaler Variable in beiden Klassen wird bei jedem Zyklus erhöht bis bei dem zugewiesenen Baustein der gewählte Wert erreicht wurde oder bei falls beim Baustein kein prescaler zugewiesen ist ohne Beachtung des gewählten Wertes.

Zur Berechnung der Abstufungen wird für den Timer folgende mathematische Gleichung benutzt:

$$f(PS_{value}) = 2^{PS_{value} * 2}$$

Die Berechnung für den Watchdog sieht folgendermaßen aus:

$$f(PS_{value}) = 2^{PS_{value}}$$

Hat die prescaler Variable den errechneten Wert erreicht wird die entsprechende Funktionalität ausgeführt und die Variable auf 0 gesetzt.

5.6 EEPROM

Das EEPROM sorgt für als löschbarer Festwertspeicher dafür, dass Werte auch im Stromlosen Zustand gespeichert werden können. Hierfür haben wir eine Textdatei ins Projekt eingebunden. Diese kann mit der benötigten Write Sequenz beschrieben sowie über Read gelesen werden und bleibt bestehen, auch wenn der Simulator geschlossen ist. Die Datei wird beim Start des Simulators in einen lokalen Member der Klasse EEPROM gespiegelt. Hierdurch muss bei einem Read des EEPROMs

nicht immer direkt von der Datei gelesen werden, wodurch sich Optimierungen der Laufzeit ergeben. Da bei einem Write die Programmierzeit von einer Millisekunde implementiert ist, kann in dieser Zeit das neue Datum in die Datei geschrieben werden.

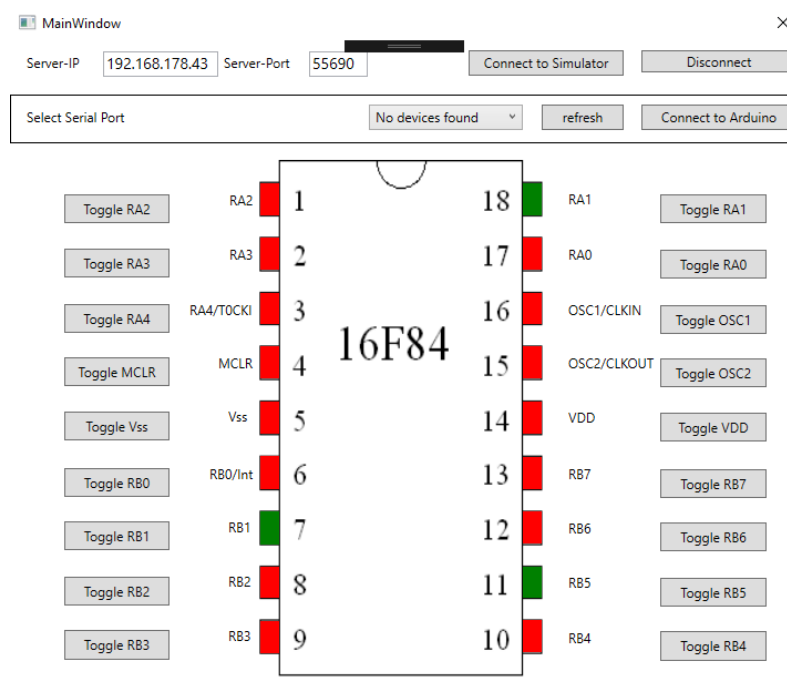
Der Status der Statemachine wird zyklisch bei jedem Befehl aktualisiert. Dabei wird immer die Speicherstelle des EEPROM Data überprüft, somit können auch andere Befehle ausgeführt werden und der Status der Statemachine wird nicht zurückgesetzt. Erst bei einem falschen Wert im EEPROM Data Register wird der Status zurückgesetzt. Beim Erreichen des letzten Status wird der Schreibbefehl ausgeführt und die Textdatei aktualisiert.

5.7 TRIS-Register

Die Latchfunktion der IO Ports wurden mittels eines Zwischenspeichers realisiert. Alle Schreibfunktionen, welche auf die Ports schreiben wollen, schreiben in diesen Zwischenspeicher. Anschließend wird bitweise durch die TRIS-Register iteriert. Von allen Bits, welche hierbei auf Ausgang stehen, wird der Wert aus dem Zwischenspeicher am Pin sowie im Speicher übernommen. Die Werte der eingangs-Bits werden direkt in den Speicher geschrieben, der Zwischenspeicher bleibt hierbei unverändert.

6 Hardwareansteuerung

Als zusätzliche Funktion wurde eine Möglichkeit geschaffen über eine zweite Software Signale der Pins auszutauschen. Im folgenden Bild ist die grafische Oberfläche dieser Software zu sehen.

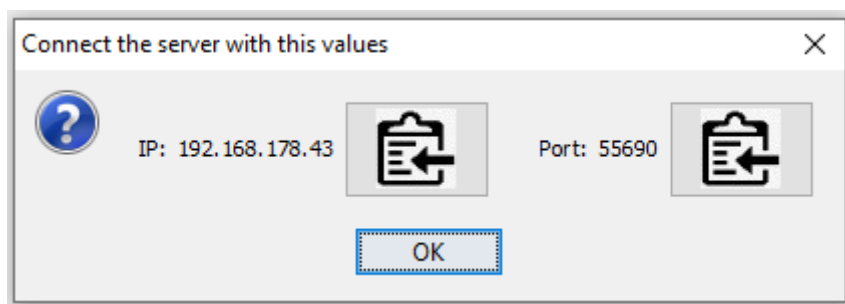


Diese Software wurde in der Programmiersprache C# und mithilfe des WPF Desktopentwicklung Frameworks entwickelt.

Grundlage ist eine TCP-Verbindung zwischen der Simulator-Software und der Hardware-Software. Hier dient der Simulator als TCP-Server und die Hardware-Software stellt den TCP-Client.

Auf Hardwareseite werden geänderte Werte in einer Send Variable gespeichert, um im nächsten zu sendenden Datensatz an den Simulator übertragen zu werden. Wenn der Simulator diesen Wert übernimmt gibt er dies zurück und ändert damit eine Receive Variable. Diese sorgt dafür, dass die gedrückten Pins ihre Farbe entsprechend ändern. Die Farbe Rot steht hier für den Bit Wert 0, die Farbe grün für den Bit Wert 1.

Um sich mit dem Simulator zu verbinden, verbirgt sich im Menü unter dem Reiter Simulator ein Button IO Server. Hiermit öffnet sich ein kleines Fenster in dem die benötigte IP-Adresse sowie der zugehörige Port zu finden ist. Mit den beiden Buttons können diese Werte in die Zwischenablage kopiert werden und müssen nicht abgetippt werden.



7 Fazit

Das Projekt hat einige Stunden an Zeit in Anspruch genommen. Jedoch konnten wir hierdurch unsere Java Kenntnisse sowie die erlernten Projektmanagement Strategien vertiefen.

Eine Hardware in digitaler Form abzubilden ist im Sinne von Industrie 4.0 aktueller denn je zuvor. Der PIC bietet hierfür viele interessante Eckpunkte wie Speicherabbildung oder Befehlsabarbeitung. Das theoretische Wissen konnte vertieft und praxisnah angewendet werden.

Aufgefallen ist uns, dass die Hardware meist sehr simpel arbeitet. An anderen Stellen mussten wir dann aber merken, dass bestimmte Verhalten wie Interrupts oder Timer doch etwas mehr Code benötigen als es auf den ersten Blick vermuten lässt.

Generell konnte die Software gut im 3-Schichten-Modell entwickelt werden. Als Modelle konnten die einzelnen Bestandteile des PIC wie Memory, Processor, usw. möglichst einfach erstellt und über einen Controller verknüpft und dann auch der grafischen Oberfläche zur Verfügung gestellt werden. Da wir nach der Umsetzung des Projekts ein wesentlich besseres Verständnis der Funktionalität sowie des Aufbaus des Mikroprozessors haben, würden wir bei einer erneuten Realisierung das die komplette Struktur im Vorhinein planen.

Durch unsere Verwendung von GitHub als Versionsverwaltung konnten wir sehr gut parallel an unterschiedlichen Stellen arbeiten und so in kurzer Zeit große Fortschritte erzielen. Wir nutzten das Projekt auch um andere Arbeitsweisen auszuprobieren, hier hat sich Extreme Programming bei komplexeren Zusammenhängen als sehr hilfreich erwiesen. Beim Pair Programming – Bestandteil von Extreme Programming – ist ein Teammitglied der aktive Programmierer während ein anderes Teammitglied zuschaut und so logische Fehler fast komplett vermieden werden können.