|  |
| --- |
|  |
| Dokumentation des PIC16C84 Simulators |
| Studiengang Informationstechnik an der  Dualen Hochschule Baden-Württemberg Karlsruhe |
|  |
| **Benedikt Bock, Mario Waxenegger** |
| **19.05.2014** |

|  |
| --- |
|  |

# Realisierung

## Wahl der Programmiersprache und des Grundkonzepts

Bei der Wahl der Programmiersprache für dieses Projekt fiel unsere Wahl auf C++. Diese Sprache haben wir bereits theoretisch im 2. Semester erlernt. Das Projekt bot nun die Möglichkeit auch praktische Erfahrungen mit dieser Sprache zu sammeln.

Durch die Vorlesung Software Engineering I im 3. Semester haben wir bereits Einblicke in das Themenfeld Entwurfsmuster bekommen. Dies führte zu der Idee den Simulator gemäß dem Model-View-Controller-Entwurfsmuster (MVC-Entwurfsmuster) zu strukturieren. Dies hat zur Folge, dass die GUI als eigenständiges Modul aufgefasst wird. Dieses Entwurfsmuster wurde mit Sicherheit, vor allem bei der Trennung von Model und Controller, nicht in Gänze umgesetzt, da dies einen immensen Aufwand nach sich gezogen hätte und nicht in Relation zum eigentlich Projektziel stand.

## Programmstruktur

Das Programm besteht im Wesentlichen aus drei Klassen:

* Mainwindow Klasse
* Steuwerwerk Klasse
* Prozessor Klasse

Die Mainwindow Klasse verwaltet hierbei die GUI und enthält alle nötigen Methoden um die GUI aktuell zu halten. Außerdem enthält die Klasse eine Referenz auf ein Steuerwerkobjekt. Die Steuerwerkklasse übernimmt zum einen die Aufgaben des Controllers aus dem MVC-Entwurfsmuster und zum anderen grob die Aufgaben des Steuerwerks des PIC. Die Mainwindow Klasse ruft Methoden des Steuerwerks auf. Wenn die Steuerwerk Klasse Methoden ausführt, die Inforamtionen ändert, welche für die GUI wichtig sind, dann wird die Mainwindow Klasse benachrichtigt. Diese führt dann entsprechende Aktualisierungen aus.  
Die Prozessor Klasse stellt die Funktionalität der ALU des PIC nach. Im Wesentlichen werden hier die eigentlichen Befehle ausgeführt. Das Steuerwerk enthält ein Prozessorobjekt.

Es gibt noch ein paar zusätzliche Klassen, die im Folgenden aufgeführt werden. Die Prozessor Klasse beinhaltet ein Objekt der Speicher Klasse. Diese Klasse stellt den Speicher (incl. EEPROM) des PIC dar.   
Zusätzlich gibt es eine Codeline Klasse. Ein Objekt dieser Klasse repräsentiert eine Codezeile des PIC-Programms. Dazu werden in dem Objekt die Zeile selbst, die dezimale Befehlsrepräsentation, sowie ein boolescher Wert gespeichert. Der boolesche Wert gibt an ob auf diese Zeile ein Breakpoint gesetzt wurde. Organisiert werden die Objekte in einem Vektor, der im Steuerwerkobjekt hinterlegt ist. Erzeugt wird der Vektor durch eine weitere Klasse namens Parser. Diese Klasse stellt nur eine statische Methode zur Verfügung. Diese Methode liest die angegebene LST-Datei zeilenweise ein und erzeugt die Codelineobjekte.

Abschließend gibt es noch zwei weitere Klassen. Diese stellen allerdings keine eigenständige Funktionalität dar. Viel mehr beschreiben die beiden Klassen Funktionen des Simulators die in einem separaten Thread laufen müssen um die GUI flüssig zu halten. Die eine Klasse heißt Goklasse. Diese implementiert den „Go“ Button (siehe Kapitel goButton). Die andere Klasse ist die Sleep Klasse. Diese implementiert den Befehl „sleep“ (siehe Kapitel sleep).

Im Folgenden werden die wesentlichen Abläufe des Simulators dargestellt. Dabei ist zu beachten, dass hier nicht weiter auf die GUI eingegangen wird. In den Programmablaufplänen sind die Benachrichtigungen an die GUI nicht berücksichtigt.

### executeStep

Diese Funktion führt einen Befehl des Programms aus. Abbildung 1 zeigt den PAP der Methode. Der PAP ist selbsterklärend. Die Funktionsaufrufe (z.B. Interrupthandler) werden im Folgenden aufgeführt. Im Anschluss an den PAP ist der Programmcode zu finden. Funktionsaufrufe mit dem vorangestellten Schlüsselwort „emit“ stellen Benachrichtigungen an die GUI dar.  
Aufgerufen wird diese Methode zum einen durch das Mainwindow wenn der „Execute Step“ Button gedrückt wird und zum anderen zyklisch durch die Goklasse. Siehe hierzu Kapitel goklasse

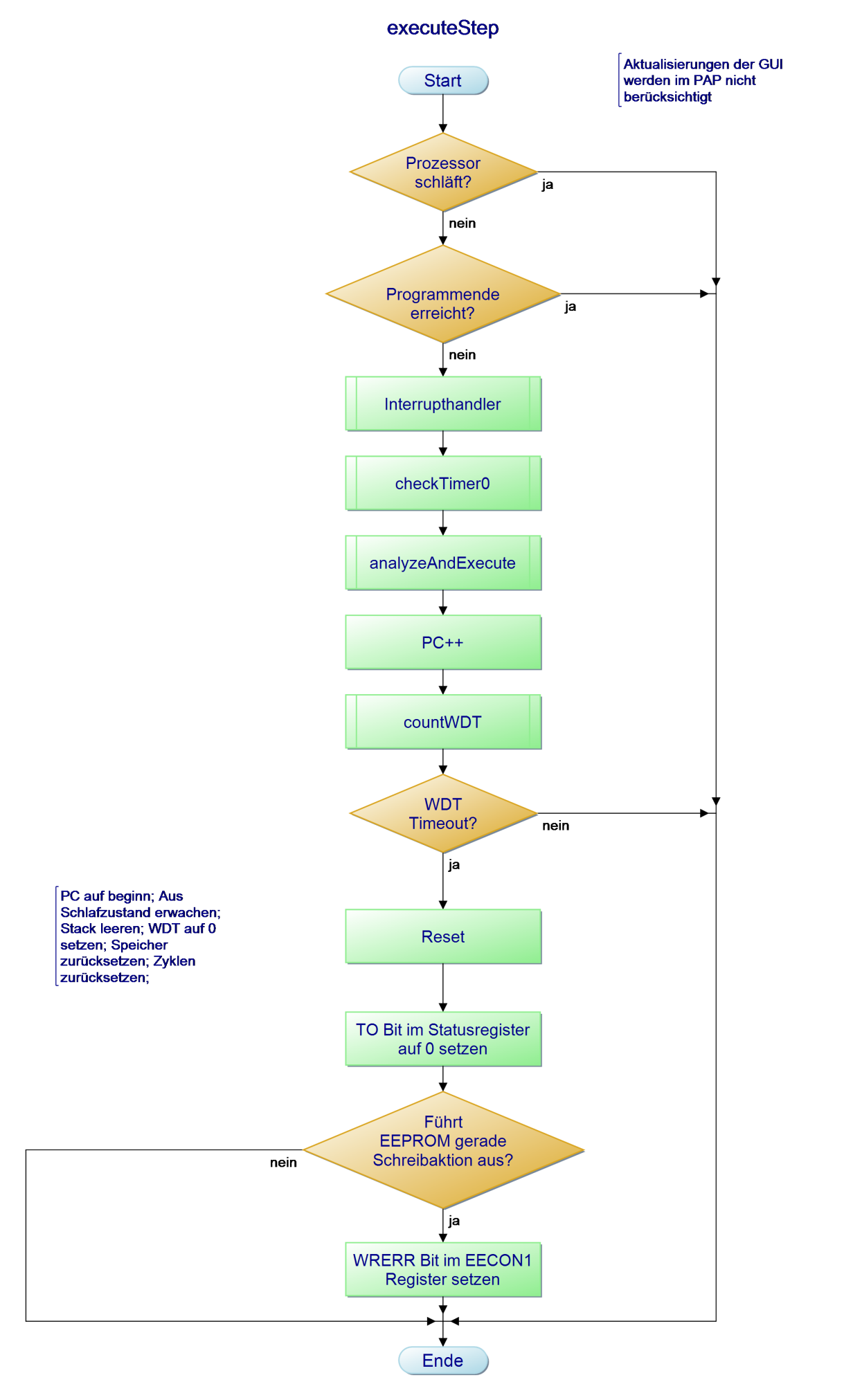


Abbildung : PAP Steuerwerk::executeStep()

|  |  |
| --- | --- |
|  | /\* |
|  | \* Gibt false zurück, wenn PC nicht mehr auf Codezeile zeigt, |
|  | \* d.h. wenn das Programm am Ende angelangt ist. |
|  | \* |
|  | \* Zeigt der PC auf einen gültigen Befehl, wird true zurückgegeben. |
|  | \*/ |
|  | bool Steuerwerk**::**executeStep**(**void**)** |
|  | **{** |
|  | **if(**isSleeping**())** |
|  | **return** **false;** |
|  | **if(**programmEndeErreicht**())** |
|  | **return** **false;** |
|  |  |
|  | testForInterrupt**();** |
|  | **if(**pc**->**breakpoint**)** |
|  | emit setLineColorRed**(**getCurrentLineNumber**()-**1**);** |
|  | **else** |
|  | emit setLineColorWhite**(**getCurrentLineNumber**()-**1**);** |
|  |  |
|  | **if(**pc **!=** maschinencode**.**end**())** |
|  | **{** |
|  | //Ursprünglichen Cycle Wert für Watchdog speichern |
|  | alu**->**vorherigeCycles **=** alu**->**cycles**;** |
|  | checkTimer0**();** |
|  | analyzeAndExecute**(**pc**->**command**);** |
|  | pc**++;** |
|  | alu**->**speicher**.**writePC**(**pc **-** maschinencode**.**begin**());** |
|  | countWDT**();** |
|  | **}** |
|  | **else** |
|  | **return** **false;** |
|  | emit refreshStorageGUI**();** |
|  |  |
|  | **if(**isWDTTimeOut**())** |
|  | **{**//WDT reset, Power on Reset |
|  | emit reset**();** |
|  | alu**->**speicher**.**writeOnBank**(**0**,**3**,**0x0008**);** //TO Bit im Status register clearen |
|  | **if(**int currentEECON1**=**alu**->**speicher**.**readOnBank**(**1**,**8**)&**0x0002**)**// Führt eeprom gerade ein schreibaktion durch? |
|  | alu**->**speicher**.**writeOnBank**(**1**,**8**,**currentEECON1 **|=** 0x0008**);** //-> WRERR bit (EECON1) setzen |
|  | emit refreshStorageGUI**();** |
|  | **return** **false;** |
|  | **}** |
|  |  |
|  | **if(!**programmEndeErreicht**())** |
|  | **{** |
|  |  |
|  | emit setLineColorGreen**(**getCurrentLineNumber**()-**1**);** |
|  | emit gotoLineNumber**(**getCurrentLineNumber**()-**1**);** |
|  | **}** |
|  | **else** |
|  | **{** |
|  | //GoKlasse muss beendet werden |
|  | **return** **false;** |
|  | **}** |
|  |  |
|  | **return** **true;** |
|  | **}** |

Listing : Steuerwerk::executeStep()

### Interrupthandler

Die Funktion Steuerwerk::testForInterrupt() entspricht dem Interrupthandler. Diese Funktion wird in Abbildung 2 als PAP dargestellt und Listing 2 zeigt den entsprechenden Auszug aus dem Programm. Zunächst wird geprüft ob Interrupts global aktiviert sind. Ist dies der Fall, so werden die einzelnen Interrupts des PIC (External RB0 Pin, TMR0 Overflow, PORTB change und EEPROM write complete) geprüft. Ist einer der Interrupts aktiviert und das entsprechende Flagbit gesetzt, so wird die Funktion callInterrupt aufgerufen. Näheres dazu, siehe weiter hinten im Kapitel.

Hier muss noch realisierung der Flags geschrieben werden.

Nach dem Quellcode muss callInterrupt erläutert werden.



Abbildung : PAP Steuerwerk::testForInterrupt()

|  |  |
| --- | --- |
|  | void Steuerwerk**::**testForInterrupt**()** |
|  | **{** |
|  | int intcon **=** alu**->**speicher**.**readOnBank**(**0**,**0x0b**);** //Intconregister lesen |
|  | **if(**intcon **&** 0x0080**)** //ist das GIE Bit gesetzt? |
|  | **{** |
|  | **if((**intcon**&**0x0020**)&&(**intcon**&**0x0004**))** //sind T0IE unf T0IF gesetzt? |
|  | **{** |
|  | callInterrupt**();** |
|  | **return;** |
|  | **}** |
|  | **if((**intcon**&**0x0010**)&&(**intcon**&**0x0002**))** //sind INTE unf INTF gesetzt? |
|  | **{** |
|  | callInterrupt**();** |
|  | **return;** |
|  | **}** |
|  | **if((**intcon**&**0x0008**)&&(**intcon**&**0x0001**))** //sind RBIE unf RBIF gesetzt? |
|  | **{** |
|  | callInterrupt**();** |
|  | **return;** |
|  | **}** |
|  | **if((**intcon**&**0x0040**)&&(**alu**->**speicher**.**readOnBank**(**1**,**0x08**)&**0x0010**))** //sind EEIE unf EEIF (in EECON1 Register) gesetzt? |
|  | **{** |
|  | callInterrupt**();** |
|  | **return;** |
|  | **}** |
|  | **}** |
|  | **}** |

Listing : Steuerwerk::testForInterrupt()